

objc.io | objc 中国

Swift 进阶

已对应 Swift 3

Chris Eidhof, Ole Begemann, Airspeed Velocity 著
王巍 译

英文版本 2.0 (2016 年 9 月), 中文版本 2.1 (2017 年 3 月)

© 2016 Kugler, Eggert und Eidhof GbR

版权所有

ObjC 中国

在中国地区独家翻译和销售授权

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

1 介绍

本书所面向的读者 8

主题 8

术语 11

Swift 风格指南 15

2 内建集合类型

数组 18

字典 35

Set 40

Range 43

3 集合类型协议

序列 47

集合类型 58

索引 69

切片 82

专门的集合类型 88

总结 96

4 可选值

哨岗值 98

通过枚举解决魔法数的问题 100

可选值概览 102

强制解包的时机 127

多灾多难的隐式可选值 131

总结 132

5 结构体和类

值类型 134

可变性 136

结构体 139
写时复制 147
闭包和可变性 155
内存 156
闭包和内存 161
闭包和内存 164
总结 167

6 函数

函数的灵活性 175
局部函数和变量捕获 185
函数作为代理 187
inout 参数和可变方法 193
计算属性和下标 197
自动闭包 202
总结 205

7 字符串

不再固定宽度 207
字符串和集合 211
简单的正则表达式匹配器 217
ExpressibleByStringLiteral 220
String 的内部结构 222
编码单元表示方式 229
CustomStringConvertible 和 CustomDebugStringConvertible 231
文本输出流 233
字符串性能 238
展望 241

8 错误处理

Result 类型 244
抛出和捕获 246
带有类型的错误 248
将错误桥接到 Objective-C 249

错误和函数参数 251
使用 defer 进行清理 253
错误和可选值 254
错误链 256
高阶函数和错误 257
总结 259

9 泛型

重载 261
对集合采用泛型操作 270
使用泛型进行代码设计 282
泛型的工作方式 286
总结 290

10 协议

面向协议编程 294
协议的两种类型 298
带有 Self 的协议 304
协议内幕 306
总结 308

11 互用性

实践：封装 CommonMark 311
低层级类型概览 324
函数指针 326

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。

当我们开始本书第一版的写作的时候，Swift 才刚刚一岁。我们推测这门语言会在进入第二个年头的时候继续高速地发展，不过尽管我们十分犹豫，我们还是决定在 Swift 2.0 测试版发布以前就开始写作。几乎没有别的语言能够在如此短的时间里就能吸引这么多的开发者前来使用。

但是这留给了我们一个问题，你如何写出“符合语言习惯”的 Swift 代码？对某一个任务，有正确的做法吗？标准库给了我们一些提示，但是我们知道，即使是标准库本身也会随时间发生变化，它常常抛弃一切约定，又去遵守另一些约定。不过，在过去两年里，Swift 高速进化着，而优秀的 Swift 代码标准也日益明确。

对于从其他语言迁移过来的开发者，Swift 可能看起来很像你原来使用的语言，特别是它可能拥有你原来的语言中你最喜欢的那一部分。它可以像 C 一样进行低层级的位操作，但又可以避免许多未定义行为的陷阱。Ruby 的教徒可以在像是 map 或 filter 的轻量级的尾随闭包中感受到宾至如归。Swift 的泛型和 C++ 的模板如出一辙，但是额外的类型约束能保证泛型方法在被定义时就是正确的，而不必等到使用的时候再进行判定。灵活的高阶函数和运算符重载让你能够以 Haskell 或者 F# 那样的风格进行编码。最后 @objc 关键字允许你像在 Objective-C 中那样使用 selector 和各种运行时的动态特性。

有了这些相似点，Swift 可以去适应其他语言的风格。比如，Objective-C 的项目可以自动地导入到 Swift 中，很多 Java 或者 C# 的设计模式也可以直接照搬过来使用。在 Swift 发布的前几个月，一大波关于单子 (monad) 的教程和博客也纷至沓来。

但是失望也接踵而至。为什么我们不能像 Java 中接口那样将协议扩展 (protocol extension) 和关联类型 (associated type) 结合起来使用？为什么数组不具有我们预想那样的协变 (covariant) 特性？为什么我们无法写出一个“函子” (functor)？有时候这些问题的答案是 Swift 还没有来得及实现这部分功能，但是更多时候，这是因为在 Swift 中有其他更适合这门语言的方式来完成这些任务，或者是因为 Swift 中这些你认为等价的特性其实和你原来的想象大有不同。

译者注：数组的协变特性指的是，包含有子类型对象的数组，可以直接赋值给包含有父类型对象的数组的变量。比如在 Java 和 C# 中 string 是 object 的子类型，而对应的数组类型 string[] 可以直接赋值给声明为 object[] 类型的变量。但是在 Swift 中，Array<Parent> 和 Array<Child> 之间并没有这样的关系。

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，重载或者是静态调用和动态派发之间的区别这些知识。你可能永远都不会需要去调用 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的性能，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 中会是这个结果”这样的问题，这种问题遍布各个论坛。我们希望你一旦阅读过本书，就能把握这些语言基础的知识，并且了解很多 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，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，或者是想要从其他比如 Java 或者 C++ 这样的语言转行过来的程序员。如果你想要把你的 Swift 相关知识技能提升到和你原来已经熟知的 Objective-C 或者其他语言的同一水平线上的话，这本书会非常适合你。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 Apple 的 Swift 相关书籍（在 [iBooks](#) 以及 [Apple 开发者网站](#) 上均有下载）。如果你很有把握，你可以尝试同时阅读我们的这本书和 Apple 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，但是我们更希望这本书可以对非 Apple 平台的程序员也有所帮助。

主题

我们按照基本概念的主题来组织本书，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，有一些主题可以描绘出 Swift 给人的总体印象：

Swift 既是一门高层级语言，又是一门低层级语言。你可以在 Swift 中用 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 让你有能力快速完成代码编写，并将它们直接编译为原生的二进制可执行文件，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内容。

Swift 是一门多范式的语言。你可以用 Swift 来编写面向对象的代码，也可以使用不变量的值来写纯函数式的程序，在必要的时候，你甚至还能使用指针运算来写和 C 类似的代码。

这是一把双刃剑。好的一面，在 Swift 中你将有很多可用工具，你也不会被限制在一种代码写法里。但是这也让你身临险境，因为可能你实际上会变成使用 Swift 语言来书写 Java 或者 C 或者 Objective-C 的代码。

Swift 仍然可以使用大部分 Objective-C 的功能，包括消息发送，运行时的类型判定，以及 KVO 等。但是 Swift 还引入了很多 Objective-C 中不具备的特性。

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。在 Swift 早期的时候，这门语言为世界带来了很多关于单子 (monad) 的博客。不过等到 Swift 2.0 发布并引入协议扩展的时候，大家研究的趋势也随之变化。

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

富有的 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 的出现比 Lisp 要晚得多，不过，我们能强烈感受到 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本身中定义的，相反，它们是在标准库中被实现的。

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，而使得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；泛型和协议扩展让你免于重复，并且把通用的操作封装到可以复用的方法中去。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) 比较类似。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。但是，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。

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。但是强大能力的背后是更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
    // ptr 只在这个 block 中有效
    // 不过你完全可以将它返回给外部世界：
    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。方法名里已经警告了你这是不安全的，所以对此你需要自己负责。

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，我们有着坚定的自己的看法。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。Swift 还是一门年轻的语言，许多事情还未成定局。更糟糕的是，很多博客或者文章是不正确的，或者已经过时（包括我们曾经写过的一些内容，特别是早期就完成了的内容）。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。带着批判的眼光去审视和思考，并且警惕那些已经过时的信息。

术语

你用，或是不用，术语就在那里，不多不少。你懂，或是不懂，定义就在那里，不偏不倚。

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，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。如果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，并且确定你能接受我们的表述。

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，那就是这个值可能会被程序的两个不同的部分所改变。

类 (class) 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，我们只能在变量里持有对它的引用，然后使用这个引用来访问它。

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 `=` 来判断两个变量是否相等。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，类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。不过类是使用起来最简单引用类型，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处理指针的“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。

我们通过值类型是否执行深复制来对它们分类，判断它们是否具有值语义 (value semantics)。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。

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` 结构体实际上是对引用类型 `NSData` 的一个封装。不过，`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 `NSData` 对象进行深复制。他们使用一种名为“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直接具有的，它需要额外进行实现。

Swift 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。在下一章，我们会讨论 Swift 中的集合类型与 Foundation 框架中 `NSArray` 和 `NSDictionary` 这些集合类型的不同之处。

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所有元素上)，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 (比如把这个函数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。有时候我们只把通过闭包表达式创建的函数叫做“闭包”，不过不要让这种叫法蒙蔽了你的双眼。实际上使用 func 关键字定义的函数也是闭包。

函数是引用类型。也就是说，将一个通过闭合变量保存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。如果一个函数不是接受多个参数，而是只接受部分参数，然后返回一个接受其余参数的函数的话，那么这个函数就是一个**柯里化函数 (curried function)**。我们将在函数中讲解一个方法是如何成为柯里化函数的。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。

自由函数和那些在结构体上调用的方法是**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，编译器可能能够**内联 (inline)** 这些函数，也就是说，完全不去做函数调用，而是将这部分代码替换为需要执行的函数。静态派发还能够帮助编译器丢弃或者简化那些在编译时就能确定不会被执行的代码。

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 @objc 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。另一种方式是函数重载 (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，泛型中的方法在编译期间就是静态已知的。我们会在泛型章节中提及关于这方面的更多内容。

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。
- 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的时候不要使用类型推断。(比如 `func` 就需要显式地指定返回类型)
- 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`。
- 除非一个闭包后面立即跟随有左括号，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。
- 试着去使用 `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。用函数将可变的部分封装起来，可以把它带来的副作用进行隔离。
- Swift 的泛型可能会导致非常长的函数签名。坏消息是我们现在除了将函数声明强制写成几行以外，对此并没有什么好办法。我们会在示例代码中在这点上保持一贯性，这样你能看到我们是如何处理这个问题的。
- 除非你确实需要，否则不要使用 `self.`。在闭包表达式中，使用 `self` 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，而不是写一些全局函数。这有助于提高可读性，让别人更容易发现你的代码。

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。在语言层面上对于不同类型的容器的良好支持，是决定编程效率和幸福指数的重要因素。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话题的内容所投入的精力远超其他部分。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。在下一章中，我们会沿着抽象的阶梯蜿蜒而上，去探究标准库中的集合协议的工作原理。

数组

数组和可变性

在 Swift 中最常用的集合类型非数组莫属。数组是一系列相同类型的元素的有序的容器，对于其中每个元素，我们可以使用下标对其进行访问（这又被称作随机访问）。举个例子，要创建一个数字的数组，我们可以这么写：

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

要是我们使用像是 `append(_)` 这样的方法来修改上面定义的数组的话，会得到一个编译错误。这是因为在上面的代码中数组是用 `let` 声明为常量的。在很多情景下，这是正确的做法，它可以避免我们不小心对数组做出改变。如果我们想按照变量的方式来使用数组，我们需要将它用 `var` 来进行定义：

```
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。使用 `let` 定义的变量因为其具有不变性，因此更有理由被优先使用。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。这在你需要通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` 定义的类实例对象 (也就是说对于引用类型) 时，它保证的是这个引用永远不会发生变化，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。

数组和标准库中的所有集合类型一样，是具有值语义的。当你创建一个新的数组变量并且把一个已经存在的数组赋值给它的时候，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` 末尾的时候，`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；方法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：

```
let a = NSMutableArray(array: [1,2,3])

// 我们不想让 b 发生改变
let b: NSArray = a

// 但是事实上它依然能够被 a 影响并改变
a.insert(4, at: 3)
b // (1, 2, 3, 4)
```

正确的方式是在赋值时，先手动进行复制：

```
let c = NSMutableArray(array: [1,2,3])

// 我们不想让 d 发生改变
let d = c.copy() as! NSArray

c.insert(4, at: 3)
```

```
d // (1, 2, 3)
```

在上面的例子中，显而易见，我们需要进行复制，因为 `a` 的声明毕竟就是可变的。但是，当把数组在方法和函数之间来回传递的时候，事情可能就不那么明显了。

而在 Swift 中，相较于 `NSArray` 和 `NSMutableArray` 两种类型，数组只有一种统一的类型，那就是 `Array`。使用 `var` 可以将数组定义为可变，但是区别于与 `NS` 的数组，当你使用 `let` 定义第二个数组，并将第一个数组赋值给它，也可以保证这个新的数组是不会改变的，因为这里没有共用的引用。

创建如此多的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。

数组和可选值

Swift 数组提供了你能想到的所有常规操作方法，像是 `isEmpty` 或是 `count`。数组也允许直接使用特定的下标直接访问其中的元素，像是 `fibs[3]`。不过要牢记在使用下标获取元素之前，你需要确保索引值没有超出范围。比如获取索引值为 3 的元素，你需要保证数组中至少有 4 个元素。否则，你的程序将会崩溃。

这么设计的主要原因是我们可以使用数组切片。在 Swift 中，计算一个索引值这种操作是非常罕见的：

- 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)`
- 想要列举数组中的元素和对应的下标？`for (num, element) in collection.enumerated()`
- 想要寻找一个指定元素的位置？`if let idx = array.index { 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标准的元素？`array.filter { someCriteria($0) }`

Swift 3 中传统的 C 风格的 `for` 循环被移除了，这是 Swift 不鼓励你去做索引计算的另一个标志。手动计算和使用索引值往往可能带来很多潜在的 bug，所以最好避免这么做。如果这不可

避免的话，我们可以很容易写一个可重用的通用函数来进行处理，在其中你可以对精心测试后的索引计算进行封装，我们将在泛型一章里看到这个例子。

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，你应该已经深思熟虑，对背后的索引计算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。但实际上你是信任你自己的代码的，所以你可能会选择将结果进行强制解包，因为你知道这些下标都是有效的。这一方面十分麻烦，另一方面也是一个坏习惯。当强制解包变成一种习惯后，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，数组根本没有给你可选值的选项。

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**，但是这只是安全性的一个方面。下标操作在**内存安全**的意义上是完全安全的，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。

其他操作的行为略有不同。`first` 和 `last` 属性返回的是可选值，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，`removeLast` 将会导致崩溃，而 `popLast` 将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一个：当你将数组当作栈来使用时，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一定非空，那再去处理可选值就完全没有必要了。

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，并把操作的结果加入到新数组的末尾。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]
```

这种版本有三大优势。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，我们不会再改变它的值，所以也就不再需要用 var 来进行声明了，我们可以将其声明为 let。另外，由于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 的一个扩展，我们将在之后关于编写泛型算法的章节里继续 Sequence 的话题）：

```
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的占位符。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 的具体类型将由调用者传入给 map 的 transform 方法的返回值类型来决定。

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` 将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这个修饰，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` 的代码。是什么让它可以如此通用而且有用？

`map` 可以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代码。`map` 函数通过接受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，我们可以发现很多这样将行为进行参数化的设计模式。标准库中有不下十多个函数接受调用者传入的闭包，并将它作为函数执行的关键步骤：

- **map** 和 **flatMap** — 如何对元素进行变换
- **filter** — 元素是否应该被包含在结果中
- **reduce** — 如何将元素合并到一个总和的值中
- **sequence** — 序列中下一个元素应该是什么？
- **forEach** — 对于一个元素，应该执行怎样的操作
- **sort**, **lexicographicCompare** 和 **partition** — 两个元素应该以怎样的顺序进行排列
- **index**, **first** 和 **contains** — 元素是否符合某个条件
- **min** 和 **max** — 两个元素中的最小/最大值是哪个
- **elementsEqual** 和 **starts** — 两个元素是否相等
- **split** — 这个元素是否是一个分割符

所有这些函数的目的都是为了摆脱代码中那些杂乱无用的部分，比如像是创建新数组，对源数据进行 `for` 循环之类的事情。这些杂乱代码都被一个单独的单词替代了。这可以重点突出那些程序员想要表达的真正重要的逻辑代码。

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" }` 更容易理解。

不过在上面的例子中，它们都只是特殊情况下的简写。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。你可以不对整个元素进行操作，比如，对一个包含人的数组，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。

还有一些其他类似的很有用的函数，可以接受一个闭包来指定行为。虽然它们并不存在于标准库中，但是你可以很容易地自己定义和实现它们，我们也建议你自己尝试着做做看：

- **accumulate** — 累加，和 `reduce` 类似，不过是将所有元素合并到一个数组中，而且保留合并时每一步的值。
- **all(matching:)** 和 **none(matching:)** — 测试序列中是不是所有元素都满足某个标准，以及是不是没有任何元素满足某个标准。它们可以通过 `contains` 和它进行了精心对应的否定形式来构建。
- **count(where:)** — 计算满足条件的元素的个数，和 `filter` 相似，但是不会构建数组。
- **indices(where:)** — 返回一个包含满足某个标准的所有元素的索引的列表，和 `index(where:)` 类似，但是不会在遇到首个元素时就停止。
- **prefix(while:)** — 当判断为真的时候，将元素滤出到结果中。一旦不为真，就将剩余的抛弃。和 `filter` 类似，但是会提前退出。这个函数在处理无限序列或者是延迟计算 (`lazily-computed`) 的序列时会非常有用。
- **drop(while:)** — 当判断为真的时候，丢弃元素。一旦不为真，返回将其余的元素。和 `prefix(while:)` 类似，不过返回相反的集合。

对于上面列表中的部分函数，我们在本书的其他地方进行了定义和实现。`(prefix(while:))` 和 `drop(while:)` 已经在添加到标准库的计划之中，它们没有来得及被添加到 Swift 3.0 中，不过会在未来的版本中被添加。)

有时候你可能会发现你写了好多次同样模式的代码，比如想要在一个逆序数组中寻找第一个满足特定条件的元素：

```
let names = ["Paula", "Elena", "Zoe"]
```

```
var lastNameEndingInA: String?  
for name in names.reversed() where name.hasSuffix("a") {  
    lastNameEndingInA = name  
    break  
}  
lastNameEndingInA // Optional("Elena")
```

在这种情况下，你可以考虑为 Sequence 添加一个小扩展，来将这个逻辑封装到 last(where:) 方法中。我们使用闭包来对 for 循环中发生的变化进行抽象描述：

```
extension Sequence {  
    func last(where predicate: (Iterator.Element) -> Bool) -> Iterator.Element? {  
        for element in reversed() where predicate(element) {  
            return element  
        }  
        return nil  
    }  
}
```

现在我们就能把代码中的 for 循环换成 findElement 了：

```
let match = names.last { $0.hasSuffix("a") }  
match // Optional("Elena")
```

这么做好处和我们在介绍 map 时所描述的一样，相较 for 循环，last(where:) 的版本显然更加易读。虽然 for 循环也很简单，但是在你的头脑里你始终还是要去做个循环，这加重了理解的负担。使用 last(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。

它和 guard 一起也能很好地工作，可能你会想要在元素没被找到的情况下提早结束代码：

```
guard let match = someSequence.last(where: { $0.passesTest() })  
else { return }  
// 对 match 进行操作
```

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` 来执行一些其他操作（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。在上面这样的例子中，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。我们有一个叫做 `forEach` 的函数，看起来很符合我们的需求，但是 `forEach` 本身存在一些问题，我们一会儿会详细讨论。

这种做法和故意给闭包一个局部状态有本质不同。闭包是指那些可以捕获自身作用域之外的变量的函数，闭包再结合上高阶函数，将成为强大的工具。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进行实现：

```
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创建结果数组：

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

要注意的是，这段代码假设了变形函数是以序列原有的顺序执行的。在我们上面的 `map` 中，事实确实如此。但是也有可能对于序列的变形是无序的，比如我们可以有并行处理元素变形的实现。官方标准库中的 `map` 版本没有指定它是否会按顺序来处理序列，不过看起来现在这么做是安全的。

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数组的 `filter` 方法表示：

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

我们可以使用 Swift 内建的用来代表参数的简写 `$0`，这样代码将会更加简短。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，更好的做法应该是就像我们之前那样，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。一个不错的原则是，如果闭包可以很好地写在一行里的话，那么使用简写名会更合适。

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，而不需要引入中间数组。这会使得最终的代码变得更短更易读。比如，寻找 100 以内同时满足是偶数并且是其他数字的平方的数，我们可以对 `0..<10` 进行 `map` 来得到所有平方数，然后再用 `filter` 过滤出其中的偶数：

```
(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
    }
}
```

如果你对在 `for` 中所使用的 `where` 感兴趣的话，可以阅读[可选值](#)一章。

一个关于性能的小提示：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。上面的代码仅仅检查了是否有至少一个元素满足条件，在这个情景下，使用 contains(where:) 更为合适：

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

这种做法会比原来快得多，主要因为两个方面：它不会去为了计数而创建一整个全新的数组，并且一旦匹配了第一个元素，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。

有时候你会发现你想用 contains 完成一些操作，但是写出来的代码看起来很糟糕。比如，要是你想检查一个序列中的所有元素是否全部都满足某个条件，你可以用 !sequence.contains { !condition }，其实你可以用一个更具有描述性名字的新函数将它封装起来：

```
extension Sequence {
    public func all(matching predicate: (Iterator.Element) -> Bool) -> Bool {
        // 对于一个条件，如果没有元素不满足它的话，那意味着所有元素都满足它：
        return !contains { !predicate($0) }
    }
}

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

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值。比如，要是我们想将元素的值全部加起来，可以这样写：

```
var total = 0
for num in fibs {
    total = total + num
}
```

```
total // 12
```

reduce 方法对应这种模式，它把一个初始值 (在这里是 0) 以及一个将中间值 (total) 与序列中的元素 (num) 进行合并的函数进行了抽象。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 的输出值的类型可以和输入的类型不同。举个例子，我们可以将一个整数的列表转换为一个字符串，这个字符串中每个数字后面跟一个空格：

```
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，你可以只使用 reduce 就能实现 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的 for 循环。但是 Swift 不是 Haskell，Swift 的数组并不是列表 (list)。在这里，每次执行 `combine` 函数都会通过在前面的元素之后附加一个变换元素或者是已包含的元素，并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。

flatMap

有时候我们会想要对一个数组用一个函数进行 `map`，但是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，假如我们有一个叫做 `extractLinks` 的函数，它会读取一个 Markdown 文件，并返回一个包含该文件中所有链接的 URL 的数组。这个函数的类型是这样的：

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

如果我们有一系列的 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，我们可以尝试使用 `markdownFiles.map(extractLinks)` 来构建。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，这个数组中的每个元素都是一个文件中的 URL 的数组。为了得到一个包含所有 URL 的数组，你还要对这个由 `map` 取回的数组中的每一个数组用 `joined` 来进行展平 (`flatten`)，将它归并到一个单一数组中去：

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

`flatMap` 将这两个操作合并为一个步骤。`markdownFiles.flatMap(links)` 将直接把所有 Markdown 文件中的所有 URL 放到一个单独的数组里并返回。

flatMap 的实现看起来也和 map 基本一致，不过 flatMap 需要的是一个能够返回数组的函数作为变换参数。另外，在附加结果的时候，它使用的是 append(contentsOf:) 而不是 append(_:)，这样它将能把结果展平：

```
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，然后对另一个进行 map 操作：

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

使用 forEach 进行迭代

我们最后要讨论的操作是 forEach。它和 for 循环的作用非常类似：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。技术上来说，我们可以不假思索地将一个 for 循环替换为 forEach：

```
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。你只需要将函数或者方法直接通过参数的方式传递给 `forEach` 就行了，这可以改善代码的清晰度和准确性。比如在一个 `view controller` 里你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带有条件的 `where` 和 `for` 循环完成的：

```
extension Array where Element: Equatable {
    func index(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 index_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` 并不能返回到外部函数的作用域之外，它仅仅只是返回到闭包本身之外，这和原来的逻辑就不一样了。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，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，它做的仅仅是从闭包中返回。

在某些情况下，比如上面的 `addSubview` 的例子里，`forEach` 可能会更好。它作为一系列链式操作使用时可谓适得其所。想像一下，你在同一个语句中有一系列 `map` 和 `filter` 的调用，这时候你想在调试时打印出操作链中间某个步骤的数组值，插入一个 `forEach` 步骤应该是最快的选择。

不过，因为 `return` 在其中的行为不太明确，我们建议大多数情况下不要用 `forEach`。这种时候，使用常规的 `for` 循环可能会更好。

数组类型

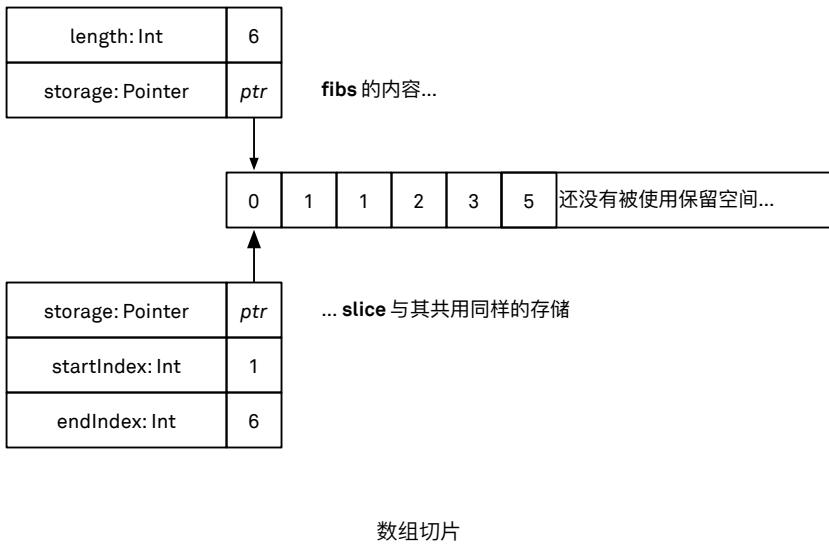
切片

除了通过单独的下标来访问数组中的元素（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的其他元素，我们可以这么做：

```
let slice = fibs[1..<fibs.endIndex]  
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到最后一个元素的数据。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。这意味着原来的数组并不需要被复制。`ArraySlice` 具有的方法和 `Array` 上定义的方法是一致的，因此你可以把它们当做数组来进行处理。如果你需要将切片转换为数组的话，你可以通过将切片传递给 `Array` 的构建方法来完成：

```
Array(fibs[1..<fibs.endIndex]) // [1, 1, 2, 3, 5]
```



桥接

Swift 数组可以桥接到 Objective-C 中。实际上它们也能被用在 C 代码里，不过我们稍后才会涉及到这个问题。因为 NSArray 只能持有对象，所以对 Swift 数组进行桥接转换时曾经有一个限制，那就是数组中的元素需要能被转换为 AnyObject。这限制了只有当数组元素是类实例或者是像是 Int, Bool, String 这样的一小部分能自动桥接到 Objective-C 对应类型的值类型时，Swift 数组才能被桥接。

不过在 Swift 3 中这个限制已经不复存在了。Objective-C 中的 id 类型现在导入 Swift 中时变成了 Any，而不再是 AnyObject，也就是说，任意的 Swift 数组都可以被桥接转换为 NSArray 了。NSArray 本身仍旧只接受对象，所以，编译器和运行时将自动在后台把不匹配的那些值用类来进行包装。反方向的解包同样也是自动进行的。

使用统一的桥接方式来处理所有 Swift 类型到 Objective-C 的桥接工作，不仅仅使数组的处理变得容易，像是字典 (dictionary) 或者集合 (set) 这样的其他集合类型，也能从中受益。除此之外，它还为未来 Swift 与 Objective-C 之间互用性的增强带来了可能。比如，现在 Swift 的值可以桥接到 Objective-C 的对象，那么在未来的 Swift 版本中，一个 Swift 值类型完全有可能去遵守一个被标记为 @objc 的协议。

字典

Swift 中另一个关键的数据结构是 Dictionary，字典。字典包含键以及它们所对应的值。在一个字典中，每个键都只能出现一次。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(true),
    "Name": .text("My iPhone"),
]

defaultSettings["Name"] // Optional(setting.text("My iPhone"))
```

我们使用下标的方式可以得到某个设置的值（比如 defaultSettings["Name"]）。字典查找将返回的是可选值，当特定键不存在时，下标查询返回 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。

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由数组属性计算得来的（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。即使在 name 键下存在某个值，你也还是无法确定 address 键下是否有值。

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。如果想要定义一个可变的字典，你需要使用 `var` 进行声明。想要将某个值从字典中移除，可以用下标将对应的值设为 `nil`，或者调用 `removeValue(forKey:)`。后一种方法除了删除这个键以外，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，想要进行改变的话，首先需要进行复制：

```
var localizedSettings = defaultSettings
localizedSettings["Name"] = "Mein iPhone"
localizedSettings["Do Not Disturb"] = true
```

再次注意，`defaultSettings` 的值并没有改变。和键的移除类似，除了下标之外，还有一种方法可以更新字典内容，那就是 `updateValue(_:forKey:)`，这个方法将在更新之前有值的时候返回这个更新前的值：

```
let oldName = localizedSettings
    .updateValue(.text("Il mio iPhone"), forKey: "Name")
localizedSettings["Name"] // Optional(Setting.text("Il mio iPhone"))
oldName // Optional(Setting.text("Mein iPhone"))
```

有用的字典扩展

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。标准库中并没有这样的函数，不过我们可以自己写一个。

我们扩展 `Dictionary` 类型，为它添加一个 `merge` 方法，该方法接受待合并的字典作为参数。我们可以将这个参数指明为 `Dictionary` 类型，不过更好的选择是用更加通用的泛型方法来进行实现。我们对参数的要求是，它必须是一个序列，这样我们就可以对其进行循环枚举。另外，序列的元素必须是键值对，而且它必须和接受方法调用的字典的键值对拥有相同类型。对于任意的 `Sequence`，如果它的 `Iterator.Element` 是 `(Key, Value)` 的话，它就满足我们的要求，因此我们将其作为泛型的约束（这里的 `Key` 和 `Value` 是我们所扩展的 `Dictionary` 中已经定义的泛型类型参数）：

```
extension Dictionary {
    mutating func merge<S>(_ other: S)
```

```
where S: Sequence, S.Iterator.Element == (key: Key, value: Value) {
    for (k, v) in other {
        self[k] = v
    }
}
```

正如下例所示，我们可以将一个字典合并进另一个字典了。另外，方法的参数还可以是键值对数组或者其他类似的任意序列，而不一定必须是字典：

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

另一个有意思的扩展是从一个 (Key, Value) 键值对的序列来创建字典。标准库中为数组提供了一个类似的初始化方法，我们也经常用到它。当你每次将一个范围转变为数组时 (Array(1...10))，或者是将一个 ArraySlice 转换回数组时 (Array(someSlice))，你都用到了从序列创建数组的方法。但是，对于 Dictionary，还没有这样的初始化方法。(在 Swift-Evolution 的提案中，有人提议添加这样的方法，所以我们有可能在未来能见到这样的初始化方法。)

我们可以先创建一个空字典，然后将序列合并到字典中去。这样一来，我们就可以重用上面的 merge 方法，让它来做实际的工作：

```
extension Dictionary {
    init<S: Sequence>(_ sequence: S)
        where S.Iterator.Element == (key: Key, value: Value) {
            self = [:]
            self.merge(sequence)
    }
}

// 所有 alert 默认都是关闭的
let defaultAlarms = (1..<5).map { (key: "Alarm \($0)", value: false)}
let alarmsDictionary = Dictionary(defaultAlarms)
```

我们要添加的第三个有用扩展是一个 map 函数，它可以用来操作并转换字典中的值。因为 Dictionary 已经是一个 Sequence 类型，它已经有一个 map 函数来产生数组。不过我们有时候

想要的是结果保持字典的结构，只对其中的值进行映射。我们的 `mapValues` 方法将首先调用标准的 `map`，来创建一个(**键, 转换后的值**)的数组。接下来，使用上面定义的初始化方法将其转换回字典：

```
extension Dictionary {  
    func mapValues<newValue>(transform: (Value) -> newValue)  
        -> [Key: newValue] {  
        return Dictionary<Key, newValue>(map { (key, value) in  
            return (key, transform(value))  
        })  
    }  
}  
  
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": "true"]
```

Parcelable 要求

字典其实是哈希表。字典通过键的 `hashValue` 来为每个键指定一个位置，以及它所对应的存储。这也就是 `Dictionary` 要求它的 `Key` 类型需要遵守 `Parcelable` 协议的原因。标准库中所有的基本数据类型都是遵守 `Parcelable` 协议的，它们包括字符串，整数，浮点数以及布尔值。不带有关联值的枚举类型也会自动遵守 `Parcelable`。

如果你想要将自定义的类型用作字典的键，那么你必须手动为你的类型添加 `Parcelable` 并满足它，这需要你实现 `hashValue` 属性。另外，因为 `Parcelable` 本身是对 `Equatable` 的扩展，因此你还需要为你的类型重载 `==` 运算符。你的实现必须保证哈希不变原则：两个同样的实例（由你实现的 `==` 定义相同），**必须** 拥有同样地哈希值。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。

哈希值可能重复这一特性，意味着 `Dictionary` 必须能够处理哈希碰撞。不必说，优秀的哈希算法总是能给出较少的碰撞，这将保持集合的性能特性。理想状态下，我们希望得到的哈希值在

整个整数范围内平均分布。在极端的例子下，如果你的实现对所有实例返回相同的哈希值（比如 0），那么这个字典的查找性能将下降到 $O(n)$ 。

优秀哈希算法的第二个特质是它应该很快。记住，在字典中进行插入，移除，或者查找时，这些哈希值都要被计算。如果你的 `hashValue` 实现要消耗太多时间，那么它很可能会拖慢你的程序，让你从字典的 $O(1)$ 特性中得到的好处损失殆尽。

写一个能同时做到这些要求的哈希算法并不容易。对于一些由本身就是 `Hashable` 的数据类型组成的类型来说，将成员的哈希值进行“异或”（XOR）运算往往是一个不错的起点：

```
struct Person {
    var name: String
    var zipCode: Int
    var birthday: Date
}

extension Person: Equatable {
    static func ==(lhs: Person, rhs: Person) -> Bool {
        return lhs.name == rhs.name
        && lhs.zipCode == rhs.zipCode
        && lhs.birthday == rhs.birthday
    }
}

extension Person: Hashable {
    var hashValue: Int {
        return name.hashValue ^ zipCode.hashValue ^ birthday.hashValue
    }
}
```

异或计算方法的一个限制是，这个操作本身是左右对称的（也就是说 $a \wedge b == b \wedge a$ ），对于某些数据的哈希计算，这有时候会造成不必要的碰撞。你可以添加一个位旋转并混合使用它们来避免这个问题。

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与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 // [2, 3, 1]
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；Set 也支持你在高中数学中学到的那些基本集合操作。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"]
let currentIPods = iPods.subtracting(discontinuedIPods)
// ["iPod shuffle", "iPod nano", "iPod touch"]
```

我们也可以求两个集合的交集，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 touch", "iPod nano"]
```

或者，我们能求两个集合的**并集**，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(discontinued!Pods)
// ["iBook", "iPod mini", "Powerbook", "Power Mac", "iPod Classic"]
```

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。

索引集合和字符集合

`Set` 是标准库中唯一实现了 `SetAlgebra` 的类型，但是这个协议在 `Foundation` 中还被另外两个很有意思的类型实现了：那就是 `IndexSet` 和 `CharacterSet`。两者都是在 Swift 诞生之前很久就已经存在的东西了。包括这两个类在内的其他一些 Objective-C 类现在被完全以值类型的方式导入到 Swift 中，在此过程中，它们还遵守了一些常见的标准库协议。这对 Swift 开发者来说非常友好，这些类型立刻就变得熟悉了。

`IndexSet` 表示了一个由正整数组成的集合。当然，你可以用 `Set<Int>` 来做这件事，但是 `IndexSet` 更加高效，因为它内部使用了一组范围列表进行实现。打个比方，现在你有一个含有 1000 个用户的 `table view`，你想要一个集合来管理已经被选中的用户的索引。使用 `Set<Int>` 的话，根据选中的个数不同，最多可能会要存储 1000 个元素。而 `IndexSet` 不太一样，它会存储连续的范围，也就是说，在选取前 500 行的情况下，`IndexSet` 里其实只存储了选择的首位和末位两个整数值。

不过，作为 `IndexSet` 的用户，你不需要关心这个数据结构的内部实现，所有这一切都隐藏在我们所熟知的 `SetAlgebra` 和 `Collection` 接口之下。(除非你确实需要直接操作内部的范围，对于这种需求，`IndexSet` 暴露了它的 `rangeView` 属性，代表了集合内部的范围)。举例来说，你可以向一个索引集合中添加一些范围，然后对这些索引 `map` 操作，就像它们是独立的元素一样：

```
var indices = IndexSet()
indices.insert(integersIn: 1..5)
indices.insert(integersIn: 11..15)
let evenIndices = indices.filter { $0 % 2 == 0 } // [2, 4, 12, 14]
```

同样地，CharacterSet 是一个高效的存储 Unicode 字符的集合。它经常被用来检查一个特定字符串是否只包含某个字符子集 (比如字母数字 alphanumerics 或者数字 decimalDigits) 中的字符。不过和 IndexSet 有所不同，CharacterSet 并不是一个集合类型。我们会在字符串一章中对 CharacterSet 进行更多讨论。

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 Set 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 Iterator.Element: Hashable {
    func unique() -> [Iterator.Element] {
        var seen: Set<Iterator.Element> = []
        return filter {
            if seen.contains($0) {
                return false
            } else {
                seen.insert($0)
                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背后的技术。

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  
  
// 包含 "z"  
let lowercaseLetters = Character("a")...Character("z")
```

范围看起来很自然地会是一个序列或者集合类型，但是可能出乎你的意料，它并非这两者之一 - 至少不是所有的范围都是序列或者集合类型。

在标准库中，现在有四种范围类型。它们能够被归类到一个 2×2 的矩阵中：

	半开范围	闭合范围
元素满足 Comparable	Range	ClosedRange
元素满足 Strideable (以整数为步长)	CountableRange	CountableClosedRange

矩阵中的列对应了我们上面提到的两种生成范围的操作符，它们分别创建半开的 [Countable]Range 或者闭合的 [Countable]ClosedRange。半开和闭合的范围各有所用：

- 只有半开范围能够表达空区间的概念 (当范围的上下边界相等时，比如 `5..5)。`
- 只有闭合范围能够包含它的元素类型所能表达的最大值 (比如 `0...Int.max`)。半开范围总是最少会有一个值比范围所表达的只要大。

(在 Swift 2 中，即使一个范围是由 `...` 操作符创建的，在技术上来说，所有的范围实际上都是半开的。标准库中曾经用了额外的 HalfOpenInterval 和 ClosedInterval 类型来弥补这个问题，不过在 Swift 3 中它们被移除了。)

上表中的行区分了元素类型仅仅只是满足了 Comparable 的“普通”范围 (这是范围元素的最小要求)，以及那些元素满足 Strideable 并且使用整数作为步长的范围。只有后一种范围是集合类型，它继承了我们在本章中所看到的一系列强大的功能和方法。

Swift 把这些功能更强的范围叫做**可数范围**(Countable Range)，这是因为只有这类范围可以被迭代。对于可数范围，因为类型的 Stride 需要用整数进行约束，所以整数和指针类型是这类范围的有效边界值，但是浮点类型不是。如果你需要用连续的浮点值迭代某个可数范围，你需要使用 `stride(from:to:by)` 和 `stride(from:through:by)` 来创建一个这样的序列。

也就是说，你只能迭代某些范围。比如，我们上面定义的 `Character` 范围值就不是一个序列，下面的代码将无法工作：

```
for char in lowercaseLetters {  
    // ...  
}  
// 错误: 'ClosedRange<Character>' 类型不遵守 'Sequence' 协议
```

(对字符集进行迭代粗看起来应该没什么难度，但是实际却并非如此，这涉及到 `Unicode` 的相关知识，我们会在关于字符串的章节中再深入这个问题。)

不过，下面的这种方式就没有问题，因为整数范围是一个可数范围，它是一个集合类型：

```
singleDigitNumbers.map { $0 * $0 }  
// [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

现在标准库需要将可数范围与其他普通的范围分开，成为 `CountableRange` 和 `CountableClosedRange`。理想状态下，它们不应该被区分对待，但是不这么做的话，我们就需要为泛型参数满足约束的 `Range` 和 `ClosedRange` 添加扩展，来让它们满足集合类型的要求。我们会在下一章详细讨论这个方面的内容，我们现来看看如果这么做的话，代码会是怎样的：

```
// Invalid in Swift 3  
extension Range: RandomAccessCollection  
    where Bound: Strideable, Bound.Stride: SignedInteger  
{  
    // 实现 RandomAccessCollection  
}
```

啊咧，Swift 3 的类型系统并不支持这样的表达方式，你还不能针对泛型参数条件添加扩展，所以这里我们只能使用另外的类型。对于按照条件进行扩展的支持有望在 Swift 4 中被加入，届时 `CountableRange` 和 `CountableClosedRange` 也将会被归类到 `Range` 和 `ClosedRange` 中去。

半开的 `Range` 和闭合的 `ClosedRange` 之间的差异应该会一直存在，这个差异有时候会使得对范围的使用变得十分困难。比如说你有一个方法接受 `Range<Character>` 作为参数，而同时你

想要将我们上面创建的闭合的字符范围传递给它。这时候你会惊奇地发现，这是不可能完成的任务！可能你无法解释，为什么没有一种方法将 `ClosedRange` 转换为 `Range` 呢？如果想要将一个闭合范围转换为等效的半开范围，那么你就需要找到原来的闭合范围上界的后一个元素。除非元素本身满足 `Strideable`，否则那是不可能的。而满足 `Strideable` 的元素所对应的则是可数范围。

也就是说，这个函数的调用者必须提供合适的类型。如果一个函数接受 `Range` 作为参数，那么你就不能用 ... 来创建它。在实践中，我们不太确定这会带来多大的限制，因为大部分的范围都是基于整数的，不过可以肯定的是，这不太符合我们的直觉。

集合类型协议

3

在前一章中，我们看到了 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的。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，它们为什么要以这样的方式工作，以及你如何写出自己的序列和集合类型等话题。

序列

Sequence 协议是集合类型结构中的基础。一个序列 (sequence) 代表的是一系列具有相同类型的值，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)  
}
```

Sequence 协议提供了许多强大的功能，满足该协议的类型都可以直接使用这些功能。上面这样步进式的迭代元素的能力看起来十分简单，但它却是 Sequence 可以提供这些强大功能的基础。我们已经在上一章提到过不少这类功能了，每当你遇到一个能够针对元素序列进行的通用的操作，你都应该考虑将它实现在 Sequence 层的可能性。在本章和书中接下来的部分，我们将会看到许多这方面的例子。

满足 Sequence 协议的要求十分简单，你需要做的所有事情就是提供一个返回迭代器 (iterator) 的 makeIterator() 方法：

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

对于迭代器，我们现在只能从 Sequence 的定义中知道它应该是一个满足 IteratorProtocol 协议的类型。所以我们首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。迭代器每次产生一个序列的值，并且当遍历序列时对遍历状态进行管理。在 IteratorProtocol 协议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?  
}
```

关联类型 `Element` 指定了迭代器产生的值的类型。比如 `String.CharacterView` 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；当你在处理 `Sequence` 的函数签名或者泛型约束的时候，你经常會见到 `Iterator.Element`，实际上这里的 `Element` 就是 `IteratorProtocol` 中所定义的。我们会在本章以及接下来的协议中看到很多关于带有关联类型的协议的内容。

一般来说你只有在想要实现一个你自己的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，这正是 `for` 循环背后帮我们做的事情：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，我们在上面看到的 `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。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，永不枯竭的序列。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：

```
struct ConstantIterator: IteratorProtocol {  
    typealias Element = Int  
    mutating func next() -> Int? {  
        return 1  
    }  
}
```

显示地使用 `typealias` 指定 `Element` 的类型其实并不是必须的（不过通常可以用为文档的目的帮助理解代码，特别是在更大的协议中这点尤为明显）。如果我们去掉它，编译器也将会从 `next()` 的返回值类型中推断出 `Element` 的类型：

```
struct ConstantIterator: IteratorProtocol {
```

```
mutating func next() -> Int? {
    return 1
}
}
```

注意这里 `next()` 被标记为了 `mutating`。对于我们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)
}
```

我们来看一个更有意义的例子。`FibsIterator` 迭代器可以产生一个斐波那契序列。它将记录接下来的两个数字，并作为状态存储，`next` 函数做的事情是为接下来的调用更新这个状态，并且返回第一个数。和之前的例子一样，这个迭代器也将产生“无穷”的数字，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（我们暂时先不考虑这个问题）：

```
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

我们也可以创造有限序列的迭代器，比如下面这个 `PrefixGenerator` 就是一个例子。它将顺次生成字符串的所有前缀（也包含字符串本身）。它从字符串的开头开始，每次调用 `next` 时，会返回一个追加之后一个字符的字符串切片，直到达到整个字符串尾部为止：

```
struct PrefixIterator: IteratorProtocol {
```

```
let string: String
var offset: String.Index

init(string: String) {
    self.string = string
    offset = string.startIndex
}

mutating func next() -> String? {
    guard offset < string.endIndex else { return nil }
    offset = string.index(after: offset)
    return string[string.startIndex..
```

(`string[string.startIndex.. 是一个对字符串的切片操作，它将返回从字符串开始到偏移量为止的子字符串。我们稍后会再对切片进行一些讨论。)`

有了 `PrefixIterator`，定义一个 `PrefixSequence` 类型就很容易了。和之前一样，我们不需要指明关联类型 `Iterator` 的具体类型，因为编译器可以从 `makeliterator` 方法的返回类型中自己推断出来：

```
struct PrefixSequence: Sequence {
    let string: String

    func makeliterator() -> PrefixIterator {
        return PrefixIterator(string: string)
    }
}
```

现在，我们可以使用 `for` 循环来进行迭代，并打印出所有的前缀了：

```
for prefix in PrefixSequence(string: "Hello") {
    print(prefix)
}
/*
H
He
Hel
```

```
Hell  
Hello  
*/
```

我们还可以对它执行 Sequence 提供的所有操作：

```
PrefixSequence(string: "Hello").map { $0.uppercased() }  
// ["H", "HE", "HEL", "HELL", "HELLO"]
```

通过同样的方式，我们可以为 ConstantIterator 和 FibsIterator 创建对应的序列。我们在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

我们至今为止所看到的迭代器都具有值语义。如果你复制一份，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，这是我们所期待的。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，我们首先来看看 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。现在，我们对它进行复制：

```
var i2 = i1
```

现在原有的迭代器和新复制的迭代器是分开且独立的了，在下两次 `next` 时，它们分别都会返回 2 和 3：

```
i1.next() // Optional(2)  
i1.next() // Optional(3)
```

```
i2.next() // Optional(2)  
i2.next() // Optional(3)
```

这是因为 `StrideTolterator` 是一个很简单的结构体，它的实现和我们上面的斐波纳契迭代器没有太大不同，它也具有值语义。

现在我们来看一个不具有值语义的迭代器的例子。`Anyliterator` 是一个对别的迭代器进行封装的迭代器，它可以用来将原来的迭代器的具体类型“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。`Anyliterator` 进行封装的做法是将另外的迭代器包装到一个内部的对象中，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抹消部分的内容。)

这造成了行为上的不同。我们创建一个包装了 `i1` 的 `Anyliterator`，然后进行复制：

```
var i3 = Anyliterator(i1)  
var i4 = i3
```

在这种情况下，原来的迭代器和复制后的迭代器并不是独立的，因为实际的迭代器不再是一个结构体，`Anyliterator` 并不具有值语义。`Anyliterator` 中用来存储原来迭代器的盒子对象是一个类实例，当我们把 `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` 进行的调用，都会增加底层那个相同的迭代器的取值：

```
i3.next() // Optional(4)  
i4.next() // Optional(5)  
i3.next() // Optional(6)  
i3.next() // Optional(7)
```

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。这是因为平时你一般不会在你的代码里把迭代器这样来回传递赋值。你基本上会在本地创建迭代器，这种创建有时候是显式进行的，但更多的时候是通过使用 `for` 循环隐式地进行创建的。你只用它来循环元素，然后就将其抛弃。如果你发现你要与其他对象共享一个迭代器的话，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 还有另一个初始化方法，那就是直接接受一个 next 函数作为参数。与对应的 AnySequence 类型结合起来使用，我们可以做到不定义任何新的类型，就能创建迭代器和序列。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。

因为 AnySequence 提供了一个初始化方法，可以接受返回值为迭代器的函数作为输入，所以创建序列就非常容易了：

```
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 Swift 3 中引入的 sequence 函数。这个函数有两种版本。首先 sequence(first:next:) 将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。在下面的例子中，我们生成了一个随机数的序列，其中每个元素都要比前一个小，直到到达 0 为止：

```
let randomNumbers = sequence(first: 100) { (previous: UInt32) in
    let newValue = arc4random_uniform(previous)
    guard newValue > 0 else {
        return nil
    }
    return newValue
}
Array(randomNumbers) // [100, 83, 27, 13, 10, 2, 1]
```

另一个版本是 `sequence(state:next:)`，因为它可以在两次 `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。我们可以用它来通过以及一个单一的方法调用来构建出斐波纳契序列：

```
let fibsSequence2 = sequence(state: (0, 1)) {
    // 在这里编译器需要一些类型推断的协助
    (state: inout (Int, Int)) -> 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`。这个术语来自函数式编程，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 `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 **展开** 形成一个序列。

这两个 `sequence` 方法非常有用，它们经常用来代替传统的 C 风格的循环，特别是当下标的步长不遵守线性关系的时候。

无限序列

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。

不稳定序列

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来进行建模。但是这些都和数组不太一样，对于数组，你可以多次遍历其中的元素，而上面这些例子中你并非对所有的序列都能这么做。

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，但是像是网络包的流这样的序列将会随着遍历被消耗。你就算再次对其进行迭代，它也不会再次产生同样的值。两者都是有效的序列，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 循环将继续之前的循环迭代或者是从头再次开始：

```
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，而在序列中该属性却不存在。调用一个属性的 getter 方法应该是非破坏的，但序列无法对此进行保证。

举一个破坏性的可消耗序列的例子：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 {  
    return 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型 Unix 中 cat 命令的函数：

```
let numberedStdIn = standardIn.enumerated()

for (i, line) in numberedStdIn {
    print("\(i+1): \(line)")
}
```

enumerate 将一个序列转化为另一个带有递增数字的新序列。和 readLine 进行的封装一样，这里的元素也是延迟生成的。对于原序列的消耗只在你通过迭代器移动到被迭代的序列的下一个值时发生，而不是在序列被创建时发生的。因此，你在命令行中运行上面的代码的话，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，程序会打印出相应的内容。当按下 control-D 结束输入的时候，程序才会停止等待。不过无论如何，每次 enumerate 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。

如果你在写一个 Sequence 的扩展的话，你并不需要考虑这个序列在迭代时是不是会被破坏。但是如果你是一个序列类型上的方法的调用者，你应该时刻提醒自己注意访问的破坏性。

如果一个序列遵守 Collection 协议的话，那就可以肯定这个序列是稳定的了，因为 Collection 在这方面进行了保证。但是反过来却不一定，稳定的序列并不一定需要是满足 Collection。在标准库中，就有一些非集合的序列是可以被多次遍历的。这样的例子包括 stride(from:to:by:) 返回的 StrideTo 类型，以及 stride(from:through:by:) 所返回的 StrideThrough 类型。因为你可以很巧妙地使用浮点数步长来获取值，这使得它们无法被描述为一个集合，所以它们只能作为序列存在。

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？我们为什么不能直接把 IteratorProtocol 的功能包含到 Sequence 中去呢？对于像是前面标准输入例子那种破坏性消耗的序列来说，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稳定序列来说，它的内部状态是不能随着 for 循环而改变的，它们需要独立的遍历状态，这就是迭代器所提供的（当然还需要遍历的逻辑，不过这部分是序列的内容）。makeiterator 方法的目的就是创建这样一个遍历状态。

其实对于所有的迭代器来说，都可以将它们看作是即将返回的元素所组成的**不稳定序列**。事实上，你可以单纯地将迭代器声明为满足 Sequence 来将它转换为一个序列，因为 Sequence 提供了一个默认的 makeIterator 实现，对于那些满足协议的迭代器类型，这个方法将返回 self 本身。Swift 团队也在 swift-evolution 的邮件列表里声明过，要不是由于语言限制的原因，IteratorProtocol 应该继承自 Sequence (具体来说，这个原因是编译器缺乏对于循环的关联类型约束的支持)。

虽然现在不可能添加这样的强制关系，不过标准库中的大部分迭代器还是满足了 Sequence 协议的。

子序列

Sequence 还有另外一个关联类型，叫做 SubSequence：

```
protocol Sequence {
    associatedtype Iterator: IteratorProtocol
    associatedtype SubSequence
    // ...
}
```

在返回原序列的切片的操作中，SubSequence 被用作返回值的子类型，这类操作包括：

- **prefix** 和 **suffix** — 获取开头或结尾 n 个元素
- **dropFirst** 和 **dropLast** — 返回移除掉前 n 个或后 n 个元素的子序列
- **split** — 将一个序列在指定的分隔元素时截断，返回子序列的数组

如果你没有明确指定 SubSequence 的类型，那么编译器会将它推断为 AnySequence<Iterator.Element>，这是因为 Sequence 以这个类型作为返回值，为上述方法提供了默认的实现。如果你想要使用你自己的子序列类型，你必须为这些方法提供自定义实现。

在有些时候，子序列和原来序列的类型一致，也就是 SubSequence == Self 时，会非常方便。在标准库中，String.CharacterView 就是这种情况的一个例子。在关于字符串的章节里，我们会展示一些例子，来说明这个特性是如何帮助我们更舒服地使用字符表示形式的。

在理想世界里，SubSequence 关联类型的声明应该要包括 (a) 其本身也是序列，(b) 子序列的元素类型和其子序列类型，要和原序列中的对应类型一致这两个约束。如果我们加上这些约束，声明看起来会是这样的：

```
// Swift 3.0 中无法编译
associatedtype SubSequence: Sequence
    where Iterator.Element == SubSequence.Iterator.Element,
          SubSequence.SubSequence == SubSequence
```

在 Swift 3.0 中，编译器缺乏两个必要的特性：现在没有循环协议约束 (Sequence 会对自身进行引用)，也没有关联类型中的 where 语句支持。我们希望两者都能在未来的 Swift 版本中引入。在那之前，你需要自己发现这些约束条件，并将它们添加到你自己的 Sequence 扩展方法中，否则编译器将无法理解相关类型。

下面这个例子检查了一个序列的开头和结尾是否以同样的 n 个元素开始。它通过比较序列的 prefix 的 n 个元素以及 suffix 的 n 个元素的逆序序列来实现。用来比较的 elementsEqual 方法只能在我们告诉编译器子序列也是一个序列，并且它的元素类型和原序列的元素类型相同的情况下才能工作 (序列中的类型已经遵守了 Equatable)：

```
extension Sequence
    where Iterator.Element: Equatable,
          SubSequence: Sequence,
          SubSequence.Iterator.Element == Iterator.Element
{
    func headMirrorsTail(_ n: Int) -> Bool {
        let head = prefix(n)
        let tail = suffix(n).reversed()
        return head.elementsEqual(tail)
    }
}
```

```
[1,2,3,4,2,1].headMirrorsTail(2) // true
```

在泛型我们还会有一个这方面的例子。

集合类型

集合类型 (Collection) 指的是那些稳定的序列，它们能够被多次遍历且保持一致。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被获取到。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)，这有时候让使用起来不那么直观。集合的索引值可以构成一个有限的范围，它具有定义好了的开始和结束索引。也就是说，和序列不同，集合类型不能是无限的。

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，如果序列是不稳定的，那么对序列计数将会消耗序列中的元素，这显然不是我们的目的。但是对于稳定的集合类型，我们就可以对其进行计数。

即使你用不到集合类型提供的这些特性，你依旧可以让你自己的序列满足 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，但却必须去选一个合适的索引类型，确实会显得比较奇怪。在实现 Collection 协议时，最难的部分在于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迭代序列的协议，因为它和 Sequence 拥有同样的要求，但是语义却不一样，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 以外，String 的四种表示方式都是集合类型。另外还有 CountableRange 和 UnsafeBufferPointer 也是如此。更进一步，我们可以看到标准库外的一些类型也遵守了 Collection 协议。有两个我们熟知的类型通过这种方法获得了很多新的能力，它们是 Data 和 IndexSet，它们都来自 Foundation 框架。

为了展示 Swift 中集合类型的工作方式，我们将实现一个我们自己的集合类型。可能在 Swift 中标准库中没有被实现，但是最有用的容器类型就是队列了。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，所以移除非数组尾部元素时，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的时间就能完成)。

为队列设计协议

在实际实现队列之前，我们应该先定义它到底是什么。定义一个协议来描述队列到底是什么，会是一个好方法。让我们来看看下面的定义：

```
/// 一个能够将元素入队和出队的类型
protocol Queue {
    /// 在 `self` 中所持有的元素的类型
    associatedtype Element
    /// 将 `newElement` 入队到 `self`
    mutating func enqueue(_ newElement: Element)
    /// 从 `self` 出队一个元素
    mutating func dequeue() -> Element?
}
```

就这么简单，它表述了我们通常所说的队列的定义：这个协议是通用的，通过设定关联类型 Element，它能够包含任意的类型。协议中没有任何关于 Element 的限制，它只需要是某个特定的类型就可以了。

需要特别指出的是，上面协议中方法前的注释非常重要，它和实际的方法名及类型名一样，也是协议的一部分，这些注释用来保证协议应有的行为。在这里，我们并没有给出超过我们现在所做的范围的承诺：我们没有保证 enqueue 或者 dequeue 的复杂度。我们其实可以写一些要求，比如这两个操作的复杂度应该是常数时间 ($O(1)$)。这将会给协议的使用者一个大概的概念，满足这个协议的**所有**类型都应该非常快。但是实际上这取决于最终实现所采用的数据结构，比如对于一个优先队列来说，入队操作的复杂度就可能是 $O(\log n)$ 而非 $O(1)$ 。

我们也没有提供一个 peek 操作来在不出队的前提下检视队列的内容。也就是说，我们的队列定义里就不包含这样的特性，你只能进行出队，而不能 peek。另外，它没有指定这两个操作是否是线程安全的，也没有说明这个队列是不是一个 Collection (虽然我们稍后的实现里将它作为集合类型进行了实现)。

我们也没有说过这是一个先进先出 (FIFO) 队列，它甚至可以是一个后进先出 (LIFO) 队列，这样的话我们就可以用 Array 来实现它了，只需要用 append 来实现 enqueue，然后用结合使用 isEmpty 和 popLast 来实现 dequeue 就行了。

话说回来，这里协议**确实**还是指定了一些东西的：和 Array 的 popLast 类似 (但是和它的 removeLast 不同)，dequeue 返回的是可选值。如果队列为空，那么它将返回 nil，否则它将移除需要出队的元素并返回它。我们没有提供一个 Array.removeLast 那样的当数组为空时就让程序崩溃的类似的方法。

通过将 dequeue 设置为可选值，你可以将这个操作缩短到一行中，同时这种做法是安全的，不会出现错误：

```
while let x = q.dequeue() {  
    // 处理队列元素  
}
```

缺点是就算你已经知道队列**不可能**为空时，你也还是需要进行解包。你最好通过权衡你的使用方式来决定选择合适的数据类型。(不过，如果你的自定义类型满足 Collection 协议的话，你就可以免费得到 popFirst 和 removeFirst 这两个版本的方法了，它们都已经被 Collection 实现了。)

队列的实现

现在我们定义了队列，让我们开始实现它吧。下面是一个很简单的先进先出队列，它的 `enqueue` 和 `dequeue` 方法是基于一系列数组来构建的。

因为我们将队列的泛型占位符命名为了与协议中所要求的关联值一样的 `Element`，所以我们就不需要再次对它进行定义了。这个占位类型并不是一定需要被叫做 `Element`，我们只是随意选择的。我们也可以把它叫做 `Foo`，不过这样的话，我们就还需要定义 `typealias Element = Foo`，或者让 Swift 通过 `enqueue` 和 `dequeue` 的实现所返回的类型来进行隐式的类型推断：

```
/// 一个高效的 FIFO 队列，其中元素类型为 `Element`
struct FIFOQueue<Element>: Queue {
    fileprivate var left: [Element] = []
    fileprivate 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()
            right.removeAll()
        }
        return left.popLast()
    }
}
```

这个实现使用两个栈（两个常规的 Swift 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”栈的反序数组的“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 $O(1)$ 感到有一点奇怪。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，这也同样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过去。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”这个理论已然是有效的。

遵守 Collection 协议

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，想要找出要遵守某个协议所需要提供的最少实现有时候并不容易。

在本章写作的时候，Collection 协议有四个关联类型，四个属性，七个实例方法，以及两个下标方法：

```
protocol Collection: Indexable, Sequence {
    associatedtype Iterator: IteratorProtocol = IndexingIterator<Self>
    associatedtype SubSequence: IndexableBase, Sequence = Slice<Self>
    associatedtype IndexDistance: SignedInteger = Int
    associatedtype Indices: IndexableBase, Sequence = DefaultIndices<Self>

    var first: Iterator.Element? { get }
    var indices: Indices { get }
    var isEmpty: Bool { get }
    var count: IndexDistance { get }

    func makeIterator() -> Iterator
```

```
func prefix(through position: Index) -> SubSequence
func prefix(upTo end: Index) -> SubSequence
func suffix(from start: Index) -> SubSequence
func distance(from start: Index, to end: Index) -> IndexDistance
func index(_ i: Index, offsetBy n: IndexDistance) -> Index
func index(_ i: Index, offsetBy n: IndexDistance, limitedBy limit: Index) -> Index?

subscript(position: Index) -> Iterator.Element { get }
subscript(bounds: Range<Index>) -> SubSequence { get }
}
```

而且它还继承了 `Sequence` 和 `Indexable`，所以想要实现我们自己的类型，我们还需要将那些协议的需求也添加到 `to-do` 列表中。想想就让人绝望吧？

好吧，其实实际上事情没那么糟。可以看到，所有的关联类型都有默认值，所以除非你的类型有特殊的要求，否则你都不必去关心它。对于大部分方法、属性和下标，这一点也同样适用：`Collection` 的协议扩展为我们提供了默认的实现。这些扩展有一些包含关联类型的约束，它们要求协议中的关联类型需要是默认类型；比如，`Collection` 只为那些其中 `Iterator` 为 `IndexingIterator<Self>` 的类型提供 `makeliterator` 方法的默认实现：

```
extension Collection where Iterator == IndexingIterator<Self> {
    /// 返回一个基于集合元素的迭代器
    func makeliterator() -> IndexingIterator<Self>
}
```

如果你的类型需要一个不同的迭代器类型的话，你就需要自己实现这个方法。

寻找哪些是必须的，哪些已经默认提供了，这件事本身并不十分困难，不过它需要很多手工作，而且一旦你不小心，就可能会陷入到和编译器进行猜测的境地，这十分烦人。其中最恼人的部分在于，编译器会将所有的信息提供给你，这些信息其实并没有给我们足够的帮助。

结果，其实你最应该给予希望的是满足一个协议的最小需求已经被写在文档里了，`Collection` 就是这么做的。

** 遵守 `Collection` 协议 **

... 要使你的类型满足 Collection，你可以声明 startIndex 和 endIndex 属性，并提供一个下标方法，使其至少能够读取你的类型的元素。最后，你还需要提供 index(after:) 方法来在你的集合索引之间进行步进。

于是最后，我们需要实现的有：

```
protocol Collection: Indexable, Sequence {  
    /// 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那个索引值  
    func index(after i: Index) -> Index  
    /// 访问特定位置的元素  
    subscript(position: Index) -> Element { get }  
}
```

相比最初 Collection 协议的要求，只有下标被保留了。其他的要求是通过 IndexableBase 以及 Indexable 的路径继承下来的。IndexableBase 和 Indexable 这两个协议可以被看作是只存在于 Swift 3 中的实现细节，这是由于 Swift 3 中缺少对循环协议约束的支持所导致的。我们希望它们能够在 Swift 4 里能被移除，并折叠进 Collection 中。你应该不会需要直接用到这些协议。

我们能够让 FIFOQueue 满足 Collection 了：

```
extension FIFOQueue: Collection {  
    public var startIndex: Int { return 0 }  
    public var endIndex: Int { return left.count + right.count }  
  
    public func index(after i: Int) -> Int {  
        precondition(i < endIndex)  
        return i + 1  
    }  
  
    public subscript(position: Int) -> Element {  
        precondition((0..        if position < left.endIndex {  
            return left[left.count - position - 1]  
        }  
    }  
}
```

```
    } else {
        return right[position - left.count]
    }
}
}
```

我们使用 Int 来作为 Index 类型。这里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。注意索引集合开头开始返回元素的，所以 Queue.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)。

有了这几行代码，我们的队列已经拥有超过 40 个方法和属性供我们使用了。我们可以迭代队列：

```
var q = FIFOQueue<String>()
for x in ["1", "2", "foo", "3"] {
    q.enqueue(x)
}

for s in q {
    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(q) // ["1", "2", "foo", "3"]
a.append(contentsOf: q[2...3]) //
```

我们可以调用那些 Sequence 的扩展方法和属性：

```
q.map { $0.uppercased() } // ["1", "2", "FOO", "3"]
q.flatMap { Int($0) } // [1, 2, 3]
q.filter { $0.characters.count > 1 } // ["foo"]
q.sorted() // ["1", "2", "3", "foo"]
q.joined(separator: " ") // 1 2 foo 3
```

我们也可以调用 Collection 的扩展方法和属性：

```
q.isEmpty // false
q.count // 4
```

```
q.first // Optional("1")
```

遵守 ExpressibleByArrayLiteral 协议

当实现一个类似队列这样的集合类型时，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。要做到这个非常简单：

```
extension FIFOQueue: ExpressibleByArrayLiteral {
    public init(arrayLiteral elements: Element...) {
        self.init(left: elements.reversed(), right: [])
    }
}
```

对于我们的队列逻辑来说，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，但是因为在出队时我们迟早要将它们复制到左侧去，所以直接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 queue: 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，我们可以用它来创建任意的遵守 ExpressibleByArrayLiteral 的类型。在这个字面量里面还包括了其他的字面量类型，比如能够创建任意遵守 ExpressibleByIntegerLiteral 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，那些 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`。但是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 以外的关联类型提供了默认值。虽然你不需要太多关心其他的那些关联类型，不过如果你能简单看看它们到底是什么，会对你理解它们做的事情有所帮助。

Iterator。这是从 Sequence 所继承的关联类型。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>，这个类型是一个很简单的结构体，它是对集合的封装，并用集合本身的索引来迭代每个元素。它的实现非常直接：

```
struct IndexingIterator<Elements: IndexableBase>: IteratorProtocol, Sequence {  
    private let _elements: Elements  
    private var _position: Elements.Index  
  
    init(_elements: Elements) {  
        self._elements = _elements  
        self._position = _elements.startIndex  
    }  
  
    mutating func next() -> Elements._Element? {  
        guard _position < _elements.endIndex else {  
            return nil  
        }  
        let element = _elements[_position]  
        _elements.formIndex(after: &_position)  
        return element  
    }  
}
```

(<Elements: IndexableBase> 这个泛型约束其实应该是 <Elements: Collection>，不过编译器还不允许循环关联类型约束。)

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写一个你自己的迭代器类型。

SubSequence。也是从 `Sequence` 继承的，不过 `Collection` 用更严格的约束重新定义了这个类型。一个集合类型的 `SubSequence` 本身也应该是一个 `Collection` (我们这里使用了“应该”而不是“必须”这个词，是因为现在类型系统中无法表达出这个约束)。它的默认值是 `Slice<Self>`，它是对原来的集合的封装，并存储了切片相对于原来集合的起始和终止索引。

集合类型可以对它的 `SubSequence` 的类型进行自定义，这种情况并不少见，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)。我们将在本章稍后的切片部分再回到这个话题。

IndexDistance。一个有符号整数类型，代表了两个索引之间的步数。应该没有理由去改变它的默认值 `Int`。

Indices。集合的 `indices` 属性的返回值类型。它代表对于集合的所有有效下标的索引所组成的集合，并以升序进行排列。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。

在 Swift 2 中，`indices` 属性返回的是 `Range<Index>`，我们可以用它来迭代集合类型中的所有有效索引。在 Swift 3 里，因为索引自己不再能够进行步进，而是由集合来决定索引的迭代，所以 `Range<Index>` 不再能够进行迭代了。`Indices` 类型也取代了 `Range<Index>` 来使对索引的迭代能够保持工作。

`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。

在结构体和类中我们会涉及写时复制的内容，现在，你只需要知道如果在为自定义集合提供另外的 `Indices` 类型作为替换的话，你不需要让它保持对原集合的引用，这样做可以带来性能上的提升。这对于那些计算索引时不依赖于集合本身的集合类型都是有效的，比如数组或者我们的队列就是这样的例子。如果你的索引是整数类型，你可以直接使用 `CountableRange<Index>`：

```
extension FIFOQueue: Collection {  
    ...  
    typealias Indices = CountableRange<Int>  
    var indices: CountableRange<Int> {
```

```
    return startIndex..
```

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之后的位置。所以 `endIndex` 并不是一个有效的下标索引，你可以用它来创建索引的范围 (`someIndex..)，或者将它与别的索引进行比较，比如来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我们集合的索引。Array 如此，我们的 `FIFOQueue` 类型在经过一些操作之后亦是如此。整数索引十分直观，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。但是这是行不通的，因为你无法增加一个键，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，不过它包含了集合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，因为字典可能被传递到 Objective-C 的 API 中去，也可能是由 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，因为(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，所以它总是返回非可选值：

```
protocol Collection {  
    subscript(position: Index) -> Element { get }  
}
```

返回的类型是 `Element`。对于字典来说，`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`。

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“安全”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的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，一个集合通常不能够区分它“自己的”索引和一个来自同样类型集合的索引之间的区别。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.index(of: 4)! // 3  
squares[numbersIndex] // 16
```

这对那些像是 `String.Index` 的不透明的索引类型也是适用的。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，你能这么做并不意味着这么做会是个好主意。如果我们用这个索引来通过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。

在集合之间共享索引是有其合理用法的，在切片上我们会大量使用这种方式。子序列通常会与它们的原序列共享底层的存储，所以原序列的索引在切片中也会定位到同样的元素。

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 2 的时候，索引是可以自己进行步进移动的。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都是一些简单的加法运算。但是比如对于一个字符串索引，因为字符在 Swift 中是尺寸可变的，所以计算索引时需要考虑实际字符的数据到底是什么。

在原来的自己管理的索引模型中，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。当你实现你自己的索引类型时，请记住尽可能地避免持有集合类型的引用。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，有时候它的语法会显得有一些啰嗦。

链表

作为非整数索引的例子，让我们来实现最基础的集合类型：单向链表。在此之前，我们先来看看实现数据结构的另一种方法：使用间接枚举 (indirect enum)。

一个链表的节点有两种可能：要么它有值，并且指向对下一个节点的引用，要么它代表链表的结束。我们可以这样来定义它：

```
/// 一个简单的链表枚举
enum List<Element> {
    case end
    indirect case node(Element, next: List<Element>)
}
```

在这里使用 `indirect` 关键字可以告诉编译器这个枚举值应该被看做引用。Swift 的枚举是值类型，这意味着一个枚举将直接在变量中持有它的值，而不是持有一个指向值位置的引用。这样做有很多好处，我们在结构体和类一章中介绍它们。但是这同时也意味着它们不能包含一个对自己的引用。`indirect` 关键字允许一个枚举成员能够持有引用，这样一来，它就能够持有自己。

我们通过创建一个新的节点，并将 `next` 值设为当前节点的方式来在链表头部添加一个节点。为了使用起来简单一些，我们为它创建了一个方法。我们将这个添加方法命名为 `cons`，这是因为在 LISP 中这个操作就是叫这个名字 (它是“构造”(construct) 这个词的缩写，在列表前方追加元素的操作有时候也被叫做“consing”)：

```
extension List {
    /// 在链表前方添加一个值为 `x` 的节点，并返回这个链表
    func cons(_ x: Element) -> List {
        return .node(x, next: self)
    }
}

// 一个拥有 3 个元素的链表 (3 2 1)
let list = List<Int>.end.cons(1).cons(2).cons(3)
// node(3, List<Swift.Int>.node(2, List<Swift.Int>.node(1, List<Swift.Int>.end)))
```

链式的语法调用使链表的构建过程十分清晰，但是它看起来却十分丑陋。和我们的队列类型一样，我们可以为它添加 `ExpressibleByArrayLiteral`，这样我们就能用数组字面量的方式来初始化一个链表了。我们首先对输入数组进行逆序操作 (因为链表是从末位开始构建的)，然后使用 `reduce` 以及 `.end` 为初始值，来将元素一个一个地添加到链表中：

```

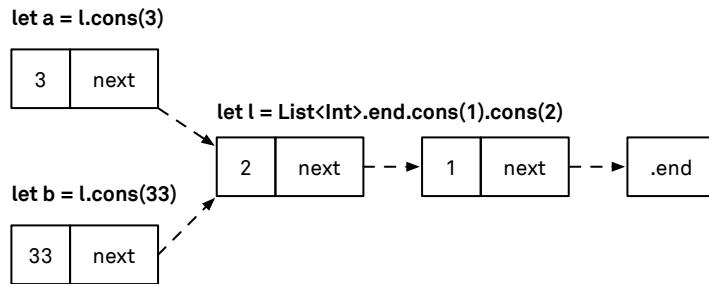
extension List: ExpressibleByArrayLiteral {
    init(arrayLiteral elements: Element...) {
        self = elements.reversed().reduce(.end) { partialList, element in
            partialList.cons(element)
        }
    }
}

let list2: List = [3,2,1]
// node(3, List<Swift.Int>.node(2, List<Swift.Int>.node(1, List<Swift.Int>.end)))

```

这个列表类型有一个很有趣的特性，它是“持久”的。节点都是不可变的，一旦它们被创建，你就无法再进行更改了。将另一个元素添加到链表中并不会复制这个链表，它仅仅只是给你一个链接在既存的链表的前端的节点。

也就是说，两个链表可以共享链表尾：



List Sharing

链表的不可变性的关键就在这里。假使你可以改变链表（比如移除最后的条目，或者对某个节点的元素进行升级等），那么这种共享就将出现问题 — x 将会改变链表，而改变将会影响到 y。

栈

链表其实是一个栈，构建链表相当于进栈，获取下一个值相当于出栈。我们前面提到过，数组也可以看作是栈。现在让我们像对待队列那样，来定义一个通用的栈的协议：

```
/// 一个进栈和出栈都是常数时间操作的后进先出 (LIFO) 栈
protocol Stack {
    /// 栈中存储的元素的类型
    associatedtype Element

    /// 将 `x` 入栈到 `self` 作为栈顶元素
    /// - 复杂度: O(1).
    mutating func push(_ x: Element)

    /// 从 `self` 移除栈顶元素，并返回它
    /// 如果 `self` 是空，返回 `nil`
    /// - 复杂度: O(1)
    mutating func pop() -> Element?
}
```

我们这次在 `Stack` 的协议定义的文档注释中做了详细一些的约定，包括给出了性能上的保证。

`Array` 可以遵守 `Stack`:

```
extension Array: Stack {
    mutating func push(_ x: Element) { append(x) }
    mutating func pop() -> Element? { return popLast() }
}
```

`List` 也行:

```
extension List: Stack {
    mutating func push(_ x: Element) {
        self = self.cons(x)
    }

    mutating func pop() -> Element? {
        switch self {
            case .end: return nil
            case let .cons(x, xs): return x
        }
    }
}
```

```
case let .node(x, next: xs):
    self = xs
    return x
}
}
}
```

但是我们刚才才说了列表是不可变的，否则它就无法稳定工作了。一个不可变的列表怎么能有被标记为可变的方法呢？

实际上这并没有冲突。这些可变方法改变的不是列表本身的内容。它们改变的只是变量所持有的列表的节点到底是哪一个。

```
var stack: List<Int> = [3,2,1]
var a = stack
var b = stack
```

```
a.pop() // Optional(3)
a.pop() // Optional(2)
a.pop() // Optional(1)
```

```
stack.pop() // Optional(3)
stack.push(4)
```

```
b.pop() // Optional(3)
b.pop() // Optional(2)
b.pop() // Optional(1)
```

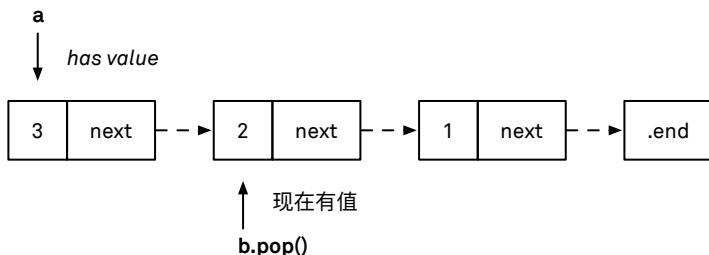
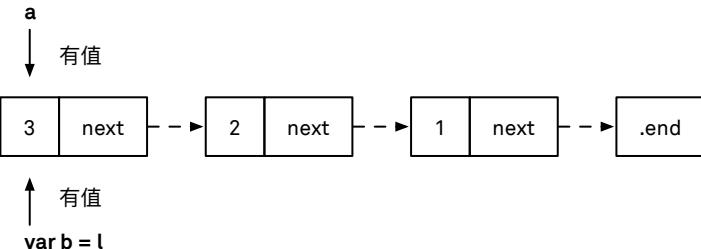
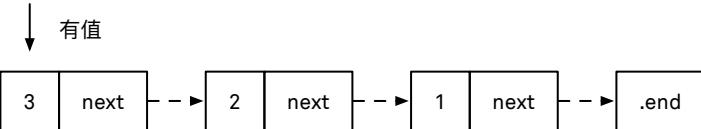
```
stack.pop() // Optional(4)
stack.pop() // Optional(2)
stack.pop() // Optional(1)
```

这足以说明值和变量之间的不同。列表的节点是值，它们不会发生改变。一个存储了 3 并且指向确定的下一个节点的节点永远不会变成其他的值，就像数字 3 不会发生改变一样，这个节点也不会发生改变。虽然这些值是可以通过引用的方式互相关联，但是这并不改变它们是结构体的事实，它们所表现出的也完全就是值类型的特性。

另一方面，变量 a 所持有的值是可以改变的，它能够持有任意一个我们可以访问到节点值，也可以持有 end 节点。不过改变 a 并不会改变那些节点，它只是改变 a 到底是持有哪个节点。

这正是结构体上的可变方法所做的事情，它们其实接受一个隐式的 inout 的 self 作为参数，这样它们就能够改变 self 所持有的值了。这并不改变列表，而是改变这个变量现在所呈现的是列表的哪个部分。这样一来，通过 indirect，变量就变成链表的迭代器了：

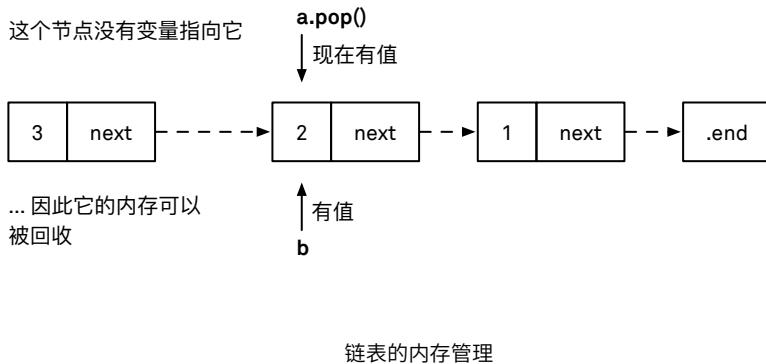
```
let a = List<Int>.end.cons(1).cons(2).cons(3)
```



链表迭代器

当然，你可以使用 `let` 而不是 `var` 来声明这些变量。这样一来，这些变量将变为常量，当它们所持有的值一旦设定以后，也不能再被更改了。不过 `let` 是和变量相关的概念，它和值没什么关系。值天生就是不能变更的，这是定义使然。

现在，一切就顺理成章了。在实际中，这些互相引用的节点会被放在内存中。它们会占用一些内存空间，如果我们不再需要它们，那么这些空间应该被释放。Swift 使用自动引用计数 (ARC) 来管理节点的内存，并在不再需要的时候释放它们：



我们会在函数中详细介绍 `input`，对于可变方法和 ARC 的有关内容，我们也将在结构体和类再深入探讨。

让 List 遵守 Sequence

因为列表的变量可以进行列举，所以你能够使用它们来让 List 遵守 Sequence 协议。事实上，和我们在序列和迭代器的关系中看到过的一样，List 是一个拥有自己的迭代状态的不稳定序列。我们只需要提供一个 `next()` 方法，就能一次性地使它遵守 `IteratorProtocol` 和 `Sequence` 协议。实现的方式就是使用 `pop()`：

```
extension List: IteratorProtocol, Sequence {
    mutating func next() -> Element? {
        return pop()
    }
}
```

现在，你能够在列表上使用 `for...in` 了：

```
let list: List = ["1", "2", "3"]
for x in list {
    print("\u2022(x)", terminator: "")
}
// 1 2 3
```

同时，得益于协议扩展的强大特性，我们可以在 `List` 上使用很多标准库中的函数：

```
list.joined(separator: ",") // 1,2,3
list.contains("2") // true
list.flatMap { Int($0) } // [1, 2, 3]
list.elementsEqual(["1", "2", "3"]) // true
```

让 `List` 遵守 `Collection`

接下来，我们要让 `List` 实现 `Collection`。要做到这一点，我们需要确定 `List` 的索引类型。我们说过，对于集合来说，最好的索引类型是关于集合存储的一个简单的整数偏移值，但是在这里并不适用，因为链表并不是连续的存储结构。想要通过基于整数的索引（比如从链表开头节点到某个特定节点所需要步数）来获取元素，每次都只能对链表从 `startIndex` 开始进行遍历，这会使得下标索引变成一个 $O(n)$ 操作。然而，`Collection` 的文档指出，下标访问应该是 $O(1)$ 的，“因为很多集合操作都依赖于 $O(1)$ 的下标操作，否则它们也将无法达到保证的性能。”

如此一来，我们的索引必须要直接引用链表的节点。这不会造成性能问题，因为 `List` 是不可变的，它没有使用任何的写时复制优化。

因为我们已经将 `List` 直接作为迭代器来使用了，采用同样的方式，将枚举本身作为索引来使用也是一件很诱惑人的事。不过这可能会导致问题。比如索引和集合需要的 `==` 实现并不相同：

- 索引需要知道从相同列表中取出的两个索引是否在相同的位置。它不需要列表中的元素满足 `Equatable`。
- 而对于集合来说，它应该可以比较两个不同的列表，检查它们是否含有相同的元素。这需要列表元素满足 `Equatable`。

通过创建不同的类型来分别代表索引和集合，我们可以为它们实现两个不同的 `==` 运算符。另外，由于索引不是节点枚举，我们可以将节点的实现标记为私有，从而将这个细节隐藏起来。新的 `ListNode` 类型看起来和我们一开始的 `List` 非常相似：

```
/// List 集合类型的私有实现细节
fileprivate enum ListNode<Element> {
    case end
    indirect case node(Element, next: ListNode<Element>)

    func cons(_ x: Element) -> ListNode<Element> {
        return .node(x, next: self)
    }
}
```

索引类型封装了 `ListNode`。某个类型必须满足 `Comparable`，才能成为集合类型的索引。而 `Comparable` 有两个要求：它需要实现一个小于运算符 (`<`) 和一个从 `Equatable` 继承的等于运算符 (`==`)。一旦有了这两个运算符实现，另外的像是 `>`，`<=` 和 `>=` 都可以使用默认的实现。

要实现 `==` 和 `<`，我们还需要一些额外的信息。我们前面提到过，节点是值，值是不具有同一性的。那我们怎么说明两个变量是指向同一个节点的呢？为了解决这个问题，我们使用一个递增的数字来作为每个索引的标记值 (tag) (`.end` 节点的标记值为 0)。我们之后会看到，在索引中存储这些标记可以让操作变得非常高效。列表的工作原理决定了如果相同列表中的两个索引拥有同样的标记值，那么它们就是同一个索引：

```
public struct ListIndex<Element>: CustomStringConvertible {
    fileprivate let node: ListNode<Element>
    fileprivate let tag: Int

    public var description: String {
        return "ListIndex(\(tag))"
    }
}
```

另一个值得注意的地方是，虽然 `ListIndex` 是被标记为公开的结构体，但是它有两个私有属性 (`node` 和 `tag`)。这意味着该结构体不能被从外部构建，因为它的“默认”构造函数 `ListIndex(node:tag:)` 对外部用户来说是不可见的。你可以从一个 `List` 中拿到 `ListIndex`，但是你不能自己创建一个 `ListIndex`。这是非常有用的技术，它可以帮助你隐藏实现细节，并提供安全性。

我们还需要实现 `Equatable`。我们上面已经讨论过，通过检查 `tag` 标记值就可以判定它们是否相等了：

```
extension ListIndex: Comparable {
```

```

public static func == <T>(lhs: ListIndex<T>, rhs: ListIndex<T>) -> Bool {
    return lhs.tag == rhs.tag
}

public static func <<T>>(lhs: ListIndex<T>, rhs: ListIndex<T>) -> Bool {
    // startIndex 的 tag 值最大, endIndex 最小
    return lhs.tag > rhs.tag
}
}

```

现在我们有合适的索引类型了，下一步是创建满足 Collection 协议的 List 结构体：

```

public struct List<Element>: Collection {
    // Index 的类型可以被推断出来，不过写出来可以让代码更清晰一些
    public typealias Index = ListIndex<Element>

    public let startIndex: Index
    public let endIndex: Index

    public subscript(position: Index) -> Element {
        switch position.node {
            case .end: fatalError("Subscript out of range")
            case let .node(x, _): return x
        }
    }

    public func index(after idx: Index) -> Index {
        switch idx.node {
            case .end: fatalError("Subscript out of range")
            case let .node(_, next): return Index(node: next, tag: idx.tag - 1)
        }
    }
}

```

注意 List 除了 startIndex 和 endIndex 以外并不需要存储其他东西。因为索引封装了列表节点，而节点又相互联系，所以你可以通过 startIndex 来访问到整个列表。endIndex 对于所有的实例(至少在我们之后谈及切片前)来说，都是相同的 ListIndex(node: .end, tag: 0)。

为了让创建列表变得简单一些，我们还实现了 ExpressibleByArrayLiteral：

```
extension List: ExpressibleByArrayLiteral {
    public init(arrayLiteral elements: Element...) {
        startIndex = ListIndex(node: elements.reversed().reduce(.end) {
            partialList, element in
            partialList.cons(element)
        }, tag: elements.count)
        endIndex = ListIndex(node: .end, tag: 0)
    }
}
```

从 Sequence 继承得到的能力可以让我们很容易地写出一个简单的 description 实现，从而能更好地进行调试输出。我们将列表元素进行 map 操作，将它们转为字符串的表示形式，然后再拼接成一个单独的字符串：

```
extension List: CustomStringConvertible {
    public var description: String {
        let elements = self.map { String(describing: $0) }
            .joined(separator: ", ")
        return "List: (\(elements))"
    }
}
```

现在，我们的列表可以使用 Collection 的扩展了：

```
let list: List = ["one", "two", "three"] // List: (one, two, three)
list.first // Optional("one")
list.index(of: "two") // Optional(ListIndex(2))
```

除此之外，因为 tag 计算了添加到 .end 之前的节点的数量，所以 List 可以拥有一个常数时间的复杂度 count 属性。而一般的链表的 count 是 $O(n)$ 复杂度的：

```
extension List {
    public var count: Int {
        return startIndex.tag - endIndex.tag
    }
}
list.count // 3
```

被减去的列表末尾的 tag 值，到现在为止都一定会是 0。我们在这里将它写为更一般的 endIndex，是因为我还想要支持马上就要提到的切片操作。

最后，因为 List 和 ListIndex 是两个不同类型，我们可以为 List 实现一个不同的 == 运算符，用来比较集合中的元素：

```
public func == <T: Equatable>(lhs: List<T>, rhs: List<T>) -> Bool {  
    return lhs.elementsEqual(rhs)  
}
```

在一个完美的类型系统中，我们不仅会去重载 ==，还应该将 List 本身声明为 Equatable 的同时，将它的 Element 类型也约束为 Equatable，就像这样：

```
extension List: Equatable where Element: Equatable {}  
// 错误: 含有约束的 'List' 类型不能拥有继承语句关系
```

如果能做到这样，我们就将可以比较一系列的列表，或者将 List 用在其他任何需要 Equatable 的地方。不过现在这门语言还不支持这样的约束方式。不过，按条件遵守的协议是一个被高度期望的特性，很可能我们在 Swift 4 中可以看到它的出现。

切片

所有的集合类型都有切片操作的默认实现，并且有一个接受 Range<Index> 作为参数的下标方法。下面的操作等价于 list.dropFirst():

```
let list: List = [1,2,3,4,5]  
let onePastStart = list.index(after: list.startIndex)  
let firstDropped = list[onePastStart..Array(firstDropped) // [2, 3, 4, 5]
```

因为像是 list[somewhere..

```
let firstDropped2 = list.suffix(from: onePastStart)
```

默认情况下，firstDropped 不是一个列表，它的类型是 Slice<List<String>>。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
    typealias IndexDistance = Base.IndexDistance

    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.Iterator.Element {
        return collection[position]
    }

    typealias SubSequence = Slice<Base>

    subscript(bounds: Range<Base.Index>) -> Slice<Base> {
        return Slice(base: collection, bounds: bounds)
    }
}

```

除了保存对原集合类型的引用之外，Slice 还存储了切片边界的开始索引和结束索引。所以在 List 的场合，因为列表本身是由两个索引组成的，所以切片占用的大小也将会是原来列表的两倍：

```

//一个拥有两个节点 (start 和 end) 的列表的大小:
MemoryLayout.size(ofValue: list) //32

//切片的大小是列表的大小再加上子范围的大小
//(两个索引之间的范围。在 List 的情况下这个范围也是节点)

```

```
MemoryLayout.size(ofValue: list.dropFirst()) // 64
```

实现自定义切片

我们可以改善这一点，因为列表可以通过返回自身，但是持有不同的起始索引和结束索引来表示一个子序列，我们可以用自定义的 List 方法来进行实现：

```
extension List {  
    public subscript(bounds: Range<Index>) -> List<Element> {  
        return List(startIndex: bounds.lowerBound, endIndex: bounds.upperBound)  
    }  
}
```

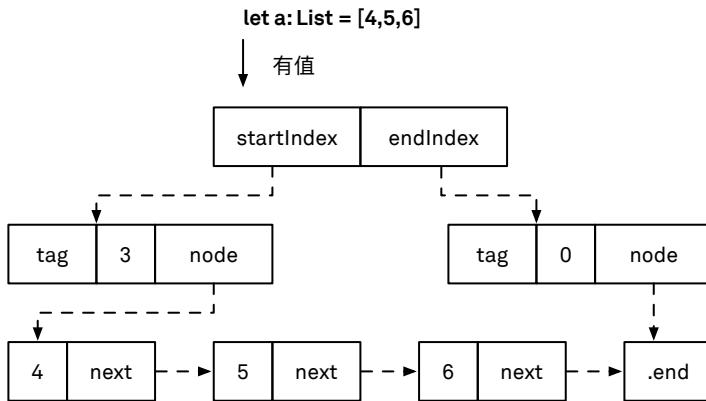
在这样的实现下，列表切片也就变成了列表本身，所以它们尺寸依然只有 32 字节：

```
let list: List = [1,2,3,4,5]  
MemoryLayout.size(ofValue: list.dropFirst()) // 32
```

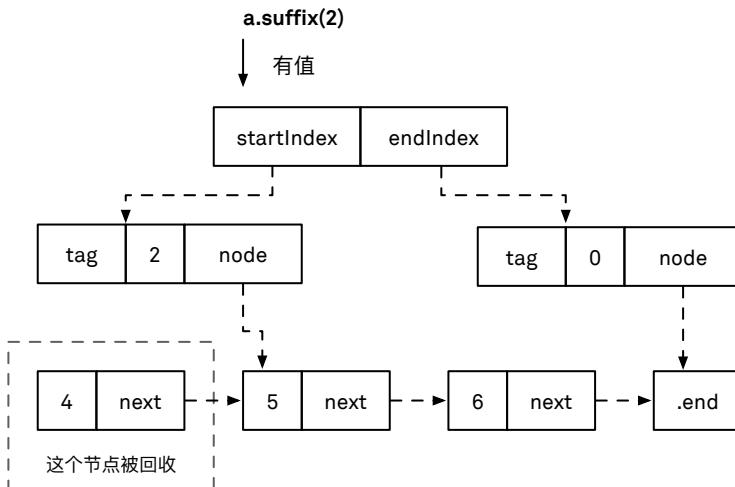
也许除了尺寸的优化以外，这么做带来的更重要的好处是现在序列和集合的子序列将使用同样的类型，你不必再处理另外的类型。比如，你精心设计的 CustomStringConvertible 实现现在可以直接用在子序列上，而不必增加额外的代码了。

另一件需要考虑的事情是，对于包括 Swift 数组和字符串在内的很多可以被切片的容器，它们的切片将和原来的集合共享存储缓冲区。这会带来一个不好的副作用：切片将在它的整个生命周期中持有集合的缓冲区，而不论集合本身是不是已经超过了作用范围。如果你将一个 1 GB 的文件读入到数组或者字符串中，然后获取了它很小的一个切片，整个这 1 GB 的缓冲区会一直存在于内存中，直到集合和切片都被销毁时才能被释放。这也是 Apple 在文档中特别警告“只应当将切片用作临时计算的目的” 的原因。

对于 List 来说，这个问题不是很严重。我们看到，节点是通过 ARC 来管理的：当切片是仅存的复制时，所有在切片前方的节点都会变成无人引用的状态，这部分内存将得到回收：

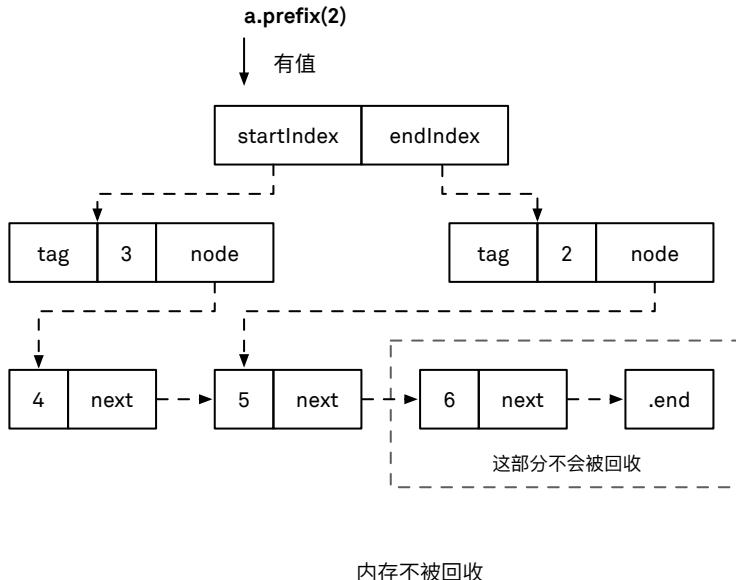


列表共享和 ARC



内存回收

不过，切片后方的节点并不会被回收。因为切片的最后一个节点仍然持有者对后方节点的引用。



切片与原集合共享索引

切片的索引基本上可以和原来集合的索引互换使用。这不是正式的要求，不过因为标准库中的所有切片类型都是这么做的，所以在你自己进行实现的时候，最好也遵循这条规则。

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，而不去手动地用 `for index in collection.indices` 进行索引计算的另一个原因。不过有一个例外：如果你在通过集合类型的 `indices` 进行迭代时，修改了集合的内容，那么 `indices` 所持有的任何对原来集合类型的强引用都会破坏写时复制的性能优化，因为这会造成不必要的复制操作。如果集合的尺寸很大的话，这会对性能造成很大的影响。(不是所有集合的 `Indices` 类型都持有对原集合的强引用，不过很多集合都是这么做的。)

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，要记住一定要从 `collection.startIndex` 开始进行循环，而不要把 0 作为开始。

通用的 PrefixIterator

现在我们知道所有的集合都能够进行切片了，我们现在来回顾一下本章早些时候实现过的前缀迭代器代码，并写出一个对任意集合都适用的版本：

```
struct PrefixIterator<Base: Collection>: IteratorProtocol, Sequence {
    let base: Base
    var offset: Base.Index

    init(_ base: Base) {
        self.base = base
        self.offset = base.startIndex
    }

    mutating func next() -> Base.SubSequence? {
        guard offset != base.endIndex else { return nil }
        base.formIndex(after: &offset)
        return base.prefix(upTo: offset)
    }
}
```

通过使迭代器直接满足 `Sequence`，我们可以直接对它使用序列的那些函数，而不用另外定义类型：

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

```
Array(PrefixIterator(numbers))
// [ArraySlice([1]), ArraySlice([1, 2]), ArraySlice([1, 2, 3])]
```

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，协议本身应该只要求和功能相关所绝对必须的内容，满足该协议的类型可以不需要提供那些非相关功能。

对于一个 Collection 来说，有两个特别有意思的限制，Collection 不能将自己的索引后退移动，另外，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。这并不是说满足 Collection 的类型没有这些能力，只是这个协议并没有提供这方面的功能。

有些操作需要作出额外的要求，虽然只有一部分集合类型可以使用它们，不过最好还是将它们的共通点提取出来，作为通用的变形来使用。标准库中就是这么做的，在标准库中，有四个专门的集合类型协议，每一个都用特定的方式在 Collection 的基础上进行了追加定义，这使得我们可以使用新的功能 (下面的引用来自于标准库文档)：

- **BidirectionalCollection** — “一个既支持前向又支持后向遍历的集合。”
- **RandomAccessCollection** — “一个支持高效随机存取索引遍历的集合。”
- **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。”

让我们一个一个来进行讨论。不过在此之前，我们需要先看看只支持前向遍历的索引是什么。

前向索引

单向链表的特性非常有名：它只支持前向索引。你不能一下子跳到链表的中间去，也不能从后往前进行遍历，你只能能够从前往后对链表进行操作。

正因如此，我们的列表集合只有 `first` 属性，而没有 `last` 属性。想要得到列表中的最后一个元素，你需要一路对链表进行列举，直到最后一个元素，这是一个 $O(n)$ 操作。所以，为获取最后一个元素提供一个简单的小属性可能会造成误解。要获取一个有百万级别的元素列表的最后一个元素可能会耗费相当长的时间，而不是像访问一个属性这样表面上看起来那么简单。

对于属性来说，一个通用的原则是“它们必须是常数时间复杂度的操作；如果不是的话，那这个操作应该显而易见地不可能在常数时间内完成”。这个定义可能有点模棱两可。只保留“必须是常数时间复杂度的操作”可能会更好，不过在标准库中，有一些例外。比如，`count` 属性的默认实现现在文档中被写为 $O(n)$ ，虽然大多数集合都像我们上面为 `List` 做的那样，提供了 $O(1)$ 的实现。

函数的话就不一样了。我们的 `List` 包含一个 `reversed` 操作：

```
let list: List = ["red", "green", "blue"]
let reversed = list.reversed() // ["blue", "green", "red"]
```

在这里，实际调用的是标准库中 `Sequence` 的扩展所提供的 `reversed` 方法，它将把逆序后的元素作为一个数组返回：

```
extension Sequence {
    /// 返回一个包含序列中元素的逆向数组。
    /// 序列必须是有限的。
    func reversed() -> [Self.Iterator.Element]
}
```

但是可能你更想要的是返回类型仍然是列表的版本。这种情况下，我们就可以通过扩展 `List` 来重载原来的默认实现：

```
extension List {
    public func reversed() -> List<Element> {
        let reversedNodes: ListNode<Element> =
            self.reduce(.end) { $0.cons($1) }
        return List(
            startIndex: ListIndex(node: reversedNodes, tag: self.count),
            endIndex: ListIndex(node: .end, tag: 0))
    }
}
```

现在，当你在列表上调用 `reversed` 时，你会得到另一个列表。这个方法会被 Swift 的重载解析机制选择为默认的实现，这个解析机制总是会挑选那些更加精确和专用的函数以及方法。在这个例子中，直接在 `List` 上实现的 `reversed` 要比定义在 `Sequence` 上的通用同名函数更加专用。

```
let list: List = ["red", "green", "blue"]
let reversed = list.reversed() // List: (blue, green, red)
```

但是也有你确实需要数组的情况，这种时候，直接使用序列的 `reversed` 方法获取一个数组，要比先获取反向列表，然后再从列表创建数组的两步操作要高效。如果你想要使用返回数组的版本，你可以将结果的类型显式声明为数组（而不是让类型推断帮助你决定类型），这样就可以强制 Swift 使用定义在序列上的版本：

```
let reversedArray: [String] = list.reversed() // ["blue", "green", "red"]
```

或者如果你要将结果传递给另一个函数的话，你还可以使用 `as` 关键字。比如，下面的代码证明了在列表上调用 `reversed` 所得到的结果和数组的结果是一样的：

```
list.reversed().elementsEqual(list.reversed() as [String]) // true
```

一个测试用的小提示：请确定在列表上的重载版本是存在的，否则上面的这行代码中两次对 `reversed` 的调用将会是相同的，且都返回数组作为结果，这样一来，这个例子就没有意义了，它们一定会是相等的。

你还可以使用 `is List` 来进行验证。如果重载的方法被使用的话，编译器会直接警告你这个 `is` 没有意义（因为使用的是重载的方法，结果一定会是 `List`）。想要避免这个警告，我们可以先将其转换为 `Any`，再转回 `List`：

```
list.reversed() as Any is List<String> // true
```

双向索引

`BidirectionalCollection` 在前向索引的基础上只增加了一个方法，但是它非常关键，那就是获取上一个索引值的 `index(before:)`。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: Iterator.Element? {
        return isEmpty ? nil : self[index(before: endIndex)]
    }
}
```

标准库中的 `String.CharacterView` 是双向索引集合的一个例子。因为 Unicode 相关的原因，一个字符集合不能含有随机存取的索引，我们会在后面字符串的章节中详述原因。不过你是可以从后往前一个一个对字符进行遍历的。

受益于集合向前索引，BidirectionalCollection 还添加了一些高效的实现，比如 suffix，removeLast 和 reversed。这里的 reversed 方法不会直接将集合反转，而是返回一个延时加载的表示方式：

```
extension BidirectionalCollection {  
    /// 返回集合中元素的逆序表示方式似乎数组  
    /// - 复杂度: O(1)  
    public func reversed() -> ReversedCollection<Self> {  
        return ReversedCollection(_base: self)  
    }  
}
```

和上面 Sequence 的 enumerated 封装一样，它不会真的去将元素做逆序操作。ReverseCollection 会持有原来的集合，并且使用逆向的索引。集合会将所有索引遍历方法逆序，这样一来就可以通过前向遍历来在原来的集合中进行后向遍历了，相反也一样。

这种做法之所以能成，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 ReversedCollection 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。我们会在结构体和类中看到这一点，在像是 Array 这样的写时复制类型(或者是像 List 这样的不变的稳定结构，或者像 FIFOQueue 这样由两个写时复制的类型组成的类型)中，这依然是高效的操作。

随机存取索引

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 $O(1)$ 时间常数的操作。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，你可能觉得这没什么大不了的。即使是像我们的 List 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。但是其中有很大区别，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，在泛型中我们会看到一些例子。在泛型章节中，我们会实现一个泛型的二分搜索算法，这个搜索算法将被限制在随机存取协议中，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。

在前面我们实现链表的时候，我们通过获取 tag 标记值写了一个自定义版本的 count 函数。因为满足随机存取索引的集合类型可以在常数时间内计算 startIndex 和 endIndex 的距离，所以这样的集合也能够在常数时间内计算集合中元素的个数。

在 Swift 2 原本的索引模型中，前向、双向和随机遍历的能力是作为索引的属性存在的，而非现在的集合属性。老版本中没有 BidirectionalCollection 和 RandomAccessCollection 这样的东西，而是 ForwardIndexType, BidirectionalIndexType 和 RandomAccessIndexType。在老版本中，Swift 通过索引类型对协议扩展进行约束，来根据索引能力的不同为集合提供不同的实现。

在新模型中，索引的遍历特性被定义在改进了的 Collection 中，这更加清晰，也和已经存在的 MutableCollection 和 RangeReplaceableCollection 的模式相匹配。

MutableCollection

可变集合支持原地的元素更改。相比于 Collection，MutableCollection 增加了一个要求，下标访问方法 subscript 现在必须提供一个 setter。我们没法让 List 可变，不过对于我们的 FIFOQueue 类型，我们可以让它遵守 MutableCollection：

```
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 {
        return 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..<endIndex).contains(position), "Index out of bounds")
    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")
swap(&playlist[0], &playlist[1])
playlist.first // Optional("Blank Space")

```

相对来说标准库中很少有满足 MutableCollection 的类型。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。后面一点解释了为什么 Dictionary 和 Set 虽然本身当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**它的元素顺序也是通过实现而唯一确定的。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 indices 指向的是对应元素所在的内部存储，而一旦元素发生变化，这个存储也会发生改变。

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新的空集合。
- `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 是展示协议扩展的强大能力的绝佳例子。你只需要实现一个超级灵活的 `replaceSubrange` 方法，协议扩展就可以为你引申出一系列有用的方法，比如：

- `append(_:_)` 和 `append(contentsOf:)` — 将 `endIndex.. (也就是说末尾的空范围) 替换为单个或多个新的元素。`
- `remove(at:)` 和 `removeSubrange(_:_)` — 将 `i...i` 或者 `subrange` 替换为空集合。
- `insert(at:)` 和 `insert(contentsOf:)` — 将 `atIndex.. (或者说在数组中某个位置的空范围) 替换为单个或多个新的元素。`
- `removeAll` — 将 `startIndex.. 替换为空集合。`

如果特定的集合类型能够依据自身为这些函数提供更高效的实现方式，它也可以提供自己的版本，这些版本在使用时将比协议扩展中的默认实现具有更高的优先级。

我们选择为我们的队列类型提供一个简单的不那么高效的实现。在我们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.Iterator.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 结构体默认就有这个方法了。

RandomAccessCollection 是对 BidirectionalCollection 的扩展，而与之不同，RangeReplaceableCollection 却并不是对 MutableCollection 的扩展，它们拥有各自独立继承关系。在标准库中，String.CharacterView 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 CharacterView 不能保证这一点。我们会在之后字符串的章节再次讨论这个话题。

现在标准库中有十二种不同类型的集合，它们是三种遍历方式(前向，双向和随机存取)以及四种变更方式(不变，可变，范围可替换，可变且范围可替换)的组合。

由于每种集合类型都需要一个专门的默认子序列类型，所以你会看到像是 MutableRangeReplaceableBidirectionalSlice 这样的类型。不要让这些看起来可怕的怪物熄灭你使用它们的勇气，它们和普通的 Slice 表现起来并无二致，只是针对它们的原集合类型加入了对应的功能，你几乎不需要考虑特殊化后的类型。如果一旦 Swift 支持按条件遵守协议的特性，这些类型就将会被移除，然后全部被归并到带有条件约束的 Slice 的扩展中去。

组合能力

这些专门的集合协议可以被很好地组合起来，作为一组约束，来匹配每个特定操作的要求。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 上，并包括 RandomAccessCollection 和 Element: Comparable 作为附加的约束：

```
extension MutableCollection
    where Self: RandomAccessCollection, Iterator.Element: Comparable {
    /// 原地对集合进行排序
    public mutating func sort() { ... }
}
```

总结

`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。像是 `MutableCollection` 或者 `RandomAccessCollection` 这样的专门的集合类型，为你在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，在未来很多的代码就会立即看起来非常熟悉，因为它是基于相同的抽象构建的，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。

下一章我们要讲的是 Swift 中的另一个基础概念：可选值。

可选值

4

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。如果到达文件末尾，getchar 将返回 -1。

又或者返回空值意味着“未找到”，就像 C++ 中的那样：

```
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() 是容器的“末尾再超一位”的迭代器。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这个值。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 null 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 被自动解开成一个 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:&e];
```

在这里，`NSString` 有可能是 `nil`，在这种情况下 — 而且只有在这种情况下 — 你应该去检查错误指针。如果得到的 `NSString` 是非 `nil` 的话，错误指针并不一定会是有效值。

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

不过这种策略是有问题的。返回的结果不管从哪个角度看都很像一个真实值。`-1` 的 `int` 值依然是一个有效的整数，但是你并不会想将它打印出来。`v.end()` 是一个迭代器，但是当你使用它的时候，结果却是未定义的。另外，所有人都会把你那陷于 `NullPointerException` 困境之中的 Java 程序当作一段笑话来看待。

哨岗值很容易产生问题，因为你可能会忘记检查哨岗值，并且不小心使用了它们。使用它们还需要预先的知识。有时候会有像是 C++ 的 `end` 迭代器这样的约定俗成的用法，有时候又没有这种约定。你通常需要查看文档才能知道需要怎么做。另外，一个函数也没有办法来表明自己 **不会失败**。也就是说，当一个函数的调用返回指针时，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。

在 Objective-C 中，对 `nil` 发送消息是安全的。如果这个消息签名返回一个对象，那么 `nil` 会被返回；如果消息返回的是一个结构体，那么它的值都将为零。不过，让我们来看看下面这个例子：

```
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。

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。

通过枚举解决魔法数的问题

当然，每个好程序员都知道使用魔法数是不对的。大多数语言都支持某种类型的枚举，它是一种用来表达某个类型的一组不相关可能值的更安全的做法。

Swift 更进一步，它的枚举中含有“关联值”的概念。也就是说，枚举可以在它们的值中包含另外的关联的值：

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

在一些语言中，这个特性被称为“标签联合”(tagged unions) (或者“可辨识联合”(discriminated union))。将若干种不同的可能类型保存在内存的同一空间中，并使用一个标签来区分到底被持有的是什么类型，这样的结构就是联合。在 Swift 枚举中，枚举的 case 就是标签。

获取关联值的唯一方法是使用 switch 或者 if case 语句。和哨岗值不同，除非你显式地检查并解包，你是不可能意外地使用到一个 Optional 中的值的。

因此，Swift 中与 find 等效的方法 index(of:) 所返回的不是一个索引值，而是一个 Optional<Index>。它是通过协议扩展实现的：

```
extension Collection where Iterator.Element: Equatable {
    func index(of element: Iterator.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 (optional) 在 Swift 中非常基础，所以有很多让它看起来更简单的语法：Optional<Index> 可以被写为 Index?；可选值遵守 ExpressibleByNilLiteral 协议，因此你可以用 nil 来替代 .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)。

现在，用户就不会错误地使用一个无效的值了：

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

如果你得到的可选值不是 .none，现在想要取出可选值中的实际的索引的话，你必须对其进行“解包”：

```
var array = ["one", "two", "three"]
switch array.index(of: "four") {
case .some(let idx):
    array.remove(at: idx)
case .none:
    break // 什么都不做
}
```

在这个 switch 语句中我们使用了完整的可选值枚举语法，在当值为 some 的时候，将其中的“关联类型”进行了解包。这种做法非常安全，但是写起来和读起来都不是很顺畅。Swift 2.0 中引入了使用 ? 作为在 switch 中对 some 进行匹配的模式后缀的语法，另外，你还可以使用 nil 字面量来匹配 none：

```
switch array.index(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

在这门语言中，可选值得到了很多内建的支持。如果你已经在使用 Swift 编写代码了的话，下面这些可能看起来会很简单，但是请务必确认你准确地理解了这些概念，因为我们在整本书中不断地使用它们。

if let

使用 if let 来进行可选绑定 (optional binding) 要比上面使用 switch 语句要稍好一些：

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

和 switch 语句一样，使用 if 的可选绑定也可以跟一个布尔限定语句搭配。比如要是匹配的目标恰好是数组中的第一个元素的话，就不移除它：

```
if let idx = array.index(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，后面的绑定值可以基于之前的成功解包的值来进行操作。这在你想要多次调用一些返回可选值的函数时会特别有用。比如 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，或者返回的页面是个错误，或者下载的数据是被损坏的，这些方法都会返回 nil。Data 的初始化方法可能会抛出错误，我们可以通过 try? 来吧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 = "http://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  
}
```

多个 let 的每一部分也能拥有一个布尔值限定的语句：

```
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)  
}
```

如果你需要在指定 if let 绑定之前执行某个检查的话，可以为 if 提供一个前置的条件。假设你正在使用 storyboard，并且想要在将 view controller 转换为某个指定类型之前检查一下 segue 的 identifier 的话：

```
if segue.identifier == "showUserDetailsSegue",  
    let userDetailVC = segue.destination  
        as? UserDetailViewController  
{  
    userDetailVC.screenName = "Hello"  
}
```

你可以在同一个 if 中将可选值绑定，布尔语句和 case let 混合并匹配起来使用。

if let 在 Foundation 框架的 Scanner 类型中也有用处，Scanner 将返回一个代表是否扫描到某个值的布尔值，在之后，你可以解包得到的结果：

```
let scanner = Scanner(string: "myUserName123")  
var username: NSString?  
let alphas = CharacterSet.alphanumerics  
  
if scanner.scanCharacters(from: alphas, into: &username),  
    let name = username {  
    print(name)  
}
```

while let

`while let` 语句和 `if let` 非常相似，它代表一个当遇到 `nil` 时终止的循环。

标准库中的 `readLine` 函数从标准输入中读取一个可选字符串。当到达输入末尾时，这个方法将返回 `nil`。所以，你可以使用 `while let` 来实现一个非常基础的和 Unix 中 `cat` 命令等价的函数。

```
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)
}
```

我们在集合协议一章中提到，`for x in sequence` 这样的语句需要 `sequence` 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，当序列中值耗尽的时候，`nil` 将被返回。`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
```

所以，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了，这里我们需要添加一个 `where` 关键字：

```
for i in 0..10 where i % 2 == 0 {
    print(i)
}
```

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 iterator = (0..<10).makeiterator()
while let i = iterator.next() {
    if i % 2 == 0 {
        print(i)
    }
}
```

for 循环的这个特性带来了一个很有趣的行为，它避免了由于变量捕获而造成的一个非常奇怪的 bug，而这个 bug 可能发生在其他一些语言中。看看下面的 Ruby 代码：

```
a = []
for i in 1..3
    a.push(lambda {i})
end

for f in a
    print "#{f.call()}"
end
```

Ruby 的 lambda 和 Swift 的闭包表达式类似，而且和 Swift 中一样，它们也会捕获局部变量。上面的代码在 1 到 3 中进行循环，然后将一个捕获了 i 的闭包添加到数组中。当闭包被调用时，它会将 i 的值打印出来。接下来我们遍历了这个含有闭包的数组，并调用每个闭包。你可以猜猜看会打印出什么，如果你有一台 Mac 的话，可以试着将上面的代码复制到一个文件中去，并在命令行用 ruby 执行它。

运行的结果是打印出三个 3。虽然 i 在每次闭包创建时的值都不同，但是这些闭包捕获的是同一个 i 变量。所以当你调用闭包时，i 的值会是循环结束时的值，也就是 3。

现在来看看类似的 Swift 版本：

```
var a: [() -> Int] = []
for i in 1...3 {
    a.append {i}
}
```

```
for f in a {
    print("\f0)", terminator: " ")
}
// 1 2 3
```

输出将是 1, 2 和 3。一旦你理解 for...in 其实是 while let 后，这一切就能解释得通了。为了让事情更加清楚，想像一下要是我们没有 while let 的话，我们使用迭代器来实现时可能是这样的：

```
var g = (1...3).makelteator()
var o: Int? = g.next()
while o != nil {
    let i = o!
    a.append { i }
    o = g.next()
}
```

很容易看出来，每次迭代时 i 都是一个新的局部变量，所以闭包所捕获的是正确的值，每次之后的迭代中定义的 i 都会是不同的局部变量。

作为对比，Ruby 和 Python 的代码看起来更像是下面这样的：

```
var a: [() -> Int] = []

do {
    var g = (1...3).makelteator()
    var i: Int
    var o: Int? = g.next()
    while o != nil {
        i = o!
        a.append { i }
        o = g.next()
    }
}
```

这里，i 是声明在循环外部的，它将被重用，所以每个闭包捕获的都是同样的 i。如果你对它们进行调用，它们都返回 3。这里使用了 do 的原因是，虽然 i 是声明在循环外部的，但是它还是应该有正确的作用域，使用 do 可以让这个变量在整个循环外部不被访问到，i 会被夹在内部循环和循环外部之间。

C# 在 C# 5 之前的行为和 Ruby 是一样的，之后，C# 的维护者意识到了这个行为十分危险，因此他们不惜在新版本中引入破坏性的变化来修复该问题，现在这样的 C# 代码与 Swift 的工作方式相同。

双重可选值

需要指出，一个可选值本身也可以被使用另一个可选值包装起来，这会导致可选值嵌套在可选值中。这其实不是一个奇怪的边界现象，编译器也不应该自动去将这种情况进行合并处理。假设你有一个字符串数组，其中的字符串是数字，你现在想将它们转换为整数。最直观的方式是用一个 map 来进行转换：

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

你现在得到了一个元素类型为 `Optional<Int>` 的数组，这是因为 `Int.init(String)` 是可能失败的，当字符串不包括一个有效的整数时，转换会得到 `nil`。我们的例子中，最后一个元素就将是 `nil`，因为字符串 `"three"` 并不代表一个整数。

当使用 `for` 循环遍历这个结果数组时，显然每个元素都会是可选整数值，因为 `maybeInts` 含有的就是这样的值：

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

前面我们已经知道 `for...in` 是 `while` 循环加上一个迭代器的简写方式，`next` 函数返回的其实是一个 `Optional<Optional<Int>>` 值，或者说是一个 `Int??`。`next` 将把序列中的每个元素用可选值的方式包装起来，然后 `while let` 会检查这个值是不是 `nil`，如果不是，则将其解包并运行循环体部分：

```
var iterator = maybeInts.makeIterator()
while let maybeInt = iterator.next() {
    print(maybeInt)
}
/*
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。如果没有双重可选值，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 if 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 {  
}
```

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("\(j) 在范围内")  
} // 5 在范围内
```

因为 case 匹配可以通过实现 ~= 运算符来进行扩展，所以你可以将 if case 和 for case 进行一些有趣的扩展：

```
struct Substring {
    let s: String
    init(_ s: String) { self.s = s }
}

func ~=(pattern: Substring, value: String) -> Bool {
    return value.range(of: pattern.s) != nil
}

let s = "Taylor Swift"
if case Substring("Swift") = s {
    print("拥有 \"Swift\" 后缀")
}
// 拥有 "Swift" 后缀
```

这为我们提供了无限可能，不过使用起来你需要小心一些，因为写 `~=` 操作符有可能意外地匹配到比你预想的更多的内容。将下面的代码插入到一个比较通用的库里也许会有很“好”的恶作剧效果：

```
func ~=<T, U>(_: T, _: U) -> Bool { return true }
```

除非还定义了更精确的版本，否则这段代码将会使所有情形都匹配。千万别这么做，要是被你捉弄的人找到了这段代码的话，他一定会来找你的麻烦的，我可不保证你会不会被拖出去打死...

if var 和 while var

除了 `let` 以外，你还可以使用 `var` 来搭配 `if`、`while` 和 `for`：

```
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提取出来。所以使用 `if var` 这个变形和在函数参数上使用 `var` 类似，它只是获取一个能在作用域内使用的副本的简写，而并不会改变原来的值。

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有点不爽，但是这其实和其他一些做法并无不同。

举个例子，数组有个 `first` 方法，它将会返回数组的首个元素的可选值，如果数组为空的话，返回 `nil`。这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，你依然可以使用 a[0]，但无法保证它的有效性
```

如果你使用 `first` 的话，你就必须先对其解包才能使用它的值。这十分安全，你不会由于粗心而忘掉这件事情：

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

不过如果你从函数中提早退出的话，情况就完全不同了。有时候你可能会这么写：

```
func doStuff(withArray a: [Int]) {
    if a.isEmpty { return }
    // 现在可以安全地使用 a[0]
}
```

提早退出有助于避免恼人的 if 嵌套，你也不再需要在函数后面的部分再次重复地进行判断。

我们可以使用 Swift 延迟初始化的能力来改善在作用域外使用解包后的可选值的代码。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 = characters.index(of: ".") {
            period = idx
```

```
    } else {
        return nil
    }
    let extensionRange = characters.index(after: period)..<characters.endIndex
    return self[extensionRange]
}
}

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

Swift 会检查你的代码，并且发现这段代码有两条可能的路径：一条是没有找到 “.” 时的提早返回，另一条是 period 被正确初始化。因为 period 不是一个可选值，所以它不可能为 nil；period 也不可能未被初始化，因为 Swift 不会让你使用未被初始化的变量。所以在 if 语句后，你的代码可以随意使用 period，而不需要考虑它是可选值的可能性。

但是这段代码看起来很丑。我们在这里真正需要的其实是一个 if not let 语句，其实这正是 guard let 所做的事情。

```
extension String {
    var fileExtension: String? {
        guard let period = characters.index(of: ".") else { return nil }
        let extensionRange = index(after: period)..<characters.endIndex
        return self[extensionRange]
    }
}

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

和 if 或者 else 语句的块一样，在 guard 的 else 中你可以做任何事，包括执行多句代码。唯一的限制是你必须在 else 中离开当前的作用域，也就是说，在代码块的最后你必须写 return 或者调用 fatalError (或者其他被声明为返回 Never 的方法)。如果你是在循环中使用 guard 的话，那么最后应该是 break 或者 continue。

一个函数的返回值如果是 Never 的话，就意味着告诉了编译器这个函数不会返回。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进行控制流诊断。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。它的唯一作用就是给编译器提供信号。一个返回值被声明为无人类型的函数将永远不能正常返回。在 Swift 中，无人类型是由一个不包含任意成员的 enum 来实现的。

一般来说你不会需要自己定义一个返回 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，当你在写新代码时，可能会遇到一个很复杂的 switch 语句，你正在慢慢填上所有的 case，这时候编译器会用空的 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先处理一个 case 语句的逻辑。这时候，你可以放几个 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.")  
}
```

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

和可选绑定的情况不同，单单使用 guard 并没有太多好处。实际上它还要比原来的 if return 的写法稍微啰嗦一些。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()
```

和 Objective-C 不同的是，Swift 编译器会在你的值是可选值的时候警告你。如果你的可选值值中确实有值，那么编译器能够保证方法肯定会被实际调用。如果没有值的话，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!.uppercased()  
} else {  
    // 这里没有合理的处理方法  
    fatalError("no idea what to do now...")  
}
```

如果 str 是非 nil，那么 upper 会有我们想要的值。但是如果 str 是 nil 的话，那么 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 result 只能是可选值，因为它需要考虑 str 可能为 nil 的情况：

```
let result = 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")
```

然而，这看起来有点出乎意料。我们不是刚才才说过可选链调用的结果是一个可选值么？为什么在第一个 `uppercased()` 后面不需要加上问号？这是因为可选链是一个“展平”操作。

`str?.uppercased()` 返回了一个可选值，如果你再对它调用 `?.lowercased()` 的话，逻辑上来说你将得到一个可选值的可选值。不过其实你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包含了。

不过，如果 `uppercased` 方法本身也返回一个可选值的话，你就需要在它后面加上 `?` 来表示你正在链接这个可选值。比如，让我们想像一下为 `Int` 类型添加一个 `half` 方法，这个方法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 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)
```

```
Optional(2)
```

可选链对下标和函数调用也同样适用，比如：

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

另外，还有这样的情况：

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

你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，你可以这样来更新它的属性：

```
let splitViewController: UISplitViewController? = nil  
let myDelegate: UISplitViewControllerDelegate? = nil  
if let viewController = splitViewController {  
    viewController.delegate = myDelegate  
}
```

你也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：

```
splitViewController?.delegate = myDelegate
```

nil 合并运算符

很多时候，你会想要解包一个可选值，如果可选值是 nil 时，就用一个默认值来替代它。你可以使用 nil 合并运算符来完成这件事：

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

这里，如果字符串代表一个整数的话，number 将会是那个解包后的整数值。如果字符串不能转换为整数，Int.init 将返回 nil，它将会被默认值 0 替换掉并赋值给 number。也就是说 lhs ?? rhs 做的事情类似于这样的代码 lhs != nil ? lhs! : rhs。

“真是了不起！”Objective-C 程序员可能会有点反讽地说道，“不过我们早就有 ?: 操作符了”。确实，?? 和 Objective-C 中的 ?: 十分相似，但是它们还是有不同的地方。我们需要强调在 Swift 中思考可选值时很重要的一点，那就是可选值**不是**指针。

确实，在大部分时间里你遇到的可选值都是在和 Objective-C 的库打交道时的引用值。但是正如我们所见，可选值也可以封装值类型。所以在上面的例子中 number 只是一个 Int，而不是 NSNumber。

通过使用可选值，除了确保不会取到 null 指针以外，你还能用它来做很多事情。举个例子，比如你想要获取数组中的第一个值，但是当数组为空的时候，你想要提供一个默认值：

```
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 ?? 0 // 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，先进行检查，然后取值，再然后指定默认值的三值逻辑版本。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，你不可能会忘记检查 first 的可选值特性并且不小心使用它。

当你发现你在检查某个语句来确保取值满足条件的时候，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(safe idx: Int) -> Element? {  
        return idx < endIndex ? self[idx] : nil  
    }  
}
```

现在你就可以这样写：

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

合并操作也能够进行链接 — 如果你有多个可能的可选值，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 // Optional(42)
```

有时候你可能会有多个可选值，并且想要按照顺序选取其中非 `nil` 的值。但是当它们全都是 `nil` 时，你又没有一个合理的默认值。这种情况下，你依然可以使用 `??` 操作符。不过如果最终的值是可选值的话，你得到的整个结果也将是可选值：

```
let m = i ?? j ?? k  
type(of: m) // Optional<Int>
```

这种方式经常和 `if let` 配合起来使用，你可以将它想像成 “or” 版本的 `if let`：

```
if let n = i ?? j {  
    // 和 if i != nil || j != nil 类似  
    print(n)  
}
```

如果你将 if let 的 ?? 操作符看作是和 “or” 语句类似的话，多个 if let 语句并列则等价于 “and”：

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

因为可选值是链接的，如果你要处理的是双重嵌套的可选值，并且想要使用 ?? 操作符的话，你需要特别小心 a ?? b ?? c 和 (a ?? b) ?? c 的区别。前者是合并操作的链接，而后者是先解包括号内的内容，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
```

可选值 map

我们现在有一个字符数组，我们想要将第一个字符转换为字符串：

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

不过，如果 characters 可能为空的话，我们在就需要用 if let，只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，当字符数组至少含有一个元素时，firstCharAsString 将会是一个含有该元素的 String。如果字符数组为空的话，firstCharAsString 将会为 nil。

这种获取一个可选值，并且在当它不是 nil 的时候进行转换的模式十分常见。Swift 中的可选值里专门有一个方法来处理这种情况，它叫做 map。这个方法接受一个闭包，如果可选值有内容，则调用这个闭包对其进行转换。上面的函数用 map 可以重写成：

```
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 要么在零值的情况下不做处理，要么在有值的时候会对其进行转换。

鉴于可选值 map 与集合的 map 的相似性，可选值 map 的实现和集合 map 也很类似：

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

当你想要的就是一个可选值结果时，可选值 map 就非常有用。设想你想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 nil，所以这个结果应当是一个可选值。你可能会这样来实现它：

```
extension Array {  
    func reduce(_ nextPartialResult: (Element, Element) -> Element) -> Element? {  
        // first will be nil if the array is empty  
        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，可选值的 map 也会返回 nil，所以我们可以使用不包含 guard 的单 return 形式来重写 reduce：

```
extension Array {  
    func reduce_alt(_ nextPartialResult: (Element, Element) -> Element)  
        -> Element?  
    {  
        return 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结果的话，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 (first 可能会是 nil)，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：

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

这么做得到的结果 y 将是 Int? 类型。

你可能会使用 if let 来写，因为值被绑定了，我们可以使用它来计算后续的值：

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

这说明 flatMap 和 if let 非常相似。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。我们可以使用 map 和 flatMap 来重写它们：

```
let urlString = "http://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进行实现：

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

使用 flatMap 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，你可以忽略掉那些 nil 值。

设想你需要处理一个字符串数组中的数字。在有可选值模式匹配时，用 for 循环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 flatMap 正是你想要的：

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

我们之前已经看过两个 flatMap 了：一个作用在数组上展平一个序列，另一个作用在可选值上展平可选值。这里的 flatMap 是两者的混合：它将把一个映射为可选值的序列进行展平。

回到我们将可选值类比于包含零个或者一个值的集合这一假设上，一切就能说得通了。如果这个集合就是一个数组，那 flatMap 恰好做了我们想要的事情。

我们来实现一下我们自己版本的这个操作，让我们先来定义一个可以将 nil 过滤掉并且返回非可选值数组的 flatten 函数：

```
func flatten<S: Sequence, T>  
(source: S) -> [T] where S.Iterator.Element == T? {  
    let filtered = source.lazy.filter { $0 != nil }  
    return filtered.map { $0! }  
}
```

唉？一个全局函数？为什么不用协议扩展？很不幸，现在没有办法写一个只作用于可选值序列的 Sequence 的扩展。因为在这里你需要两个占位符语句（一个用于限定 S，另一个用于 T，就像这里我们做的一样），但是协议扩展现在还不支持这么做。

不过，这个方法确实可以让 flatMap 更容易写：

```
extension Sequence {  
    func flatMap<U>(transform: (Iterator.Element) -> U?) -> [U] {  
        return flatten(source: self.lazy.map(transform))  
    }
```

```
}
```

在这两个函数里，我们都使用了 `lazy` 来将数组的实际创建推迟到了使用前的最后一刻。这是一个小的优化，不过如果在处理很大的数组时，这么做可以避免不必要的中间结果的缓冲区内存申请，还是值得的。

可选值判等

通常，在判等时你不需要关心一个值是不是 `nil`，你只需要检查它是否包含某个（非 `nil` 的）特定值即可：

```
let regex = "^Hello$"  
// ...  
if regex.characters.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 `if` 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.characters[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，`==` 有一个接受两个可选值的版本，它的实现类似这样：

```
func ==<T: Equatable>(lhs: T?, rhs: T?) -> Bool {  
    switch (lhs, rhs) {  
        case (nil, nil): return true  
        case let (x?, y?): return x == y  
        case (_, nil), (nil, _?): return false  
    }  
}
```

这个重载只对那些可以判等的可选值类型有效。在这个保证下，输入的左右两个值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。

但是这只是故事的一半。注意一下，我们**并不一定要写**这样的代码：

```
// 我们其实并不需要将 "^" 声明为可选值
if regex.characters.first == Optional("^") {
    // 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，如果需要匹配可选值类型，Swift 总是会将它“升级”为一个可选值然后使用。

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，我们需要 == 可以同时作用于可选值和非可选值。这样一来，我们可能需要三个版本的函数：

```
// 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，我们在整本书中都在依赖这个隐式的转换。比方说，当我们在实现可选 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 的返回值其实是个可选值。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，因为可能有键不存在的情况，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，如果你使用下标为一个字典的某个键赋值 nil 的时候，实际上做的是将这个键从字典中移除。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 // ["none": nil, "one": Optional(1)]
```

它将会把 "two" 这个键移除。

我们可以使用下面中的任意一个来改变这个键的值，你可以选择一个你觉得清晰的方式，它们都可以正常工作：

```
dictWithNils["two"] = Optional(nil)  
dictWithNils["two"] = .some(nil)  
dictWithNils["two"]? = nil  
dictWithNils // ["none": nil, "one": Optional(1), "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，当把 "three" 设置 nil 时，并没有值被更新或者插入。

Equatable 和 ==

尽管可选值有 == 操作符，但是这并不是说它们实现了 Equatable 协议。如果你尝试做下面这样的事情的话，这个细微但是重要区别将会啪啪打你脸：

```
// 两个可选值整数的数组  
let a: [Int?] = [1, 2, nil]  
let b: [Int?] = [1, 2, nil]  
  
// 错误：binary operator '==' cannot be applied to two [Int?] operands
```

```
a == b
```

问题在于数组的 `==` 操作符需要数组中的元素是遵守 Equatable 协议的：

```
func ==<Element : Equatable>(lhs: [Element], rhs: [Element]) -> Bool
```

可选值没有遵守 Equatable 协议，想要遵守它的话，需要可选值所能包含的所有类型都实现 `==` 操作符，而事实上只有那些本身可以判等的类型才满足这个要求。也许在未来 Swift 会支持带有条件的协议实现，如果有这个特性的话，代码将会是类似这样的：

```
extension Optional: Equatable where Wrapped: Equatable {  
    // 没有必要写任何东西，因为 == 已经被实现了  
}
```

不过现在的话，你可以为可选值的数组实现一个 `==`：

```
func ==<T: Equatable>(lhs: [T?], rhs: [T?]) -> Bool {  
    return lhs.elementsEqual(rhs) { $0 == $1 }  
}
```

switch-case 匹配可选值

可选值没有实现 Equatable 的另一个后果是，你将不能用 case 语句对它们进行检查。在 Swift 里 case 匹配是通过 `~=` 运算符来进行的，它的相关定义和我们上面看到的可选值数组无法判等的例子中的定义有点儿类似：

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

想要为可选值创建一个匹配版本非常简单，只需要调用 `==` 就行了：

```
func ~=<T: Equatable>(pattern: T?, value: T?) -> Bool {  
    return pattern == value  
}
```

同时，我们可以实现对范围进行匹配的方法：

```
func ~=<Bound>(pattern: Range<Bound>, value: Bound?) -> Bool {  
    return value.map { pattern.contains($0) } ?? false
```

```
}
```

这里，我们使用了 map 来判断一个非 nil 值是不是落在范围中。因为我们不希望 nil 匹配到任何范围，所以在 nil 的情况下我们直接返回 false。

有了这个，我们现在就可以通过 switch 来匹配可选值了：

```
for i in ["2", "foo", "42", "100"] {
    switch Int(i) {
        case 42:
            print("The meaning of life")
        case 0..<10:
            print("A single digit")
        case nil:
            print("Not a number")
        default:
            print("A mystery number")
    }
}
/*
A single digit
Not a number
The meaning of life
A mystery number
*/
```

可选值比较

和 == 类似，对于可选值曾经也有像是 <、>、<= 和 >= 这些操作符。在 [SE-0121](#) 中这些操作符被移除了，因为它们可能会导致意外的结果。

比如说，`nil <.some(_)` 以前将返回 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 是小于 0 的，所以它将被包含在 belowFreezing 温度中，这显然是不合情理的。

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”，“当可以让代码更清晰时使用”，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不巧意外地是 nil 的话，这句程序直接挂掉。

举个例子，看看之前提到过的 flatten 的实现：

```
func flatten<S: Sequence, T>
  (source: S) -> [T] where S.Iterator.Element == T? {
  let filtered = source.lazy.filter { $0 != nil }
  return filtered.map { $0! }
}
```

这里，因为在 filter 的时候已经把所有 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 值的情况。我们当然可以把强制解包运算符从这个函数中消除掉，并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。

不过使用强制解包还是很罕见的。如果你完全掌握了本章中提到的解包的知识，你一般应该可以找到更好的方法。每当你发现你需要使用 ! 时，可以回头看看是不是真的别无他法了。比如，我们可以使用 source.flatMap { \$0 } 这个方法来实现 flatten，这样就不需要强制解包了。

第二个例子，下面这段代码会根据特定的条件来从字典中找到值满足这个条件的对应的所有的键：

```
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"]
```

这里使用 ! 非常安全 — 因为所有的键都是来源于字典的，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 filter 处理剔除掉不满足条件的值，然后再对姓名进行映射和排序。

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

这样的写法还额外带来了一些性能上的收益，它避免了不必要的用键进行的查找。

还有些时候，造化弄人，虽然你**确实**知道一个值不可能是 nil，但别人就是以可选值的方式把它传递给你了。在这种情况下，选择让程序挂掉会比让它继续运行**要好**，因为继续的话很可能会为你的逻辑引入非常严重的 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打扫这种理论上不可能存在的情况的方法，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。现在，当程序发生错误时，你从输出中无法知道发生问题的原因是什么。

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)\""
```

@autoclosure 注解确保了我们只在需要的时候会执行操作符右侧的语句。我们在[函数](#)一章中会详细说明它的用法。

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 !?
(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，但是在发布版本中打印 0：

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

对其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 !?
(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多元组为参数的版本，多元组中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")
```

对于返回 Void 的函数，使用可选链进行调用时将返回 Void?。利用这一点，你可以写一个非泛型的版本来检测一个可选链调用碰到 nil，且并没有进行完操作的情况：

```
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，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 比较类型，但是在发布版本中它不会被移除，也就是说，只要条件被判定为 false，执行就会被停止。

多灾多难的隐式可选值

隐式可选值是那些不论何时你使用它们的时候就自动强制解包的可选值。别搞错了，它们依然是可选值，现在你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，那你到底为什么会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。

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。但是几乎没有 Objective-C 的 API 真的会返回一个空引用，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，所以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 Objective-C 桥接代码中看到它们的身影。但是在纯的原生 Swift API 中，你**不应该**使用隐式可选值，也不应该将它们在回调中进行传递。

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。

比如你有一个分两步的初始化操作，当你的类最后准备好被使用时，这个隐式可选值将有一个实际值。这就是 Xcode 和 Interface Builder 使用它们的方式。

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 和 map，所有的操作都是一样的：

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

不过虽然隐式可选值会尽可能地隐藏它们的可选值特性，它们在行为上也还是有些不一样的地方。比如，你不能将一个隐式解包的值通过 inout 的方法传递给一个函数：

```
func increment(inout x: Int) {  
    x += 1  
}  
  
var i = 1 // 普通的 Int  
increment(&i) // 将 i 增加为 2  
  
var j: Int! = 1 // 隐式解包的 Int  
increment(&j) // 错误  
// cannot invoke 'increment' with an argument list of type '(inout Int!)'
```

总结

在处理有可能是 `nil` 的值的时候，可选值会非常有用。相比于使用像是 `NSNotFound` 这样的魔法数，我们可以用 `nil` 来代表一个值为空。Swift 中有很多内置的特性可以处理可选值，所以你能够避免进行强制解包。隐式解包可选值在与遗留代码协同工作时会有用，但是在有可能的情况下还是应该尽可能使用普通的可选值。最后，如果你需要比单个可选值更多的信息（比如，在结果不存在时你可能需要一个错误信息提示），你可以使用抛出错误的方法，我们会在 [错误处理](#)一章中介绍相关内容。

结构体和类

5

在 Swift 中，要存储结构化的数据，我们有多种不同的选择：结构体、枚举、类以及使用闭包捕获变量。在 Swift 标准库中，绝大多数的公开类型都是结构体，而枚举和类只占很小一部分。这可能是标准库中那些类型的特性使然，但是不管从什么方面这个事实都提醒我们 Swift 中结构体有多么重要。在 Swift 3 中，许多 Foundation 框架中的类现在有专门针对 Swift 构建的对应结构体类型了。在本章中，我们主要来看看结构体和类有哪些区别。我们可能不会花太多精力在枚举类型上，因为它的行为和结构体十分相似。

这里是结构体和类的主要不同点：

- 结构体 (和枚举) 是**值类型**，而类是**引用类型**。在设计结构体时，我们可以要求编译器保证不可变性。而对于类来说，我们就得自己来确保这件事情。
- 内存的管理方式有所不同。结构体可以被直接持有及访问，但是类的实例只能通过引用间接地访问。结构体不会被引用，但是会被复制。也就是说，结构体的持有者是唯一的，但是类却能有很多个持有者。
- 使用类，我们可以通过继承来共享代码。而结构体 (以及枚举) 是不能被继承的。想要在不同的结构体或者枚举之间共享代码，我们需要使用不同的技术，比如像是组合、泛型以及协议扩展等。

在本章中，我们将会探索这些不同之处的细节。我们会从实体和值的区别谈起，然后继续讨论可变性所带来的问题。之后，我们会向你展示如何将一个引用类型封装到结构体里，这样我们就能够把它当作一个值类型来使用。最后，我们会详细谈谈两者之间内存管理上的差异，特别是引用类型与闭包一起使用时内存的管理方式的问题，以及如何避免引用循环。

值类型

我们经常会处理一些需要有明确的**生命周期**的对象，我们会去初始化这样的对象，改变它，最后摧毁它。举个例子，一个文件句柄 (file handle) 就有着清晰的生命周期：我们会打开它，对其进行一些操作，然后在使用结束后我们需要把它关闭。如果我们想要打开两个拥有不同属性的文件句柄，我们就需要保证它们是独立的。想要比较两个文件句柄，我们可以检查它们是否指向同样的内存地址。因为我们对地址进行比较，所以文件句柄最好是由引用类型来进行实现。这也正是 Foundation 框架中 `FileHandle` 类所做的事情。

其他一些类型并不需要生命周期。比如一个 URL 在创建后就不会再被更改。更重要的是，它在被摧毁时并不需要进行额外的操作 (对比文件句柄，在摧毁时你需要将其关闭)。当我们比较两个 URL 变量时，我们并不关心它们是否指向内存中的同一地址，我们所比较的是它们是否指向同样的 URL。因为我们通过它们的属性来比较 URL，我们将其称为值。在 Objective-C 里，我们用 `NSURL` 来实现一个不可变的对象。不过在 Swift 中对应的 URL 却是一个结构体。

软件中拥有生命周期的对象非常多 — 比如文件句柄，通知中心，网络接口，数据库连接，view controller 都是很好的例子。对于这些类型，我们想在初始化和销毁的时候进行特定的操作。在对它们进行比较的时候，我们也不是去比较它们的属性，而是检查两者的内存地址是否一样。所有这些类型的实现都使用了对象，它们全都是引用类型。

在大多数软件里值类型也扮演着重要的角色。URL，二进制数据，日期，错误，字符串，通知以及数字等，这些类型只通过它们的属性来定义。当对它们进行比较的时候，我们不关心内存地址。所有这些类型都可以使用结构体来实现。

值永远不会改变，它们具有不可变的特性。这 (在绝大多数情况下) 是一件好事，因为使用不变的数据可以让代码更容易被理解。不可变性也让代码天然地具有线程安全的特性，因为不能改变的东西是可以在线程之间安全地共享的。

Swift 中，结构体是用来构建值类型的。结构体不能通过引用来进行比较，你只能通过它们的属性来比较两个结构体。虽然我们可以用 var 来在结构体中声明可变的变量属性，但是这个可变性只体现在变量本身上，而不是指里面的值。改变一个结构体变量的属性，在概念上来说，和为整个变量赋值一个全新的结构体是等价的。我们总是使用一个新的结构体，并设置被改变的属性值，然后用它替代原来的结构体。

结构体只有一个持有者。比如，当我们将结构体变量传递给一个函数时，函数将接收到结构体的复制，它也只能改变它自己的这份复制。这叫做值语义 (value semantics)，有时候也被叫做复制语义。而对于对象来说，它们是通过传递引用来工作的，因此类对象会拥有很多持有者，这被叫做引用语义 (reference semantics)。

因为结构体只有一个持有者，所以它不可能造成引用循环。而对于类和函数这样的引用类型，我们需要特别小心，避免造成引用循环的问题。我们会在内存部分讨论这个问题。

值总是需要复制这件事情听来可能有点低效，不过，编译器可以帮助我们进行优化，以避免很多不必要的复制操作。因为结构体非常基础和简单，所以这是可能的。结构体复制的时候发生的是按照字节进行的浅复制。除非结构体中含有类，否则复制时都不需要考虑其中属性的引用计数。当使用 let 来声明结构体时，编译器可以确定之后这个结构体的任何一个字节都不会被改变。另外，和 C++ 中类似的值类型不同，开发者没有办法知道和干预何时会发生结构体的复制。这些简化给了编译器更多的可能性，来排除那些不必要的复制，或者使用传递引用而非值的方式来优化一个常量结构体。

编译器所做的对于值类型的复制优化和值语义类型的写时复制行为并不是一回事儿。写时复制必须由开发者来实现，想要实现写时复制，你需要检测所包含的类是否有共享的引用。

和自动移除不必要的值类型复制不同，写时复制是需要自己实现的。不过编译器会移除那些不必要的“无效”浅复制，以及像是数组这样的类型中的代码会执行“智能的”写时复制，两者互为补充，都是对值类型的优化。我们接下来很快就会看到如何实现你自己的写时复制机制的例子。

如果你的结构体只由其他结构体组成，那编译器可以确保不可变性。同样地，当使用结构体时，编译器也可以生成非常快的代码。举个例子，对一个只含有结构体的数组进行操作的效率，通常要比对一个含有对象的数组进行操作的效率高得多。这是因为结构体通常要更直接：值是直接存储在数组的内存中的。而对象的数组中包含的只是对象的引用。最后，在很多情况下，编译器可以将结构体放到栈上，而不用放在堆里。

当和 Cocoa 以及 Objective-C 交互时，我们可能通常都需要类。比如在实现一个 table view 的代理时，我们除了使用类以外别无它选。Apple 的很多框架都重度依赖于子类，不过在某些问题领域，我们仍然能创建一个对象为值的类。举个例子，在 Core Image 框架里，`CILImage` 对象是不可变的：它们代表了一个永不变化的图像。

有些时候，决定你的新类型应该是结构体还是类不容易。两者表现得不太一样，知晓其中的区别将有助于作出决定。在本章之后的例子中，我们会更清楚地看到这种行为所带来的影响，然后提供一些关于何时应该使用结构体的指导建议。

可变性

在最近几年，可变性的名声一直不好，这是因为很多 bug 的主要诱因就是可变性。大部分专家都会推荐你尽可能地使用不可变的对象，以便写出安全可维护的代码。幸运的是，Swift 可以让我们在写出安全代码的同时，保留直观的可变代码的风格。

我们从展示一些可变性带来的问题入手。在 Foundation 中，有两个数组类型，一个是 `NSArray`，另一个是它的子类 `NSMutableArray`。我们可以用 `NSMutableArray` 写出下面这样（会崩溃）的程序：

```
let mutableArray: NSMutableArray = [1,2,3]
for _ in mutableArray {
    mutableArray.removeLastObject()
}
```

当迭代一个 `NSMutableArray`，你不能去改变它，因为迭代器是基于原始的数组工作的，改变数组将会破坏迭代器的内部状态。这个限制是很合理的，一旦你知道了这一点，你应该就不会再犯这个错误。不过考虑下要是在 `mutableArray.removeLastObject()` 的位置上是一个另外的

方法调用，而这个调用在其中改变了 `mutableArray`。这时如果你不是特别清楚这个方法内部做了什么，对于数组的破坏就很难被发现了。

现在我们来看一个相同的例子，不过这次我们使用的是 Swift 数组：

```
var mutableArray = [1, 2, 3]
for _ in mutableArray {
    mutableArray.removeLast()
}
```

这个例子不会崩溃，这是因为迭代器持有了数组的一个本地的，独立的复制。要更清晰地观察这一点，可以打开一个 Playground，然后将 `removeLast` 换成 `removeAll`，你在 Playground 中可以看到这条语句被调用了三次。这是因为不论如何移除，数组的迭代器的复制依然持有最开始的三个元素。

类是引用类型。如果你创建一个类的实例，并且将它赋值给新的变量，新旧两个变量都将指向同一个对象：

```
let mutableArray: NSMutableArray = [1, 2, 3]
let otherArray = mutableArray
mutableArray.add(4)
otherArray // (1, 2, 3, 4)
```

因为两个变量都指向了同样地对象，现在它们两都引用着数组 `[1, 2, 3, 4]`，改变其中一个变量中的值，也会使另一个变量的值发生改变。这是一个非常强大的特性，但是也是一个非常容易产生 bug 的来源。调用一个方法有可能会让意料之外的东西发生改变，不变性不再有效。

在类中，我们可以使用 `var` 和 `let` 来控制属性的可变和不可变性。比如，我们可以创建一个和 Foundation 中 `Scanner` 类似的扫描器，不过区别是它将读取二进制数据。在 `Scanner` 类中，你可以从一个字符串中扫描值，每次成功获得一个扫描值后就进行步进。类似地，我们的 `BinaryScanner` 类将持有一个位置属性（它是可变的），以及原始的数据（它是不可变的）。想要模仿 `Scanner` 的行为，这两个属性是我们所需要存储的全部内容了：

```
class BinaryScanner {
    var position: Int
    let data: Data
    init(data: Data) {
        self.position = 0
        self.data = data
    }
}
```

```
    }
}
```

我们还可以添加一个方法，来扫描一个字节。注意，这个方法是可变的：它会改变 `position` 的值，除非我们到达数据的末尾：

```
extension BinaryScanner {
    func scanByte() -> UInt8? {
        guard position < data.endIndex else {
            return nil
        }
        position += 1
        return data[position-1]
    }
}
```

要测试这个 `Scanner`，我们可以写一个方法来扫描所有剩余数据：

```
func scanRemainingBytes(scanner: BinaryScanner) {
    while let byte = scanner.scanByte() {
        print(byte)
    }
}

let scanner = BinaryScanner(data: "hi".data(using: .utf8)!)
scanRemainingBytes(scanner: scanner)

/*
104
105
*/
```

所有部分都按预想工作了。但是，我们很容易构造出一个含有竞态条件的例子。如果我们使用 GCD (Grand Central Dispatch) 来从两个不同的线程调用 `scanRemainingBytes` 的话，就有可能进入竞态条件。在下面的代码中，`position < data.endIndex` 这个条件可能会在一个线程里为真，但是考虑接下来 GCD 切换到另一个线程然后扫描了最后的字节。现在，如果再切换回原线程，`position` 将是被增加后的，下标访问也将越界：

```
for _ in 0..Int.max {
    let newScanner = BinaryScanner(data: "hi".data(using: .utf8)!)
```

```
DispatchQueue.global().async {
    scanRemainingBytes(scanner: newScanner)
}
scanRemainingBytes(scanner: newScanner)
}
```

这个竞态条件不会发生的很频繁(否则我们也不需要用 Int.max 来重现它)，所以在测试的时候我们很难发现这个问题。如果我们将 BinaryScanner 换成结构体的话，这个问题就完全不会再出现了，我们接下来就看看原因。

结构体

值类型意味着一个值变量被复制时，这个值本身也会被复制，而不只限于对这个值的引用的复制。在几乎所有的编程语言中，标量类型都是值类型。这意味着当一个值被赋给新的变量时，并不是传递引用，而是进行值的复制：

```
var a = 42
var b = a
b += 1
b // 43
a // 42
```

在上面的代码执行之后，b 的值将是 43，但是 a 依然是 42。这看上去非常自然，也很明显。然而，在 Swift 中，不仅是标量类型，所有的结构体都是这么工作的。

我们先来看看一个简单的结构体，Point。这个结构体和 CGPoint 很相似，只不过它包含的是 Int 值，而 CGPoint 包含的是 CGFloat：

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

对于结构体，Swift 会自动按照成员变量为它添加初始化方法。也就是说，我们现在可以初始化一个新的变量了：

```
let origin = Point(x: 0, y: 0)
```

因为 Swift 中的结构体拥有值语义，对于使用 `let` 定义的结构体变量，我们不能改变它的任何属性。比如，下面的代码将无法工作：

```
origin.x = 10 // 错误
```

虽然我们在结构体中将 `x` 定义为了一个 `var` 属性，但是因为 `origin` 是用 `let` 定义的，所以我们不能对它进行改变。这么做的主要好处是，当你读到 `let point = ...` 这样的代码，并且你知道 `point` 是一个结构体变量时，你就知道它永远不会发生改变。这对读懂代码很有帮助。

要创建一个可变的变量，我们需要使用 `var`：

```
var otherPoint = Point(x: 0, y: 0)
otherPoint.x += 10
otherPoint // (x: 10, y: 0)
```

和对象不同，所有的结构体变量都是唯一的。举个例子，我们可以创建一个新的变量 `thirdPoint` 并且把 `origin` 的值赋给它。现在我们可以改变 `thirdPoint`，但是我们之前使用 `let` 定义的不可变变量 `origin` 并不会发生改变：

```
var thirdPoint = origin
thirdPoint.x += 10
thirdPoint // (x: 10, y: 0)
origin // (x: 0, y: 0)
```

当你将一个结构体赋值给一个新的变量时，Swift 会自动对它进行复制。虽然听起来这会很昂贵，不过大部分的复制都会被编译器优化掉，Swift 也竭尽全力让这些复制操作更加高效。实际上，标准库中有很多结构体都使用了写时复制的技术进行实现，我们稍后会进行介绍。

如果我们有一个想要经常使用的结构体值，我们可以把它作为静态属性定义在扩展里。比如，我们可以在 `Point` 上定义一个 `origin` 属性，这样我们就可以在我们需要的任何地方写 `Point.origin` 来获得它了：

```
extension Point {
    static let origin = Point(x: 0, y: 0)
}
Point.origin // (x: 0, y: 0)
```

结构体也可以包含其他的结构体。比如，如果我们再定义一个 `Size` 结构体的话，我们就能创建 `Rect` 结构体了，它由一个点和一个尺寸组合而成：

```
struct Size {  
    var width: Int  
    var height: Int  
}  
  
struct Rectangle {  
    var origin: Point  
    var size: Size  
}
```

和上面一样，对 Rectangle 我们也有基于成员的初始化函数。参数的顺序和属性定义的顺序是一致的：

```
Rectangle(origin: Point.origin,  
          size: Size(width: 320, height: 480))
```

如果我们想要为我们的结构体定义一个自定义的初始化方法，我们可以将它直接添加到结构体的定义中去。不过，对于包含自定义初始化方法的结构体，Swift 就不再为它生成基于成员的初始化方法了。通过在扩展中定义自定义方法，我们就可以同时保留原来的初始化方法：

```
extension Rectangle {  
    init(x: Int = 0, y: Int = 0, width: Int, height: Int) {  
        origin = Point(x: x, y: y)  
        size = Size(width: width, height: height)  
    }  
}
```

除了直接设置 origin 和 size，我们也可以调用 self.init(origin:size:) 来进行初始化。

如果我们定义了一个可变变量 screen，我们可以为它添加 didSet，这样每当 screen 改变时这个代码块都将被调用。这个 didSet 对所有结构体定义都会有效，不管是定义在 Playground 里，或者是一个类里，甚至对定义在全局的结构体变量也适用：

```
var screen = Rectangle(width: 320, height: 480) {  
    didSet {  
        print("Screen changed: \(screen)")  
    }  
}
```

也许有点出乎意料，当我们只是改变深入结构体中的某个属性的时候，`didSet` 也会被触发：

```
screen.origin.x = 10 // Screen changed: (10, 0, 320, 480)
```

理解值类型的关键就是理解为什么这里会被调用。对结构体进行改变，在语义上来说，与重新为它进行赋值是相同的。当我们改变了结构体中某个深层次的属性时，其实还是意味着我们改变了结构体，所以 `didSet` 依然会被触发。

虽然语义上来说，我们将整个结构体替换为了新的结构体，但是编译器依然会原地进行变更。由于这个结构体没有其他所有者，实际上我们没有必要进行复制。不过如果有多个持有者的话，重新赋值意味着发生复制。对于写时复制的结构体，工作方式会略有不同（我们稍后再说）。

因为数组是结构体，所以这个特性也适用。如果我们定义了一个包含其他值类型的数组，我们可以更改数组中某个元素的一个属性，整个数组的 `didSet` 也将被触发：

```
var screens = [Rectangle(width: 320, height: 480)] {
    didSet {
        print("Array changed")
    }
}
screens[0].origin.x += 100 // Array changed
```

如果 `Rectangle` 是类的话，`didSet` 就不会被触发了，因为在那种情况下，数组存储的引用不会发生改变，只是引用指向的对象发生了改变。

要将两个 `Point` 相加，我们可以对 `+` 操作符进行重载。在方法中，我们可以将成员相加，并返回新的 `Point`：

```
func +(lhs: Point, rhs: Point) -> Point {
    return Point(x: lhs.x + rhs.x, y: lhs.y + rhs.y)
}
screen.origin + Point(x: 10, y: 10) // (x: 20, y: 10)
```

对于矩形，我们可以添加一个 `translate` 方法来做类似的事，这个方法按照输入偏移量将矩形进行位移。不过我们的第一次尝试似乎失败了：

```
extension Rectangle {
    func translate(by offset: Point) {
        // 错误：不能赋值属性: 'self' 是不可变的
```

```
    origin = origin + offset
}
}
```

编译器告诉我们因为 `self` 是不可变量，所以我们不能给 `origin` 进行赋值（`origin =` 其实是 `self.origin =` 的简写形式）。我们可以把 `self` 想像为一个传递给 `Rectangle` 所有方法的额外的隐式参数。你不需要自己去传递这个参数，但是在函数体内部你可以随时使用 `self`。如果我们想要改变 `self`，或是改变 `self` 自身或者嵌套的（比如 `self.origin.x`）任何属性，我们就需要将方法标记为 `mutating`：

```
extension Rectangle {
    mutating func translate(by offset: Point) {
        origin = origin + offset
    }
}
screen.translate(by: Point(x: 10, y: 10))
screen // (20, 10, 320, 480)
```

编译器会强制我们添加 `mutating` 关键字。只有使用了这个关键字，我们才能在方法内部进行改变。将方法标记为 `mutating`，意味着我们改变了 `self` 的行为。现在它将表现得像是一个 `var`，而不再是 `let`：我们可以任意改变属性。（不过要更精准说的话，它其实并不是一个 `var`，我们马上会进行说明）。

如果我们用 `let` 定义 `Rectangle` 的话，我们不能对它调用 `translate`，这是因为只有 `var` 定义的 `Rectangle` 才能被改变：

```
let otherScreen = screen
// 错误：不能对不可变的量使用可变成员
otherScreen.translate(by: Point(x: 10, y: 10))
```

回想一下内建集合一章，我们现在可以理解将 `let` 和 `var` 应用到集合上的区别了。数组的 `append` 方法被定义为 `mutating`，所以当数组被定义为 `let` 时，编译器不让我们调用这个方法。

Swift 会自动将属性的 `setter` 标记为 `mutating`，你无法调用一个 `let` 的 `setter`。对于下标 `setter` 来说，也是一样的：

```
let point = Point.origin
// 错误：无法赋值属性：'point' 是一个 'let' 常量
point.x = 10
```

在很多情况下，一个方法会同时有可变和不可变版本。比如数组有 `sort()` 方法 (这是个 `mutating` 方法，将在原地排序) 以及 `sorted()` 方法 (返回一个新的数组)。我们也可以为我们的 `translate(by:)` 提供一个非 `mutating` 的版本。这次我们不再改变 `self`，而是创建一个复制，改变它，然后返回这个新的 `Rectangle`：

```
extension Rectangle {
    func translated(by offset: Point) -> Rectangle {
        var copy = self
        copy.translate(by: offset)
        return copy
    }
}
screen.translated(by: Point(x: 20, y: 20)) // (40, 30, 320, 480)
```

`sort` 和 `sorted` 的名字选择是有所考究的，确切说，它们的名字遵循了 Swift API 设计准则。对于 `translate` 和 `translated`，我们也使用了同样的规则。对于拥有可变和不可变版本的方法，甚至还有专门的文档解释：因为 `translate` 拥有副作用，所以它应该读起来是一个祈使动词短语。而不可变的版本应该以 `-ed` 或者 `-ing` 结尾。

我们在本章开头就看到过，当处理可变代码时，很容易就会引入 bug；因为我们刚刚检查过的对象可能会被另一个线程更改 (或者甚至是被同一个线程的其他方法更改)，所以你的假设可能是无效的。

含有 `mutating` 方法和属性的 Swift 结构体就没有这个问题。对于结构体的更改所产生的副作用只会影响本地，它只被应用在当前的结构体变量上。因为所有的结构体变量都是唯一的 (或者说，所有的结构体值都有且仅有一个持有者)，它几乎不可能产生类似的 bug。唯一会出现问题的地方是你在不同的线程中引用了同一个全局的结构体变量。

想要理解 `mutating` 是如何工作的，我们先来看看 `inout` 的行为。在 Swift 中，我们可以将一个函数参数标记为 `inout`。我们先来定义一个全局函数，它可以将一个矩形在两个轴方向上各移动 10 个点。我们不能简单地对 `rectangle` 参数调用 `translate`，因为所有的函数参数默认都是不可变的，它们都以复制的方式被传递进来。所以这里我们需要使用 `translated(by:)`。接下来，我们将函数的结果重新赋值给 `screen`：

```
func translatedByTenTen(rectangle: Rectangle) -> Rectangle {
    return rectangle.translated(by: Point(x: 10, y: 10))
}
screen = translatedByTenTen(rectangle: screen)
```

```
screen // (30, 20, 320, 480)
```

我们能不能写一个原地的函数来改变 `rectangle` 呢。回头看看，`mutating` 关键字做的正是此事。它将隐式的 `self` 参数变为可变的，并且改变了这个变量的值。

在函数中，我们可以将参数标记为 `inout` 来使其可变。就和一个普通的参数一样，值被复制并作为参数被传到函数内。不过，我们可以改变这个复制(就好像它是被 `var` 定义的一样)。然后当函数返回时，原来的值将被覆盖掉。这样，我们就可以用 `translate(by:)` 来替代 `translated(by:)` 了：

```
func translateByTwentyTwenty(rectangle: inout Rectangle) {
    rectangle.translate(by: Point(x: 20, y: 20))
}

translateByTwentyTwenty(rectangle: &screen)
screen // (50, 40, 320, 480)
```

`translateByTwentyTwenty` 函数接受矩形 `screen`，在本地改变它的值，然后将新的值复制回去(覆盖原来的 `screen` 的值)。这个行为和 `mutating` 方法如出一辙。实际上，`mutating` 标记的方法也就是结构体上的普通方法，只不过隐式的 `self` 被标记为了 `inout` 而已。

我们不能对一个 `let` 定义的矩形调用 `translateByTwentyTwenty`。我们只能对可变值调用这个方法：

```
let immutableScreen = screen
// 错误：不能将不可变值作为可变参数传入
translateByTwentyTwenty(rectangle: &immutableScreen)
```

现在，我们就明白怎么写一个像是 `+=` 这样的可变运算符了。这样的运算会改变左侧的值，所以输入的参数必须是 `inout` 的：

```
func +=(lhs: inout Point, rhs: Point) {
    lhs = lhs + rhs
}

var myPoint = Point.origin
myPoint += Point(x: 10, y: 10)
myPoint // (x: 10, y: 10)
```

在函数一章里，我们将更深入地探讨 `inout` 的细节。现在，我们只需要知道 `inout` 被用在了很多地方。比如，现在我们也知道通过下标改变值的原理是什么了：

```
var array = [Point(x: 0, y: 0), Point(x: 10, y: 10)]
array[0] += Point(x: 100, y: 100)
array // [(x: 100, y: 100), (x: 10, y: 10)]
```

array[0] 被自动地作为 inout 变量传入。在函数章节中，我们会解释为什么像是 array[0] 这样的表达式可以被当作 inout 参数使用。

我们来回顾一下本章介绍部分的 BinaryScanner 的例子。我们在下面有一段有问题的代码：

```
for _ in 0..
```

如果 BinaryScanner 是一个结构体，而非类的话，每次 scanRemainingBytes 的调用都将获取它自己的 newScanner 的独立的复制。这样一来，这些调用将能够在数组上保持安全的迭代，而不必担心结构体被另一个方法或者线程所改变。

结构体并不意味着你的代码就可以像魔法一般做到线程安全。比如说，我们保持 BinaryScanner 是一个结构体，但是我们将 scanRemainingBytes 方法的内容内联使用的话，我们就会和上面一样面临竞态条件的问题。在闭包内的 while 循环和闭包外的 while 循环都引用了同一个 newScanner 变量，两者会在同时发生改变：

```
for _ in 0..
```

写时复制

在 Swift 标准库中，像是 Array, Dictionary 和 Set 这样的集合类型是通过一种叫做**写时复制**(copy-on-write) 的技术实现的。我们这里有一个整数数组：

```
var x = [1,2,3]
var y = x
```

如果我们创建了一个新的变量 y，并且把 x 赋值给它时，会发生复制，现在 x 和 y 含有的是独立的结构体。在内部，这些 Array 结构体含有指向某个内存的引用。这个内存就是数组中元素所存储的位置，它们位于堆(heap)上。在这个时候，两个数组的引用指向的是内存中同一个位置，这两个数组共享了它们的存储部分。不过，当我们改变 x 的时候，这个共享会被检测到，内存将会被复制。这样一来，我们得以独立地改变两个变量。昂贵的元素复制操作只在必要的时候发生，也就是我们改变这两个变量的时候发生复制：

```
x.append(5)
y.removeLast()
x // [1, 2, 3, 5]
y // [1, 2]
```

如果 Array 结构体中的引用在数组被改变的一瞬间时是唯一的话(比如，没有声明 y)，那么也不会有复制发生，内存的改变将在原地进行。这种行为就是**写时复制**，作为一个结构体的作者，你并不能免费获得这种特性，你需要自己进行实现。当你的类型内部含有一个或多个可变引用，同时你想要保持值语义，并且避免不必要的复制时，为你的类型实现写时复制是有意义的。

为了展示写时复制的工作方式，我们会用 `NSMutableData` 作为内部引用类型来重新实现 Foundation 框架中的 Data 结构体。Data 是值类型，它的行为正如你所愿：

```
var input: [UInt8] = [0x0b,0xad,0xf0,0x0d]
var other: [UInt8] = [0x0d]
var d = Data(bytes: input)
var e = d
d.append(contentsOf: other)
d // 5 bytes
e // 4 bytes
```

如上所示，d 和 e 是独立的，向 d 中添加一个字节并不会改变 e 的值。

用 `NSMutableData` 写同样的代码，我们可以看到对象是共享的：

```
var f = NSMutableData(bytes: &input, length: input.count)
var g = f
f.append(&other, length: other.count)
f //<0badf00d 0d>
g //<0badf00d 0d>
```

`f` 和 `g` 都引用了同样的对象 (换句话说，它们指向了相同的一块内存)，所以对其中一个进行改变，也会改变另一个变量。我们也可以通过 `==` 运算符来验证它们引用的是同一个对象：

```
f == g // true
```

如果我们只是简单地将 `NSMutableData` 封装到结构体中，我们并不能自动得到值语义。比如，我们可以尝试下面的代码：

```
struct MyData {
    var _data: NSMutableData
    init(_ data: NSData) {
        self._data = data.mutableCopy() as! NSMutableData
    }
}
```

如果我们复制结构体变量，里面进行的是浅复制。这意味着对象本身不会被复制，而只有对象的引用会被复制：

```
let theData = NSData(base64Encoded: "wAEP/w==", options: [])
let x = MyData(theData)
let y = x
x._data === y._data
```

我们可以添加一个 `append` 函数，它将在底层操作 `_data` 属性，再一次，我们可以看到我们创建了一个不包含值语义的结构体：

```
extension MyData {
    func append(_ other: MyData) {
        _data.append(other._data as Data)
    }
}
```

```
x.append(x)
y // <c0010fff c0010fff>
```

因为我们只是更改了 `_data` 指向的对象，我们甚至不需要将 `append` 标记为 `mutating`。因为引用本身是常量，结构体也是常量。所以，虽然 `x` 和 `y` 都发生了改变，我们依然可以将两者都声明为 `let`。

写时复制 (昂贵方式)

要实现写时复制，我们首先将 `_data` 标记为结构体的私有属性。我们不再直接变更 `_data`，而是通过一个计算属性 `_dataForWriting` 来访问它。这个计算属性总是会复制 `_data` 并将其返回：

```
struct MyData {
    fileprivate var _data: NSMutableData
    var _dataForWriting: NSMutableData {
        mutating get {
            _data = _data.mutableCopy() as! NSMutableData
            return _data
        }
    }
    init(_ data: NSData) {
        self._data = data.mutableCopy() as! NSMutableData
    }
}
```

因为 `_dataForWriting` 会更改结构体 (它对 `_data` 属性进行了重新赋值)，这个属性的 `getter` 需要被标记为 `mutating`。这意味着我们只能通过 `var` 的方式声明的变量来使用它。

我们可以在 `append` 方法中使用 `_dataForWriting`，现在 `append` 也需要被标记为 `mutating` 了：

```
extension MyData {
    mutating func append(_ other: MyData) {
        _dataForWriting.append(other._data as Data)
    }
}
```

现在，这个结构体具有值语义了。如果我们将 `x` 赋值给变量 `y`，两个变量将继续指向底层相同的 `NSMutableData` 对象。不过，当我们对其中某个调用 `append` 时，将会进行复制：

```
let theData = NSData(base64Encoded: "wAEP/w==", options: [])!
var x = MyData(theData)
let y = x
x._data === y._data
x.append(x)
y // <c0010fff>
x._data === y._data
```

这种策略可以奏效，但是多次更改同一个变量时，这种方法效率很低。比如说，下面这个例子：

```
var buffer = MyData(NSData())
for _ in 0..5 {
    buffer.append(x)
}
```

每次我们调用 `append` 时，底层的 `_data` 对象都要被复制一次。因为我们没有共享 `buffer` 变量，所以对它进行原地变更会高效得多。

写时复制 (高效方式)

为了提供高效的写时复制特性，我们需要知道一个对象 (比如这里的 `NSMutableData`) 是否是唯一的。如果它是唯一引用，那么我们就可以直接原地修改对象。否则，我们需要在修改前创建对象的复制。在 Swift 中，我们可以使用 `isKnownUniquelyReferenced` 函数来检查引用的唯一性。如果你将一个 Swift 类的实例传递给这个函数，并且没有其他变量强引用这个对象的话，函数将返回 `true`。如果还有其他的强引用，则返回 `false`。对于 Objective-C 的类，它会直接返回 `false`。所以，直接对 `NSMutableData` 使用这个函数的话没什么意义。我们可以创建一个简单的 Swift 类，来将任意的 Objective-C 对象封装到 Swift 对象中：

```
final class Box<A> {
    var unbox: A
    init(_ value: A) { self.unbox = value }
}

var x = Box(NSMutableData())
isKnownUniquelyReferenced(&x) // true
```

如果我们有多个引用指向相同的对象，这个函数将返回 `false`：

```
var y = x
isKnownUniquelyReferenced(&x) // false
```

不单单是全局变量，对于结构体中的引用，这种方法也适用。有了这个方法，我们就可以重写 `MyData`，在发生改变前先检查对 `_data` 的引用是否是唯一的。我们还添加了一个 `print` 语句来在调试时快速查看创建复制的频度：

```
struct MyData {
    fileprivate var _data: Box<NSMutableData>
    var _dataForWriting: NSMutableData {
        mutating get {
            if !isKnownUniquelyReferenced(&_data) {
                _data = Box(_data.unbox.mutableCopy() as! NSMutableData)
                print("Making a copy")
            }
            return _data.unbox
        }
    }
    init(_ data: NSData) {
        self._data = Box(data.mutableCopy() as! NSMutableData)
    }
}
```

在 `append` 方法中，我们需要对 `other` 的 `_data` 属性调用 `unbox`。对于追加操作中原来的值进行复制的逻辑已经被封装在了 `_dataForWriting` 这个计算属性中了：

```
extension MyData {
    mutating func append(_ other: MyData) {
        _dataForWriting.append(other._data.unbox as Data)
    }
}
```

要测试我们代码，让我们再写一个循环：

```
let someBytes = MyData(NSData(base64Encoded: "wAEP/w==", options: [])!)
var empty = MyData(NSData())
```

```
var emptyCopy = empty
for _ in 0..<5 {
    empty.append(someBytes)
}
empty // <c0010fff c0010fff c0010fff c0010fff c0010fff>
emptyCopy // <>
```

运行代码，你会看到上面加入的调试语句只在第一次调用 `append` 的时候被打印了一次。在接下来的循环中，引用都是唯一的，所以也就没有进行复制操作。

这项技术让你能够在创建值类型的自定义结构体的同时，保持像对象和指针那样的高效操作。作为结构体的使用者，你不需要操心去手动复制那些结构体，结构体的实现已经帮你处理好了。得益于写时复制和与其关联的编译器优化，大量的不必要的复制操作都可以被移除掉。

当你定义你自己的结构体和类的时候，需要特别注意那些原本就可以复制和可变的行为。结构体应该是具有值语义的。当你在一个结构体中使用类时，我们需要保证它确实是不可变的。如果办不到这一点的话，我们就需要（像上面那样的）额外的步骤。或者就干脆使用一个类，这样我们的数据的使用者就不会期望它表现得像一个值。

Swift 标准库中的大部分数据结构是使用了写时复制的值类型。比如数组，字典，集合，字符串等这些类型都是结构体。这种设计让我们能在使用这些类型时更容易理解它们的行为。当我们把数组传递给函数时，我们知道这个函数一定不会修改原来的数组，因为它所操作的只是数组的一个复制。同样地，通过数组的实现方式，我们也知道不会发生不必要的复制。对比 Foundation 框架中的数据类型，我们总是需要手动地去复制像是 `NSArray` 或者 `NSString` 这样的类型。当使用 Foundation 的数据类型时，忘记手动复制对象，进而写出不安全的代码，简直就是家常便饭。

在当创建结构体时，类也还是有其用武之地的。有时候你会想要定义一个只有单个实例的从不会被改变的类型，或者你想要封装一个引用类型，而并不想要写时复制。还有时候你可能需要将接口暴露给 Objective-C，这种情况下我们也是无法使用结构体的。通过为你的类定义一个带有限制的使用接口，还是能够做到让其不可变的。

将已经存在的类型封装到枚举中也很有意思。在 [CommonMark 封装](#)一章中，我们会提供一个基于枚举的，对引用类型进行了封装的接口。

写时复制的陷阱

不幸的是，想要意外地引入复制简直太容易了。作为例子，我们将创建一个非常简单的结构体，它将包含指向一个空 Swift 类的引用。这个结构体中有一个 `change` 方法，它是结构体的唯一的 `mutating` 方法，这个方法中将检查引用是否应该非复制，不过我们不会进行实际的复制，而只是打印结果：

```
final class Empty {}

struct COWStruct {
    var ref = Empty()

    mutating func change() -> String {
        if isKnownUniquelyReferenced(&ref) {
            return "No copy"
        } else {
            return "Copy"
        }
        // 进行实际改变
    }
}
```

比如，我们创建一个变量，然后马上改变它，变量没有被共享，所以引用是唯一的，也没有必要进行复制：

```
var s = COWStruct()
s.change() // No copy
```

当我们创建了第二个变量后，引用现在被共享了，所以当我们调用 `change` 的时候，有必要进行复制，一切按部就班：

```
var original = COWStruct()
var copy = original
original.change() // Copy
```

当我们将结构体放到数组中，我们可以直接改变数组元素，且不需要进行复制。这是因为我们使用数组下标直接访问到了内存的位置：

```
var array = [COWStruct()]
```

```
array[0].change() // No copy
```

不过如果我们加一个中间变量的话，就将进行复制：

```
var otherArray = [COWStruct()]
var x = array[0]
x.change() // Copy
```

让人会觉得惊讶的是，所有的其他类型，包括字典、集合以及你自己的类型，会表现得非常不同。举例来说，字典的下标将会在字典中寻找值，然后将它返回。因为我们是在值语义下处理，所以返回的是找到的值的复制。这样一来，因为 COWStruct 已经不会被唯一引用了，所以对这个值进行 change() 将会发生复制：

```
var dict = ["key": COWStruct()]
dict["key"]?.change() // Optional("Copy")
```

如果在你将一个写时复制的结构体放到字典中，又想要避免这种复制的话，你可以将值用类封装起来，这将为值赋予引用语义。

当你在使用自己的结构体时，也需要将这一点牢记于心。比如，我们可以创建一个储存某个值的简单地容器类型，通过直接访问存储的属性，或者间接地使用下标，都可以访问到这个值。当我们直接访问它的时候，我们可以获取写时复制的优化，但是当我们用下标间接访问的时候，复制会发生：

```
struct ContainerStruct<A> {
    var storage: A
    subscript(s: String) -> A {
        get { return storage }
        set { storage = newValue }
    }
}

var d = ContainerStruct(storage: COWStruct())
d.storage.change() // No copy
d["test"].change() // Copy
```

Array 的下标使用了特别的处理，来让写时复制生效。但是不幸的是，现在其他类型都没有使用这种技术。Swift 团队提到过它们希望提取该技术的范式，并将其应用在字典上。

Array 通过使用**地址器**(addressors) 的方式实现下标。地址器允许对内存进行直接访问。数组的下标并不是返回元素，而是返回一个元素的地址器。这样一来，元素的内存可以被原地改变，而不需要再进行不必要的复制。你可以在你自己的代码中使用地址器，但是因为它们没有被官方文档化，所以也许会发生改变。要了解更多信息，可以看看 Swift 仓库中关于 [Accessors](#) 的文档。

闭包和可变性

在这一节中，我们会看看闭包是怎么存储数据的。

例如，有一个函数在每次被调用时生成一个唯一的整数，直到 Int.max。这可以通过将状态移动到函数外部来实现。换句话说，这个函数对变量 i 进行了**闭合**(close)。

```
var i = 0
func uniqueInteger() -> Int {
    i += 1
    return i
}
```

每次我们调用该函数时，共享的变量 i 都会改变，一个不同的整数值将被返回。函数也是引用类型，如果我们将 uniqueInteger 赋值给另一个变量，编译器将不会复制这个函数或者 i。相反，它将会创建一个指向相同函数的引用。

```
let otherFunction: () -> Int = uniqueInteger
```

调用 otherFunction 所发生的事情与我们调用 uniqueInteger 是完全一样的。这对所有的闭包和函数来说都是正确的：如果我们传递这些闭包和函数，它们会以引用的方式存在，并共享同样的状态。

回顾一下在集合协议中基于函数的 fibsIterator 的例子吧，其实我们已经看到过这种行为了。当我们使用迭代器时，迭代器本身就是一个函数，它会改变自己的状态。为了在每次迭代时创建新的迭代器，我们必须将它封装到一个 AnySequence 中。

如果我们想要有多个独立的整数发生器的话，我们可以使用相同的技术：我们不返回一个整数，而是返回一个捕获可变量的闭包。函数返回的闭包是一个引用类型，传递它将导致共享状态。然而，每次调用 uniqueIntegerProvider 都将会返回一个新的从零开始的方法：

```
func uniqueIntegerProvider() -> () -> Int {
```

```
var i = 0
return {
    i += 1
    return i
}
}
```

除了返回闭包，我们也可以将它封装成 AnyIterator。这样，我们就可以在 for 循环里用我们自己的整数发生器了：

```
func uniqueIntegerProvider() -> AnyIterator<Int> {
    var i = 0
    return AnyIterator {
        i += 1
        return i
    }
}
```

Swift 的结构体一般被存储在栈上，而非堆上。不过这其实是一种优化：默认情况下结构体是存储在堆上的，但是在绝大多数时候，这个优化会生效，并将结构体存储到栈上。当结构体变量被一个函数闭合的时候，优化将不再生效，此时这个结构体将存储在堆上。因为变量 i 被函数闭合了，所以结构体将存在于堆上。这样一来，就算 uniqueIntegerProvider 退出了作用域，i 也将继续存在。与此相似，如果结构体太大，它也会被存储在堆上。

内存

值类型在 Swift 中非常普遍。在标准库中大部分的类型要么是结构体，要么是枚举，对它们来说内存管理是很容易的。因为它们只会有一个持有者，所以它们所需要的内存可以被自动地创建和释放。当使用值类型时，你不会被循环引用的问题所困扰。举个例子，来看看下面的代码：

```
struct Person {
    let name: String
    var parents: [Person]
}

var john = Person(name: "John", parents: [])
john.parents = [john]
john // John, parents: [John, parents: []]
```

因为值类型的特点，当你把 john 加到数组中的时候，其实它被复制了。更精确地说的话，应该是“你把 john 的值加到了数组中。”要是 Person 是一个类的话，我们就会引入一个循环引用。但是在这些结构体版本中，john 只有一个持有者，那就是原来的变量值 john。

对于类，Swift 使用自动引用计数 (ARC) 来进行内存管理。在大多数情况下，这意味着所有事情都将按预想工作。每次你创建一个对象的新的引用（比如为类变量赋值），引用计数会被加一。一旦引用失效（比如变量离开了作用域），引用计数将被减一。如果引用计数为零，对象将被销毁。遵循这种行为模式的变量也被叫做**强引用**（它是相对于我们稍后要提到的 `weak` 和 `unowned` 引用而言的）。

比如有下面的代码：

```
class View {  
    var window: Window  
    init(window: Window) {  
        self.window = window  
    }  
}  
  
class Window {  
    var rootView: View?  
}
```

我们可以创建一个变量，它将申请内存并初始化对象：

```
var myWindow: Window? = Window()  
myWindow = nil
```

第一行创建了一个新的实例，此时引用计数为 1。当之后变量被设为 `nil` 时，我们的 `Window` 实例的引用计数将变为 0，这个实例将被销毁。

当把 Swift 和使用垃圾回收机制的语言进行对比时，第一印象是它们在内存管理上似乎很相似。大多数时候，你都不太需要考虑它。不过，看看下面的例子：

```
var window: Window? = Window()  
var view: View? = View(window: window!)  
window?.rootView = view  
view = nil  
window = nil
```

首先，我们创建了 window 对象，window 的引用计数将为 1。之后创建 view 对象时，它持有了 window 对象的强引用，所以这时候 window 的引用计数为 2，view 的计数为 1。接下来，将 view 设置为 window 的 rootView 将会使 view 的引用计数加一。此时 view 和 window 的引用计数都是 2。当把两个变量都设置为 nil 后，它们的引用计数都会是 1。即使它们已经不能通过变量进行访问了，但是它们却互相有着对彼此的强引用。这就被叫做**引用循环**，当处理类似于这样的数据结构时，我们需要特别小心这一点。因为存在引用循环，这样的两个对象在程序的生命周期中将永远无法被释放。

weak 引用

要打破引用循环，我们需要确保其中一个引用要么是 weak，要么是 unowned。weak 引用表示不增加引用计数，并且当被引用的对象被释放时，将该 weak 引用自身设置为 nil。比如，我们可以将 rootView 声明为 weak，这样 window 就不会强引用 view，当 view 被释放时，这个属性也将自动地变为 nil。当我们处理弱引用变量时，我们必须将它声明为可选值。为了调试内存行为，我们可以添加一个析构函数，它会在类实例被回收的时候被调用：

```
class View {  
    var window: Window  
    init(window: Window) {  
        self.window = window  
    }  
    deinit {  
        print("Deinit View")  
    }  
}  
  
class Window {  
    weak var rootView: View?  
    deinit {  
        print("Deinit Window")  
    }  
}
```

在下面的代码中，我们创建了一个 window 和一个 view。view 强引用着 window，但是因为 window 的 rootView 被声明为 weak，所以它不会对 view 有强引用。这样，我们就打破了引用循环，在将两个变量都设为 nil 后，view 和 window 都将被释放：

```
var window: Window? = Window()
```

```
var view: View? = View(window: window!)
window?.rootView = view!
window = nil
view = nil
/*
Deinit View
Deinit Window
*/
```

在使用 `delegate` 的时候，弱引用会非常有用。调用 `delegate` 方法的对象不应该持有这个 `delegate`。所以，`delegate` 一般都会被标记为 `weak`。确保 `delegate` 在需要的时候一直存在通常是另外的对象的职责。

unowned 引用

因为 `weak` 引用的变量可以变为 `nil`，所以它们必须是可选值类型，但是有些时候这并不是你想要的。例如，也许我们知道我们的 `view` 将一定有一个 `window`，这样这个属性就不应该是可选值，而同时我们又不想一个 `view` 强引用 `window`。这种情况下，我们可以使用 `unowned` 关键字，这将不持有引用的对象，但是却假定该引用会一直有效：

```
class View {
    unowned var window: Window
    init(window: Window) {
        self.window = window
    }
    deinit {
        print("Deinit View")
    }
}

class Window {
    var rootView: View?
    deinit {
        print("Deinit Window")
    }
}
```

现在，我们可以创建 window，创建 view，然后设置 window 的 root view。这里没有引用循环，但是我们要负责保证 window 的生命周期比 view 长。如果 window 先被销毁，然后我们访问了 view 上这个 unowned 的变量的话，就会造成运行崩溃。在下面的代码中，我们可以看到两个对象都被释放了：

```
var window: Window? = Window()
var view: View? = View(window: window!)
window?.rootView = view
view = nil
window = nil
/*
Deinit Window
Deinit View
*/
```

对每个 unowned 的引用，Swift 运行时将为这个对象维护另外一个引用计数。当所有的 strong 引用消失时，对象将把它的资源(比如对其他对象的引用)释放掉。不过，这个对象本身的内存将继续存在，直到所有的 unowned 引用也都消失。这部分内存将被标记为无效(有时候我们也把它叫做僵尸(zombie)内存)，当我们试图访问这样的 unowned 引用时，就会发生运行时错误。

不过注意这其实并不是未定义的行为。我们还有第三种选择，那就是 unowned(unsafe)，它不会做运行时的检查。当我们访问一个已经无效的 unowned(unsafe) 引用时，这时候结果将是未定义的。

当你不需要 weak 的时候，还是建议使用 unowned。一个 weak 变量总是需要被定义为 var，而 unowned 变量可以使用 let 来定义。不过，只有在你确定你的引用将一直有效时，才应该使用 unowned。

个人来说，我经常发现自己即使是在那些可以用 unowned 的场合，也还一直在用 weak。我们可能会时不时地对一些代码进行重构，而这可能会导致我们之前对于对象有效的假设失效，这种情况下使用 unowned 就很危险。当使用 weak 时，一个好处是编译器强制我们需要处理引用为 nil 时的可能性。

闭包和内存

结构体和类使用实践

我们在本章介绍时已经指出，选择到底是使用值还是实体来代表你的数据，是高度依赖于你的数据种类和你要解决的相关问题的。在本节中，我们会看一个在银行账户中转账的例子。我们会使用三种方式来进行实现：一种是使用类，一种是使用结构体和纯函数，最后是使用带有 `inout` 参数的结构体。

类

使用类来对银行帐号进行建模是很常见的。因为账户拥有同一性 (identity)，这正适合用类来表达。我们创建了一个很简单的类来表示银行账户，它只存储了一个整数来代表账户中资金情况：

```
typealias USDCents = Int

class Account {
    var funds: USDCents = 0
    init(funds: USDCents) {
        self.funds = funds
    }
}
```

我们创建两个账户：

```
let alice = Account(funds: 100)
let bob = Account(funds: 0)
```

为它们创建一个用于转账的函数 `transfer` 是很容易的。我们选择返回一个 `Bool` 值，来表示转账是否成功。现在转账失败的唯一原因是资金不足：

```
func transfer(amount: USDCents, source: Account, destination: Account)
    -> Bool
{
    guard source.funds >= amount else { return false }
    source.funds -= amount
    destination.funds += amount
}
```

```
    return true  
}
```

调用转账函数也很简单，只需要将数额和两个账户传递进去就可以了，账户将在原地进行更改：

```
transfer(50, source: alice, destination: bob)
```

上面的代码写起来很容易，理解起来也很容易。不过它有一个问题：它不是线程安全的。也许在你要处理的情景里，这并不是一个问题。但是如果你要处理多线程环境中的情况的话，就必须特别小心不要在同一时间里从不同线程中调用这个函数（也就是说，你需要在一个串行队列中执行所有调用）。否则，并发线程将导致系统中的资金莫名其妙地消失或者出现。

纯结构体

我们也可以使用结构体来对帐号建模。定义和前面的例子很相似，但是我们可以去掉初始化方法，因为编译器会帮我们按照成员自动生成默认的初始化方法：

```
struct Account {  
    var funds: USDCents  
}
```

但是 transfer 函数就要复杂一些，它依然接受一个数值和两个帐号。我们使用 var 来创建输入帐号参数的复制，这样它们就能在函数内部进行更改了。不过，这样的变更**不会**改变传进来的原来的值。为了将更改过的帐号信息回传给调用者，我们将返回一个更新过的帐号的多元组，而不是像我们上面做的那样只返回一个布尔值。如果转账没有成功，我们返回 nil：

```
func transfer(amount: USDCents, source: Account, destination: Account)  
    -> (source: Account, destination: Account)?  
{  
    guard source.funds >= amount else { return nil }  
    var newSource = source  
    var newDestination = destination  
    newSource.funds -= amount  
    newDestination.funds += amount  
    return (newSource, newDestination)  
}
```

`transfer` 函数的一个很好的地方是，只需要看看类型，我们就知道它不能改变我们的帐号。因为 `Account` 是结构体，我们知道函数是不能改变它的。和上面一样，值类型让我们的代码更容易理解。

同样地，因为结构体是值，我们知道一个帐号不会被另外的线程更改。至少，这两个帐号的状态都是稳定的。

在现在，我们需要保存一个可变的持有程序中所有账户的变量。在我们的程序中，这就是我们的单一数据源 (*single source of truth*)。我们能够通过这样的方式来更新这个变量：

```
if let (newAlice, newBob) = transfer(50, source: alice, destination: bob) {  
    // 更新数据源  
}
```

同样地，我们需要确保对这个单一数据源的更新是一个接一个进行的。不过，只在一个地方做这件事要比在代码的各个地方都去确保进行线程安全调用要来得容易得多。代价是程序会变得稍微啰嗦一些。

inout 结构体

最后一个例子我们会来看看使用结构体和带有 `inout` 参数的函数的情况。它使用和上面一样的结构体，但是转账的函数略有不同。我们不将帐号参数使用 `var` 的方式进行复制，而是将它们标记为 `inout`。这样一来，这些值在传入时将被复制，在函数的内部，它们不会被其他线程更改。在函数返回时，它们会被复制回原来的值里：

```
func transfer  
(amount: USDCents, inout source: Account, inout destination: Account)  
    -> Bool  
{  
    guard source.funds >= amount else { return false }  
    source.funds -= amount  
    destination.funds += amount  
    return true  
}
```

当我们调用含有 `inout` 修饰的参数的函数时，我们需要为变量加上 `&` 符号。不过注意，和传递 C 指针的语法不同，这里不代表引用传递。当函数返回的时候，被改变的值会被复制回调用者中去：

```
var alice = Account(funds: 100)
var bob = Account(funds: 0)
transfer(50, source: &alice, destination: &bob)
```

这种策略的优势在于既保证了函数体内的稳定性，又使得写起来和基于类的策略一样容易。

在上面三个例子中，我们都需要仔细考虑并行性。第一种方法很容易造成错误，就算在转账函数内部都有可能出错。另外两种方法里，虽然我们依然需要考虑并行，但是不需要在转账函数本身里进行应对，我们可以只在一个地方来解决这个问题。

闭包和内存

在 Swift 中，除了类以外，函数 (包括闭包) 也是引用类型。我们在闭包和可变性的部分已经看到过，闭包可以捕获变量。如果这些变量是引用类型的话，闭包将持有对它们的强引用。比如，如果在 `FileHandle` 对象的 `handle` 里有一个变量，然后你在回调中访问这个变量，回调将增加 `handle` 的引用计数：

```
let handle = FileHandle(forWritingAtPath: "out.html")
let request = URLRequest(url: URL(string: "http://www.objc.io")!)
URLSession.shared.dataTask(with: request) { (data, _, _) in
    guard let theData = data else { return }
    handle?.write(theData)
}
```

一旦回调结束，闭包将被释放，它所闭合的那些变量 (在这个例子中就是 `handle`) 的引用计数将被减去。闭包对变量的强引用是必要的，否则你就有可能在闭包中访问到已经被释放了的变量。

只有那些会逃逸 (escape) 的闭包需要保持它们变量的强引用。在[函数](#)一章里，我们会看到逃逸函数和非逃逸函数的更多细节。

引用循环

闭包捕获它们的变量的一个问题是它可能会 (意外地) 引入引用循环。常见的模式是这样的：对象 A 引用了对象 B，但是对象 B 引用了一个包含对象 A 的回调。让我们考虑之前的例子，当一个视图引用了它的窗口时，窗口通过一个弱引用指向这个根视图。在此基础上，窗口现在多了一个 `onRotate` 回调，它是一个可选值，初始值为 `nil`：

```
class View {  
    var window: Window  
    init(window: Window) {  
        self.window = window  
    }  
    deinit {  
        print("Deinit View")  
    }  
}  
  
class Window {  
    weak var rootView: View?  
    deinit {  
        print("Deinit Window")  
    }  
  
    var onRotate: ((() -> ())?)  
}
```

如果我们像之前那样创建视图，设置窗口，一切照旧，我们不会引入引用循环：

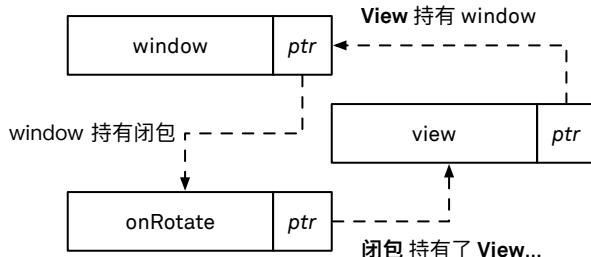
```
var window: Window? = Window()  
var view: View? = View(window: window!)  
window?.rootView = view!
```

视图强引用了窗口，但是窗口只是弱引用视图，一切安好。但是，如果我们对 `onRotate` 回调进行配置，并在其中使用 `view` 的话，我们就会引入一个引用循环：

```
window?.onRotate = {  
    print("We now also need to update the view: \(view)")  
}
```

视图引用了窗口，窗口引用回调，回调引用视图：循环形成。

从图表来看的话，会是这样的：



视图，窗口和闭包之间的引用循环

我们需要找到一种办法来打破这个引用循环。有三种方式可以打破循环，每种方式都在图表中用箭头表示出来了：

- 我们可以让指向 Window 的引用变为 weak。不过不幸的是，这会导致 Window 消失，因为没有其他指向它的强引用了。
- 我们可以将 Window 的 onRotate 闭包声明为 weak。不过这也不可行，因为闭包其实是没有办法被标记为 weak 的，而且就算 weak 闭包是可能的，所有的 Window 的用户需要知道这件事情，因为有时候会需要手动引用这个闭包。
- 我们可以通过使用捕获列表 (capture list) 来让闭包不去引用视图。这在上面这些例子中是唯一正确的选项。

在我们这个生造的例子中，我们很容易就看出这里存在一个引用循环。但是，通常来说实际上不会那么容易。有时候涉及到的对象的数量会很多，引用循环也更难被定位。甚至更糟糕的地方在于，你的代码有可能刚写的时候是正确的，但是在重构的时候有可能你一不小心就引入了引用循环。

捕获列表

为了打破上面的循环，我们需要保证闭包不去引用视图。我们可以通过使用**捕获列表**并将捕获变量 view 标记为 weak 或者 unowned 来达到这个目的。

```
window?.onRotate = { [weak view] in
    print("We now also need to update the view: \(view)")
}
```

捕获列表也可以用来初始化新的变量。比如，如果我们想要用一个 weak 变量来引用窗口，我们可以将它在捕获列表中进行初始化，我们甚至可以定义完全不相关的变量，就像这样：

```
window?.onRotate = { [weak view, weak myWindow=window, x=5*5] in
    print("We now also need to update the view: \(view)")
    print("Because the window \(myWindow) changed")
}
```

这和上面闭包的定义几乎是一样的，只有在捕获列表的地方有所不同。这些变量的作用域只在闭包内部，在闭包外面它们是不能使用的。

总结

我们研究了 Swift 中结构体和类的种种不同。对于需要同一性保证的实体，类会是更好的选择。而对于值类型，结构体会更好。当我们想要在结构体中包含对象时，我们往往需要像是写时复制这样的额外步骤，来确保这个值真的是值类型。我们还讨论了在处理类时，要如何避免引用循环的问题。通常来说，一个问题既可以用结构体解决，也可以用类解决。具体使用哪个，要根据你的需求来决定。不过，就算是那些一般来说会使用引用来解决问题，也可能可以从使用值类型中受益。

函数

6

在开始本章之前，我们先来回顾一下关于函数的事情。如果你已经对头等函数 (first-class function) 的概念很熟悉了的话，你可以直接跳到下一节。但是如果你对此还有些懵懵懂懂的话，可以浏览一下这些内容。

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：

1. 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。
2. 函数能够捕获存在于其局部作用域之外的变量。
3. 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重要顺序是反过来的，或者是遗漏其中的某点，或者把**闭包**和**闭包表达式**混淆了，这确实有时候会很让人迷惑。然而三者鼎足而立，互为补充，如果你少了其中任何一条，那么整个架构将不复存在。

1. 函数可以被赋值给变量，也能够作为函数的输入和输出

在 Swift 和其他很多现代语言中，函数被称为“头等对象”。你可以将函数赋值给变量，稍后，你可以将它作为参数传入给要调用的函数，函数也可以返回一个函数。

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。

这里有一个将函数赋值给变量并将它传递给其他函数的例子：

```
// 这个函数接受 Int 值并将其打印
func printInt(i: Int) {
    print("you passed \(i)")
}
```

要将函数赋值给一个变量，比如 funVar，我们只需要将函数名字作为值就可以了。注意在函数名后没有括号：

```
let funVar = printInt
```

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

```
funVar(2) // 将打印 "you passed 2"
```

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
```

为什么将函数作为变量来处理这件事情如此关键？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中看到过它的威力了。

你也可以在其他函数中返回一个函数：

```
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范围之外的变量

当函数引用了在函数作用域外部的变量时，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，我们再来看看 `returnFunc` 函数。这次我们添加了一个计数器，每次我们调用这个函数时，计数器将会增加。

```
func counterFunc() -> (Int) -> String {
    var counter = 0
```

```
func innerFunc(i: Int) -> String {  
    counter += i // counter is captured  
    return "running total: \(counter)"  
}  
return innerFunc  
}
```

一般来说，因为 `counter` 是一个 `counterFunc` 的局部变量，它在 `return` 语句执行之后应该离开作用域并被摧毁。但是这个因为 `innerFunc` 捕获了它，它将继续存在。我们在结构体和类讨论过，`counter` 将存在于堆上而非栈上。我们可以多次调用 `innerFunc`，并且看到 `running total` 的输出在增加：

```
let f = counterFunc()  
f(3) // running total: 3  
f(4) // running total: 7
```

如果我们再次调用 `counterFunc()` 函数，将会生成并“捕获”新的 `counter` 变量：

```
let g = counterFunc()  
g(2) // running total: 2  
g(2) // running total: 4
```

这不影响我们的第一个函数，它拥有它自己的 `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外部声明的非局部变量 `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。一种是像上面所示那样使用 `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**，就和 1 是整数字面量，"hello" 是字符串面量那样。与 func 相比较，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。使用它们的唯一方法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。

使用闭包表达式声明的 doubler，和之前我们使用 func 关键字声明的函数，其实是完全等价的。它们甚至存在于同一个“命名空间”中，这一点和一些其他语言有所不同。

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 的例子用短得多的形式进行了重写：

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

之所以看起来和原来很不同，是因为我们使用了 Swift 中的一些特性，来让代码更加简洁。我们来一个个看看这些用到的特性：

1. 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，就没有必要现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。
2. 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，从闭包的乘法结果的类型可以推断出闭包返回的也是 Int。
3. 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。
4. 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。

5. 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**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。
6. 最后，如果一个函数除了闭包表达式外没有别的参数，那么方法名后面的调用时的圆括号也可以一并省略。

依次将上面的每个规则使用到最初的表达式里，我们可以逐步得到最后的结果：

```
[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的风格的话，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。

有时候，Swift 可能需要你在类型推断的时候给一些提示。还有某些情况下，你可能会得到和你想象中完全不同的错误类型。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其他代码的话，在这个过程中相信你已经修复好这些代码了。

Swift 有时候会要求你用更明显的方式进行调用。比如，你不能完全省略掉输入参数。假设你想对一组随机数的数组，一种快速的方式是对一个范围进行 `map` 操作，在 `map` 中生成一个随机数。但无论如何你还是要为 `map` 接受的函数提供一个输入参数。在这里，你可以使用单下划线`_`来告诉编译器你承认这里有一个参数，但是你并不关心它究竟是什么：

```
(0..3).map { _ in arc4random() } // [3625814234, 3043175062, 594585166]
```

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，你也可以为参数和闭包表达式中的返回值指定类型：

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

```
}
```

或者，我们也可以选择为所有的 Integer 定义一个顶层函数：

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

如果你想要把这个顶层函数赋值给变量的话，你需要决定它到底要操作哪个类型。变量不能持有泛型函数，它只能是一个指定的版本：

```
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 {} 声明的是一样的。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

在内建集合一章中，我们谈到过将函数作为参数传递来实现参数化。现在让我们来看一个另外的例子：排序。

如果你想要用 Foundation 在 Objective-C 中实现排序的话，你会用到一长串的选择。这些选项为我们提供了灵活性和强大的功能，但是这是以复杂性为代价的。即使是最简单的排序可能你也得回到文档中来查看要如何使用。

在 Swift 中为集合排序就很简单：

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

实际上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 时进行升序排序的重载方法。重载方法意味着你可以通过简单地使用 sorted() 来达到最简单的排序目的。如果你需要用不同于默认升序的顺序进行排序的话，可以提供一个排序函数：

```
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，比如可选值就是一个例子：

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

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.characters.reversed()
    let r = rhs.characters.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（也就是那些返回 `NSComparisonResult` 的函数）来对集合进行排序。这使得 Swift 排序非常强大，而且它也使得这一个函数就能在很大程度上代替 Foundation 框架中各种不同的排序方法的功能。

为了展示这一点，我们来看一个稍微复杂的例子，这个例子受到 Apple 文档中关于排序描述符编程话题的启发。`NSArray` 上的 `sortedArray(using:)` 方法非常灵活，它可以很好地说明 Objective-C 动态特性的强大之处。对于 `selector` 和动态派发的支持在 Swift 中是被保留了的，但是标准库的选择更偏向于基于函数的实现方式。我们一会儿将展示把函数看作参数，以及把函数看作数据，所能够带来的同样的动态效果。

我们从定义一个 `Person` 对象开始。因为我们想要展示 Objective-C 强大的运行时的工作方式，所以我们将它定义为 `NSObject` 的子类（在纯 Swift 中，使用结构体会是更好的选择）：

```
final class Person: NSObject {
    var first: String
    var last: String
    var yearOfBirth: Int
    init(first: String, last: String, yearOfBirth: Int) {
        self.first = first
        self.last = last
    }
}
```

```
    self.yearOfBirth = yearOfBirth  
}  
}
```

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: "Jo", last: "Smith", yearOfBirth: 1970),  
    Person(first: "Joe", last: "Smith", yearOfBirth: 1970),  
    Person(first: "Joe", last: "Smyth", yearOfBirth: 1970),  
    Person(first: "Joanne", last: "smith", yearOfBirth: 1985),  
    Person(first: "Joanne", last: "smith", yearOfBirth: 1970),  
    Person(first: "Robert", last: "Jones", yearOfBirth: 1970),  
]
```

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。在排序的时候我们还会忽略大小写，并且使用用户的区域设置来决定顺序。我们可以使用 NSSortDescriptor 对象来描述如何排序对象，通过它可以表达出各个排序标准：

```
let lastDescriptor = NSSortDescriptor(key: #keyPath(Person.last),  
    ascending: true,  
    selector: #selector(NSString.localizedCaseInsensitiveCompare(_:)))  
let firstDescriptor = NSSortDescriptor(key: #keyPath(Person.first),  
    ascending: true,  
    selector: #selector(NSString.localizedCaseInsensitiveCompare(_:)))  
let yearDescriptor = NSSortDescriptor(key: #keyPath(Person.yearOfBirth),  
    ascending: true)
```

要对数组进行排序，我们使用 NSArray 的 sortedArray(using:) 方法。这个方法可以接受一系列排序描述符。为了确定两个元素的顺序，它会先使用第一个描述符，并检查其结果。如果两个元素在第一个描述符下相同，那么它将使用第二个描述符，以此类推：

```
let descriptors = [lastDescriptor, firstDescriptor, yearDescriptor]  
(people as NSArray).sortedArray(using: descriptors)  
/*  
[Robert Jones (1970), Jo Smith (1970), Joanne smith (1970), Joanne smith (1985),  
Joe Smith (1970), Joe Smyth (1970)]  
*/
```

排序描述符用到了 Objective-C 的两个运行时特性：首先是键路径 key 以及键值编程 (key-value-coding)。我们可以通过键值编程的方式在运行时通过键查找一个对象上的对应值。其次是 selector 参数，它接受一个 selector (实际上就是一个用来描述方法名字的 String)，在运行时，这个 selector 将被转变为用来比较两个对象的函数，对象上指定键所对应的值将被用来进行比较。

这是运行时编程的一个很酷的用例，排序描述符的数组可以在运行时构建，这一点在实现比如用户点击某一列时按照该列进行排序这种需求时会特别有用。

我们要怎么用 Swift 的 sort 来复制这个功能呢？要复制这个排序的**部分**功能是很简单的，比如，你想要使用 localizedCaseInsensitiveCompare 来排序一个数组的话：

```
var strings = ["Hello", "hallo", "Hallo", "hello"]
strings.sort { $0.localizedCaseInsensitiveCompare($1) == .orderedAscending}
strings // ["hallo", "Hallo", "Hello", "hello"]
```

如果只是想用一个对象的某一个属性进行排序的话，也非常简单：

```
people.sorted { $0.yearOfBirth < $1.yearOfBirth }
/*
[Jo Smith (1970), Joe Smith (1970), Joe Smyth (1970), Joanne smith (1970),
Robert Jones (1970), Joanne smith (1985)]
*/
```

不过，当你要把可选值属性与像是 localizedCaseInsensitiveCompare 这样的方法结合起来使用的话，这条路就有点儿走不通了。代码会迅速变得丑陋不堪：

```
var files = ["one", "file.h", "file.c", "test.h"]
files.sort { l, r in r.fileExtension.flatMap {
    l.fileExtension?.localizedCaseInsensitiveCompare($0)
} == .orderedAscending }
files // ["one", "file.c", "file.h", "test.h"]
```

稍后我们会让可选值的排序稍微容易一些。不过现在，我们要先来尝试对多个属性进行排序。要同时排序姓和名，我们可以用标准库的 lexicographicalCompare 方法来进行实现。这个方法接受两个序列，并对它们执行一个电话簿方式的比较，也就是说，这个比较将顺次从两个序列中各取一个元素来进行比较，直到发现不相等的元素。所以，我们可以用姓和名构建两个数组，然后使用 lexicographicalCompare 来比较它们。我们还需要一个函数来执行这个比较，这里我们把使用了 localizedCaseInsensitiveCompare 的比较代码放到这个函数中。

```
people.sorted { p0, p1 in
    let left = [p0.last, p0.first]
    let right = [p1.last, p1.first]

    return left.lexicographicallyPrecedes(right) {
        $0.localizedCaseInsensitiveCompare($1) == .orderedAscending
    }
}

/*
[Robert Jones (1970), Jo Smith (1970), Joanne smith (1985), Joanne smith (1970),
Joe Smith (1970), Joe Smyth (1970)]
*/
```

至此，我们用了差不多相同的行数重新实现了原来的那个排序方法。不过还有很大的改进空间：在每次比较的时候都构建一个数组是非常没有效率的，比较操作是被写死的，而且使用方法我们将无法实现对 yearOfBirth 的排序。

函数作为数据

我们不会选择去写一个更复杂的函数来进行排序，先回头看看现状。排序描述符的方式要清晰不少，但是它用到了运行时编程。我们写的函数没有使用运行时编程，不过它们不太容易写出来或者读懂。

排序描述符其实是一种描述对象顺序的方式。我们现在不去将这些信息存储在类里，而是尝试定义一个函数来描述对象的顺序。最简单的定义是获取两个对象，当它们的顺序正确的时候返回 `true`。这个函数的类型正是标准库中 `sort(by:)` 和 `sorted(by:)` 所接受的参数的类型。我们通过一个通用的 `typealias` 来表述排序描述符：

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

现在，我们就可以定义比较两个 `Person` 对象的排序描述符了，它可以比较出生年份，也可以比较姓的字符串：

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

除了手写这些排序描述符外，我们也可以创建一个函数来生成它们。将相同的属性写两次不太好，比如在 `sortByLastName` 中，我们很容易就会不小心弄成 `$0.last` 和 `$1.first` 进行比较。而且写排序描述符本身也挺无聊的：想要通过名来排序的时候，很可能你就把姓排序的 `sortByLastName` 复制粘贴一下，然后再进行修改。

为了避免复制粘贴，我们可以定义一个函数，它和 `NSSortDescriptor` 大体相似，但不涉及运行时编程的接口。这个函数接受一个键和一个比较方法，返回排序描述符（这里的描述符将是函数，而非 `NSSortDescriptor`）。在这里，`key` 将不再是一个字符串，而是一个函数。要比较两个键，我们使用 `areInIncreasingOrder` 函数。最后，得到的结果也是一个函数，只不过它被 `typealias` 稍微包装了一下：

```
func sortDescriptor<Value, Key>(
    key: @escaping (Value) -> Key,
    _ areInIncreasingOrder: @escaping (Key, Key) -> Bool)
-> SortDescriptor<Value>
{
    return { areInIncreasingOrder(key($0), key($1)) }
}
```

通过这个，我们可以用另一种方式来定义 `sortByYear` 了：

```
let sortByYearAlt: SortDescriptor<Person> =
    sortDescriptor(key: { $0.yearOfBirth }, <)
people.sorted(by: sortByYearAlt)
/*
[Jo Smith (1970), Joe Smith (1970), Joe Smyth (1970), Joanne smith (1970),
Robert Jones (1970), Joanne smith (1985)]
*/
```

我们还可以为所有的 `Comparable` 类型定义一个重载版本的函数：

```
func sortDescriptor<Value, Key>(key: @escaping (Value) -> Key)
-> SortDescriptor<Value> where Key: Comparable
{
    return { key($0) < key($1) }
}
let sortByYearAlt2: SortDescriptor<Person> =
    sortDescriptor(key: { $0.yearOfBirth })
```

上面的两个 `sortDescriptor` 返回的都是与布尔值相关的函数。`NSSortDescriptor` 类有一个初始化方法，它接受类似 `localizedCaseInsensitiveCompare` 这样的比较函数，这类函数将返回三种值(升序，降序，相等)。增加这类函数的支持很简单：

```
func sortDescriptor<Value, Key>(
    key: @escaping (Value) -> Key,
    ascending: Bool = true,
    _ comparator: @escaping (Key) -> (Key) -> ComparisonResult)
-> SortDescriptor<Value>
{
    return { lhs, rhs in
        let order: ComparisonResult = ascending
            ? .orderedAscending
            : .orderedDescending
        return comparator(key(lhs))(key(rhs)) == order
    }
}
```

这样，我们就可以用简短清晰得多的方式来写 `sortByFirstName` 了：

```
let sortByFirstName: SortDescriptor<Person> =
    sortDescriptor(key: { $0.first }, String.localizedCaseInsensitiveCompare)
people.sorted(by: sortByFirstName)
/*
[Jo Smith (1970), Joanne smith (1985), Joanne smith (1970), Joe Smith (1970),
Joe Smyth (1970), Robert Jones (1970)]
*/
```

`SortDescriptor` 现在与 `NSSortDescriptor` 拥有了同样地表达能力，不过它是类型安全的，而且不依赖于运行时编程。

我们现在只能用一个单一的 `SortDescriptor` 函数对数组进行排序。如果你还记得，我们曾经用过 `NSArray.sortedArray(using:)` 方法来用一组比较运算符对数组进行排序。我们可以很容易地为 `Array`，甚至是 `Sequence` 协议添加一个相似的方法。不过，我们需要添加两次：一次是可变版本，另一次是不可变版本。

使用稍微不同的策略，就可以避免添加多次扩展。我们可以创建一个函数来将多个排序描述符合并为单个的排序描述符。它的工作方式和 `sortedArray(using:)` 类似：首先它会使用第一个描

述符，并检查比较的结果。然后，当结果是相等的话，再使用第二个，第三个描述符，直到全部用完：

```
func combine<Value>
    (sortDescriptors: [SortDescriptor<Value>]) -> SortDescriptor<Value> {
    return { lhs, rhs in
        for areInIncreasingOrder in sortDescriptors {
            if areInIncreasingOrder(lhs, rhs) { return true }
            if areInIncreasingOrder(rhs, lhs) { return false }
        }
        return false
    }
}
```

使用这个新的描述符，最后我们可以把一开始的例子重写为这样：

```
let combined: SortDescriptor<Person> = combine(
    sortDescriptors: [sortByLastName, sortByFirstName, sortByYear]
)
people.sorted(by: combined)
/*
[Robert Jones (1970), Jo Smith (1970), Joanne smith (1970), Joanne smith (1985),
Joe Smith (1970), Joe Smyth (1970)]
*/
```

最终，我们得到了一个与 Foundation 中的版本在行为和功能上等价的实现方法，但是这个方法要更安全，也更符合 Swift 的语言习惯。因为 Swift 的版本不依赖于运行时，所以编译器有机会对它进行更好的优化。另外，我们也可以使用它来对结构体或是非 Objective-C 的对象进行排序。

这种方式的实质是将函数用作数据，我们将这些函数存储在数组里，并在运行时构建这个数组。这将动态特性带到了一个新的高度，这也是像 Swift 这样的编译时就确定了静态类型的语言仍然能实现像是 Objective-C 或者 Ruby 的部分动态行为的一种方式。

我们也看到了合并其他函数的函数的用武之地。比如，`combine(sortDescriptors:)` 函数接受一个排序描述符的数组，并将它们合并成了单个的排序描述符。在很多不同的应用场景下，这项技术都非常强大。

除此之外，我们甚至还可以写一个自定义的运算符，来合并两个排序函数：

```

infix operator <||> : LogicalDisjunctionPrecedence
func <||><A>(lhs: @escaping (A,A) -> Bool, rhs: @escaping (A,A) -> Bool)
    -> (A,A) -> Bool
{
    return { x,y in
        if lhs(x,y) { return true }
        if lhs(y,x) { return false }

        // Otherwise, they're the same, so we check for the second condition
        if rhs(x,y) { return true }

        return false
    }
}

```

大部分时候，自定义运算符不是什么好主意。因为自定义运算符的名字无法描述行为，所以它们通常都比函数更难理解。不过，当使用得当的时候，它们也会非常强大。有了上面的运算符，我们可以重写合并排序的例子：

```

let combinedAlt = sortByLastName <||> sortByFirstName <||> sortByYear
people.sorted(by: combinedAlt)
/*
[Robert Jones (1970), Jo Smith (1970), Joanne smith (1970), Joanne smith (1985),
Joe Smith (1970), Joe Smyth (1970)]
*/

```

这样的代码读起来非常清晰，而且可能比原来函数的作法更简洁一些。不过这有一个前提，那就是你(和这段代码的读者)都已经习惯了该操作符的意义。相比自定义操作符的版本，我们还是倾向于选择 `combine(sortDescriptors:)` 函数。它在调用方看来更加清晰，而且显然增强了代码的可读性。除非你正在写一些高度专用的代码，否则自定义的操作符很可能都是在用牛刀杀鸡。

对比我们的版本，Foundation 中的实现仍然还有一个功能上的优势：不用再写额外的代码就可以处理可选值的情况。比如，如果我们想将 Person 上的 last 属性换成可选值字符串，要是使用的是 NSSortDescriptor 的话，我们什么改变都不需要。

但是基于函数的版本就需要一些额外代码。你应该能知道接下来会发生什么：我们再一次写一个接受一个函数，并返回另一个函数的函数。我们可以接受一个普通的比较两个字符串的函数，比如 `localizedCaseInsensitiveCompare`，然后将它转换为一个接受两个可选值字符串的函数。如果两个比较值都是 nil，那么它们相等。如果左侧的值是 nil，而右侧不是的话，返回

升序，相反的时候返回降序。最后，如果它们都不是 nil 的话，我们使用 compare 函数来对它们进行比较：

```
func lift<A>(_ compare: @escaping (A) -> (A) -> ComparisonResult) -> (A?) -> (A?)  
-> ComparisonResult  
{  
    return { lhs in { rhs in  
        switch (lhs, rhs) {  
            case (nil, nil): return .orderedSame  
            case (nil, _): return .orderedAscending  
            case (_, nil): return .orderedDescending  
            case let (l?, r?): return compare(l)(r)  
            default: fatalError() // Impossible case  
        }  
    }}  
}
```

这让我们能够将一个普通的比较函数“抬举”(lift)到可选值的作用域中，这样它就能够和 sortDescriptor 函数一起使用了。如果你还记得我们之前的 files 数组，你会知道因为需要处理可选值的问题，按照 fileExtension 对它进行排序的代码十分难看。不过现在有了新的 lift 函数，它就又变得很清晰了：

```
let lcic = lift(String.localizedCaseInsensitiveCompare)  
let result = files.sorted(by: sortDescriptor(key: {$0.fileExtension}, lcic))  
result // ["one", "file.c", "file.h", "test.h"]
```

我们可以为返回 Bool 的函数写一个类似的 lift。在可选值一章中我们提到过，标准库现在不再为可选值提供像是 `>` 这样的运算符了。如果你使用不小心，可能会产生预想之外的结果，因此它们被删除了。Bool 版本的 lift 函数可以让你轻而易举地将现有的运算符提升为可选值也能使用的函数，以满足你的需求。

基于函数的实现方法有一个不足，那就是函数是不透明的。在使用 NSSortDescriptor 时，我们可以将它打印到控制台，并了解这个排序描述符的信息：比如键路径，selector 的名字，以及排序方式等。而在基于函数的方法里就没法这么做了。对于排序描述符来说，在实践中这不成问题。如果这些信息非常重要的话，我们可以将排序函数包装到一个结构体或者类中，然后在其中存储额外的调试信息。

这样的做法也让我们能够清晰地区分排序方法和比较方法的不同。Swift 的排序算法使用的是多个排序算法的混合。在写这本书的时候，排序算法基于的是内省排序 (introsort)，而内省排序本身其实是快速排序和堆排序的混合。但是，当集合很小的时候，会转变为插入排序 (insertion sort)，以避免那些更复杂的排序算法所需要的显著的启动消耗。

内省排序不是一个“稳定”的排序算法。也就是说，它不保证维持两个相等的值在排序前后的顺序是一致的。不过如果你实现了一个稳定排序，那么因为排序方法和比较方法是分离的，你可以很容易地将两个值进行交换：

```
people.stablySorted(by: combine(  
    sortDescriptors: [sortByLastName, sortByFirstName, sortByYear]  
)
```

局部函数和变量捕获

如果你想要实现一个这样的稳定的排序算法的话，归并排序 (merge sort) 就是一个不错的选择。归并排序由两部分组成：首先将待排数组分解为单个元素的字列表，然后将这些列表进行合并。通常，将 `merge` 定义为一个单独的函数会是不错的选择。但是它也会带来一个问题 — `merge` 需要一些临时的存储空间。

```
extension Array where Element: Comparable {  
    private mutating func merge(lo: Int, mi: Int, hi: Int) {  
        var tmp: [Element] = []  
        var i = lo, j = mi  
        while i != mi && j != hi {  
            if self[j] < self[i] {  
                tmp.append(self[j])  
                j += 1  
            } else {  
                tmp.append(self[i])  
                i += 1  
            }  
        }  
  
        tmp.append(contentsOf: self[i..        tmp.append(contentsOf: self[j..        replaceSubrange(lo..    }  
}
```

```
mutating func mergeSortInPlaceInefficient() {
    let n = count
    var size = 1

    while size < n {
        for lo in stride(from: 0, to: n-size, by: size*2) {
            merge(lo: lo, mi: (lo+size), hi: Swift.min(lo+size*2,n))
        }
        size *= 2
    }
}
```

注意，因为 Array 有一个叫做 min() 的方法，所以我们需要用 Swift.min。这会告诉编译器去使用标准库中的名叫 min 的全局函数，而不是用 Array 上的 min。

当然，你可以在外部初始化存储，并将它作为参数传递进去，但是这么写会有点儿不舒服。因为数组是值类型，因此将一个在外部创建的数组传递到方法内并不能帮助进行存储。一个可能的优化是将 input 的参数换成引用类型，不过文档特别告诉了我们不要依赖这种做法。

另一种方法是，将 merge 定义为一个内部函数，并让它捕获在外层函数作用域中定义的存储：

```
extension Array where Element: Comparable {
    mutating func mergeSortInPlace() {
        // 定义所有 merge 操作所使用的临时存储
        var tmp: [Element] = []
        // 并且确保它的大小足够
        tmp.reserveCapacity(count)

        func merge(lo: Int, mi: Int, hi: Int) {
            // 清空存储，但是保留容量不变
            tmp.removeAll(keepCapacity: true)

            // 和上面的代码一样
            var i = lo, j = mi
            while i != mi && j != hi {
                if self[j] < self[i] {
```

```

        tmp.append(self[j])
        j += 1
    } else {
        tmp.append(self[i])
        i += 1
    }
}

tmp.append(contentsOf: self[i..<mi])
tmp.append(contentsOf: self[j..<hi])
replaceSubrange(lo..<hi, with: tmp)
}

let n = count
var size = 1

while size < n {
    for lo in stride(from: 0, to: n-size, by: size*2) {
        merge(lo: lo, mi: (lo+size), hi: Swift.min(lo+size*2,n))
    }
    size *= 2
}
}
}

```

因为闭包 (也包括内部函数) 通过引用的方式来捕获变量，所以在单次的 `mergeSortInPlace` 调用中，每个对于 `merge` 的调用都将共享这个存储。不过它依然是一个局部变量，不同的并行的 `mergeSortInPlace` 中使用的将是分开的实例。使用这项技术可以为排序带来巨大的速度提升，而并不需要在原来的版本上做特别大的改动。

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。

如果代理只是一个简单的方法，那么使用一个函数来取代它是可行的。不过，在这里你必须进行一些权衡。

Foundation 框架的代理

让我们从以 Foundation 中定义协议的方式入手来定义一组协议和代理。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: class {
    func buttonTapped(atIndex: Int)
}
```

这是一个只针对类的协议，因为在 AlertView 类中，我们希望持有的是一个对代理的弱引用。这样一来，我们就可以不必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。举例来说，我们可以创建一个 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 {
    init() {
        let av = AlertView(buttons: ["OK", "Cancel"])
        av.delegate = self
    }

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

```
    }
}
```

将代理标记为 `weak` 在实践中非常常见，这让内存管理变得很容易。实现代理协议的类不需要考虑引入引用循环的问题。

有时候我们会想让结构体来实现代理协议。在现在 `AlertViewDelegate` 的声明下，这是不可能的，因为这个协议是一个只针对类的协议。

结构体代理

我们可以放宽 `AlertViewDelegate` 的定义，让它不限于只针对类。同时，我们可以将 `buttonTapped(atIndex:)` 标记为 `mutating`。这样，结构体就可以在方法被调用时改变自身的内 容了：

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

我们还需要对 `AlertView` 的 `delegate` 属性进行更改，因为它不能再是弱引用了：

```
class AlertView {
    var buttons: [String]
    var delegate: AlertViewDelegate?

    init(buttons: [String] = ["OK", "Cancel"]) {
        self.buttons = buttons
    }

    func fire() {
        delegate?.buttonTapped(atIndex: 1)
    }
}
```

如果我们将一个对象赋值给 `delegate` 属性，这个对象就将被强引用。当在处理代理的时候，这种强引用意味着我们很容易就在某个时候引入引用循环。不过，现在我们可以使用结构体。举例来说，我们可以创建一个记录所有按钮事件的结构体：

```
struct TapLogger: AlertViewDelegate {
    var taps: [Int] = []
    mutating func buttonTapped(atIndex index: Int) {
        taps.append(index)
    }
}
```

一开始，可能看起来一切正常。我们创建一个 alert view 和一个 logger，然后将两个关联起来。不过，如果我们在事件被触发后再检查 logger.taps，会发现数组依然为空：

```
let av = AlertView()
var logger = TapLogger()
av.delegate = logger
av.fire()
logger.taps // []
```

当我们给 av.delegate 赋值的时候，我们其实是将结构体的复制进行了赋值。所以 taps 并没有被记录在 logger 中，而是被添加到了 av.delegate 里。更糟糕的是，当我们这么赋值后，我们将失去结构体的信息。想要将记录值取回，我们还需要一个条件类型转换：

```
if let theLogger = av.delegate as? TapLogger {
    print(theLogger.taps)
}
```

很显然这种方式并不能很好地工作。当使用类时，很容易造成引用循环，当使用结构体时，原来的值不会被改变。一句话总结：在代理和协议的模式中，并不适合使用结构体。

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，我们完全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: ((Int) -> ())?

    init(buttons: [String] = ["OK", "Cancel"]) {
```

```
    self.buttons = buttons
}

func fire() {
    buttonTapped?(1)
}
}
```

和之前一样，我们可以创建一个 logger 结构体，然后初始化 alert view 实例以及 logger 变量：

```
struct TapLogger {
    var taps: [Int] = []

    mutating func logTap(index: Int) {
        taps.append(index)
    }
}

let av = 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”：

```
av.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：

```
av.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：

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

当与类和回调打交道时，我们有一些忠告。比如现在我们创建了一个含有 `buttonTapped` 方法的 `Test` 类：

```
class Test {  
    func buttonTapped(atIndex: Int) {  
        print(atIndex)  
    }  
}  
  
let test = Test()
```

我们可以将 `Test` 的实例方法 `buttonTapped` 赋值给 `alert view`：

```
av.buttonTapped = test.buttonTapped
```

这样依赖，`alert view` 就（通过闭包）持有一个对 `test` 对象的强引用了。在上面的例子中，因为 `test` 并没有引用 `alert view`，所以不存在引用循环。不过我们来看看刚才 `view controller` 中的例子，就知道这种使用方式非常容易引入引用循环。要避免强引用，通常我们需要在闭包中使用捕获列表：

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

这样一来，`alert view` 就不会强引用 `test` 了。如果 `test` 引用在闭包被调用之前就被释放回收了的话，它在闭包中将是 `nil`，`buttonTapped` 方法也就不会被调用。

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灵活的实现，这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，当你在处理一组紧密联系的多个回调函数（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。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

如果你有一些 C 或者 C++ 背景的话，在 Swift 中 inout 参数前面使用的 & 符号可能会给你一种它是传递引用的印象。但事实并非如此，inout 做的事情是通过值传递，然后复制回来，而**并不是**传递引用。引用官方《Swift 编程语言》中的话：

inout 参数将一个值传递给函数，函数可以改变这个值，然后将原来的值替换掉，并从函数中传出。

在结构体和类中，我们提到过 inout 参数，并且了解了一些 mutating 方法和接受 inout 参数的方法之间的异同。

为了了解到底什么样的表达式可以被当作 inout 参数传递，我们需要对 lvalue 和 rvalue 进行区分。lvalue 描述的是一个内存地址。rvalue 描述的是一个值。比如 array[0] 是一个 lvalue，它代表的是数组中第一个元素所在的内存位置。2 + 2 是一个 rvalue，它描述的是 4 这个值。

对于 inout 参数，你只能传递 lvalue 给他，因为我们不可能对一个 rvalue 进行改变。当你在普通的函数或者方法中使用 inout 时，你需要显式地将它们传入：每个 lvalue 需要在前面添加上 & 符号。比如，当调用一个接受 inout Int 的 increment 函数式，我们为变量添加 & 前缀后将它传入：

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

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

如果是用 let 定义这个变量的话，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，我们还能传入很多其他东西。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，对于所有的下标（包括你自定义的那些下标），只有它同时拥有 get 和 set 两个方法，这都是适用的。类似地，我们也可以将属性值用作 lvalue，只要它们的 get 和 set 都被定义了：

```
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++
```

可变运算符甚至还可以与可选链一起使用。这不仅适用于普通的可选值，对于字典下标也依然可用，如下所示：

```
var dictionary = ["one": 1]
dictionary["one"]?++
```

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，文档已经明确指出了我们不应该依赖 `inout` 的这个行为。

嵌套函数和 `inout`

你可以在嵌套函数中使用 `inout` 参数，Swift 依然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（我们会在本章最后详细提到逃逸函数的内容）：

```
func escapeIncrement(value: inout Int) -> () -> () {
    func inc() {
        value += 1
    }
    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，应该要怎么做呢？是说值应该在改变之后再复制吗？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) 的方法，你应该小心 & 的另一种含义。& 除了在将变量传递给 inout 以外，还可以用来将变量转换为一个不安全的指针。

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，将一个 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，你在传递指针。

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，假设你在和上面相似的作用域情况下，传入一个数组：

```
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。

想要搞清楚为什么，你需要弄明白你传进去的到底是什么。当你为一个参数添加 & 时，你可能调用的是安全的 Swift 的 inout 语义，你也可能是把你的可怜的变量强制转换成了不安全的指针。当处理不安全的指针时，你需要非常小心变量的生命周期。我们会在接下来的互用性的章节里继续深入这方面的内容。

计算属性和下标

有两种方法和其他普通的方法有所不同，那就是计算属性和下标方法。计算属性看起来和常规的属性很像，但是它并不使用任何内存来存储自己的值。相反，这个属性每次被访问时，返回值都将被实时计算出来。下标的话，就是一个遵守特殊的定义和调用规则的方法。

让我们来看看定义属性的不同的方法。我们以代表 GPS 追踪信息的结构体作为开始，它在 record 存储属性中存储了所有的已记录点：

```
struct GPSTrack {  
    var record: [(CLLocation, Date)] = []  
}
```

我们也可以使用 let 而非 var 来定义 record，这样一来，这个值将成为常量存储属性，将不能再被变更。

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 追踪中所有记录的日期值，我们可以创建一个计算属性：

```
extension GPSTrack {  
    /// 返回 GPS 追踪的所有日期  
    /// - 复杂度: O(n), n 是记录点的数量。  
    var dates: [Date] {  
        return record.map {$0.1}  
    }  
}
```

因为我们没有指定 setter，所以 dates 属性是只读的。它的结果不会被缓存，每次在你调用 dates 属性时，结果都要被计算一遍。Swift API 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一个计算属性的耗时是常数时间。

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，Swift 为此准备了一个特殊的 `lazy` 关键字来定义一个延迟属性 (`lazy property`)。需要注意，延迟属性会被自动声明为 `mutating`，因此，这个属性也必须被声明为 `var`。延迟修饰符是编程记忆化的一种特殊形式。

比如，如果我们有一个 `view controller` 来显示 `GPSTrack`，我们可能会想展示一张追踪的预览图像。通过将属性改为延迟加载，我们可以将昂贵的图像生成工作推迟到属性被首次访问：

```
class GPSTrackViewController: UIViewController {  
    var track: GPSTrack = GPSTrack()  
  
    lazy var preview: UIImage = {  
        for point in self.track.record {  
            // 进行昂贵的计算  
        }  
        return UIImage()  
    }  
}
```

关注我们是如何定义延迟属性的：我们使用了一个闭包表达式，来返回想要存储的值 (在我们的例子中，这个值是一张图片)。当属性被第一次访问时，闭包将被执行 (注意闭包后面的括号)，它的返回值被存储在变量中。对延迟属性来说，多于一行的初始化方法是很常见的。

因为延迟属性需要存储，所以在我们需要在 `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，我们需要在闭包中访问实例成员的时候要使用 `self.` (这里我们需要写 `self.track`)。

如果 `track` 属性发生了改变，`preview` 并不会自动更新。让我们来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 = 0  
    var y: Double = 0  
    lazy var distanceFromOrigin: Double = self.x*self.x + self.y*self.y  
  
    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 // 25.0  
point.x += 10  
point.distanceFromOrigin // 25.0
```

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，更好地解决方式是，我们一开始就将 `distanceFromOrigin` 设置为一个普通的（非延迟）计算属性。

我们在结构体和类也已经看到过，对于属性和变量来说，我们还可以实现 `willSet` 和 `didSet` 回调。这两个方法分别会在 `setter` 被调用之前和之后被调用。一个很有用的场景是把它们和 Interface Builder 一起使用：我们可以为一个 `IBOutlet` 实现 `didSet`，这样我们就有机会在 `IBOutlet` 被连接的时候运行代码，比如可以对视图进行一些配置。举个例子，如果我们想在一个 `label` 可用的时候设置它的文本颜色，可以这样做：

```
class SettingsController: UIViewController {  
    @IBOutlet weak var label: UILabel?  
    didSet {  
        label?.textColor = .black  
    }  
}
```

使用不同参数重载下标

在 Swift 中，我们已经看到过一些下标的特殊的语法了。比如我们可以通过 `dictionary[key]` 这样的方式来在字典中进行查找。这些下标的行为很像普通的函数，只不过它们使用了特殊的语法。它们既可以是只读的（使用 `get`），也可以既可读又可写（使用 `get set`）的。和普通的函数一样，我们可以提供不同的类型来对下标进行重载。比如，我们使用数组下标可以获取一个单独的元素，也可以获取一个切片：

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

我们也可以为自己的类型添加下标支持，或者也可以为已经存在的类型添加新的下标重载。举个例子，让我们来定义一个接受半有界区间为参数的 Collection 下标方法。半有界区间指的是那些只有一个端点（比如 lowerBound 或者 upperBound）被指定了的范围。

在 Swift 中，Range 类型代表的是有界区间 (bounded intervals)：每一个 Range 都有起始位置和终止位置。我们之前演示过，我们可以使用它来在数组中（更确切地说，在任何 Collection 类型中）寻找一个子序列。现在，我们将会扩展 Collection，让它支持相似运算符得到的半有界区间。使用 suffix(from:) 和 prefix(upTo:)，我们已经可以获取到这些字序列了。

要表示一个半有界区间，我们创建了两个新的结构体：

```
struct RangeStart<I> { let start: I }
struct RangeEnd<I> { let end: I }
```

我们可以定义两个简便操作符来创建半有界区间。这两个操作符被 prefix 和 postfix 所修饰，这样它们将只接受一个操作数。有了这个简便操作符，我们就可以把 RangeStart(x) 写作 x..
，把 RangeEnd(x) 写作 ..x：

```
postfix operator ..<
postfix func ..<<I>>(lhs: I) -> RangeStart<I> {
    return RangeStart(start: lhs)
}

prefix operator ..<
prefix func ..<<I>>(rhs: I) -> RangeEnd<I> {
    return RangeEnd(end: rhs)
}
```

最后，我们可以将 Collection 进行扩展，添加两个新的下标方法，让它支持半有界的范围：

```
extension Collection {
    subscript(r: RangeStart<Index>) -> SubSequence {
        return suffix(from: r.start)
    }
    subscript(r: RangeEnd<Index>) -> SubSequence {
```

```
    return prefix(upTo: r.end)
}
}
```

有了这些，我们就可以通过这样的半有界下标来读取了：

```
fibs[2..  
]
```

直接使用 `suffix(from:)` 和 `prefix(upTo:)` 其实可以省去很多努力，而为它们添加一个自定义的运算符也有点小题大做。不过这个例子依然可以很好地说明如何使用 `prefix` 和 `postfix` 运算符以及自定义下标。

下标进阶

现在我们已经知道如何添加简单的下标了。我们可以更进一步，除了接受单个参数以外，下标也可以像函数那样接受多个参数。下面的扩展方法可以让我们在字典中进行带默认值的查找。在查找时，当键不存在的情况下，我们将返回默认值。在 `setter` 中，因为 `newValue` 并不是可选值，所以我们可以简单地将其忽略：

```
extension Dictionary {
    subscript(key: Key, or defaultValue: Value) -> Value {
        get {
            return self[key] ?? defaultValue
        }
        set(newValue) {
            self[key] = newValue
        }
    }
}
```

这让我们可以写出非常简短的函数，用以计算一个序列中元素出现的次数。我们在开始时创建一个空字典，然后对于我们遇到的每个元素，我们将它对应的次数加一。如果这个元素还没有存在于字典中，那么默认值 0 将会在查询时被返回：

```
extension Sequence where Iterator.Element: Hashable {
    var frequencies: [Iterator.Element: Int] {
        var result: [Iterator.Element: Int] = [:]
        for x in self {
            result[x] = result[x] + 1
        }
    }
}
```

```
        result[x, or: 0] += 1
    }
    return result
}
}

"hello".characters.frequencies // ["e": 1, "o": 1, "l": 2, "h": 1]
```

自动闭包

我们都对 `&&` 操作符很熟悉了，它又被叫做短路求值。它接受两个操作数，左边的会首先被求值。只有当左边的求值为 `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。也就是说，这种情况下我们可以把右边的操作数“短路”掉，而不去执行它。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。

在几乎所有的语言中，短路求值操作都是通过 `&&` 和 `||` 操作符内建在语言值中的。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。如果一门语言支持闭包，我们就可以通过提供闭包而非值的方式，来模拟短路求值操作。比如，让我们设想一下定义一个和 Swift 中 `&&` 操作符相同的 `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。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(!events.isEmpty, { events[0] > 10 }) {
    // 执行操作
}
```

在 Swift 中有一个很好的特性，能让代码更漂亮。我们也可以使用 `@autoclosure` 标注来自动为一个参数创建闭包。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()
}
```

现在，`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` 值那样来使用它：

```
if and(!events.isEmpty, events[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 Swift 标准库中，`assert` 和 `fatalError` 也使用了 `autoclosure`，因为它们只在确实需要时才对参数进行求值。通过将断言条件从调用时就要求求知推迟到在 `assert` 函数的内部才去求值，可以将这些昂贵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优化版中。

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` 的时候会执行并获取输出信息的 `log` 函数：

```
func log(condition: Bool,
         message: @autoclosure () -> (String),
         file: String = #file, line: Int = #line) {
    if condition { return }
    print("myAssert failed: \(message()), \(file):(function) (\(line) \((line))")
}

log(condition: true, message: "This is a test")
```

这个 `log` 函数使用了像是 `#file`, `#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。

@escaping 标注

正如我们在之前一章中看到的那样，在处理闭包时我们需要对内存格外小心。回想一下捕获列表的例子，在那个例子中为了避免引用循环，我们将 `view` 标记为了 `weak`:

```
window?.onRotate = { [weak view] in
    print("We now also need to update the view: \(view)")
}
```

但是，在我们使用 `map` 这样的函数的时候，我们从来不会去把什么东西标记为 `weak`。因为 `map` 将同步执行，这个闭包不会被任何地方持有，也不会有引用循环被创建出来，所以并不需要这么做。和我们传递给 `map` 的闭包相比，这里存储在 `onRotate` 中的闭包是**逃逸的** (`escape`)，两者有所区别。

一个被保存在某个地方等待稍后 (比如函数返回以后) 再调用的闭包就叫做**逃逸闭包**。而传递给 `map` 的闭包会在 `map` 中被直接使用。这意味着编译去不需要改变在闭包中被捕获的变量的引用计数。

在 Swift 3 中，闭包默认是非逃逸的。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：

```
func sortDescriptor<Value, Key>(
    key: @escaping (Value) -> Key,
    _ areInIncreasingOrder: @escaping (Key, Key) -> Bool)
-> SortDescriptor<Value>
{
    return { areInIncreasingOrder(key($0), key($1)) }
}
```

在 Swift 3 之前，事情完全相反：那时候逃逸闭包是默认的，对非逃逸闭包，你需要标记出 `@noescape`。Swift 3 的行为更好，因为它默认是安全的：如果一个函数参数

可能导致引用循环，那么它需要被显式地标记出来。`@escaping` 标记可以作为一个警告，来提醒使用这个函数的开发者注意引用关系。非逃逸闭包可以被编译器高度优化，快速的执行路径将被作为基准而使用，除非你在有需要的时候显式地使用其他方法。

注意默认非逃逸的规则只对那些**直接参数位置** (*immediate parameter position*) 的函数类型有效。也就是说，类型是函数的存储属性将会是逃逸的 (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多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，以及一个不可选，不逃逸的函数来绕过这个限制：

```
func transform(_ input: Int, with f: ((Int) -> Int)?) -> Int {
    print("Using optional overload")
    guard let f = f else { return input }
    return f(input)
}

func transform(_ input: Int, with f: (Int) -> Int) -> Int {
    print("Using non-optional overload")
    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
```

总结

函数是 Swift 中的头等对象。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地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，以及计算属性 (实际上它是一种特殊的函数)。最后我们介绍了 `@autoclosure` 和 `@escaping` 标注。在泛型和协议两章中，我们还将看到关于如何使用 Swift 中的函数来获取更多灵活性的内容。

字符串

7

不再固定宽度

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。

但这只有当你是用英语书写，并且受众是美国人时才是这样。其他国家和语言需要其他的字符（就连说英语的英国人都需要一个 £ 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，那就不怎么走运了。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，而且大家也都承认其实两个字节也还是不够用，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：由编码单元 (code unit) 组成编码点 (code point)；由编码点组成字符。

Unicode 数据可以被编码成许多不同宽度的编码单元，最普遍的使用的是 8 比特 (UTF-8) 或者 16 比特 (UTF-16)。UTF-8 额外的优势在于可以向后兼容 8 比特的 ASCII。这也使其超过 ASCII 成为网上最流行的编码方式。

Unicode 中的“编码点”在 Unicode 编码空间中是介于 0 到 0x10FFFF 之间的一个单一值。对于 UTF-32，一个编码点会占用一个编码单元。对于 UTF-8 一个编码点会占用一至四个编码单元。起始的 256 个 Unicode 编码点组成的字符和 Latin-1 中的一致。

Unicode “标量” (scalar) 是另一种单元。除了那些“代理” (surrogate) 编码点 (用来标示成对的 UTF-16 编码的开头或者结尾的编码点) 之外的所有编码点都是 Unicode 标量。标量在 Swift 字

字符串字面量中以 "\u{xxxx}" 来表示，其中的 xxxx 是十六进制的数字。比如欧元符号 € 在 Swift 中写作 "\u{20AC}"。

但是即使在编码时使用了 32 位编码单元，用户所认为的在屏幕上显示的“单个字符”可能仍需要由多个编码点组合而成。多数操作字符串的代码对 Unicode 可变长的本质都表现出一定程度的背离。而这会导致恼人的 bug。

Swift 的字符串实现竭尽全力去符合 Unicode 规范。当无法保证符合时，至少也要确保你知悉此事。这是有代价的。在 Swift 中 String 不是一个集合，相反，它是一种提供了多种方式来察看字符串的类型：你可以将其看作一组 Character 的集合；也可以看作一组 UTF-8、UTF-16 或者 Unicode 标量的集合。

Swift 的 Character 类型与其它表示方式不同。它可以编码任意数量的编码点，将它们合在一起可以组成单个“字位簇”(grapheme cluster)。我们很快就会看到这样的例子。

对于除了 UTF-16 之外的所有其它表示方式来说，只能通过索引来访问集合，**而并不能**使用随机存取。也就是说，测量两个索引之间的距离或者将一个索引步进后移若干步都不是 $O(1)$ 的操作。即使是 UTF-16 的表达，也只在你导入了 Foundation 后才是随机存取(稍后会说明)。在进行繁重的文本处理时，一些表示方式可能会比其它方式更慢。在本章中，我们会来探寻这背后的原因，以及一些涉及功能和性能的技术。

字位簇和标准等价

一种区分 Swift.String 与 NSString 处理 Unicode 数据时差异的快速途径是来考察 “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，而且用户多半也对两个都显示为 “résumé”的字符串彼此相等且含有六个字符有着合理的预期，而不管里面的两个 “é” 是由哪种方式生成的。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"
let double = "Pok\u{0065}\u{0301}mon"
```

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.characters.count // 7  
double.characters.count // 7
```

只有当你深入察看其底层形态时才能发现它们的不同：

```
single.utf16.count // 7  
double.utf16.count // 8
```

将此和 NSString 对照：两个字符串并不相等，并且 length 属性（许多程序员大概会用此属性计算显示在屏幕上的字符数）给出了不同的结果：

```
let nssingle = NSString(characters: [0x0065,0x0301], length: 2)  
nssingle.length // 2  
let nsdouble = NSString(characters: [0x00e9], length: 1)  
nsdouble.length // 1  
nssingle == nsdouble // false
```

这里 == 被定义成比较两个 NSObject 的版本：

```
extension NSObject: Equatable {  
    static func ==(lhs: NSObject, rhs: NSObject) -> Bool {  
        return lhs.isEqual(rhs)  
    }  
}
```

就 NSString 而言，这会按字面值做一次比较，而不会将不同字符组合起来的等价性纳入考虑。NSString.isEqualToString 也是这么做的。如果你真要进行标准的比较，你必须使用 NSString.compare。你不知道这点？将来那些不好查的 bug 和暴脾气的国际用户们可够你受的。

当然，只比较编码单元有一个很大的好处：快很多！Swift 的字符串通过 utf16 的表示方式也能达成这个效果：

```
single.utf16.elementsEqual(double.utf16) // false
```

预组合字符的存在使得开放区间的 Unicode 编码点可以和拥有 “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。

抛弃它们也不会有帮助，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，可以用三种不同的形式来书写：通过组合 ö 和一个点；通过组合 ø 和一个尖音符；或者是通过组合 œ 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}",  
    "\u{F2}\u{323}",  
    "\u{6F}\u{323}\u{300}",  
    "\u{6F}\u{300}\u{323}"  
]  
chars.dropFirst().all { $0 == chars.first } // true
```

(all 方法对序列中的所有元素进行条件检查，判断是否为真。此方法的定义见[内建集合](#)。)

实际上，某些变音符号可以被无限地添加：

```
let zalgo = "soon"  
          ^ ^ ^  
          | | |  
          ˘ ˘ ˘
```

在上面，zalgo.characters.count 返回 4，而 zalgo.utf16.count 返回 36。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？

含有 emoji 的颜文字的字符串也令人有些惊讶。比如，一行 emoji 旗帜被认为是一个字符：

```
let flags = "🏳️‍🌈"  
flags.characters.count // 1  
// the scalars are the underlying ISO country codes:  
(flags.unicodeScalars.map { String($0) }).joinWithSeparator(",")  
// 🇺 , 🇩 , 🇬 , 🇪
```

另一方面，"👩🏿".characters.count 返回 2 (一个是通用的人物，一个是皮肤色调)，而 "👩🏿‍👩🏿".characters.count 返回 4，因为多人小组是由单个成员的 emoji 和零宽度结合符组合而成的。

"👩🏿\u{200D}👩🏿\u{200D}👩🏿\u{200D}👩🏿" == "👩🏿‍👩🏿"

虽然连接起来的旗帜认为是一个字符本身很奇怪，但是它就是期望的结果，这些 emoji 就应该被当作单个字符来使用。希望这些结果能随着 Swift 将其字位簇的规则更新到 2016 年六月发布的 Unicode 9.0 而发生变化。

字符串和集合

在 Swift 中字符串有一个叫做 `Index` 的关联类型，字符串的 `startIndex` 和 `endIndex` 属性都是这个类型，`subscript` 也接受这个类型的参数来获取一个指定的字符，`index(after:)` 可以获取某个索引加一之后的索引。

这意味着 `String` 符合遵循 `Collection` 协议的所有要求。然而 `String` 不是一个集合。你不能将其和 `for...in` 一起使用。它也没有继承 `Collection` 或 `Sequence` 的协议扩展。

理论上，你可以自己来扩展 `String` 对此进行改变：

```
extension String: Collection {  
    // 什么都不用做，需要的实现已经有了  
}  
  
var greeting = "Hello, world!"  
greeting.dropFirst(7) // "world!"
```

然而，这样做可能并不明智。字符串不是集合是有原因的，而不是因为 Swift 开发团队忘记了。当 Swift 2.0 引入协议扩展时带来了巨大的益处，所有的集合和序列都可以使用非常多的有用的方法。但这也带来了一些忧虑，处理集合的算法如果被当作字符串的方法来使用的话，会给人一种 **这些方法完全安全并且符合 Unicode 规范的暗示**，而这种暗示并不一定是真的。尽管 `Character` 尽其所能将组合字符序列展现成单个值（上面已经讲过），在某些情况下逐字符地处理字符串仍然可能出现错误的结果。

因此，字符串的字符集合的表示方式被移到了 `characters` 属性上，使其与其他集合类型的表示方式，比如 `unicodeScalars`、`utf8` 和 `utf16` 相仿。选择一种特定的表示方式意味着你清楚自己已经进入一种“集合处理”模式，并且你会仔细考虑所运行的算法的结果。

在这些表示方式中，`CharacterView` 有着特殊的地位。`String.Index` 事实上只是 `CharacterView.Index` 的类型别名。这意味着一旦你获得了一个字符表达的索引，你就能直接用其索引访问该字符串中的字符。

但是这些表示方式都并不是可随机存取的集合（原因在上一节的例子中应该已经讲清楚了）。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编码点有任何帮助。所以这些集合的索引是不可能随机存取的。

因此 `CharacterView` 只遵循 `BidirectionalCollection` 协议。你可以从字符串的任意一端开始，向前或者向后移动。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。

字符串的索引也遵循 `Comparable` 协议。你可能不知道两个索引之间有几个字符，但你至少可以知道其中一个索引是否在另一个之前。

你可以调用 `index(_:offsetBy:)` 方法一口气迭代多个字符：

```
let s = "abcdef"  
// 从开始之后的 5 个字符  
let idx = s.index(s.startIndex, offsetBy: 5)  
s[idx] // f
```

如果有越过字符串尾端的风险的话，你可以通过增加一个界限值来防止 `advancedBy` 越过给定的索引。当到达目标索引之前就遇到界限值的情况下，这个方法将返回 `nil`：

```
let safelidx = s.index(s.startIndex, offsetBy: 400, limitedBy: s.endIndex)  
safelidx // nil
```

这个行为是 Swift 3.0 新加入的。在 Swift 2.2 中，对应的方法是 `advancedBy(_:_limit:)`，但它并不会区分恰好达到边界和超越了边界的行为，而统一地返回最终的值。现在它返回可选值，新的 API 拥有了更强的表达能力。

现在，你可能在看了之后会想，“我知道了！我可以让字符串的下标支持整数！”于是你可能会这么做：

```
extension String {  
    subscript(idx: Int) -> Character {  
        guard let strIdx = index(startIndex, offsetBy: idx, limitedBy: endIndex)  
            else { fatalError("String index out of bounds") }  
        return self[strIdx]  
    }  
}  
s[5] // 返回 "f"
```

然而，就和上面扩展 String 使其成为一个集合一样，我们最好还是避免这类扩展。你也可能会被诱惑来写出这样的代码：

```
for i in 0..<5 {  
    print(s[i])  
}
```

这代码虽然看起来很简单，实际上却是极其低效的。每次通过整数来访问 s 时，都会执行一个 $O(n)$ 的函数来从其起始索引处开始进行迭代。而在一个线性循环中运行另一个线性循环意味着这个 for 循环是 $O(n^2)$ 的。随着字符串长度的增加，该循环所需用的时间将按平方关系增加。

对习惯于处理固定长度字符的人来说，起初这看上去颇具挑战性。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[s.startIndex..
```

不过令人欣慰的是，String 提供了 characters 这个集合方式来访问字符串，也就是说你能随意使用很多有用的技术。许多操作 Array 的函数一样可以操作 String.characters。使用 prefix 方法，同样的事情就清楚多了（注意它返回的是 CharacterView，要将它转换回 String，我们需要将它传递给 String.init）：

```
String(s.characters.prefix(4)) // abcd
```

不使用整数索引就可以很容易地遍历字符串中的字符，你只需要用一个 for 循环就行了。如果你还需要每个字符的序号，可以使用 enumerated：

```
for (i, c) in "hello".characters.enumerated() {  
    print("\u2028(i): \u2028(c)")  
}
```

或许你要找到某个特定的字符。这种情况你可以用 index(of:)：

```
var hello = "Hello!"  
if let idx = hello.characters.index(of: "!) {  
    hello.insert(contentsOf: ", world".characters, at: idx)  
}  
hello // Hello, world!
```

这里注意虽然索引是用 `characters.index(of:)` 找到的，但 `insert(contentsOf:)` 却是直接在字符串上调用的，这是因为 `String.Index` 仅只是 `Character.Index` 的别名而已。

`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后。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。

和 `Array` 一样，`String` 支持 `RangeReplaceableCollection` 协议中的所有方法。但是再一次地，它并不遵循该协议。你可以自己添加相应的声明，但是我们还是不建议这么做，因为它会错误地假设所有的集合操作在任何情况下都是 `Unicode` 安全的：

```
extension String: RangeReplaceableCollection {}
```

```
if let comma = greeting.index(of: ",") {
    print(greeting[greeting.startIndex..
```

和集合类型的特性相比，字符串**没有**提供 `MutableCollection` 协议中的那些方法，该协议为集合在单个元素的下标的 `get` 之外，增加了 `set` 的特性。不过这并非意味着字符串是不可变的——字符串有多个改变自身的方法，但是你不能通过下标操作符来替换单个字符。究其原因，还是因为可变长度字符的关系。大多数人可能会凭直觉认为通过下标操作更新单个元素就像 `Array` 中那样，只需常量时间。但是由于在字符串中一个字符可能是可变长度的，更新单个字符可能会花费的时间是和字符串长度成线性比例的，这是因为改变单个元素的长度需要在内存中将其后的所有元素往前或是往后挪动。因此，即使只是一个元素，你也必须使用 `replaceRange`。

字符串与切片

一个集合函数可以和字符串相处融洽的好迹象就是得到的结果是输入的一个 `SubSequence`。我们知道，在数组上进行切片操作有点不便，因为你获得的返回值不是一个 `Array`，而是一个 `ArraySlice`。这使得用递归函数来将输入进行切片异常痛苦。

`String` 的集合表示方式就没有这种问题。它们的 `SubSequence` 被定义成了 `Self` 的实例，所以那些接受可切片类型作为参数，并返回其子切片的泛型函数可以很好地操作字符串。比如说，这里 `world` 的类型会是 `String.CharacterView`：

```
let world = "Hello, world!".characters.suffix(6).dropLast()
String(world) // world
```

`split` 返回的是一个子切片的数组，对于字符串处理也很好用。它的定义如下：

```
extension Collection {  
    func split(maxSplits: Int = default,  
              omittingEmptySubsequences: Bool = default,  
              whereSeparator isSeparator: (Self.Iterator.Element) throws -> Bool)  
        rethrows -> [AnySequence<Self.Iterator.Element>]  
}
```

最简单的使用方式如下：

```
let commaSeparatedArray = "a,b,c".characters.split {$0 == ","}  
commaSeparatedArray.map(String.init) // ["a", "b", "c"]
```

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。因为 `split` 函数接受一个闭包作为参数，所以除了单纯的字符比较以外，它还能做更多的事情。这里有一个简单的按词折行的例子，其中闭包里捕获了当前行中的字符数：

```
extension String {  
    func wrapped(after: Int = 70) -> String {  
        var i = 0  
        let lines = self.characters.split(omittingEmptySubsequences: false) {  
            character in  
            switch character {  
                case "\n",  
                    " " where i >= after:  
                    i = 0  
                    return true  
                default:  
                    i += 1  
                    return false  
            }  
        }.map(String.init)  
        return lines.joined(separator: "\n")  
    }  
}  
  
let paragraph = "The quick brown fox jumped over the lazy dog."
```

```
paragraph.wrapped(after: 15)
/*
The quick brown
fox jumped over
the lazy dog.
*/
```

在 `split` 后的 `map` 是必要的，因为我们想要的是一个由 `String` 组成的数组，而不是 `String.CharacterView` 数组。

很有可能大多数时候你会想按照字符来切分。使用 `split` 接受一个分隔符作为参数的变种会比较方便：

```
extension Collection where Iterator.Element: Equatable {
    public func split(separator: Self.Iterator.Element,
                      maxSplits: Int = default,
                      omittingEmptySubsequences: Bool = default)
        -> [Self.SubSequence]
}

"1,2,3".characters.split(separator: ",").map(String.init) // ["1", "2", "3"]
```

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 Iterator.Element: Equatable {
    func split<S: Sequence>(separators: S) -> [SubSequence]
        where Iterator.Element == S.Iterator.Element
    {
        return split { separators.contains($0) }
    }
}
```

现在，你就可以写出下列语句了：

```
"Hello, world!".characters.split(separators: ",! ".characters).map(String.init)
// ["Hello", "world"]
```

简单的正则表达式匹配器

为了展示字符串的切片也是字符串这一点是多么有用，我们将基于 Brian W. Kernighan 和 Rob Pike 所著的《程序设计实践》中的正则表达式匹配器来实现一个类似的简单的匹配器。原来的代码尽管优雅简洁，却大量使用了 C 的指针，这使其通常不能很好地用其他语言改写。而通过使用 Swift 的可选值和切片，在简洁性上几乎可以匹敌用 C 写的版本。

首先，让我们定义一个基础的正则表达式类型：

```
/// 简单的正则表达式类型，支持 ^ 和 $ 锚点，  
/// 并且匹配 . 和 *  
public struct Regex {  
    fileprivate let regexp: String  
  
    /// 从一个正则表达式字符串构建进行  
    public init(regexp: String) {  
        self.regexp = regexp  
    }  
}
```

由于该正则表达式的功能将非常简单，通过其构造方法不太可能生成一个“无效”的正则表达式。如果对于表达式的支持更复杂一些（比如支持通过 [] 做多字符匹配等），你就可能会为其定义一个可失败的构造方法了。

接下来，我们为 Regex 添加一个 match 函数，使其接受一个字符串作参数并且当表达式匹配时返回 true。

```
extension Regex {  
    /// 当字符串参数匹配表达式是返回 true  
    public func match(_text: String) -> Bool {  
  
        // 如果表达式以 ^ 开头，那么它只从头开始匹配输入  
        if regexp.characters.first == "^" {  
            return Regex.matchHere(regexp: regexp.characters.dropFirst(),  
                text: text.characters)  
        }  
  
        // 否则，在输入的每个部分进行搜索，直到发现匹配
```

```

var idx = text.startIndex
while true {
    if Regex.matchHere(regex: regexp.characters,
        text: text.characters.suffix(from: idx))
    {
        return true
    }
    guard idx != text.endIndex else { break }
    text.characters.formIndex(after: &idx)
}

return false
}
}

```

匹配函数很简单，它只是从头至尾遍历输入参数的所有可能子串，检查其是否与正则表达式匹配。但是如果这个正则表达式以 ^ 开头，那么只需要从头开始匹配即可。

正则表达式的大部分处理逻辑都在 `matchHere` 里：

```

extension Regex {
    /// 从文本开头开始匹配正则表达式
    fileprivate static func matchHere(
        regexp: String.CharacterView, text: String.CharacterView) -> Bool
    {
        // 空的正则表达式可以匹配所有
        if regexp.isEmpty {
            return true
        }

        // 所有跟在 * 后面的字符都需要调用 matchStar
        if let c = regexp.first, regexp.dropFirst().first == "*" {
            return matchStar(character: c, regexp: regexp.dropFirst(2), text: text)
        }

        // 如果已经是正则表达式的最后一个字符，而且这个字符是 $,
        // 那么当且仅当剩余字符串的空时才匹配
        if regexp.first == "$" && regexp.dropFirst().isEmpty {
            return text.isEmpty
        }
    }
}

```

```

}

// 如果当前字符匹配了，那么从输入字符串和正则表达式中将其丢弃，
// 然后继续进行接下来的匹配
if let tc = text.first, let rc = regexp.first, rc == "." || tc == rc {
    return matchHere(regexp: regexp.dropFirst(), text: text.dropFirst())
}

// 如果上面都不成立，就意味着没有匹配
return false
}

/// 在文本开头查找零个或多个 `c` 字符，
/// 接下来是正则表达式的剩余部分
fileprivate static func matchStar
(character c: Character, regexp: String.CharacterView,
text: String.CharacterView)
-> Bool
{
    var idx = text.startIndex
    while true { // 一个 * 号匹配零个或多个实例
        if matchHere(regexp: regexp, text: text.suffix(from: idx)) {
            return true
        }
        if idx == text.endIndex || (text[idx] != c && c != ".") {
            return false
        }
        text.formIndex(after: &idx)
    }
}
}

```

匹配器用起来很简单：

```
Regex("^h..lo*!$").match("helloooo!") // true
```

这段代码大量使用了切片 (基于范围的下标和 dropFirst 函数) 及可选值 (特别是比较一个可选值与一个非可选值是否相等的能力)。比方说，if regexp.first == "^" 中的 regexp 即使是空字符

串，表达式也能工作。尽管 `".first` 返回 `nil`，你还是可以将其与非可选的 `"^"` 比较。当 `regexp.first` 为 `nil` 时，表达式的值为 `false`。

这段代码最丑陋的部分大概就是 `while true` 的循环了。我们的需求是要遍历所有可能的子串，**包括**字符串尾部的空串。这是为了确保像 `Regex("$").match("abc")` 这样的表达式返回 `true`。如果字符串可以像数组那样使用整数作为索引，我们就能这么写：

```
// ... 表示直到并且包括 endIndex
for idx in text.startIndex...text.endIndex {
    // idx 和结尾之间的字符串切片
    if Regex.matchHere(regexp: _regexp, text: text[idx..<text.endIndex]) {
        return true
    }
}
```

`for` 循环的最后一轮，`idx` 将会等于 `text.endIndex`，于是 `text[idx..<text.endIndex]` 会是一个空串。

那么为什么这行不通呢？我们在内建集合一章中提到过，默认情况下范围既不是序列也不是集合。正因为范围不是序列，所以我们不能对一个字符串索引范围进行迭代。而且因为它不包含 `endIndex`，我们也不能使用字符表达方式的 `indices` 集合。所以，这里我们就只好被迫使用 C 风格的 `while` 循环。

ExpressibleByStringLiteral

在本章中，我们一直将 `String("blah")` 和 `"blah"` 交换着使用。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` 来让你自己的类型也可以通过字符串字面量进行初始化。

字符串字面量隶属于 `ExpressibleByStringLiteral`、`ExpressibleByExtendedGraphemeClusterLiteral` 和 `ExpressibleByUnicodeScalarLiteral` 这三个层次结构的协议，所以实现起来比数组字面量稍费劲一些。这三个协议都定义了支持各自字面量类型的 `init` 方法，你必须对这三个都进行实现。不过除非你真的需要区分是从一个 `Unicode` 标量还是从一个字符簇来创建实例这样细粒度的逻辑，不然都按字符串来实现大概是最容易的了。比如这样：

```
extension Regex: ExpressibleByStringLiteral {
    public init(stringLiteral value: String) {
```

```
    regexp = value
}
public init(extendedGraphemeClusterLiteral value: String) {
    self = Regex(stringLiteral: value)
}
public init(unicodeScalarLiteral value: String) {
    self = Regex(stringLiteral: value)
}
}
```

一旦定义好，你只需要显式地标明类型，就可以开始用字符串字面量来创建正则表达式匹配器了：

```
let r: Regex = "^h..lo*!$"
```

当类型已经标明时就更好用了，因为编译器可以帮助你进行推断：

```
func findMatches(in strings: [String], regex: Regex) -> [String] {
    return strings.filter { regex.match($0) }
}
findMatches(in: ["foo", "bar", "baz"], regex: "^b..") // ["bar", "baz"]
```

默认情况下，字符串字面量会生成 String 类型。这是由于标准库中有如下的 typealias：

```
typealias StringLiteralType = String
```

不过如果你希望为你的应用改变这种默认行为（比方说你写了个在特定场景下更快的字符串类型，假设该类型实现了短字符串优化算法，自身可以直接存储数个字符），你可以重新定义这个 typealias 值：

```
typealias StringLiteralType = StaticString
```

```
let what = "hello"
what is StaticString // true
```

String 的内部结构

(注意：此节描述了 Swift.String 的内部结构。尽管在 Swift 3.0 中该描述是正确的，但由于随时可能产生的改动，你不应该在生产环境中依赖本节中的描述。在此展示该结构的目的是为了帮助读者更好地理解 Swift 字符串的性能特征。如果你想要更深入了解和跟踪字符串的变化，可以参考 [String.swift](#) 和 [StringCore.swift](#) 的源码。)

字符串的内部存储是由这样的东西构成的：

```
struct String {  
    var _core: _StringCore  
}  
  
struct _StringCore {  
    var _baseAddress: UnsafeMutableRawPointer?  
    var _countAndFlags: UInt  
    var _owner: AnyObject?  
}
```

属性 `_core` 当前是公开的，因此可以很容易地访问到。不过即使在未来的版本中变成非公开，你应该依旧可以通过 `unsafeBitCast` 将任意字符串转成 `_StringCore`：

```
let hello = "hello"  
let bits = unsafeBitCast(hello, to: _StringCore.self)
```

(字符串实际上只聚合了一个内部类型。由于其是结构体，除了自身成员外不会有其他开销。故此使用 `unsafeBitCast` 转换这样的外层容器不会有任何问题。)

这就足以通过 `print(bits)` 语句将所有的内容都打印出来了。不过你可能会注意到无法单独获取像是 `_countAndFlags` 这样的内部字段，这是因为它们都是私有的。为了绕过这个限制，我们可以通过自己的代码来复制一个 `_StringCore` 结构，这样就可以对其调用 `unsafeBitCast` 了：

```
/// 对于 Swift._StringCore 的克隆，用来能绕过访问权限控制  
struct StringCoreClone {  
    var _baseAddress: UnsafeMutableRawPointer?  
    var _countAndFlags: UInt  
    var _owner: AnyObject?  
}
```

```
let clone = unsafeBitCast(bits, to: StringCoreClone.self)
clone._countAndFlags // 5
```

于是当你调用 `print(clone._countAndFlags)` 时你会看到输出了 5，这是该字符串的长度。`_baseAddress` 字段是指向 ASCII 字符序列所在内存的指针。你可以通过 C 的 `puts` 函数将该指针所指向的缓冲区打印出来。`puts` 接受一个 `Int8` 的指针，所以你需要先将无类型的指针转换为 `UnsafePointer<Int8>`：

```
if let pointer = clone._baseAddress?.assumingMemoryBound(to: Int8.self) {
    puts(pointer)
}
```

当你按照上面说的做时，可能会输出 `hello`。也可能在输出 `hello` 后跟着一串垃圾内容。因为这块缓冲区并不一定会按照通常的 C 字符串那样以 `\0` 结尾。

那这是否意味着 Swift 内部是以 UTF-8 形式来存储字符串内容的呢？这可以通过存储一个非 ASCII 的字符串来一探究竟：

```
let emoji = "Hello, 🌎"
let emojiBits = unsafeBitCast(emoji, StringCoreClone.self)
```

当你这么做后，你会发现这次和之前有两处不同。第一处是 `_countAndFlags` 属性变成了一个巨大的数。这是因为它不只存储长度了。其高位的比特被用来存放表明此字符串含有非 ASCII 字符的旗标（另外还有一个旗标用来表明该字符串所指向的缓冲区属于一个 `NSString` 实例）。`_StringCore` 有一个公开的属性 `count` 可以很方便地返回编码单元的长度：

```
emoji._core.count // 9
```

第二个变化是 `_baseAddress` 现在指向 16 位字符，这可以从 `elementWidth` 属性反映出来：

```
emoji._core.elementWidth // 2
```

当有一个以上的字符不是 ASCII 时，就会触发 `String` 将其缓冲区以 UTF-16 形式存储。不管你在字符串中存放了什么非 ASCII 字符，即使该字符需要 32 位的存储空间，也还是以 UTF-16 形式存储。以 UTF-32 形式存储的第三种模式并不存在。

`_StringCore` 的最后一个属性 `_owner` 是一个空指针：

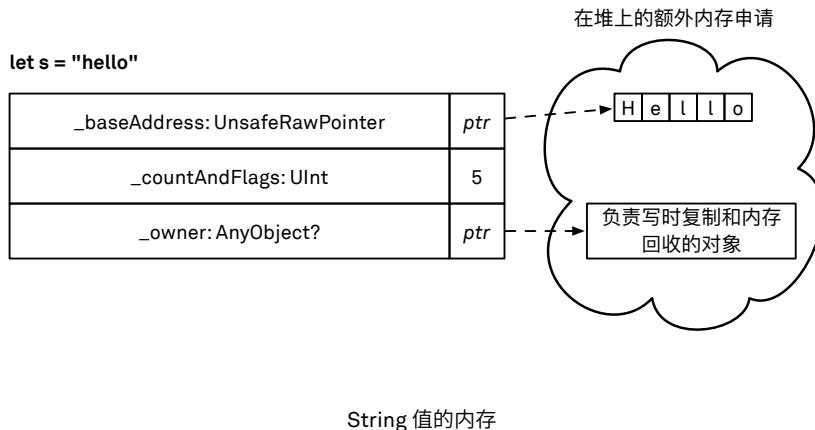
```
emojiBits._owner // nil
```

这是由于迄今为止所有的字符串都是从字符串字面量构造来的。所以缓冲区指向了内存中的常量字符串，后者位于二进制文件中的只读数据部分。反之，如果创建一个非常量的字符串：

```
var greeting = "hello"
greeting.append(" world")
let greetingBits = unsafeBitCast(greeting, to: StringCoreClone.self)
greetingBits._owner
// Optional(Swift._HeapBufferStorage<Swift._StringBufferVars, Swift.UInt16>)
```

这个字符串的 `_owner` 字段将包含有一个值。这将是一个指向由 ARC 管理的类的指针。将其配合 `isKnownUniquelyReferenced` 这样的函数使用，就可以赋予字符串的写时复制行为的值语义。

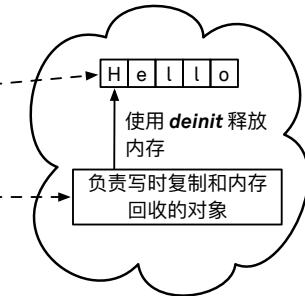
这个 `_owner` 管理被分配以存储字符串的内存。至此我们所勾勒出的画面是这样的：



因为 `_owner` 是一个类，所以它可以有 `deinit` 方法。当 `deinit` 方法被调用时，就释放内存：

`s` 离开作用域

<code>_baseAddress: UnsafeRawPointer</code>	<code>ptr</code>
<code>_countAndFlags: UInt</code>	5
<code>_owner: AnyObject?</code>	<code>ptr</code>



当 String 离开作用域

字符串和数组等其它标准库的集合类型一样，都是写时复制的。当你将一个字符串赋值给另一个字符串变量时，前者的缓冲区并不会立即被复制。相反，正如任何结构体的复制一样，这只会对其所含的字段做一次浅拷贝，而两个变量的字段在最初时会共享存储内容：

`s`

<code>_baseAddress: UnsafeRawPointer</code>	<code>ptr</code>
<code>_countAndFlags: UInt</code>	5
<code>_owner: AnyObject?</code>	<code>ptr</code>

`var s2 = s`

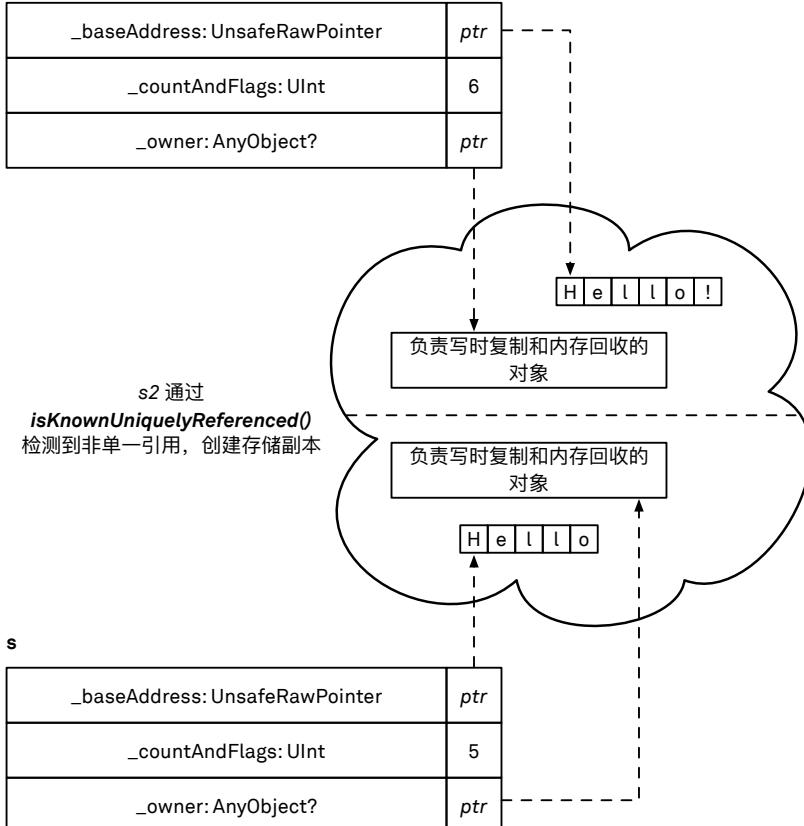
<code>_baseAddress: UnsafeRawPointer</code>	<code>ptr</code>
<code>_countAndFlags: UInt</code>	5
<code>_owner: AnyObject?</code>	<code>ptr</code>



两个 String 共享同样的内存

之后，当其中一个字符串改变内容时，代码通过检查 `_owner` 是否是唯一引用来检测这样的共享情况。如果其不是唯一引用的，那么在改变内容前需要先复制共享的缓冲区，之后这块缓冲区就不再共享了：

```
s2.append("!")
```



更多关于写时复制的内容，请参见结构体与类一章。

这个结构在切片方面还有一个优势。如果从一个字符串创建切片，这些切片的内部是这样的：

分解 "hello, world"

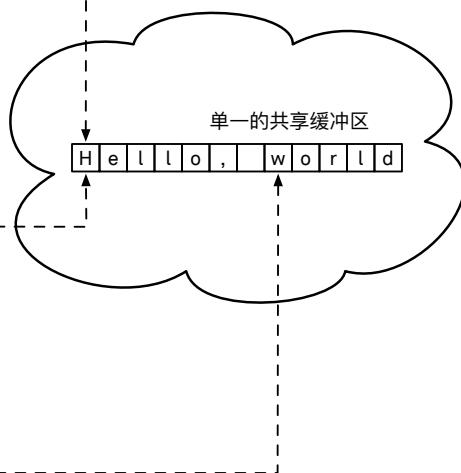
_baseAddress	ptr
_countAndFlags	12
_owner	ptr

"hello"

_baseAddress	ptr
_countAndFlags	5
_owner	ptr

"world"

_baseAddress	ptr
_countAndFlags	5
_owner	ptr



字符串切片

这意味着对一个字符串调用 `split`，本质上创建的是一个起始/结束指针的数组，这些指针指向原始字符串内部的缓冲区。调用 `split` 并不会做大量的拷贝。而这也是有代价的：只要持有了字符串的某个切片，即使那个切片只是几个字符大小，而原始字符串有好几兆大小，它也会导致整个字符串的内存无法释放。

当从 `NSString` 创建 `String` 时还有一个优化。该 `String` 的 `_owner` 将引用原始的 `NSString`，而其缓冲区将指向 `NSString` 的存储空间。我们可以通过字符串的 `owner` 引用可以转换为 `NSString` 这一事实来证明这一点，因为该字符串本来就是个 `NSString`：

```
let ns = "hello" as NSString
let s = ns as String
let nsBits = unsafeBitCast(s, to: StringCoreClone.self)
nsBits._owner is NSString // true
nsBits._owner === ns // true
```

Character 的内部组织结构

正如我们所看到的，`Swift.Character` 表示一串任意长度的编码点。`Character` 是如何对此进行管理的呢？如果你查看[源码](#)，会发现 `Character` 的定义是这样的：

```
struct Character {
    enum Representation {
        case large(Buffer)
        case small(Builtin.Int63)
    }

    var _representation: Representation
}
```

这种将一小部分元素存储在内部并且切换到堆上缓冲区的技术有时被称作“短字符串优化”。由于字符大多数时候就是几个字节，这种技术特别适用。

`Builtin.Int63` 是一个内部的 LLVM 类型，它只对标准库可用。63 位这个不太常见的数字暗示了另一个可能的优化。因为区别两个枚举成员需要一位，所以想要在一个机器字长范围内放下整个结构体，63 就是可用的最大宽度。不过这个优化现在还没有起效，因为 `large` 成员的关联值是一个占用整个 64 位宽度的指针。指针对齐原则可以让一个有效对象的指针的一些位为 0，这样就可以用作枚举成员的存储，但是这个优化在 [Swift 3.0 中还没有实现](#)。结果就是，一个 `Character` 现在长度为 9 个字节：

```
MemoryLayout<Character>.size // 9
```

编码单元表示方式

有时我们需要降低一个抽象层次，直接操作 Unicode 编码点而不是字符。这样做有几个常见的原因。

首先，可能你确实需要操作编码单元。比如要将其渲染到一个 UTF-8 编码的网页中，又或者要和一个接受编码单元作为参数的非 Swift API 协同工作。

作为使用编码单元的 API 的例子，不妨看看 Swift 的字符串和 Foundation 框架中的 CharacterSet 的结合使用。CharacterSet API 大多数都是定义在 Unicode 标量上的，所以如果你要用 CharacterSet 来拆分一个字符串，可以通过 unicodeScalars 方式来操作：

```
extension String {
    func words(with charset: CharacterSet = .alphanumerics) -> [String] {
        return self.unicodeScalars.split {
            !charset.contains($0)
        }.map(String.init)
    }
}

let s = "Wow! This contains _all_ kinds of things like 123 and \"quotes\"?"
s.words()
/*
["Wow", "This", "contains", "all", "kinds", "of", "things", "like", "123",
"and", "quotes"]
*/
```

这段代码在非字母和数字的字符位置将原字符串分解，并输出一个含有 String.UnicodeScalarView 切片的数组。这些切片可以通过 map 函数调用 String 接收 UnicodeScalarView 类型参数的构造方法来转回成字符串。

好消息是，即使在经过那么复杂的系列操作之后，这些由 words 方法生成的字符串切片依然只是原始字符串的表示方式；这种特性并不会在转成 UnicodeScalarView 再转回来后丢失。

使用这些表示方式的第二个可能的理由是操作编码单元比操作完全合成后的字符要快得多。因为为了合成字位簇，必须检查每个字符后是否跟着组合字符。想要知道这些表示方式到底会快多少，可以看看后面的 [性能一节](#)。

除开 `UnicodeScalarView` 以外，`UTF16View` 也是一个常用的表示方式，特别是在和 `Foundation` 框架结合使用时我们会见到很多 `UTF-16` 的字符串。`UTF-16` 表示方式有一个其它方式所没有的优势：它支持随机存取。正如我们之前看到的，字符串在 `String` 类型内部存储的方式决定了只有该表示形式类型可以支持随机存取。也就是说第 `n` 个 `UTF-16` 编码单元总是在缓冲区的第 `n` 个位置上（即使字符串是“ASCII 缓冲模式”，这也只是每一项的步进宽度的问题）。

Swift 团队决定不在标准库这种添加 `String.UTF16View` 对 `RandomAccessCollection` 的支持。他们把相关的声明移动到了 `Foundation` 框架中，所以如果你想使用随机存取访问 `UTF16View` 的话，就需要先导入 `Foundation`。在 `Foundation` 的 源代码 中有这样一个注释解释了原因：

```
// String.UTF16View 的随机存取支持，只在 Foundation 被导入时生效。  
// 将这个 API 声明在 Foundation 中可以使 Swift 核心代码与 UTF16 表达方式解耦。  
...  
extension String.UTF16View : RandomAccessCollection {
```

如上所述，现在那些依赖于随机存取的代码中的 `String` 的背后是 `NSString`，它在内部使用的也是 `UTF-16`，所以今后就算 `String` 使用了一种不同的内部表达方式，也不会对现有代码造成破坏。

也就是说，需要随机存取的场合可能比你想得要少。实践中大多数字符串操作都只需要顺序访问存取。但是一些字符串处理算法依赖随机存取来提高效率。比如 Boyer-Moore 字符串搜索算法就依赖在被搜索字符串中跳过多个字符的能力。

假使 `UTF-16` 方式支持随机存取，你就能对其使用要求这种特性的算法。比方说，你可以使用我们在泛型一章中定义的 `search` 算法：

```
let helloWorld = "Hello, world!"  
if let idx = helloWorld.utf16.search(for: "world".utf16)?  
    .samePosition(in: helloWorld)  
{  
    print(helloWorld[idx..}
```

但是请注意！这些便利性或效率上的优势是有代价的。代价就是你的代码可能不再完全符合 `Unicode` 规范。比如很不幸，下判断言为真：

```
let text = "Look up your Pok\u{0065}\u{0301}mon in a Pok\u{00e9}dex."  
text.utf16.search(for: "Pok\u{00e9}mon".utf16) // nil
```

Unicode 规范的定义中，变音符号是被用来和字母符号一起使用，来合成一个字符的，所以下面的代码会更好一些：

```
let nonAlphas = CharacterSet.alphanumerics.inverted
text.unicodeScalars.split(whereSeparator: nonAlphas.contains).map(String.init)
// ["Look", "up", "your", "Pokémon", "in", "a", "Pokédex"]
```

CustomStringConvertible 和 CustomDebugStringConvertible

print 和 String.init 这样的函数，以及字符串插值被设计成接收任何类型的参数。甚至不需要任何自定义代码，你获得的结果也常常是可以接受的，因为结构体在默认情况下将打印它们的属性：

```
print(Regex("colou?r"))
// 打印 Regex("colou?r")
```

你可能想要更好一些，特别是你的类型有一些不希望被展示的私有变量。别怕！让你的自定义类被传给 print 时输出令人满意的格式化输出只需要很少的代码：

```
extension Regex: CustomStringConvertible {
    public var description: String {
        return "/\\($0)/"
    }
}
```

现在，如果有人将你的自定义类型通过各种手段转成字符串（比如用在类似 print 的流式函数，或者当做 String(describing:) 的参数，又或者用在某个字符串插值中使用），都会得到 /expression/：

```
let regex = Regex("colou?r")
print(regex) // /colou?r/
```

还有一个 CustomDebugStringConvertible 协议，实现该协议可以在调用 String(reflecting:) 时输出更多调试信息。

```
extension Regex: CustomDebugStringConvertible {
    public var debugDescription: String {
        return "{expression: \\(regexp)}"
    }
}
```

如果没有实现 `CustomDebugStringConvertible`, `String(describing:)` 会退回使用 `CustomStringConvertible`。所以如果你的类型很简单, 通常没必要实现 `CustomDebugStringConvertible`。不过如果你的自定义类型是一个容器, 那么遵循 `CustomDebugStringConvertible` 以打印其所含元素的调试描述信息会更考究一些。我们可以把集合协议一章中的 `FIFOQueue` 例子扩展一下:

```
extension FIFOQueue: CustomStringConvertible,
    CustomDebugStringConvertible
{
    public var description: String {
        // 使用 String(describing:) 对元素进行打印, 它将优先使用 CustomStringConvertible
        let elements = map { String(describing: $0) }.joined(separator: ", ")
        return "[\\(elements)]"
    }

    public var debugDescription: String {
        // 使用 String(describing:) 对元素进行打印, 它将优先使用 CustomDebugStringConvertible
        let elements = map { String(describing: $0) }.joined(separator: ", ")
        return "FIFOQueue: [\\(elements)]"
    }
}
```

注意这里“优先使用”这个词, 当 `CustomStringConvertible` 不可用时, `String(describing:)` 将退回使用 `CustomDebugStringConvertible`。所以你在调试时做了任何额外工作的话, 请确保也实现了 `CustomStringConvertible`。如果你的 `description` 和 `debugDescription` 是一样的话, 你可以随意实现一个就行。

还有一点, 就算使用的是 `String(describing:)`, `Array` 还是会为它的元素打印调试版本的描述。Swift 开发组的邮件列表中指出过原因, 是因为数组的描述应该永远不会有呈现给用户的需求, 它们只应该被用在调试中。另外, 一个空字符串的数组如果打印出来不含引号的话会有些奇怪, 而如果用 `String.description` 打印空字符串的话, 就是不含引号的版本。

遵循 `CustomStringConvertible` 协议意味着某个类型有着漂亮的 `print` 输出，所以你可能很想写一个像下面这样的泛型函数：

```
func doSomethingAttractive<T: CustomStringConvertible>(with value: T) {  
    // 因为 CustomStringConvertible 的输出应该很漂亮，所以能很好地打印某个值  
}
```

然而你不应该这么来使用 `CustomStringConvertible`。我们应该使用 `String(describing:)`，而不是检查某个类型是否有 `description` 属性。如果某个类型不遵循 `CustomStringConvertible`，那也只能忍受其丑陋的输出了。所以你写的任何稍微复杂一些的类型都应该实现 `CustomStringConvertible`，这要不了几行代码。

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。它们是如何工作的呢？这两个函数的默认版本的实现调用了 `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` 是标准库中唯一的输出流类型：

```
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。因为这个声明是公开的，你甚至可以用它来做很多有意思的事情：

```
var printCapture = ""
_playgroundPrintHook = { text in
    printCapture += text
}
print("This is supposed to only go to stdout")
printCapture // This is supposed to only go to stdout
```

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) {
        string.utf8CString.dropLast().withUnsafeBufferPointer {
            append($0)
        }
    }
}
```

```
    }
}
```

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`, `Character` 和 `UnicodeScalar` 满足 `TextOutputStreamable`，不过你也可以自行为你类型的添加 `TextOutputStreamable` 支持。一种方式是使用 `print(_:to:)`。不过，要特别注意很容易就会忘掉调用中的 `to:` 参数。只有在用非标准输出的其他输出目标时我们才有机会发现这个错误。另一种方式是直接调用目标流上的 `write` 方法。我们的队列就是以这种方式满足 `TextOutputStreamable` 的：

```
extension FIFOQueue: TextOutputStreamable {
    func write<Target: TextOutputStream>(to target: inout Target) {
        target.write("[")
        target.write(map { String(describing: $0) }.joined(separator: ","))
        target.write("]")
    }
}

var textRepresentation = ""
let queue: FIFOQueue = [1,2,3]
queue.write(to: &textRepresentation)
textRepresentation // [1,2,3]
```

除了更加复杂以外，这和 `let textRepresentation = String(describing: queue)` 并没有太多不同。对于输出流来说，一个有趣的事是，输出源可以多次调用 `write`，流也将立即处理每一次的 `write` 操作。如果你想要写一些像下面这样的代码的话，就会非常简单了：

```
struct SlowStreamer: TextOutputStreamable, ExpressibleByArrayLiteral {
    let contents: [String]

    init(arrayLiteral elements: String...) {
        contents = elements
    }

    func write<Target: TextOutputStream>(to target: inout Target) {
        for x in contents {
```

```
    target.write(x)
    target.write("\n")
    sleep(1)
}
}
}

let slow: SlowStreamer = [
    "You'll see that this gets",
    "written slowly line by line",
    "to the standard output",
]
print(slow)
```

随着新的行被打印入 target，输出也随着出现，而不需要等待调用结束。

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 }

        // 能够直接传递给 C 函数的字符串是
        // const char* 的，参阅互用性一章获取更多信息！
        fputs(string, stderr)
    }
}

var standarderror = StdErr()
print("oops!", to: &standarderror)
```

流还能够持有状态，或者对输出进行变形。除此之外，你也能够将多个流链接起来。下面的输出流将所有指定的短语替换为给定的字符串。和 String 一样，它也遵守 TextOutputStreamable，这让它可以同时作为文本流操作的输出目标和输出源：

```
struct ReplacingStream: TextOutputStream, TextOutputStreamable {
    let toReplace: DictionaryLiteral<String, String>
```

```
private var output = ""

init(replacing toReplace: DictionaryLiteral<String, String>) {
    self.toReplace = toReplace
}

mutating func write(_ string: String) {
    let toWrite = toReplace.reduce(string) { partialResult, pair in
        partialResult.replacingOccurrences(of: pair.key, with: pair.value)
    }
    print(toWrite, terminator: "", to: &output)
}

func write<Target: TextOutputStream>(to target: inout Target) {
    output.write(to: &target)
}

var replacer = ReplacingStream(replacing: [
    "in the cloud": "on someone else's computer"
])

let source = "People find it convenient to store their data in the cloud."
print(source, terminator: "", to: &replacer)

var output = ""
print(replacer, terminator: "", to: &output)
output
// People find it convenient to store their data on someone else's computer.
```

上面的代码中，我们使用了 `DictionaryLiteral` 而不是一个普通的字典。`Dictionary` 有两个副作用：它会去掉重复的键，并且会将所有键重新排序。如果你想要使用像是 `[key: value]` 这样的字面量语法，而又不想引入 `Dictionary` 的这两个副作用的话，就可以使用 `DictionaryLiteral`。`DictionaryLiteral` 是对于键值对数组（比如 `[(key, value)]`）的很好的替代，它不会引入字典的副作用，同时让调用者能够使用更加便捷的 `[:] 语法。`

字符串性能

不可否认，将多个变长的 UTF-16 值合并到扩展字位簇，会比仅仅只是遍历存储 16 位的值的缓冲区开销要大。但是到底开销有多大？我们可以将之前写的正则表达式匹配器适配成可以接受所有类型的集合表示形式，以此来测试性能。

但这有一个问题。理想情况下，你会写一个泛型的正则匹配器，使用一个占位符来代表字符串的表示形式。但这行不通：四种不同形式并没有一个共同遵循的“字符串表示形式”的协议。同时，在我们的正则匹配器中，需要类似 * 和 ^ 这样的特定字符常量来和正则比较。在 `UTF16View` 中，这些常量将是 `UInt16` 类型。但在字符表示中，它们将是 `Character` 类型。最后，我们想要正则匹配器的构造方法依然接受一个 `String` 类型作为参数。它怎么会知道调用哪个方法来获得合适的表示呢？

有一种技术可以将这些可变的逻辑打包成一个类型，并且基于此类型将正则匹配器参数化。首先，我们定义一个含有所有必要信息的协议：

```
protocol StringViewSelector {
    associatedtype View: Collection

    static var caret: View.Iterator.Element { get }
    static var asterisk: View.Iterator.Element { get }
    static var period: View.Iterator.Element { get }
    static var dollar: View.Iterator.Element { get }

    static func view(from s: String) -> View
}
```

这些信息包括一个关联类型来表明我们将要使用的表示形式，所需四个常量的 `get` 方法以及一个从字符串中提取相关形式的函数。

有了这些，就可以给出具体实现了：

```
struct UTF8ViewSelector: StringViewSelector {
    static var caret: UInt8 { return UInt8(ascii: "^\u0000") }
    static var asterisk: UInt8 { return UInt8(ascii: "*\u0000") }
    static var period: UInt8 { return UInt8(ascii: ".\u0000") }
    static var dollar: UInt8 { return UInt8(ascii: "$\u0000") }
```

```

static func view(from s: String) -> String.UTF8View { return s.utf8 }

}

struct CharacterViewSelector: StringViewSelector {
    static var caret: Character { return "^" }
    static var asterisk: Character { return "*" }
    static var period: Character { return "." }
    static var dollar: Character { return "$" }

    static func view(from s: String) -> String.CharacterView { return s.characters }
}

```

你大概能猜到 UTF16ViewSelector 跟 UnicodeScalarViewSelector 长什么样子。

这就是一些人称作“幻影 (phantom) 类型”的东西。这种类型只在编译时存在，并且不存储任何数据。尝试调用 `MemoryLayout<CharacterViewSelector>.size` 会返回零。里面没有任何数据。我们使用这些幻影类型就是为了将正则匹配器的行为进行参数化。用法如下：

```

struct Regex<V: StringViewSelector>
    where V.View.Iterator.Element: Equatable,
        V.View.SubSequence == V.View
{
    let regexp: String
    /// 从正则表达式字符串中构建
    init(_ regexp: String) {
        self.regexp = regexp
    }
}

extension Regex {
    /// 当表达式匹配字符串时返回 true
    func match(text: String) -> Bool {
        let text = V.view(from: text)
        let regexp = V.view(from: self.regexp)

        // 如果正则以 ^ 开头，它只从开头进行匹配
        if regexp.first == V.caret {
            return Regex.matchHere(regexp: regexp.dropFirst(), text: text)
        }
    }
}

```

```
// 否则，在输入内逐位搜索匹配，直到找到匹配内容
var idx = text.startIndex
while true {
    if Regex.matchHere(regex: regexp, text: text.suffix(from: idx)) {
        return true
    }
    guard idx != text.endIndex else { break }
    text.formIndex(after: &idx)
}
return false
}
```

```
/// 从文本开头匹配正则表达式字符串
fileprivate static func matchHere(regexp: V.View, text: V.View) -> Bool {
    // ...
}
```

将代码照此重写之后，写出基准测试代码以衡量在非常庞大的输入中匹配正则的时间就很容易了：

```
func benchmark<V: StringViewSelector>(_: V.Type)
where V.View.Iterator.Element: Equatable, V.View.SubSequence == V.View
{
    let r = Regex<V>("h..a*")
    var count = 0

    let startTime = CFAbsoluteTimeGetCurrent()
    while let line = readLine() {
        if r.match(text: line) { count = count &+ 1 }
    }
    let totalTime = CFAbsoluteTimeGetCurrent() - startTime
    print("\(V.self): \(totalTime) s")
}

func ~=<T: Equatable>(lhs: T, rhs: T?) -> Bool {
```

```

    return lhs == rhs
}

switch CommandLine.arguments.last {
case "ch": benchmark(CharacterViewSelector.self)
case "8": benchmark(UTF8ViewSelector.self)
case "16": benchmark(UTF16ViewSelector.self)
case "sc": benchmark(UnicodeScalarViewSelector.self)
default: print("unrecognized view type")
}

```

结果显示不同的表示方式在处理同一份大型英语文本语料库 (128,000行，一百万个单词) 时的速度：

表示方式	时间
UTF16	0.3 秒
UnicodeScalar	0.3 秒
UTF8	1.4 秒
Characters	4.2 秒

只有你自己能知道基于性能来选用表示形式对于你的场景是否合理。几乎可以确定，这些性能特征只在做非常繁重的字符串操作时才有影响。但是，如果你能确保自己所做的操作可以正确处理 UTF-16 的数据，那选用 UTF-16 表示方式将会给你带来相当不错的性能提升。

展望

当 Chris Lattner 在 2016 年 7 月展望 [Swift 4](#) 的目标时，加强字符串处理是被列在了主要目标之中的：

String 是语言中最重要的基础类型之一。标准库的牵头人在字符串的程序模型设计上有不少好点子，这些想法将在保持 Unicode 正确的前提下改进字符串的使用方式。我们的目标是在字符串处理上做得比 Perl 更好！

Swift 团队还在很多场合表达了为正则表达式提供自然语言支持的想法，不过这种附加功能需要看 Swift 4 的时间线才能决定要不要加入。不论具体情况如何，在未来字符串的处理方式肯定还会发生变化。

错误处理

8

Swift 提供了很多种处理错误的方式，它甚至允许我们创建自己的错误处理机制。在可选值中，我们看到过可选值和断言 (assertions) 的方法。可选值意味着一个值可能存在，也可能不存在。我们在实际使用这个值之前，必须先对其确认并解包。断言会验证条件是否为 true，如果条件不满足的话，程序将会崩溃。

如果我们仔细看看标准库中类型的接口的话，我们可以得到一个何时应该使用可选值，而何时不应该使用的大概印象。可选值被广泛用来代表那些可以清楚地表明“不存在”或者“无效输入”的情况。比如说，你在使用一个字符串初始化 Int 时的初始化方法就是可失败的，如果输入不是有效的整数数字字符串，结果就将是 nil。另一个例子是当你在字典里查找一个键时，很多时候这个键并不存在于字典中。因此，字典的查找返回的是一个可选值结果。

对比数组，当通过一个指定的索引获取数组元素时，Swift 会直接返回这个元素，而不是一个包装后的可选值。这是因为一般来说程序员都应该知道某个数组索引是否有效。通过一个超出边界的索引值来访问数组通常被认为是程序员的错误，而这也会让你的应用崩溃。如果你不确定一个索引是否在某个范围内，你应该先对它进行检查。

断言是定位你代码中的 bug 的很好的工具。使用得当的话，它可以在你的程序偏离预订状态的时候尽早对你作出提醒。它们不应该被用来标记像是网络错误那样的预期中的错误。

注意数组其实也有返回可选值的访问方式。比如 Collection 的 first 和 last 属性就将在集合为空的时候返回 nil。Swift 标准库的开发者是有意进行这样的设计的，因为当集合可能为空时还需要访问这些值的情况还是比较容易出现的。

除了从方法中返回一个可选值以外，我们还可以通过将函数标记为 throws 来表示可能会出现失败的情况。除了调用者必须处理成功和失败的情况的语法以外，和可选值相比，能抛出异常的方法的主要区别在于，它可以给出一个包含所发生的错误的详细信息的值。

这个区别决定了我们要使用哪种方法来表示错误。回顾下 Collection 的 first 和 last，它们只可能有一种错误的情况，那就是集合为空时。返回一个包含很多信息的错误并不会让调用者获得更多的情报，因为错误的原因已经在可选值中表现了。对比执行网络请求的函数，情况就不一样了。在网络请求中，有很多事情可能会发生错误，比如当前没有网络连接，或者无法解析服务器的返回等等。带有信息的错误在这种情况下就对调用者非常有用了，它们可以根据错误的不同来采取不同的对应方法，或者可以提示用户到底哪里发生了问题。

Result 类型

在继续深入 Swift 内建的错误处理之前，我们想先讨论下 Result 类型，这将帮助我们理解 Swift 的错误处理机制在去掉语法糖的包装之后，到底是如何工作的。Result 类型和可选值非

常相似。可选值其实就是有两个成员的枚举：一个不包含关联值的 `.none` 或者 `nil`，以及一个包含关联值的 `some`。Result 类型也是两个成员组成的枚举：一个代表失败的情况，并关联了具体的错误值；另一个代表成功的情况，它也关联了一个值。和可选值类似，Result 也有一个泛型参数：

```
enum Result<A> {
    case failure(Error)
    case success(A)
}
```

失败的情形的可选值被限定在 Error 协议上。我们会马上回头讨论这一点。

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，我们想要返回一个 `nil`：

```
func contentsOrNil(ofFile filename: String) -> String?
```

上面的接口非常简单，但是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？在这里，告诉调用者失败的原因是有必要的。现在，让我们定义一个 `enum` 来表明可能出现的错误的情况：

```
enum FileError: Error {
    case fileDoesNotExist
    case noPermission
}
```

我们可以改变函数的类型，让它要么返回一个错误，要么返回一个有效值：

```
func contents(ofFile filename: String) -> Result<String>
```

现在，函数的调用者可以对结果情况进行判断，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，当读取成功时，我们将内容打印出来。要是文件不存在的话，我们输出一条空文件的信息，对于其他错误的话，我们将使用另外的处理方式。

```
let result = contents(ofFile: "input.txt")
switch result {
    case let .success(contents):
        print(contents)
    case let .failure(error):
```

```
if let fileError = error as? FileError,  
    fileError == .fileDoesNotExist  
{  
    print("File not found")  
} else {  
    // 处理错误  
}  
}
```

抛出和捕获

Swift 内建的错误处理的实现方式和这很类似，只不过使用了不同的语法。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`。注意 `Result` 是作用于类型上的，而 `throws` 作用于函数。我们会在后面的章节再提到这个问题。对于每个可以抛出的函数，编译器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的调用者。对于 `contents(ofFile:)` 的情况，包含 `throws` 的时候函数是这样的：

```
func contents(ofFile filename: String) throws -> String
```

现在，我们需要将所有对 `contents(ofFile:)` 的调用标记为 `try`，否则代码将无法编译。关键字 `try` 的目的有两个：首先，对于编译器来说这是一个信号，表示我们知道我们将要调用的函数可能抛出错误。更重要的是，它让代码的读者知道代码中哪个函数可能会抛出。

通过调用一个可抛出的函数，编译器迫使我们去考虑如何处理可能的错误。我们可以选择使用 `do/catch` 来处理错误，或者把当前函数也标记为 `throws`，将错误传递给调用栈上层的调用者。如果使用 `catch` 的话，我们可以用模式匹配的方式来捕获某个特定的错误或者所有错误。在下面的例子中，我们显式地捕获了 `fileDoesNotExist` 的情况对它单独处理，然后在最后的 `catch-all` 语句中处理其他所有错误。在 `catch-all` 里，变量 `error` 是可以直接使用的 (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)  
    // 处理其他错误
```

```
}
```

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。

如果我们想要在错误中给出更多的信息，我们可以使用带有关联值的枚举。(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抛出函数作为错误抛出。)举个例子，如果我们想写一个文件解析器，我们可以用下面的枚举来代表可能的错误：

```
enum ParseError: Error {
    case wrongEncoding
    case warning(line: Int, message: String)
}
```

现在，如果我们要解析一个文件，我们可以用模式匹配来区分这些情况。在 .warning 的时候，我们可以将错误中的行号和警告信息绑定到一个变量上：

```
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 {
}
```

上面的代码中有一个问题。就算我们知道唯一可能出现的错误类型是 ParseError，并且处理了其中所有的情况，我们还是需要写出最后的 catch 块，来让编译器确信我们已经处理了所有可能的错误情况。在未来的 Swift 版本中，编译器可能可以为我们检测在同一个模块中的错误类型是否已经全部处理。不过，对于跨模块的情况，这个问题还是无法解决。究其原因，是由于 Swift 的错误抛出其实是无类型的：我们只能够将一个函数标记为 throws，但是我们并不能指定应该抛出哪个类型的错误。这是一个有意的设计，在大多数时候，你只关心有没有错误抛出。如果我们需要指定所有错误的类型，事情可能很快就会失控：它将使函数类型的签名变得特别复杂，特别是当函数调用其他的可抛出函数，并且将它们的错误向上传递的时候，这个问题将

尤为严重。另外，添加一个错误类型，可能对使用这个 API 的所有客户端来说都是一个破坏性的 API 改动。

在以后的版本中，Swift 可能会支持带类型的错误；关于这个，在邮件列表中大家做了积极的讨论。当支持类型的错误被加入 Swift 的时候，它可能会是一个可选特性。你可以指定你的函数可能抛出的具体错误的类型，但是这并不是必须的。

因为现在 Swift 中的错误是无类型的，所以通过文档来说明你的函数会抛出怎样的错误是非常重要的。Xcode 支持在文档中使用 Throw 关键字来强调这个目的，下面是一个例子：

```
/// 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: `NSError` if the file does not exist or  
///           the process doesn't have read permissions.  
func contents(ofFile filename: String) throws -> String
```

在你按住 Option 并单击这个函数名时，Xcode 弹出的快速帮助将会包含关于抛出错误的额外的信息。

带有类型的错误

不过，有时候我们还是会想通过类型系统来指定一个函数可能抛出的错误类型。如果我们在意这个的话，可以使用稍微修改后的 **Result** 类型来达成目的，只需要将错误的类型也指定为泛型就可以了：

```
enum Result<A, ErrorType: Error> {  
    case failure(ErrorType)  
    case success(A)  
}
```

通过这种方式，我们可以为函数定义一个显式的错误类型。接下来定义的 **parseFile** 要么返回一个字符串数组，要么返回一个 **ParseError**。我们就不必再处理其他的情况，而且编译器也将知道这一事实：

```
func parse(text: String) -> Result<[String], ParseError>
```

当你的代码中的错误有着很重要的意义的时候，你可以选择使用这种带有类型的 `Result` 来取代 Swift 内建的错误处理。这样一来，编译器就能够验证你是否处理了所有可能的错误。不过，对于大多数应用程序来说，使用 `throws` 和 `do/try/catch` 可以让代码更简单。使用内建的错误处理还有一个好处，那就是编译器会确保你在调用一个可能抛出异常的函数时没有忽略那些错误。如果使用上面的 `parseFile`，我们可能会写出这种代码：

```
let _ = parse(text: invalidData)
```

要是函数是被标记为 `throw` 的，编译器就会强制我们使用 `try` 来调用它。编译器还会强制我们要么将这个调用包装在一个 `do/catch` 代码块中，要么将这个错误传递给上层调用。

老实说，上面的例子是不可能实际出现的，因为忽略了 `parse` 函数的返回值的话，这个调用就毫无意义了。而且，编译器将会在解包结果的时候强迫我们去考虑失败的情况。这一点在你调用的是那些不会返回普通值的函数确实会很有意义，它将提醒你不要忘记处理错误。比如，下面这个函数：

```
func setupServerConnection() throws
```

因为这个函数被标记为 `throw`，我们在调用它时必须要加上 `try`。要是连接服务器失败了，我们可能会想要转换到另外一条代码路径上，或者是显示一个错误。通过一定要使用 `try`，我们被强制思考失败的情况。然而，如果我们选择返回一个 `Result<()>` 的话，它很可能就会由于不小心而被忽略掉。

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 的异常)

Cocoa 的通用做法是在发生错误时返回 `NO` 或者 `nil`。在此之上，我们会将一个错误对象的指针作为参数传递进方法。这个方法通过该变量将关于错误的信息回传给调用者。举个例子，如果 `contentsOfFile:` 是 Objective-C 写的话，它看起来会是这样：

```
- (NSString *)contentsOfFile:(NSString *)filename error:(NSError **)error;
```

Swift 会自动将遵循这个规则的方法转换为 `throws` 语法的版本。因为不再需要错误参数了，所以它被移除了。这个转换对处理那些既存的 Objective-C 框架很有帮助。上面的函数被导入到 Swift 后会变成这样：

```
func contents(ofFile filename: String) throws -> String
```

其他的 `NSError` 参数，比如在异步 API 调用时 `completion` 回调中回传给调用者的错误，将被导入为 `Error` 协议，所以一般来说你不再需要直接和 `NSError` 打交道了。`Error` 只有一个属性，那就是 `localizedDescription`。对纯 Swift 的错误，这个属性没有被重写，运行的时候将从错误类型名中生成一个默认的文本描述。如果你想要将这些错误值展示给用户的话，通过重写这个属性来提供有意义的描述会是更好的实践。

如果你将一个纯 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`，运行时将提供默认的值，它会使用类型名作为 `domain` 名字，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` 协议来提供更好的实现。

比如，我们可以扩展 `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，你可以实现下面的协议，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。

错误和函数参数

在接下来的例子中，我们将创建一个用来检查一系列文件有效性的函数。检查单个文件的 `checkFile` 函数有三种可能的返回值。如果返回 `true`，说明该文件是有效的。如果返回 `false`，文件无效。如果它抛出一个错误，则说明在检查文件的过程中出现了问题：

```
func checkFile(filename: String) throws -> Bool
```

作为起始，我们可以用一个简单的循环来确认对列表中的每一个文件，`checkFile` 返回的都是 `true`。如果 `checkFile` 返回了 `false`，那么我们就提前退出，以避免不必要的工作。这里我们不会捕获 `checkFile` 抛出的错误，遇到的第一个错误将会被传递给调用者，并且循环也将提前退出：

```
func checkAllFiles(filenames: [String]) throws -> Bool {
    for filename in filenames {
        guard try checkFile(filename: filename) else { return false }
    }
    return true
}
```

检查一个数组中的所有元素是否都满足某个特定的条件，在我们的应用中是很常见的操作。比如，`checkPrimes` 将检查列表中的所有数字是否是质数。它的工作方式和 `checkAllFiles` 完全一样。它将对数组进行循环，然后检查是否所有的元素都满足条件 (`isPrime`)，一旦有一个数字不是质数的时候，就提前退出：

```
func checkPrimes(_ numbers: [Int]) -> Bool {
    for number in numbers {
        guard number.isPrime else { return false }
    }
    return true
}
```

```
checkPrimes([2,3,7,17]) // true
checkPrimes([2,3,4,5]) // false
```

这两个函数都将对于序列的迭代 (for 循环) 与实际决定一个元素是否满足条件的逻辑进行了混合。对与这种模式，类似于 `map` 或者 `filter`，我们可以为它创建一个抽象。我们可以为 `Sequence` 添加一个 `all` 函数。和 `filter` 一样，`all` 接受一个执行判断并检查条件是否满足的函数

作为参数。与 filter 的不同之处在于返回的类型。当序列中的所有元素都满足条件时，all 函数将返回 true，而 filter 返回的是那些满足条件的元素本身：

```
extension Sequence {  
    /// 当且仅当所有元素满足条件时返回 `true`  
    func all(condition: (Iterator.Element) -> Bool) -> Bool {  
        for element in self {  
            guard condition(element) else { return false }  
        }  
        return true  
    }  
}
```

这让我们可以用一行就搞定 checkPrimes 函数，一旦你知道 all 做的事情以后，这个版本读起来也要简明得多。这有助于我们把注意力集中到核心部分：

```
func checkPrimes2(_ numbers: [Int]) -> Bool {  
    return numbers.all { $0.isPrime }  
}
```

然而，我们还并不能用 all 重写 checkAllFiles，因为 checkFile 是被标记为 throws 的。我们可以很容易地把 all 重写为接受 throws 函数的版本，但是那样一来，我们也需要改变 checkPrimes，要么将它标记为 throws 并且用 try! 来调用，要么将对 all 的调用放到 do/catch 代码块中。我们还有一种做法，那就是定义两个版本的 all 函数：一个接受 throws，另一个不接受。除了 try 调用以外，它们的实现应该是相同的。

Rethrows

好消息是，还有一种更好的方法，那就是将 all 标记为 rethrows。这样一来，我们就可以一次性地对应两个版本了。rethrows 告诉编译器，这个函数只会在它的参数函数抛出错误的时候抛出错误。对那些向函数中传递的是不会抛出错误的 check 函数的调用，编译器可以免除我们一定要使用 try 来进行调用的要求：

```
extension Sequence {  
    func all(condition: (Iterator.Element) throws -> Bool) rethrows  
        -> Bool {  
        for element in self {  
            guard try condition(element) else { return false }  
        }  
    }  
}
```

```
    }
    return true
}
}
```

checkAllFiles 的实现现在和 checkPrimes 很相似了，不过因为 all 现在可以抛出错误，所以我们需要添加一个额外的 try：

```
func checkAllFiles(filenames: [String]) throws -> Bool {
    return try filenames.all(condition: checkFile)
}
```

标准库的序列和集合中几乎所有接受函数作为参数的函数都被标记为了 rethrows。比如 map 函数就被标记为 rethrows，它只在变形函数也为 throws 时才会抛出错误。

使用 defer 进行清理

让我们回到本章开头的 contents(ofFile:) 文件，再来看看它的实现。在很多语言里，都有 try/finally 这样的结构，其中 finally 所围绕的代码块将一定会在函数返回时被执行，而不论最后是否有错误被抛出。Swift 中的 defer 关键字和它的功能类似，但是具体做法稍有不同。和 finally 一样，defer 块会在作用域结束的时候被执行，而不管作用域结束的原因到底是什么。比如离开作用域可以是由于一个值被从函数中成功地正常返回，也可以是发生了一个错误，或者是其他任何原因。defer 与 finally 不一样的地方在于前者不需要在之前出现 try 或是 do 这样的语句，你可以很灵活地把它放在代码中需要的位置：

```
func contents(ofFile filename: String) throws -> String
{
    let file = open("test.txt", O_RDONLY)
    defer { close(file) }
    let contents = try process(file: file)
    return contents
}
```

虽然 defer 经常会被和错误处理一同使用，但是在其他上下文中，这个关键字也很有用处。比如你想将代码的初始化工作和在关闭时对资源的清理工作放在一起时，就可以使用 defer。将代码中相关的部分放到一起可以极大提高你的代码可读性，这在代码比较长的函数中尤为有用。

标准库多次在需要给一个计数器加一并且返回这个计数器之前的值的时候使用了 defer。这让我们可以不用为了返回值而去特意创建一个本地变量。这种情况下 defer 本质上等同于已被移除的后自增运算符 (也就是 `i++`)。下面是从 `EnumeratedIterator` 的实现中截取的典型例子：

```
struct EnumeratedIterator<Base: IteratorProtocol>: IteratorProtocol, Sequence {
    internal var _base: Base
    internal var _count: Int
    ...
    func next() -> Element? {
        guard let b = _base.next() else { return nil }
        defer { _count += 1 }
        return (offset: _count, element: b)
    }
}
```

如果相同的作用域中有多个 defer 块，它们将被按照逆序执行。你可以把这种行为想象为一个栈。一开始，你可能会觉得逆序执行的 defer 很奇怪，不过你可以看看这个例子，你就能很快明白为什么要这样做了：

```
guard let database = openDatabase(...) else { return }
defer { closeDatabase(database) }
guard let connection = openConnection(database) else { return }
defer { closeConnection(connection) }
guard let result = runQuery(connection, ...) else { return }
```

打个比方，如果在 `runQuery` 调用时，发生了错误，我们会想要先断开与数据库的连接，然后再关闭数据库。因为 defer 是逆序执行的，所以这一切就显得非常自然了。`runQuery` 的执行依赖于 `openConnection` 的成功，而 `openConnection` 又依赖于 `openDatabase`。因此，对于资源的清理要按照这些操作发生的逆序来执行。

有一些情况下你的 defer 块可能会没有被调用：比如当你的程序遇到一个段错误 (segfaults) 或者发生了严重错误 (使用了 `fatalError` 或者强制解包一个 `nil`) 时，所有的执行都将立即被挂起。

错误和可选值

错误和可选值都是函数用来表达发生了问题的常见方式。在本章前面的部分，我们给过你一些关于决定你自己的函数应该选取那种方式的建议。最终你可能会两种方式都有采用，并且在标记为 `throws` 和返回可选值两类 API 之间相互地进行转换。

我们可以使用 `try?` 来忽略掉 `throws` 函数返回的错误，并将返回结果转换到一个可选值中，来告诉我们函数调用是否成功：

```
if let result = try? parse(text: input) {
    print(result)
}
```

如果我们使用了 `try?` 关键字，那么其实我们的信息是要比原来少的：我们只知道这个函数返回的是成功值还是一个错误，而关于它所抛出的错误的详细信息，我们就不得而知了。如果我们相反，从一个可选值转换为函数抛出的错误的话，我们就还要额外地为可选值为 `nil` 的情况提供错误值。这里有给出错误的 `Optional` 的扩展：

```
extension Optional {
    /// 如果 `self` 不是 `nil` 的话，解包。
    /// 如果 `self` 是 `nil` 则抛出给定的错误。
    func or(error: Error) throws -> Wrapped {
        switch self {
            case let x?: return x
            case nil: throw error
        }
    }
}

do {
    let int = try Int("42").or(error: ReadIntError.couldNotRead)
} catch {
    print(error)
}
```

在将多个 `try` 语句连结起来使用时，或者是你在一个已经被标记为 `throws` 的函数内部进行编码时，这就会非常有用。

关键字 `try?` 的存在看起来违背了 Swift 中不允许忽略错误的设计原则。不过，你确实还是需要明确写出 `try?`，这让你也无法无意地忽略错误。某些情况下，当你确实对错误信息不感兴趣的时候，这个关键字还是相当有用的。

你也可以写一些转换函数，来在 `Result` 和 `throws` 之间，或者可选值和 `Result` 之间进行等效转换。

错误链

对多个可能抛出错误的函数的链式调用在 Swift 的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。我们只需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) 去。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。

```
func checkFilesAndFetchProcessID(filenames: [String]) -> Int {
    do {
        try filenames.all(condition: checkFile)
        let pidString = try contents(ofFile: "Pidfile")
        return try Int(pidString).or(error: ReadIntError.couldNotRead)
    } catch {
        return 42 // 默认值
    }
}
```

链结果

Swift 原生的错误处理相比其他的错误处理机制有不少优势，我们会将原生处理与一个基于 Result 类型的相同的例子进行对比。将多个返回 Result 且接受前一个函数输出作为输入的函数的调用手动链接起来需要很多努力，你需要调用前一个函数，然后解包它的输出，如果遇到的是 .success，则将值传递给后一个函数并再次开始。一旦你遇到一个返回 .failure，你需要将链打断，然后立即短路后面的操作，将这个失败返回给调用者。

要重构这个行为，我们需要将解包 Result 的步骤通用化，在失败的时候进行短路操作，并在成功的时候将值传递给后一个变形函数。能够完成这一系列的函数就是 flatMap。它的结构与我们在可选值一章中提到过的已经存在的可选值 flatMap 是一致的：

```
extension Result {
    func flatMap<B>(transform: (A) -> Result<B>) -> Result<B> {
        switch self {
            case let .failure(m): return .failure(m)
            case let .success(x): return transform(x)
        }
    }
}
```

有了这个，最后结果看起来就相当优雅了：

```
func checkFilesAndFetchProcessID(filenames: [String]) -> Result<Int> {
    return filenames
        .all(condition: checkFile)
        .flatMap { _ in contents(ofFile: "Pidfile") }
        .flatMap { contents in
            Int(contents).map(Result.success)
            ?? .failure(ReadIntError.couldNotRead)
        }
}
```

(我们这里使用的 `all`, `checkFile` 和 `contents(ofFile:)` 都是返回 `Result` 值的变种版本。它们具体的实现在此没有列出)

即使这样，你也可以看到 Swift 的错误处理机制表现得更好，它比 `Result` 链式调用的代码更短，而且它显然更易读易懂。

高阶函数和错误

在异步 API 调用中有时需要将错误在回调函数中返回给调用者，Swift 的错误处理在这个领域**并不能**很好地适用。让我们来看一个异步计算一个很大的数的例子，当计算结束后，我们的代码会被回调：

```
func compute(callback: (Int) -> ())
```

我们可以通过提供一个回调函数来调用它。回调将通过唯一的参数来接收计算的结果：

```
compute { result in
    print(result)
}
```

如果计算可能失败的话，我们就需要把回调改为接受一个可选值整数的情况，在失败的情况下，这个值将会是 `nil`：

```
func computeOptional(callback: (Int?) -> ())
```

现在，在我们的回调中我们需要检查可选值是不是非 nil 的值，比如说我们可以用 ?? 操作符来指定默认值：

```
computeOptional { result in
    print(result ?? -1)
}
```

但是如果我们想要的不单单是一个可选值，而想要关于这个回调错误的详细信息的话，该怎么办呢？这种写法可能看起来像是很自然的解决方式：

```
func computeThrows(callback: Int throws -> ())
```

但是可能出乎你的意料，这个签名代表的意义完全不同。它不是指计算可能失败，而是表示回调本身可能会抛出错误。这强调了我们之前提到的表达错误时的关键区别：可选值和 Result 作用于类型，而 throws 只对函数类型起效。将一个函数标注为 throws 意味着这个函数可能会失败。

当我们使用 Result 来重写上面这个错误的尝试时，这个区别带来的问题就会比较明显了：

```
func computeResult(callback: Int -> Result<()>)
```

我们真正想要的是用一个 Result 来封装 Int 参数，而不是去封装回调的返回类型。最后，正确的解决方式是：

```
func computeResult(callback: (Result<Int>) -> ())
```

不过坏消息是，现在没有办法可以很清晰地用 throws 来表达上面的含义。我们能够做的只有将 Int 封装到另一个可以抛出的函数中去，不过这样会把类型变得更加复杂：

```
func compute(callback: ((() throws -> Int) -> ())
```

这样的变形也使调用者更加复杂。为了将整数值取出，现在在回调中也需要调用这个可抛出函数。要注意，这个可抛出函数不但能返回整数值，也可能会抛出错误，你必须对此再进行处理：

```
compute { (resultFunc: () throws -> Int) in
    do {
        let result = try resultFunc()
        print(result)
    } catch {
```

```
    print("An error occurred: \(error)")
}
}
```

这样的代码可以工作，但是显然它肯定不是 Swift 中的惯用做法。Result 是异步错误处理的正确道路。不好的地方在于，如果你已经在同步函数中使用 throws 了，再在异步函数中转为使用 Result 将会在两种接口之间导入差异。Swift 团队表达过想要将 throws 模型延伸到其他场合的意愿，但是这看起来会是为这门语言添加原生的并行特性这一更大任务的一个部分，而这在 Swift 4 里是不会发生的。

在原生并行特性被加入之前，我们会一直使用自定义的 Result 类型。Apple 也考虑过将 Result 类型加入标准库中，但是最后 Swift 团队认为在离开错误处理的领域后，这个类型并没有特别独立的价值，而且它们也不想鼓励除了 throws 以外的另一种错误处理方式，所以最终这个提案被否决了。不过，在异步 API 中使用 Result 已经在 Swift 开发者社区中建立起了良好共识，所以如果你的 API 中需要错误处理，而原生的错误处理不合适的时候，你应该积极使用 Result。它肯定要比 Objective-C 中 completion 回调里包含两个可为 nil 的参数（一个代表结果对象，一个代表错误对象）要好。

总结

当 Apple 在 Swift 2.0 中引入错误处理机制时，大家都非常吃惊。在 Swift 1.x 的年代，人们就已经使用他们自己的 Result 类型了，并用它来指定错误的类型。throws 使用无类型错误这一事实曾经被看作是偏离了这门语言其他部分严格的类型系统的要求。不过毫无意外，Swift 团队是经过深思熟虑并有意使用无类型错误的。虽然我们有所怀疑，但是事后来看，我们认为 Swift 团队作出了正确的选择，至少在开发者社区中这套错误处理的模型被广泛接受了。

添加带有类型的错误处理很可能被作为可选特性在未来被加入，同时我们也可以期待对于异步错误和将错误作为值进行传递的更好的支持。现在来看，错误处理是 Swift 作为一门实用语言的好例子，开发团队选择首先针对最常用的情况进行优化。保持使用开发者们已经熟知的基于 C 的语法风格，比引入基于 Result 和 flatMap 的函数式处理方式更加重要。

现在，我们在代码中遇到意外情况时有很多选择了。当我们不能继续运行代码时，可以选择使用 fatalError 或者是断言。当我们对错误类型不感兴趣，或者只有一种错误时，我们使用可选值。当我们需要处理多种错误，或是想要提供额外的信息时，可以使用 Swift 内建的错误，或者是自定义一个 Result 类型。当我们想要写一个接受函数的函数时，我们可以使用 rethrows 来让这个待写函数同时接受可抛出和不可抛出的函数参数。最后，defer 语句在结合内建的错误处理时非常有用。defer 语句为我们提供了集中放置清理代码的地方，不论是正常退出，还是由于错误而被中断，defer 语句所定义的代码块都将被执行并进行清理。

泛型

9

和大多数先进语言一样，Swift 拥有不少能被归类于**泛型编程**下的特性。使用泛型代码，你可以写出可重用的函数和数据结构，只要它们满足你所定义的约束，它们就能够适用于各种类型。比如，像是 Array 和 Set 等多个类型，实际上是它们中的元素类型就是泛型抽象。我们也可以创建泛型方法，它们可以对输入或者输出的类型进行泛型处理。func identity<A>(input: A) -> A 就定义了一个可以作用于任意类型 A 的函数。某种意义上，我们甚至可以认为带有关联类型的协议是“泛型协议”。关联类型允许我们对特定的实现进行抽象。IteratorProtocol 协议就是一个这样的例子：它所生成的 Element 就是一个泛型。

泛型编程的目的是表达算法或者数据结构所要求的**核心接口**。比如，考虑内建集合一章中的 last(where:) 函数。将它写为 Array 的一个扩展原本是最明显的选择，但是 Array 其实包含了很 last(where:) 并不需要的特性。通过确认核心接口到底是什么，也就是说，找到想要实现的功能的最小需求，我们可以将这个函数定义在宽阔得多的类型范围内。在这个例子中，last(where:) 只有一个需求：它需要能够逆序遍历一系列元素。所以，将这个算法定义为 Sequence 的扩展是更好的选择。

在本章中，我们会研究如何书写泛型代码。我们会先看一看什么是重载 (overloading)，因为这个概念和泛型紧密相关。然后我们会使用泛型的方式，基于不同的假设，来为一个算法提供多种实现。之后我们将讨论一些你在为集合书写泛型算法时会遇到的常见问题，了解这些问题后你就将能使用泛型数据类型来重构代码，并使它们易于测试，更加灵活。最后，我们会谈一谈编译器是如何处理泛型代码的，以及要如何优化我们的泛型代码以获取更高性能的问题。

重载

拥有同样名字，但是参数或返回类型不同的多个方法互相称为重载方法，方法的重载并不意味着泛型。不过和泛型类似，我们可以将多种类型使用在同一个接口上。

自由函数的重载

我们可以定义一个名字为 raise(_:_to:) 的函数，它可以通过针对 Double 和 Float 参数的不同重载来分别执行幂运算操作。基于类型，编译器将挑选合适的重载方法进行调用：

```
func raise(_ base: Double, to exponent: Double) -> Double {  
    return pow(base, exponent)  
}  
  
func raise(_ base: Float, to exponent: Float) -> Float {
```

```
    return powf(base, exponent)
}

let double = raise(2.0, to: 3.0) // 8.0
type(of: double) // Double
let float: Float = raise(2.0, to: 3.0) // 8.0
type(of: float) // Float
```

我们使用了 Swift 的 Darwin 模块 (或者 Linux 下的话是 Glibc 模块) 中的 pow 和 powf 函数来进行实现。

Swift 有一系列的复杂规则来确定到底使用哪个重载函数，这套规则基于函数是否是泛型，以及传入的参数是怎样的类型来确定使用优先级。整套规则十分复杂，不过它们可以被总结为一句话，那就是“选择最具体的一个”。也就是说，非通用的函数会优先于通用函数被使用。

举个例子，比如我们有一个函数来输出一个视图对象的某个属性。我们可以为 UIView 提供一个泛用的实现，用来打印视图的类名和 frame 属性，同时我们可以为 UILabel 提供一个重载，来打印 label 的文本：

```
func log<View: UIView>(_ view: View) {
    print("It's a \(type(of: view)), frame: \(view.frame)")
}

func log(_ view: UILabel) {
    let text = view.text ?? "(empty)"
    print("It's a label, text: \(text)")
}
```

传入 UILabel 将会调用专门针对 label 的重载，而传入其他的视图将会调用到泛型函数：

```
let label = UILabel(frame: CGRect(x: 20, y: 20, width: 200, height: 32))
label.text = "Password"
log(label) // It's a label, text: Password

let button = UIButton(frame: CGRect(x: 0, y: 0, width: 100, height: 50))
log(button) // It's a UIButton, frame: (0.0, 0.0, 100.0, 50.0)
```

要特别注意，重载的使用是在编译期间静态决定的。也就是说，编译器会依据变量的静态类型来决定要调用哪一个重载，而不是在运行时根据值的动态类型来决定。我们如果将上面的 label

和 button 都放到一个 UIView 数组中，并对它们迭代并调用 log 的话，使用的都是泛型重载的版本：

```
let views = [label, button] // views 的类型是 [UIView]
for view in views {
    log(view)
}
/*
It's a UILabel, frame: (20.0, 20.0, 200.0, 32.0)
It's a UIButton, frame: (0.0, 0.0, 100.0, 50.0)
*/
```

这是因为 view 的静态类型是 UIView，UILabel 本来应该使用更专门的另一个重载，但是因为重载并不会考虑运行时的动态类型，所以两者都使用了 UIView 的泛型重载。

如果你需要的是运行时的多态，也就是说，你想让函数基于变量实际指向的内容决定使用哪个函数，而不考虑变量本身是什么的话，你应该使用定义在类型上的方法，而不是自由函数。比如你可以将 log 定义到 UIView 和 UILabel 上去。

运算符的重载

当使用操作符重载时，编译器会表现出一些奇怪的行为。Matt Gallagher [指出](#)，即使泛型版本应该是更好的选择（而且应该在一个普通函数调用时被选择）的时候，类型检查器也还是会去选择那些非泛型的重载，而不去选择泛型重载。

让我们回到上面的幂运算的例子，我们可以为这个操作定义一个运算符 **：

```
// 幂运算比乘法运算优先级更高
precedencegroup ExponentiationPrecedence {
    associativity: left
    higherThan: MultiplicationPrecedence
}
infix operator **: ExponentiationPrecedence

func **(lhs: Double, rhs: Double) -> Double {
    return pow(lhs, rhs)
}
func **(lhs: Float, rhs: Float) -> Float {
```

```
    return powf(lhs, rhs)
}
```

```
2.0 ** 3.0 // 8.0
```

上面的代码和我们在前面一节中的 `raise` 函数是等价的。现在让我们为它加上一个整数的重载。我们希望这个幂运算对所有的整数都生效，所以我们对所有满足 `SignedInteger` 的类型定义了一个泛型的重载（我们其实还需要对 `UnsignedInteger` 定义一个重载，这里没有列出）：

```
func **<I: SignedInteger>(lhs: I, rhs: I) -> I {
    // 转换为 IntMax, 使用 Double 的重载计算结果,
    // 然后用 numericCast 转回原类型
    let result = Double(lhs.toIntMax()) ** Double(rhs.toIntMax())
    return numericCast(IntMax(result))
}
```

看起来似乎能工作，但是如果我们将整数字面量来调用 `**`，编译器会报错说 `**` 运算符的使用存在歧义：

```
2 ** 3 // Error: Ambiguous use of operator '**'
```

要解释原因，我们需要回到我们在本节一开始说道的：对于重载的运算符，类型检查器会去使用非泛型版本，而不考虑泛型版本。显然，编译器忽略了整数的泛型重载，因此它无法确定是去调用 `Double` 的重载还是 `Float` 的重载，因为两者对于整数字面量输入来说，是相同优先级的可选项（Swift 编译器会将整数字面量在需要时自动向上转换为 `Double` 或者 `Float`），所以编译器报错说存在歧义。要让编译器选择正确的重载，我们需要至少将一个参数显式地声明为整数类型，或者明确提供返回值的类型：

```
let intResult: Int = 2 ** 3 // 8
```

这种编译器行为只对运算符生效。`SignedInteger` 泛型重载的 `raise` 函数可以不加干预地正确工作。导致这种差异的原因是性能上的考量：Swift 团队选择了一种相对简单但是有时候会无法正确处理重载模型的类型检查器，来在保证绝大多数使用情景的前提下，降低类型检查器的复杂度。

使用泛型约束进行重载

当你在写一些可以被用多种算法表达的同样的操作，并且算法对它们的泛型参数又有不同的要求的代码的时候，你可能经常会遇到带有泛型代码的重载。假设我们要写一个算法，来确定一个数组中的所有元素是不是都被包含在另一个数组中。换句话说，我们想要知道第一个数组是不是第二个数组的子集(这里元素的顺序不重要)。标准库中提供了一个叫做 `isSubset(of:)` 的方法，不过这个方法只适用于像 `Set` 这样满足 `SetAlgebra` 协议的类型。

我们可以写一个适用于更宽广范围的 `isSubset(of:)`，它看起来可能是这样的：

```
extension Sequence where Iterator.Element: Equatable {
    /// 如果 `self` 中的所有元素都包含在 `other` 中，则返回 true
    func isSubset(of other: [Iterator.Element]) -> Bool {
        for element in self {
            guard other.contains(element) else {
                return false
            }
        }
        return true
    }
}

let oneToThree = [1,2,3]
let fiveToOne = [5,4,3,2,1]
oneToThree.isSubset(of: fiveToOne) // true
```

`isSubset` 是定义在 `Sequence` 协议上的扩展方法，它要求序列中的元素满足 `Equatable`。它接受一个同样类型的数组，并进行检查，如果本数组内的每一个元素都是参数数组中的成员的话，则返回 `true`。这个方法只要求其中元素满足 `Equatable`(只有满足该协议的元素可以使用 `contains`)，而不关心这些元素到底是什么类型。数组中的元素可以是 `Int`，可以是 `String`，也可以是任何你自定义的类型，唯一的要求是它们可以被判等。

这个 `isSubset` 的版本有一个重大缺陷，那就是性能。这里的算法的时间复杂度是 $O(n * m)$ ，其中 `n` 和 `m` 分别代表两个数组的元素个数。也就是说，随着输入的增多，这个函数的最坏情况的耗时将成平方增加。这是因为 `contains` 在数组中的复杂度是线性的 $O(m)$ ，这个函数会迭代源序列中的元素，逐个检查它是否匹配给定的元素。而 `contains` 是在另一个迭代最初数组的元素的循环中被调用了，这个循环也很类似，是一个线性时间复杂度的循环。所以我们是在一个 $O(n)$ 循环里执行了一个 $O(m)$ 的循环，结果这个函数的复杂度就是 $O(n * m)$ 。

对于小输入来说还好，如果你只是调用在包含几百个元素的数组上调用这个方法的话，应该问题不大。但是如果数组中有上万甚至上百万的数据的话，那就只能为你默哀了。

我们可以通过收紧序列元素类型的限制来写出性能更好的版本。如果我们要求元素满足 `Hashable`，那么我们就可以将 `other` 数组转换为一个 `Set`，这样查找操作就可以在常数时间内进行了：

```
extension Sequence where Iterator.Element: Hashable {
    /// 如果 `self` 中的所有元素都包含在 `other` 中，则返回 true
    func isSubset(of other: [Iterator.Element]) -> Bool {
        let otherSet = Set(other)
        for element in self {
            guard otherSet.contains(element) else {
                return false
            }
        }
        return true
    }
}
```

现在 `contains` 的检查只会花费 $O(1)$ 的时间（假设哈希值是平均分布的话），整个 `for` 循环的复杂度就可以降低到 $O(n)$ 了，也就是随着接受方法调用的数组的尺寸增长，耗费的时间将线性增长。将 `other` 转换为集合的额外消耗是 $O(m)$ ，不过它是在循环之外的，只会被执行一次。所以操作的总消耗为 $O(n + m)$ ，这远远好于 `Equatable` 版本的 $O(n * m)$ 。如果两个数组都含有 1000 个元素的话，这就是 2000 和一百万次迭代的区别。

于是现在我们有两个版本的算法了，不过两者比较并没有哪个完全比另一个好：一个版本比较快，另一个版本可以针对更多的类型。好消息是你不需要去挑选使用哪一个，你可以同时实现这两个 `isSubset` 的重载，编译器将会根据参数的类型为你挑选最合适的来使用。Swift 在重载上非常灵活，你不仅可以通过输入类型或者返回类型来重载，你也可以通过泛型占位符的不同约束来重载，就像我们在这个例子中所做的一样。

类型检查器会使用它所能找到的最精确的重载。这里 `isSubset` 的两个版本都是泛型函数，所以非泛型函数先于泛型函数的规则并不适用。不过因为 `Hashable` 是对 `Equatable` 的扩展，所以要求 `Hashable` 的版本更加精确。有了这些约束，我们可能可以像例子中的 `isSubset` 这样写出更加高效的算法，所以更加精确的函数通常都会是更好的选择。

`isSubset` 还可以更加通用，到现在位置，它只接受一个数组并对其检查。但是 `Array` 是一个具体的类型。实际上 `isSubset` 并不需要这么具体，在两个版本中只有两个函数调用，那就是两者

中都有的 `contains` 以及 `Hashable` 版本中的 `Set.init`。这两种情况下，这些函数只要求输入类型满足 `Sequence` 协议：

```
extension Sequence where Iterator.Element: Equatable {
    /// 根据序列是否包含给定元素返回一个布尔值。
    func contains(_ element: Iterator.Element) -> Bool
}

struct Set<Element: Hashable>:
    SetAlgebra, Hashable, Collection, ExpressibleByArrayLiteral
{
    /// 通过一个有限序列创建新的集合。
    init<Source: Sequence>(_ sequence: Source)
        where Source.Iterator.Element == Element
}
```

所以，`isSubset` 中 `other` 只需要是遵守 `Sequence` 的任意类型就可以了。另外，`self` 和 `other` 这两个序列类型并不需要是同样的类型。我们只需要其中的元素类型相同就能进行操作。下面是针对任意两种序列重写的 `Hashable` 版本的函数：

```
extension Sequence where Iterator.Element: Hashable {
    /// 如果 `self` 中的所有元素都包含在 `other` 中，则返回 true
    func isSubset<S: Sequence>(of other: S) -> Bool
        where S.Iterator.Element == Iterator.Element
    {
        let otherSet = Set(other)
        for element in self {
            guard otherSet.contains(element) else {
                return false
            }
        }
        return true
    }
}
```

现在两个序列不需要有相同的类型了，这为我们开启了更多的可能性。比如，你可以传入一个数字的 `CountableRange` 来进行检查：

```
[5,4,3].isSubset(of: 1...10) // true
```

我们可以对可判等的元素的函数作出同样的更改：

```
extension Sequence where Iterator.Element: Equatable {
    func isSubset<S: Sequence>(of other: S) -> Bool
        where S.Iterator.Element == Iterator.Element
    {
        for element in self {
            guard other.contains(element) else {
                return false
            }
        }
        return true
    }
}
```

使用闭包对行为进行参数化

`isSubset` 还有更加通用化的可能。对于那些元素不满足 `Equatable` 的序列要怎么办？比如，数组就不是 `Equatable` 的，如果我们将数组作为别的数组的元素，那我们就不能继续使用现在的实现了。诚然，数组类型确实有一个这样定义的 `=` 操作符：

```
/// 如果两个数组包含相同的元素，返回 true
func ==<Element: Equatable>(lhs: [Element], rhs: [Element]) -> Bool
```

但是这并不意味着你可以将 `isSubset` 用在元素是数组类型的序列上：

```
// 错误：表达式类型冲突，需要更多上下文
[[1,2]].isSubset(of: [[1,2], [3,4]])
```

这是因为 `Array` 并不遵守 `Equatable`。`Array` 确实不能遵守 `Equatable`，因为它所包含的元素类型可能本身就不是可以判等的。Swift 现在不支持**按条件满足协议**，也就是说，我们不能表达一个 `Array`（或者任意 `Sequence`）只在满足某些特定约束（比如 `Iterator.Element: Equatable`）的情况下才满足一个协议。所以虽然 `Array` 可以为那些其中元素类型可判等的数组提供 `=` 的实现，但是它**并不能**满足这个协议。

那么，要怎么才能让 `isSubset` 对不是 `Equatable` 的类型也适用呢？我们可以要求调用者提供一个函数来表明元素相等的意义，这样一来，我们就把判定两个元素相等的控制权交给了调用者。比如标准库中提供的另一个版本的 `contains` 就是这么做的：

```
extension Sequence {
    /// 根据序列是否包含满足给定断言的元素，返回一个布尔值。
    func contains(where predicate: (Iterator.Element) throws -> Bool)
        rethrows -> Bool
}
```

也就是说，它接受一个函数，这个函数从序列中取出一个元素，并对它进行一些检查。它会对每个元素进行检查，并且在检查结果为 true 的时候，它也返回 true。这个版本的 contains 要强大得多。比如，你可以用它来对一个序列进行各种条件的检查：

```
let isEven = { $0 % 2 == 0 }
(0..<5).contains(where: isEven) // true
[1, 3, 99].contains(where: isEven) // false
```

我们可以利用这个更灵活的 contains 版本来写一个同样灵活的 isSubset：

```
extension Sequence {
    func isSubset<S: Sequence>(of other: S,
        by areEquivalent: (Iterator.Element, S.Iterator.Element) -> Bool)
        -> Bool
    {
        for element in self {
            guard other.contains(where: { areEquivalent(element, $0) }) else {
                return false
            }
        }
        return true
    }
}
```

现在，我们可以将 isSubset 用在数组的数组上了，只需要为它提供一个闭包表达式，并使用 == 来对数组进行比较：

```
[[1,2]].isSubset(of: [[1,2], [3,4]]) { $0 == $1 } // true
```

只要你提供的闭包能够处理比较操作，这两个序列中的元素甚至都不需要是同样类型的元素：

```
let ints = [1,2]
let strings = ["1", "2", "3"]
```

```
ints.isSubset(of: strings) { String($0) == $1 } // true
```

对集合采用泛型操作

集合上的泛型算法通常会带出一些特殊的问题，特别在与索引和切片一起使用时更是如此。在这节中，我们通过三个依赖于正确处理索引和切片的例子，来看看如何解决这些问题。

二分查找

假设你需要使用一个操作集合的算法，你来到了你最喜欢的[算法参考网站](#)，那边的算法都是用Java写的，而你想要把它们移植到Swift。比如，这里有一个二分查找算法。虽然它是用无聊的循环写的，而没有用递归，不过我们还是可以来看看这个函数：

```
extension Array {
    /// 返回 `value` 第一次出现在 `self` 中的索引值，如果 `value` 不存在，返回 `nil`。
    ///
    /// - 要求: `isOrderedBefore` 是在 `self` 中元素上的严格弱序，且数组中的元素已经按它进行过排序
    /// - 复杂度:  $O(\log \text{count})$ 
    func binarySearch(
        for value: Element, areInIncreasingOrder: (Element, Element) -> Bool)
        -> Int?
    {
        var left = 0
        var right = count - 1

        while left <= right {
            let mid = (left + right) / 2
            let candidate = self[mid]

            if areInIncreasingOrder(candidate, value) {
                left = mid + 1
            } else if areInIncreasingOrder(value, candidate) {
                right = mid - 1
            } else {
                // 由于 `isOrderedBefore` 的要求，如果两个元素互无顺序关系，那么它们一定相等
                return mid
            }
        }
    }
}
```

```

        }
    }
    //未找到
    return nil
}
}

extension Array where Element: Comparable {
    func binarySearch(for value: Element) -> Int? {
        return self.binarySearch(for: value, areInIncreasingOrder: <)
    }
}

```

对于一个像二分查找这样著名而且看起来很简单算法，其实很难完全写对。在 Java 的实现中，有个 bug 存在了长达二十多年，我们会在本章中的泛型版本中将其修复。但是我们不保证这是二分查找实现中唯一的 bug。

从 Swift 标准库中，我们可以总结出一些值得一提的规范，并将它们应用到二分查找的 API 里：

- 和 index(of:) 类似，我们返回一个可选值索引，nil 表示“未找到”。
- 它被定义两次，其中一次由用户提供比较函数作为参数，另一次依赖于满足某个协议特性的参数，来将它作为调用时的简便版本。
- 序列元素的排序必须是严格弱序。也就是说，当比较两个元素时，要是两者互相都不能排在另一个的前面的话，它们就只能是相等的。

这个 API 对数组有效，但是如果你想要对 ContiguousArray 或者 ArraySlice 进行二分查找的话，就没那么幸运了。这是因为我们所定义的扩展实际上应该是定义在 RandomAccessCollection 上的，随机访问的要求是必须的，否则我们无法保证操作能在对数时间复杂度内完成。因为没有这个保证的话，我们将不能在常数时间内确定序列索引的中点，并进而对索引用 \leq 进行排序。

想要绕过这个问题的话，一条捷径是要求集合拥有 Int 类型的索引。这将能覆盖标准库中几乎所有随机存取的集合类型，这也让你可以将整个 Array 版本的实现直接复制粘贴过来：

```

extension RandomAccessCollection where Index == Int, IndexDistance == Int {
    public func binarySearch(for value: Iterator.Element,
        areInIncreasingOrder: (Iterator.Element, Iterator.Element) -> Bool)
}

```

```
-> Index?  
{  
    // 和 Array 中同样的实现...  
}  
}
```

警告：如果你这么做了，那么你将会引入一个更加糟糕的 bug，我们马上就会遇到这个问题。

但是这将函数限制在了整数索引的集合中，并不是所有集合都是以整数为索引的。Dictionary，Set 和各种表现方式下的 String 都拥有它们自己的索引类型。在标准库中最重要的随机存取的例子是 ReversedRandomAccessCollection。我们在[集合协议](#)一章中看到过它，这是一个由不透明的索引类型将原索引类型进行封装，并将它转换为逆序集合中等效位置的一种类型。

泛型二分查找

如果你把 Int 索引的要求去掉，将会发生一些编译错误。原来的代码需要进行一些重写才能完全满足泛型的要求。下面是完全泛型化之后的版本：

```
extension RandomAccessCollection {  
    public func binarySearch(for value: Iterator.Element,  
        areInIncreasingOrder: (Iterator.Element, Iterator.Element) -> Bool)  
        -> Index?  
{  
    guard !isEmpty else { return nil }  
    var left = startIndex  
    var right = index(before: endIndex)  
  
    while left <= right {  
        let dist = distance(from: left, to: right)  
        let mid = index(left, offsetBy: dist/2)  
        let candidate = self[mid]  
  
        if areInIncreasingOrder(candidate, value) {  
            left = index(after: mid)  
        } else if areInIncreasingOrder(value, candidate) {  
            right = index(before: mid)  
        } else {  
            // 由于 isOrderedBefore 的要求，  
        }  
    }  
}
```

```

        // 如果两个元素互无顺序关系，那么它们一定相等
        return mid
    }
}
// 未找到
return nil
}
}

extension RandomAccessCollection
where Iterator.Element: Comparable
{
    func binarySearch(for value: Iterator.Element) -> Index? {
        return binarySearch(for: value, areInIncreasingOrder: <)
    }
}

```

改动虽小，意义重大。首先，`left` 和 `right` 变量现在不再是整数类型了。我们使用了起始索引和结束索引值。这些值可能是整数，但它们也可能是像是 `String` 的索引，`Dictionary` 的索引，或者是 `Set` 的索引这样的非透明索引，它们是无法随机访问的。

第二， $(\text{left} + \text{right}) / 2$ 被用稍微丑陋一点的 `index(left, offsetBy: dist/2)` 替代了，其中 `dist` 是 `distance(from: left, to: right)`。为什么要这么做？

这里的关键概念在于，实际上有两个类型参与了这个计算，它们是 `Index` 和 `IndexDistance`。它们可以是不同类型的东西，在使用整数索引时，它们恰好可以互换，但是这并不代表对于所有其他类型的索引和距离都有这个特性。

`index(after:)` 方法将会返回当前索引的下一个索引值，因此距离指的是你想要从集合中的某个点到达另一个点时，所需要调用 `index(after:)` 方法的次数。终止索引必须能从起始索引达到，也就是说，你通过有限次地调用 `index(after:)` 应该可以达到集合的终点。这意味着距离值将一定是一个整数（虽然它可以不是一个 `Int` 类型，而可以是其他整数类型）。这是在 `Collection` 中所定义的约束条件：

```

public protocol Collection: Indexable, Sequence {
    /// 一个能够表达两个索引之间步数数字的类型
    associatedtype IndexDistance: SignedInteger = Int
}

```

这也是我们需要一个额外的 guard 来确保集合不为空的原因。当你只是做整数运算时，生成一个 -1 的 right 的值，并检查它是否小于零是没什么不妥的。但是当你再处理其他类型的索引时，你就需要确保不会往回移过头超出集合的起始位置，否则将导致无效的操作。(比如，如果你尝试从一个双向链表的起始向前访问一个节点，是没有意义的。)

因为是整数，所以索引距离可以相加，或者找到两个距离相除的余数。我们不能做的是将两个任意类型的索引相加，因为这个操作也是没有意义的。如果你有一个集合协议一章中定义的列表，很显然你不能将两个节点的指针“相加”。我们能且只能通过 `index(after:)`, `index(before:)` 或者 `index(_:offsetBy:)` 来将索引移动一段距离。

如果你已经习惯了数组中索引的思考方式，那么这种基于距离的方式可能需要一定时间才能适应。不过，你可以将数组的索引表达方式想成一种简写。比如，当我们写 `let right = count - 1` 时，实际上做的是 `right = index(startIndex, offsetBy: count - 1)`。我们之所以能写为 `count - 1`，是因为索引值类型为 `Int`, `startIndex` 是零，这把实际原来的表达式变为了 `0 + count - 1`，最终可以变为简写形式。

正是这个原因这导致了我们在将 `Array` 中的实现搬到 `RandomAccessCollection` 时引入了严重的 bug：以整数为索引的集合的索引值其实并不一定要从零开始，最常见的例子就是 `ArraySlice`。通过 `myArray[3..<5]` 所创建的切片的 `startIndex` 将会为 3。试试看用我们的简化版的泛型二分查找来在切片中查找结果，它将会在运行时崩溃。虽然我们可以要求索引的类型必须是一个整数，但是 Swift 的类型系统并不能做到要求集合是从零开始的。而且就算能这么做，加上这个限制也是一件很蠢的事情，因为我们有更好的方法。我们不应该把左右两边的索引值相加后再除以 2，而应该找到两者之间的距离的一半，然后将这个距离加到左索引上，以得到中点。

这个版本同时也修复了我们初始实现中的 bug。如果你还没有发现这个问题，那么现在我们会告诉你哪儿出错了：想一想在数组非常大的情况下，将两个索引值相加有可能会造成溢出(比如 `count` 很接近 `Int.max`，并且要搜索的元素是数组最后一个元素时的情况)。不过，将距离的一半加到左侧索引时，这个问题就不会发生。当然了，想要触发这个 bug 的机会其实很小，这也是 Java 标准库中这个 bug 能隐藏如此之久的原因所在。

现在，我们能够使用二分查找算法来搜索 `ReversedRandomAccessCollection` 了：

```
let a = ["a", "b", "c", "d", "e", "f", "g"]
let r = a.reversed()

r.binarySearch(for: "g", areInIncreasingOrder: >) == r.startIndex // true
```

我们也可以搜索那些非基于零的索引的切片类型：

```
let s = a[2..<5]
s.startIndex // 2
s.binarySearch(for: "d") // Optional(3)
```

集合随机排列

为了巩固概念，我们这里给出另一个算法例子。这次我们实现一个 Fisher-Yates 洗牌算法：

```
extension Array {
    mutating func shuffle() {
        for i in 0..<(count - 1) {
            let j = Int(arc4random_uniform(UInt32(count - i))) + i

            // 保证不会将一个元素与自己进行交换。
            guard i != j else { continue }

            swap(&self[i], &self[j])
        }
    }

    /// `shuffle` 的不可变版本
    func shuffled() -> [Element] {
        var clone = self
        clone.shuffle()
        return clone
    }
}
```

再一次，我们依照标准库的实践，提供一个原地操作的版本，这可以让整个操作更加高效。然后，生成洗牌后的数组复制的非可变的版本也就可以利用原地操作版本的实现了。

那么，我们要怎么才能写一个不依赖于整数索引的泛型版本呢？和二分查找一样，我们还是需要随机存取，但是因为我们想要提供原地版本，我们还要求这个集合是可变的。`count - 1` 肯定要改为和二分查找里一样的方式来使用。

在我们开始泛型实现之前，原来的实现里还有一处额外需要处理的地方。我们想要用 arc4random_uniform 来生成一个随机数，但是我们并不知道 IndexDistance 的整数到底是什么类型。我们知道它会是一个整数，但是它不一定是 Int。

Swift 当前的整数相关的 API 并不是很适合泛型编程。有一个关于改变整数协议的提案已经被通过了，但是没来得及在 Swift 3.0 中进行实现。

要解决这个问题，我们需要使用 numericCast，它是一个通用的在不同整数类型间进行转换的函数。使用这个函数，我们可以创建一个对所有有符号整型类型都适用的 arc4random_uniform (我们也可以写一个对无符号整数适用的版本，但是因为索引距离都是有符号的，所以我们不需要这么做)：

```
extension SignedInteger {
    static func arc4random_uniform(_ upper_bound: Self) -> Self {
        precondition(upper_bound > 0 &&
            upper_bound.toIntMax() < UInt32.max.toIntMax(),
            "arc4random_uniform only callable up to \(UInt32.max)")
        return numericCast(
            Darwin.arc4random_uniform(numericCast(upper_bound)))
    }
}
```

如果需要，你也可以写一个扩展版的 arc4random，让它的支持范围扩展到负数，或者比 UInt32 还大。但是想要达成这个目的，可能需要很多额外代码。如果你感兴趣的话，arc4random_uniform 的代码实际上是开源的，而且注释也非常完备，它可以给你一些指导来教你如何扩展这个方法。

我们现在就可以对泛型洗牌实现中的每个 IndexDistance 类型生成随机数了：

```
extension MutableCollection where Self: RandomAccessCollection {
    mutating func shuffle() {
        var i = startIndex
        let beforeEndIndex = index(before: endIndex)
        while i < beforeEndIndex {
            let dist = distance(from: i, to: endIndex)
            let randomDistance = IndexDistance.arc4random_uniform(dist)
```

```

let j = index(i, offsetBy: randomDistance)
guard i != j else { continue }
swap(&self[i], &self[j])
formIndex(after: &i)
}
}
}

extension Sequence {
func shuffled() -> [Iterator.Element] {
var clone = Array(self)
clone.shuffle()
return clone
}
}

var numbers = Array(1...10)
numbers.shuffle()
numbers // [10, 8, 5, 7, 6, 1, 2, 9, 3, 4]

```

这个 `shuffle` 函数要比非泛型的版本复杂得多，可读性也更差。这主要是因为我们将 `count - 1` 这样的简单的索引运算替换为了 `index(before: endIndex)`。另一方面，我们从 `for` 循环切换为了 `while` 循环，这是因为如果使用 `for i in indices.dropLast()` 来迭代索引的话，可能会有我们在 集合协议一章中已经提到过的潜在的性能问题：如果 `indices` 属性持有了对集合的引用，那么在遍历 `indices` 的同时更改集合内容，将会让我们失去写时复制的优化，因为集合需要进行不必要的复制操作。

不可否认，这种情况在在我们情形下发生的机率很小，因为大部分的随机存取的集合应该都使用的是整数索引，它们的 `Indices` 没有必要持有原来的集合。比如，`Array.Indices` 的类型是 `CountableRange<Int>`，而非默认的 `DefaultRandomAccessIndices`。不过有一个特例，它是随机存取的集合，但它的索引类型 `Indices` 引用了原集合，这个例子是 `String.UTF16View` (如果你还记得，在字符串一章中我们说过，当引入 `Foundation` 框架后，这个类型是满足 `RandomAccessCollection` 的)。但是它并不是 `MutableCollection`，所以它也无法满足洗牌算法的要求。

在循环中，我们计算了从当前索引到结束索引间的距离，然后使用我们的新的 `SignedInteger.arc4random` 方法来计算一个随机索引值，并进行交换。实际的交换操作和非泛型版本中的是一样的。

你可能会好奇为什么我们在实现非变更的洗牌算法时，没有扩展 MutableCollection。这其实也是一个标准库中经常能够见到的模式 — 比方说，当你对一个 ContiguousArray 进行 sort 操作时，你得到的是一个 Array 返回，而不是 ContiguousArray。

在这里，原因是我们的不可变版本是依赖于复制集合并对它进行原地操作这一系列步骤的。进一步说，它依赖的是集合的值语义。但是并不是所有集合类型都具有值语义。要是 NSMutableArray 也满足 MutableCollection 的话（实际上并不满足，对于 Swift 集合来说，不满足值语义虽然是不好的方式，但是实际上还是可能的），那么 shuffled 和 shuffle 的效果将是一样的。这是因为如果 NSMutableArray 是引用，那么 var clone = self 仅只是复制了一份引用，这样一来，接下来的 clone.shuffle 调用将会作用在 self 上，显然这可能并不是用户所期望的行为。所以，我们可以将这个集合中的元素完全复制到一个数组里，对它进行随机排列，然后返回。

我们可以稍微进行让步，你可以定义一个 shuffle 函数的版本，只要它操作的集合也支持 RangeReplaceableCollection，就让它返回和它所随机的内容同样类型的集合：

```
extension MutableCollection
    where Self: RandomAccessCollection,
          Self: RangeReplaceableCollection
{
    func shuffled() -> Self {
        var clone = Self()
        clone.append(contentsOf: self)
        clone.shuffle()
        return clone
    }
}
```

这个实现依赖了 RangeReplaceableCollection 的两个特性：可以创建一个新的空集合，以及可以将任意序列（在这里，就是 self）添加到空集合的后面。这保证了我们可以进行完全的复制。标准库没有使用这种方式，这可能是因为要照顾到创建数组总是非原地操作这一统一性。但是如果你想要的是例子中这样的完全复制的话，也是可以做到的。要记住，你还需要创建一个序列的版本，这样你才能对那些非可变的可替换区间集合以及序列也进行洗牌操作。

SubSequence 和泛型算法

我们最后举一个例子来说明你在尝试和使用泛型切片时会遇到的问题。

我们来实现一个搜索给定的子序列的算法，它和 `index(of:)` 类似，但是不是查询单个的元素，而是查询一个子序列。理论上，我们可以用一种很简单实现：对集合中的每个索引进行迭代，检查从当前索引开始的切片是否匹配子序列。我们可以使用 `index(where:)`，不过，如果你尝试这么做的话，会得到一个编译错误：

```
extension Collection where Iterator.Element: Equatable {
    func search<Other: Sequence>(for pattern: Other) -> Index?
        where Other.Iterator.Element == Iterator.Element
    {
        return indices.first { idx in
            // 错误: Missing argument for parameter 'by'
            suffix(from: idx).starts(with: pattern)
        }
    }
}
```

错误信息指出编译器在这里需要的是带有一个额外闭包的 `starts(with:by:)` 的版本，这个闭包可以判断两个元素是否相等。这看起来很奇怪，因为我们已经在扩展的时候要求了集合的元素是 `Equatable` 的（通过 `Iterator.Element: Equatable`），所以不带闭包参数的版本 `starts(with:)` 应该是可用的。

另外，我们也通过 `Other.Iterator.Element == Iterator.Element` 对 `Other` 进行了约束，让它和我们的元素是相同类型，这将可以满足 `starts(with:)` 重载的要求。不幸的是，还有另一件事情没有得到保证，那就是切片中的元素的类型 `SubSequence.Iterator.Element` 和集合中的类型是否相等的条件。当然了，逻辑上来说它们肯定是相等的。但是在 Swift 3.0 中，语言还没有强大到能写出这个约束。这让我们通过 `suffix(from: idx)` 创建的切片在编译器眼中是于 `Other` 不兼容的类型。

想在语言层面上修复这个问题，需要能够重新声明 `SubSequence`，并限定 `SubSequence.Iterator.Element` 这个关联类型必须与 `Iterator.Element` 相等。但是针对关联类型的 `where` 语句现在还不被 Swift 支持。不过，Swift 可能可以在今后的版本中获得这个特性。

在此之前，你必须将约束添加到协议扩展里，这样你能保证你使用的每个切片都和集合有同样类型的元素。一开始我们可能会尝试将子序列约束为和集合一样的类型：

```
extension Collection
    where Iterator.Element: Equatable, SubSequence == Self {
    // 和之前的 search 实现一样
```

```
}
```

当子序列和生成这个子序列的集合是同样类型的时候，比如字符串中，这么做是没问题的。但是我们在内建集合中已经看到过，一个数组的切片类型是 `ArraySlice`，它和 `Array` 本身不是同一个类型，所以你也无法在数组上调用 `search`。所以，我们需要一种稍微宽松的约束，相比于直接约束子序列类型，我们对子序列中的元素类型进行约束，让它和原集合中的元素类型相同：

```
extension Collection
  where Iterator.Element: Equatable,
        SubSequence.Iterator.Element == Iterator.Element
{
  // 和之前的 search 实现一样
}
```

现在编译器将给出另一个错误信息，这次是在 `indices.first` 调用的尾随闭包上：“不能将 `(Self.Index) -> Bool` 值转换为需要的 `(_) -> Bool`”。这是什么意思呢？其实问题本质和刚才是一样的，不过这回问题出在索引，而非集合中的元素类型上。类型系统无法表示 `Indices` 的元素类型（注意 `Indices` 本身也是一个集合）会与集合索引的类型始终相同。这导致了闭包的 `Indices.Iterator.Element` 类型的 `idx` 参数无法匹配 `suffix(from:)` 所需要的 `Index` 类型的参数。

通过为扩展添加下面的约束可以最终让代码编译通过：

```
extension Collection
  where Iterator.Element: Equatable,
        SubSequence.Iterator.Element == Iterator.Element,
        Indices.Iterator.Element == Index
{
  func search<Other: Sequence>(for pattern: Other) -> Index?
    where Other.Iterator.Element == Iterator.Element
  {
    return indices.first { idx in
      suffix(from: idx).starts(with: pattern)
    }
  }
}

let text = "It was the best of times, it was the worst of times"
text.characters.search(for: ["b", "e", "s", "t"])
/*
```

```
Optional(Swift.String.CharacterView.Index(_base: Swift.String.UnicodeScalarView.Index(_position: 11, _countUTF16: 1))  
*/
```

注意，整个过程中我们没有改变任何的实际代码，而仅只是按照类型要求和需要使用的功能对约束进行了更新。

重写与优化

最后，如果约束稍微严格一些的话，很可能你就将能够提供更高效的泛型算法。举个例子，如果你知道被搜索的集合和待搜索的子序列都满足随机存取的索引的话，你就能够改进 `search` 算法的实现。比如说，你可以完全避免在集合比子序列长度还短的时候进行搜索，也可以跳过那些集合中剩余元素不足以匹配子序列的部分。

想要这些起效，你就要保证 `Self` 和 `Other` 都遵守 `RandomAccessCollection`。然后我们就可以针对这些约束并利用它们带来的特性进行编码：

```
extension RandomAccessCollection  
    where Iterator.Element: Equatable,  
          Indices.Iterator.Element == Index,  
          SubSequence.Iterator.Element == Iterator.Element,  
          SubSequence.Indices.Iterator.Element == Index  
{  
    func search<Other: RandomAccessCollection>  
        (for pattern: Other) -> Index?  
        where Other.IndexDistance == IndexDistance,  
              Other.Iterator.Element == Iterator.Element  
    {  
        // 如果匹配模式比集合长，那么就不可能会匹配，提前退出  
        guard !isEmpty && pattern.count <= count else { return nil }  
  
        // 否则，从起始索引开始取到能容纳匹配模式的最后一个索引  
        let stopSearchIndex = index(endIndex, offsetBy: -pattern.count)  
  
        // 检查从当前位置开始的切片是否以待匹配模式开头  
        return prefix(upTo: stopSearchIndex).indices.first { idx in  
            suffix(from: idx).starts(with: pattern)  
        }  
    }
```

```
}

let numbers = 1..  
numbers.search(for: 80..  
// Optional(80)
```

我们在这里还加入了另一个约束：两个集合的距离的类型是相同的，这可以让代码保持简单。虽然实际上它们确实有可能不同，但是这种情况非常罕见，可能只有用于类型抹消的 AnyCollection 结构体使用 `IntMax`，它在 32 位系统上与 `Int` 有所不同。我们也可以稍微使用 `numericCast` 来代替这个约束，比如可以用 `guard numericCast(pattern.count) <= count else { return nil }` 进行确保。另外，我们还需要加上 `SubSequence.Indices.Iterator.Element == Index`，否则 `prefix(upTo: stopSearchIndex).indices` 的元素类型可能会令集合的索引类型对不上。虽然这应该是显而易见的事实，但是我们还是需要明确地把这个信息告诉编译器。

使用泛型进行代码设计

我们已经看到了很多将泛型用来为同样的功能提供多种实现的例子。我们可以编写泛型函数，但是却对某些特定的类型提供不同的实现。同样，使用协议扩展，我们还可以编写同时作用于很多类型的泛型算法。

泛型在你进行程序设计时会非常有用，它能帮助你提取共通的功能，并且减少模板代码。在这一节中，我们会将一段普通的代码进行重构，使用泛型的方式提取出共通部分。除了可以创建泛型的方法以外，我们也可以创建泛型的数据类型。

让我们来写一些与网络服务交互的函数。比如，获取用户列表的数据，并将它解析为 `User` 数据类型。我们创建一个 `loadUsers` 函数，它可以从网上异步加载用户，并且在完成后通过回调来传递获取到的用户。

当我们用最原始的方式来实现的话，首先我们要创建 URL，然后我们同步地加载数据（这里只是为了简化我们的例子，所以使用了同步方式。在你的产品中，你应当始终用异步方式加载你的数据）。接下来，我们解析 JSON，得到一个含有字典的数组。最后，我们将这些 JSON 对象变形为 `User` 结构体：

```
func loadUsers(callback: ([User]?) -> ()) {  
    let usersURL = webserviceURL.appendingPathComponent("/users")  
    let data = try? Data(contentsOf: usersURL)  
    let json = data.flatMap {
```

```
try? JSONSerialization.jsonObject(with: $0, options: [])
}

let users = (json as? [Any]).flatMap { jsonObject in
    jsonObject.flatMap(User.init)
}
callback(users)
}
```

这个函数有三种可能发生错误的情况：URL 加载可能失败，JSON 解析可能失败，通过 JSON 数组构建用户对象也可能失败。在这三种情况下，我们都返回 nil。通过对可选值使用 flatMap，我们能确保只对那些成功的对象进行接下来的操作。不这么做的话，第一个失败操作造成的 nil 值将传播到接下来的操作，直至结束。我们在结束的时候会调用回调，传回一个有效的用户数组，或者传回 nil。

现在，如果我们想要写一个相同的函数来加载其他资源，我们可能需要复制这里的大部分代码。打个比方，我们需要一个加载博客文章的函数，它看起来是这样的：

```
func loadBlogPosts(callback: ([BlogPost])? -> ())
```

函数的实现和前面的用户函数几乎相同。不仅代码重复，两个方法同时也很难测试，我们需要确保网络服务可以在测试时被访问到，或者是找到一个模拟这些请求的方法。因为函数接受并使用回调，我们还需要保证我们的测试是异步运行的。

提取共通功能

相比于复制粘贴，将函数中 User 相关的部分提取出来，将其他部分进行重用，会是更好的方式。我们可以将 URL 路径和解析转换的函数作为参数传入。因为我们希望可以传入不同的转换函数，所以我们将 loadResource 声明为 A 的泛型：

```
func loadResource<A>(at path: String,
    parse: (Any) -> A?,
    callback: (A?) -> ())
{
    let resourceURL = webServiceURL.appendingPathComponent(path)
    let data = try? Data(contentsOf: resourceURL)
    let json = data.flatMap {
        try? JSONSerialization.jsonObject(with: $0, options: [])
    }
}
```

```
    callback(json.flatMap(parse))
}
```

现在，我们可以将 `loadUsers` 函数基于 `loadResource` 重写：

```
func loadUsers(callback: ([User]?) -> ()) {
    loadResource(at: "/users", parse: jsonArray(User.init), callback: callback)
}
```

我们使用了一个辅助函数，`jsonArray`，它首先尝试将一个 Any 转换为一个 Any 的数组，接着对每个元素用提供的解析函数进行解析，如果期间任何一步发生了错误，则返回 nil：

```
func jsonArray<A>(_ transform: @escaping (Any) -> A?) -> (Any) -> [A]? {
    return { array in
        guard let array = array as? [Any] else {
            return nil
        }
        return array.flatMap(transform)
    }
}
```

对于加载博客文章的函数，我们只需要替换请求路径和解析函数就行了：

```
func loadBlogPosts(callback: ([BlogPost]?) -> ()) {
    loadResource(at: "/posts", parse: jsonArray(BlogPost.init), callback: callback)
}
```

这让我们能少写很多重复的代码。如果之后我们决定将同步 URL 处理重构为异步加载时，就不再需要分别更新 `loadUsers` 或者 `loadBlogPosts` 了。虽然这些方法现在很短，但是想测试它们也并不容易：它们基于回调，并且需要网络服务处于可用状态。

创建泛型数据类型

`loadResource` 函数中的 `path` 和 `parse` 耦合非常紧密，一旦你改变了其中一个，你很可能也需要改变另一个。我们可以将它们打包进一个结构体中，用来描述要加载的资源。和函数一样，这个结构体也可以是泛型的：

```
struct Resource<A> {
```

```
let path: String  
let parse: (Any) -> A?  
}  
}
```

现在，我们可以在 Resource 上定义一个新的 loadResource 方法。它使用 resource 的属性来确定要加载的内容以及如何解析结果，这样一来，方法的参数就只剩回调函数了：

```
extension Resource {  
func loadSynchronously(callback: (A?) -> ()) {  
    let resourceURL = webServiceURL.appendingPathComponent(path)  
    let data = try? Data(contentsOf: resourceURL)  
    let json = data.flatMap {  
        try? JSONSerialization.jsonObject(with: $0, options: [])  
    }  
    callback(json.flatMap(parse))  
}  
}
```

相比于之前的用顶层函数来定义资源，我们现在可以定义 Resource 结构体值，这让我们可以很容易地添加新的资源，而不必创建新的函数：

```
let usersResource: Resource<[User]> =  
    Resource(path: "/users", parse: jsonArray(User.init))  
let postsResource: Resource<[BlogPost]> =  
    Resource(path: "/posts", parse: jsonArray(BlogPost.init))
```

现在，添加一个异步的处理方法就非常简单了，我们不需要改变任何现有的描述 API 接入点的代码：

```
extension Resource {  
func loadAsynchronously(callback: @escaping (A?) -> ()) {  
    let resourceURL = webServiceURL.appendingPathComponent(path)  
    let session = URLSession.shared  
    session.dataTask(with: resourceURL) { data, response, error in  
        let json = data.flatMap {  
            try? JSONSerialization.jsonObject(with: $0, options: [])  
        }  
        callback(json.flatMap(self.parse))  
    }.resume()  
}
```

```
    }  
}
```

除了使用了异步的 URLSession API 以外，和同步版本相比，还有一个本质上的不同是回调函数现在将从方法作用域中逃逸出来，所以它必须被标记为 @escaping。如果你想了解关于逃逸闭包和非逃逸闭包的更多信息，请参阅方法一章。

现在，我们将接入点和网络请求完全解耦了。我们将 usersResource 和 postResource 归结为它们的最小版本，它们只负责描述去哪里寻找资源，以及如何解析它们。这种设计也是可扩展的：你可以进行更多配置，比如添加 HTTP 请求方法或是为请求加上一些 POST 数据等，你只需要简单地在 Resource 上增加额外属性就可以了（为了保持代码干净，你应该指定一些默认值。比如对 HTTP 请求方法，可以设定默认值为 GET）。

测试也变得容易很多。Resource 结构体是完全同步，并且和网络解耦的。测试 Resource 是否配置正确是很简单的一件事。不过网络部分的代码依然难以测试，当然了，因为它天生就是异步的，并且依赖于网络。但是这个复杂度现在被很好地隔离到了 loadAsynchronously 方法中，而代码的其他部分都很简单，也没有受到异步代码的影响。

在本节中，我们从一个非泛型的从网络加载数据的函数开始，接下来，我们用多个参数创建了一个泛型函数，允许我们用简短得多的方式重写代码。最后，我们把这些参数打包到一个单独的 Resource 数据类型中，这让代码的解耦更加彻底。对于具体资源类型的专用逻辑是于网络代码完全解耦的。更改网络层的内容不会对资源层有任何影响。

泛型的工作方式

从编译器的视角来看，泛型是如何工作的呢？要回答这个问题，我们先来看看标准库中的 min 函数（我们从 2015 年 WWDC 的 Swift 性能优化专题中挑选了这个例子）：

```
func min<T: Comparable>(_ x: T, _ y: T) -> T {  
    return y < x ? y : x  
}
```

min 的两个参数和返回值泛型的唯一约束是它们三者都必须是同样的类型 T，而这个 T 需要满足 Comparable。只要满足这个要求，T 可以是任意类型，它可以是 Int, Float, String 或者甚至是在编译时未知的定义在其他模块的某个类型。也就是说，编译器缺乏两个关键的信息，这导致它不能直接为这个函数生成代码：

→ 编译器不知道（包括参数和返回值在内的）类型为 T 的变量的大小

→ 编译器不知道需要调用的 `<` 函数是否有重载，因此也不知道需要调用的函数的地址。

Swift 通过为泛型代码引入一层间接的中间层来解决这些问题。当编译器遇到一个泛型类型的值时，它会将其包装到一个容器中。这个容器有固定的大小，并存储这个泛型值。如果这个值超过容器的尺寸，Swift 将在堆上申请内存，并将指向堆上该值的引用存储到容器中去。

对于每个泛型类型的参数，编译器还维护了一系列一个或者多个所谓的**目击表** (witness table)：其中包含一个**值目击表**，以及类型上每个协议约束一个的**协议目击表**。这些目击表 (也被叫做 **vtable**) 将被用来将运行时的函数调用动态派发到正确的实现去。

对于任意的泛型类型，总会存在值目击表，它包含了指向内存申请，复制和释放这些类型的基本操作的指针。这些操作对于像是 `Int` 这样的值类型来说，可能不需要额外操作，或者只是简单的内存复制，不过对于引用类型来说，这里也会包含引用计数的逻辑。值目击表同时还记录了类型的大小和对齐方式。

我们这个例子中的泛型类型 `T` 将会包含一个协议目击表，因为 `T` 有 `Comparable` 这一个约束。对于这个协议声明的每个方法或者属性，协议目击表中都会含有一个指针，指向该满足协议的类型中的对应实现。在泛型函数中对这些方法的每次调用，都会在运行时通过目击表准换为方法派发。在我们的例子中，`y < x` 这个表达式就是以这种方式进行派发的。

协议目击表提供了一组映射关系，通过这组映射，我们可以知道泛型类型满足的协议 (编译器通过泛型约束可以静态地知道这个信息) 和某个具体类型对于协议功能的具体实现 (这只在运行时才能知道) 的对应关系。实际上，只有通过目击表我们才能查询或者操作某个值。我们无法在不加约束地定义一个 `<T>` 参数的同时，还期望它能对任意实现了 `<` 的类型工作。如果没有满足 `Comparable` 的保证，编译器就不会让我们使用 `<` 操作，这是因为没有目击表可以让我们找到正确的 `<` 的实现。这就是我们说泛型和协议是紧密联系的原因，除了像是 `Array<Element>` 或者 `Optional<Wrapped>` 这样的容器类型，脱离了使用协议来约束泛型，泛型所能做的事情也就非常有限了。

总结一下，对于 `min` 函数，编译器生成的伪代码看上去会是这样的：

```
func min<T: Comparable>(_ x: TBox, _ y: TBox,
    valueWTable: VTable, comparableWTable: VTable)
    -> TBox
{
    let xCopy = valueWTable.copy(x)
    let yCopy = valueWTable.copy(y)
    let result = comparableWTable.lessThan(yCopy, xCopy) ? y : x
    valueWTable.release(xCopy)
```

```
valueWTable.release(yCopy)
return result
}
```

泛型参数的容器结构和我们在下一章将要提到的协议类型中使用的“**存在容器**”(existential containers)有些相似，但是并不完全一样。一个存在容器中不仅会有值的存储，还可能在一个结构体中存在指向目击表的指针。而泛型参数的容器只会包含值存储，目击表是被单独存储的，这样泛型函数中同样类型的其他变量就可以共享这个目击表了。

泛型特化

在上一节中我们描述的“编译一次，动态派发”的模型是 Swift 泛型系统的重要设计目标。我们可以将其与 C++ 的模板(template)特性进行比较。在 C++ 中，编译器会为使用了模板的每个类型生成独立的模板化的函数或者类的实例。相比下来，Swift 的实现会有更快的编译速度以及更小的二进制文件。Swift 的模型也更加灵活，不像 C++ 那样，Swift 中使用泛型 API 的代码只需要知道泛型函数或者类型的声明，而不需要关心它们的实现。

相应地，因为需要经过的代码不是那么直接，所以这种做法的缺点是运行时性能会较低。对于单个的函数调用来说这点开销是可以忽略的，但是因为泛型在 Swift 中非常普及，所以它这种性能开销很容易堆叠起来，造成性能问题。标准库到处都是泛型，包括比较值的大小在内的很多常用操作必须尽可能快速。因为这类操作十分频繁，所以尽管泛型代码只是比非泛型代码慢一点点，开发者可能也会选择不去使用泛型版本。

不过 Swift 可以通过**泛型特化**(generic specialization)的方式来避免这个额外开销。泛型特化是指，编译器按照具体的参数参数类型(比如 Int)，将 `min<T>` 这样的泛型类型或者函数进行复制。特化后的函数可以将针对 Int 进行优化，移除所有的额外开销。所以 `min<T>` 针对 Int 的特化版本是这样的：

```
func min(_x: Int, _y: Int) -> Int {
    return y < x ? y : x
}
```

这不仅能去掉派发的开销，还可以让像是内联等进一步优化成为可能。由于原来的函数是被非直接使用的，这些优化以前是无法实行的。

优化器使用启发式的方法来决定需要为哪个泛型类型或函数进行特化操作，以及使用哪个具体类型进行特化。这个决定需要在编译时间，二进制大小以及运行时性能之间进行折衷考量。如果你的代码经常使用 Int 来调用 min 函数，而只调用了一次 Float 的版本，那很有可能只有 Int 的版本会被特化处理。你应该在编译的时候确保开启优化（使用命令行的话，是 swiftc -O），这样你可以用到所有可能的启发式算法来进行优化。

即使编译器采取了非常激进的策略来进行泛型特化，如果这个泛型函数会被其他模块看到的话，该函数的泛型版本就将会始终存在。在编译这些泛型函数的时候编译器并不知道外部的类型，保留泛型版本确保了外部代码始终可以调用到这个函数。

全模块优化

泛型特化只能在编译器可以看到泛型类型的全部定义以及想要进行特化的类型的时候才能生效。但是 Swift 编译器默认情况下是对源文件进行单独编译的，所以只有在使用泛型的代码和定义泛型代码在同一个文件中时，泛型特化才能工作。

因为这是一个很严重的限制，所以编译器引入了一个标志来启用**全模块优化**。在这个模式种，当前模块的所有文件会被当作全部都在一个文件中来进行优化，这让泛型特化可以横跨整个代码库进行工作。你可以通过向 swiftc 传递 `-whole-module-optimization` 来开启全模块优化。请确保在发布版本（甚至可能的话在调试版本）中进行这项操作，因为这将大幅提升性能。不过缺点是会带来更长的编译时间。

全模块优化会同时启用一些其他的重要优化。比如，优化器将会识别出整个模块中没有子类的 internal 类。因为 internal 关键字确保了这个类不会出现在模块外部，所以这意味着编译器可以将这个类的所有方法调用从动态派发转变为静态派发。

泛型特化要求泛型类型或者函数的定义可见，所以它不能跨越模块边界使用。也就是说，相比于外部使用者来说，你的泛型代码可能在泛型定义所在的模块内部拥有更好的性能。唯一的例外是标准库中的泛型代码。因为标准库被所有其他模块使用，所以标准库中的定义对于所有模块都是可见的，也就是说，所有其他模块都能够对标准库中的泛型进行特化。

Swift 中有一个叫做 `@_specialize` 的半官方标签，它能让你将你的泛型代码进行指定版本的特化，使其在其他模块中也可用。你必须指明你想要进行特化的类型列表，所以这只在当你知道你的代码将如何被一些有限的类型使用的时候能有帮助。下面的例子将 min 函数为整数和字符串进行了特化，这将使得它们能被其他模块使用：

```
@_specialize(Int)
```

```
@_specialize(String)
public func min<T: Comparable>(_ x: T, _ y: T) -> T {
    return y < x ? y : x
}
```

注意我们添加了 `public`。为 `internal`、`fileprivate` 或者 `private` 的 API 添加 `@_specialize` 是没有意义的，因为它们对其他模块是不可见的。

总结

在本章一开始，我们将泛型编程定义为描述算法或者数据结构所要求的核心接口。我们通过 `isSubset` 的例子讲述了这一点，首先我们从一个非泛型版本开始，接着小心移除了约束。通过使用不同的约束进行重载可以让我们在为尽可能多的类型提供功能的同时，保持性能优秀。编译器将根据类型来选择使用最合适重载版本。

在异步网络请求的例子中，我们将网络部分的很多假设从 `Resource` 结构体中移除了出去。具体的资源值不会依赖于服务器的根域名或者数据的加载方式，它们所做的只是描述 API 接入点。在这个例子中，泛型编程让我们的资源类型更加简单，耦合更少，这也让测试更加容易。

如果你对泛型编程背后的理论细节感兴趣，并想了解不同的语言是如何利用泛型编程的话，我们推荐阅读 Ronald Garcia 等人在 2007 年发表的 [《关于泛型编程的语言支持的扩展深度学习》](#)一文。

最后，Swift 的泛型编程不可能离开协议。在下一章我们会对协议进行深入探讨。

协议

10

在上一章，我们看到了函数和泛型可以帮助我们写出动态的程序。协议可以与函数和泛型协同工作，让我们代码的动态特性更加强大。

Swift 的协议和 Objective-C 的协议不同。Swift 协议可以被用作代理，也可以让你对接口进行抽象（比如 `IteratorProtocol` 和 `Sequence`）。它们和 Objective-C 协议的最大不同在于我们可以让结构体和枚举类型满足协议。除此之外，Swift 协议还可以有关联类型。我们还可以通过协议扩展的方式为协议添加方法实现。我们会在面向协议编程的部分讨论所有这些内容。

协议允许我们进行动态派发，也就是说，在运行时程序会根据消息接收者的类型去选择正确的方法实现。不过，方法到底什么时候是动态派发，什么时候不是动态派发，有时却不是那么直观，并有可能造成令人意外的结果。我们会在下一节中看到这个问题。

普通的协议可以被当作类型约束使用，也可以当作独立的类型使用。带有关联类型或者 `Self` 约束的协议特殊一些：我们不能将它当作独立的类型来使用，而只能将它们用作类型约束。这听起来似乎是一个小限制，但是这在实践中让带有关联类型的协议成为了完全不同的东西。我们会在之后详细对此说明，我们还将讨论如何使用（像是 `AnyIterator` 这样的）类型消除的方法来让带有关联类型的协议更加易用。

在面向对象编程中，子类是在多个类之间共享代码的有效方式。一个子类将从它的父类继承所有的方法，然后选择重写其中的某些方法。比如，我们可以有一个 `AbstractSequence` 类，以及像是 `Array` 和 `Dictionary` 这样的子类。这么做的话，我们就可以在 `AbstractSequence` 中添加方法，所有的子类都将自动继承到这些方法。

不过在 Swift 中，`Sequence` 中的代码共享是通过协议和协议扩展来实现的。通过这么做，`Sequence` 协议和它的扩展在结构体和枚举这样的值类型中依然可用，而这些值类型是不支持子类继承的。

不再依赖于子类让类型系统更加灵活。在 Swift 中，一个类只能有一个父类。当我们创建一个类时，我们必须同时选择父类，而且我们只能选择一个父类，比如我们无法创建同时是 `AbstractSequence` 和 `Stream` 的子类的类。这有时候会成为问题。在 Cocoa 中就有一些例子，比如 `NSMutableAttributedString`，框架的设计师必须在 `NSAttributedString` 和 `NSMutableString` 之间选择一个父类。

有一些语言有多继承的特性，其中最著名的是 C++。但是这也导致了钻石问题（或者叫菱型缺陷）的麻烦。举例来说，如果可以多继承，那么我们就可以让 `NSMutableAttributedString` 同时继承 `NSMutableString` 和 `NSAttributedString`。但是要是这两个类中都重写了 `NSString` 中的某个方法的时候，该怎么办？你可以通过选择其中一个方法来解决这个问题。但是要是这个方式是 `isEqual:` 这样的通用方法又该怎么处理呢？实际上，为多继承的类提供合适的行为真的是一件非常困难的事情。

因为多继承如此艰深难懂，所以绝大多数语言都不支持它。不过很多语言支持实现多个协议的特性。相比多继承，实现多个协议并没有那些问题。在 Swift 中，编译器会在方法冲突的时候警告我们。

协议扩展是一种可以在不共享基类的前提下共享代码的方法。协议定义了一组最小可行的方法集合，以供类型进行实现。而类型通过扩展的方式在这些最小方法上实现更多更复杂的特性。

比方说，要实现一个对任意序列进行排序的泛型算法，你需要两件事情。首先，你需要知道如何对要排序的元素进行迭代。其次，你需要能够比较这些元素的大小。就这么多了。我们没有必要知道元素是如何被存储的，它们可以是在一个链表里，也可以在数组中，或者任何可以被迭代的容器中。我们也没有必要规定这些元素到底是什么，它们可以是字符串，整数，数据，或者是具体的像是“人”这样的数据类型。只要你在类型系统中提供了前面提到的那两个约束，我们就能实现 `sort` 函数：

```
extension Sequence where Iterator.Element: Comparable {  
    func sorted() -> [Self.Iterator.Element]  
}
```

想要实现原地排序的话，我们需要更多的构建代码。你需要能够通过索引访问元素，而不仅仅是进行线性迭代。`Collection` 满足这点，而 `MutableCollection` 在其之上加入了可变特性。最后，你需要能在常数时间内比较索引，并移动它们。`RandomAccessCollection` 正是用来保证这一点的。这些听起来可能有点复杂，但这正是我们能够实现一个原地排序所需要的前置条件：

```
extension MutableCollection where  
    Self: RandomAccessCollection,  
    Self.Iterator.Element: Comparable {  
    mutating func sort()  
}
```

通过协议来描述的最小功能可以很好地进行整合。你可以一点一点地为某个类型添加由不同协议所带来的不同功能。我们已经在集合协议这篇中一开始使用单个 `cons` 方法构建 `List` 类型的例子中看到过这样的应用场景了。我们让 `List` 实现了 `Sequence` 协议，而没有改变原来 `List` 结构体的实现。实际上，即使我们不是这个类型的原作者，也可以使用追溯建模 (*retroactive modeling*) 的方式完成这件事情。通过添加 `Sequence` 的支持，我们直接获得了 `Sequence` 类型的所有扩展方法。

通过共同的父类来添加共享特性就没那么灵活了；在开发过程进行到一半的时候再决定为很多不同的类添加一个共同基类往往是很困难的。你想这么做的话，可能要面临大量的重构。而且如果你不是这些子类的拥有者的话，你直接就无法这么处理！

子类必须知道哪些方法是它们能够重写而不会破坏父类行为的。比如，当一个方法被重写时，子类可能会需要在合适的时机调用父类的方法，这个时机可能是方法开头，也可能是中间某个地方，又或者是在方法最后。通常这个调用时机是不可预估和指定的。另外，如果重写了错误的方法，子类还可能破坏父类的行为，却不会收到任何来自编译器的警告。

面向协议编程

比如在一个图形应用中，我们想要进行两种渲染：我们会将图形使用 Core Graphics 的 CGContext 渲染到屏幕上，还需要创建一个 SVG 格式的图形文件。我们可以从定义绘图 API 的最小功能集的协议开始进行实现：

```
protocol Drawing {
    mutating func addEllipse(rect: CGRect, fill: UIColor)
    mutating func addRectangle(rect: CGRect, fill: UIColor)
}
```

协议的最强大的特性之一就是我们可以以追溯的方式来修改任意类型，让它们满足协议。对于 CGContext，我们可以添加扩展来让它满足 Drawing 协议：

```
extension CGContext: Drawing {
    func addEllipse(rect: CGRect, fill: UIColor) {
        setFillColor(fill.cgColor)
        fillEllipse(in: rect)
    }

    func addRectangle(rect: CGRect, fill fillColor: UIColor) {
        setFillColor(fill.cgColor)
        fill(rect)
    }
}
```

要表示 SVG 文件，我们创建一个 SVG 结构体。它包含一个带有子节点的 XMLNode，以及一个 append 方法，来将子节点添加到根节点上。(我们这里没有写出 XMLNode 的定义)

```
struct SVG {
    var rootNode = XMLNode(tag: "svg")
    mutating func append(node: XMLNode) {
        rootNode.children.append(node)
    }
}
```

```
    }
}
```

渲染 SVG 的意思是，我们需要将每一个元素都添加到节点上。我们使用了一些简单的扩展：
CGRect 上的 `svgAttributes` 属性创建一个满足 SVG 标准的代表当前矩形的字典。
`String.init(hexColor:)` 接受一个 `UIColor` 并将它转换为十六进制的字符串（比如 `#010100`）。有了这些复制方法，为 SVG 添加 Drawing 支持就水到渠成了：

```
extension SVG: Drawing {
    mutating func addEllipse(rect: CGRect, fill: UIColor) {
        var attributes: [String:String] = rect.svgAttributes
        attributes["fill"] = String(hexColor: fill)
        append(node: XMLNode(tag: "ellipse", attributes: attributes))
    }

    mutating func addRectangle(rect: CGRect, fill: UIColor) {
        var attributes: [String:String] = rect.svgAttributes
        attributes["fill"] = String(hexColor: fill)
        append(node: XMLNode(tag: "rect", attributes: attributes))
    }
}
```

现在只需要假设 `context` 是满足 Drawing 协议，我们就可以写出独立于渲染目标的代码了。如果我们决定使用 `CGContext` 来取代 SVG，我们也不需要改变代码的任何部分：

```
var context: Drawing = SVG()
let rect1 = CGRect(x: 0, y: 0, width: 100, height: 100)
let rect2 = CGRect(x: 0, y: 0, width: 50, height: 50)
context.addRectangle(rect: rect1, fill: .yellow)
context.addEllipse(rect: rect2, fill: .blue)
context
/*
<svg>
<rect cy="0.0" fill="#010100" ry="100.0" rx="100.0" cx="0.0"/>
<ellipse cy="0.0" fill="#000001" ry="50.0" rx="50.0" cx="0.0"/>
</svg>
*/
```

协议扩展

Swift 的协议的另一个强大特性是我们可以使用完整的方法实现来扩展一个协议。你可以扩展你自己的协议，也可以对已有协议进行扩展。比如，我们可以向 Drawing 添加一个方法，给定一个中心点和一个半径，渲染一个圆：

```
extension Drawing {  
    mutating func addCircle(center: CGPoint, radius: CGFloat, fill: UIColor) {  
        let diameter = radius/2  
        let origin = CGPoint(x: center.x - diameter, y: center.y - diameter)  
        let size = CGSize(width: radius, height: radius)  
        let rect = CGRect(origin: origin, size: size)  
        addEllipse(rect: rect, fill: fill)  
    }  
}
```

通过在扩展中添加 addCircle，我们就可以在 CGContext 和 SVG 中使用它了。

通过协议进行代码共享相比与通过继承的共享，有这几个优势：

- 我们不需要被强制使用某个父类。
- 我们可以让已经存在的类型满足协议 (比如我们让 CGContext 满足了 Drawing)。子类就沒那么灵活了，我们无法以追溯的方式去变更 CGContext 的父类。
- 协议既可以用于类，也可以用于结构体，而父类就无法和结构体一起使用了。
- 最后，当处理协议时，我们无需担心方法重写或者在正确的时间调用 super 这样的问题。

在协议扩展中重写方法

作为协议的作者，当你想在扩展中添加一个协议方法，你有两种方法。首先，你可以只在扩展中进行添加，就像我们上面 addCircle 所做的那样。或者，你还可以在协议定义本身中添加这个方法的声明，让它成为**协议要求**的方法。协议要求的方法是动态派发的，而仅定义在扩展中的方法是静态派发的。它们的区别虽然很微小，但不论对于协议的作者还是协议的使用者来说，都十分重要。

让我们来看一个例子，在上节中，我们将 `addCircle` 作为 Drawing 协议的扩展进行了添加，不过我们没有将其标记为协议要求的方法。如果我们想要为 SVG 类型提供一个更具体的 `addCircle` 的话，我们可以“重写”这个方法：

```
extension SVG {  
    mutating func addCircle(center: CGPoint, radius: CGFloat, fill: UIColor) {  
        var attributes: [String:String] = [  
            "cx": "\(center.x)",  
            "cy": "\(center.y)",  
            "r": "\(radius)",  
        ]  
        attributes["fill"] = String(hexColor: fill)  
        append(node: XMLNode(tag: "circle", attributes: attributes))  
    }  
}
```

现在如果我们创建一个 SVG 实例并调用它的 `addCircle` 方法，结果将和你期待的一致：编译器将选择 `addCircle` 的最具体的版本，也就是定义在 SVG 扩展上的版本。我们可以看到它正确地使用了 `circle` 标签：

```
var sample = SVG()  
sample.addCircle(center: .zero, radius: 20, fill: .red)  
print(sample)  
/*  
<svg>  
<circle cy="0.0" fill="#010000" r="20.0" cx="0.0"/>  
</svg>  
*/
```

现在，和上面一样，我们创建另一个 SVG 实例，唯一的区别在于我们明确地将变量转换为 Drawing 类型。如果我们对这个 Drawing 调用 `addCircle`，会发生什么呢？很多人可能会认为这个调用依然会派发到 SVG 上同样的实现，但是事实并非如此：

```
var otherSample: Drawing = SVG()  
otherSample.addCircle(center: .zero, radius: 20, fill: .red)  
print(otherSample)  
/*  
<svg>  
<ellipse cy="-10.0" fill="#010000" ry="20.0" rx="20.0" cx="-10.0"/>
```

```
</svg>  
*/
```

它返回的是 `ellipse` 元素，而不是我们所期望的 `circle`。也就是说，它使用了协议扩展中的 `addCircle` 方法，而没有用 SVG 扩展中的。当我们把 `otherSample` 定义为 `Drawing` 类型的变量时，编译器会自动将 SVG 值封装到一个代表协议的类型中，这个封装被称作**存在容器** (*existential container*)，我们会在本章后面讨论具体细节。现在，我们可以这样考虑这个行为：当我们对存在容器调用 `addCircle` 时，方法是静态派发的，也就是说，它总是会使用 `Drawing` 的扩展。如果它是动态派发，那么它肯定需要将方法的接收者 `SVG` 类型考虑在内。

想要将 `addCircle` 变为动态派发，我们可以将它添加到协议定义里：

```
protocol Drawing {  
    mutating func addEllipse(rect: CGRect, fill: UIColor)  
    mutating func addRectangle(rect: CGRect, fill: UIColor)  
    mutating func addCircle(center: CGPoint, radius: CGFloat, fill: UIColor)  
}
```

我们依旧可以像之前那样提供一个默认的实现。

而且和之前一样，具体的类型还是可以自由地重写 `addCircle`。因为现在它是协议定义的一部分了，它将被动态派发。在运行时，根据方法接收者的动态类型的不同，存在容器将会在自定义实现存在时对其进行调用。如果自定义实现不存在，那么它将使用协议扩展中的默认实现。`addCircle` 方法变为了协议的一个**自定义入口**。

Swift 标准库大量使用了这样的技术。像是 `Sequence` 这样的协议有非常多的必要方法，不过几乎所有的方法都有默认实现。因为方法是动态派发的，所以实现 `Sequence` 协议的类型可以自定义默认实现，不过这并不是必须的。

协议的两种类型

我们在本章介绍部分已经指出，带有关联类型的协议和普通的协议是不同的。对于那些在协议定义中在任何地方使用了 `Self` 的协议来说也是如此。Swift 3 中，这样的协议不能被当作独立的类型来使用。这个限制可能会在今后实现了完整的泛型系统后被移除，但是在那之前，我们都必须要面对和处理这个限制。

一个带有关联类型的协议的最简单的例子是 `IteratorProtocol`，它包含一个关联类型 `Element` 和一个返回该类型值的 `next()` 函数：

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

在集合协议中，我们已经看到过一个满足 `IteratorProtocol` 的类型的例子了。这个迭代器在每次被调用时，都简单地返回 1：

```
struct ConstantIterator: IteratorProtocol {  
    mutating func next() -> Int? {  
        return 1  
    }  
}
```

我们知道，`IteratorProtocol` 是集合协议的基础。和 `IteratorProtocol` 不同，`Collection` 协议的定义要复杂得多：

```
protocol Collection: Indexable, Sequence {  
    associatedtype IndexDistance: SignedInteger = Int  
    associatedtype Iterator: IteratorProtocol = IndexingIterator<Self>  
    // ... Method definitions and more associated types  
}
```

让我们看看上面这个定义的一些重要部分。集合协议继承了 `Indexable` 和 `Sequence`，因为协议的继承不存在子类继承中的那些问题，我们可以将多个协议组合起来：

```
protocol Collection: Indexable, Sequence {
```

接下来，我们定义了两个关联类型：`IndexDistance` 和 `Iterator`。两者分别都有默认值：`IndexDistance` 是一个简单的 `Int`，`Iterator` 是 `IndexingIterator`。注意我们可以使用 `Self` 来作为 `IndexingIterator` 的泛型类型参数。这两个关联类型都有约束：`IndexDistance` 需要满足 `SignedInteger` 协议，而 `Iterator` 需要满足 `IteratorProtocol`：

```
associatedtype IndexDistance: SignedInteger = Int  
associatedtype Iterator: IteratorProtocol = IndexingIterator<Self>
```

我们在实现自己的满足 `Collection` 协议的类型时，有两种选择。我们可以使用默认的关联类型，或者我们也可以定义我们自己的关联类型（比如在集合协议一章中，我们就让 `List` 拥有了自定义的 `SubSequence` 关联类型）。如果我们决定使用默认的关联类型的话，我们就可以直接获得

很多有用的功能。比如，当迭代器没有被重写的时候，Collection 有一个带有条件的协议扩展，添加了 `makeliterator()` 的实现：

```
extension Collection where Iterator == IndexingIterator<Self> {
    func makeliterator() -> IndexingIterator<Self>
}
```

还有很多其他的条件扩展，你也可以添加你自己的扩展。我们前面说过，想要弄清楚为了实现一个协议你需要实现哪些方法是一件具有挑战的事情。因为标准库中的很多协议都有默认的关联类型和匹配这些关联类型的条件扩展，就算协议定义了很多的必要方法，你通常也只需要实现其中有限几个就可以了。对于 Sequence 需要实现哪些方法，标准库在“满足 Sequence 协议”的部分进行了文档说明。如果你正在写一个有很多方法的自定义协议，你也可以考虑在你的文档中加入类似的说明。

类型抹消

在上一节里，我们可以将 Drawing 作为一个类型来使用。但是，对于 `IteratorProtocol` 来说，因为存在关联类型，这是不可能的(至少现在还不可能)。编译器会给出这样的错误：“‘`IteratorProtocol`’ 协议含有 `Self` 或者关联类型，因此它只能被当作泛型约束使用。”

```
let iterator: IteratorProtocol = ConstantIterator() // Error
```

这就是说，将 `IteratorProtocol` 作为类型使用是不完整的。我们必须为它指明关联类型，否则单是关联类型的协议是没有意义的。

Swift 团队指出过他们想要支持泛用存在(generalized existentials)。这个特性将允许那些含有关联类型的协议也可以被当作独立的值来使用，这样它们就可以用来进行类型抹消了。如果你想要了解未来这方面会如何发展，你可以在 [Swift 泛型声明](#)一文中找到详细信息。

在未来版本的 Swift 中，我们可能可以通过类似这样的代码解决该问题：

```
let iterator: Any<IteratorProtocol where .Element == Int> = ConstantIterator()
```

不过现在，我们还不能表达这个。不过，我们可以将 `IteratorProtocol` 用作泛型参数的约束：

```
func nextInt<l: IteratorProtocol>(iterator: inout l) -> Int?
```

```
where I.Element == Int {  
    return iterator.next()  
}  
}
```

类似地，我们可以将迭代器保存在一个类或者结构体中。这里的限制也是一样的，我们只能够将它用作泛型约束，而不能用作独立的类型：

```
class IteratorStore<I: IteratorProtocol> where I.Element == Int {  
    var iterator: I  
  
    init(iterator: I) {  
        self.iterator = iterator  
    }  
}
```

这是可行方式，但是却有一个缺点，存储的迭代器的指定类型通过泛型参数“泄漏”出来了。在现有的类型系统中，我们无法表达“元素类型是 Int 的任意迭代器”这样一个概念。如果你想把多个 IteratorStore 放到一个数组里，这个限制就将带来问题。数组里的所有元素都必须有相同的类型，这也包括任何的泛型参数；我们无法创建一个数组，让它能同时存储 IteratorStore<ConstantIterator> 和 IteratorStore<FibsIterator>。

幸运的是，我们有两种方式来绕开这个限制，其中一种很简单，另一种则更高效（但是比较取巧）。将（迭代器这样的）指定类型移除的过程，就被称为**类型抹消**。

简单的解决方式是实现一个封装类。我们不直接存储迭代器，而是让封装类存储迭代器的 next 函数。要做到这一点，我们必须首先将 iterator 参数复制到一个 var 变量中，这样我们就可以调用它的 mutating 的 next 方法了。接下来我们将 next() 的调用封装到闭包表达式中，然后将这个闭包赋值给属性。我们使用类来表征 IntIterator 具有引用语义：

```
class IntIterator {  
    var nextImpl: () -> Int?  
  
    init<I: IteratorProtocol>(_ iterator: I) where I.Element == Int {  
        var iteratorCopy = iterator  
        self.nextImpl = { iteratorCopy.next() }  
    }  
}
```

现在，在 `IntIterator` 中，迭代器的具体类型（比如 `ConstantIterator`）只在创建值的时候被指定。在那之后，这个指定类型被隐藏起来，并被闭包捕获。我们可以使用任意类型且元素为整数的迭代器，来创建 `IntIterator` 实例。

```
var iter = IntIterator(ConstantIterator())
iter = IntIterator([1,2,3].makeIterator())
```

上面的代码让我们能够使用 Swift 当前的类型系统指定关联类型的约束（比如 `iter` 包括了一个以 `Int` 为元素的迭代器）。我们的 `IntIterator` 也非常容易就可以满足 `IteratorProtocol`，下面的扩展实现中，类型系统会将关联类型推断为 `Int`：

```
extension IntIterator: IteratorProtocol {
    func next() -> Int? {
        return nextImpl()
    }
}
```

实际上，通过抽象 `Int` 并添加一个泛型参数，我们可以将 `IntIterator` 进行改变，让它和标准库的 `AnyIterator` 做同样的事：

```
class AnyIterator<A>: IteratorProtocol {
    var nextImpl: () -> A?

    init<I: IteratorProtocol>(_ iterator: I) where I.Element == A {
        var iteratorCopy = iterator
        self.nextImpl = { iteratorCopy.next() }
    }

    func next() -> A? {
        return nextImpl()
    }
}
```

指定的迭代器类型 (`I`) 只在初始化函数中被使用，在那之后，它被“抹消”了。

从这次重构中，我们可以总结出一套创建类型抹消的简单算法。首先，我们创建一个名为 `AnyProtocolName` 的结构体或者类。然后，对于每个关联类型，我们添加一个泛型参数。最后，对每个方法，我们将实现存储在 `AnyProtocolName` 中的一个属性中。

对于像 `IteratorProtocol` 这样的简单协议，这只需要很少的几行代码，但是对于更复杂的协议（比如 `Sequence`），就有很多事情要做了。更糟糕的是，因为每有一个协议方法，就需要一个新的闭包与其对应，所以对象或者结构体占用的尺寸会随着协议方法数量的增加而线性增长。

标准库采用了一种不同的策略来处理类型抹消。我们先创建一个满足 `IteratorProtocol` 的简单类。它的泛型类型是迭代器的 `Element`，其实具体实现只是简单地崩溃：

```
class IteratorBox<A>: IteratorProtocol {
    func next() -> A? {
        fatalError("This method is abstract.")
    }
}
```

接下来，我们可以创建另一个类，`IteratorBoxHelper`，这个类也是泛型的。这里，泛型参数就是比如 `ConstantIterator` 这样的指定的迭代器类型。`next` 方法简单地将调用转给底层迭代器的 `next` 方法：

```
class IteratorBoxHelper<I: IteratorProtocol> {
    var iterator: I
    init(iterator: I) {
        self.iterator = iterator
    }

    func next() -> I.Element? {
        return iterator.next()
    }
}
```

现在是取巧的部分了。我们把 `IteratorBoxHelper` 变为 `IteratorBox` 的子类，`IteratorBox` 的泛型参数为 `I` 的元素类型，这样我们就可以把两个泛型参数进行约束：

```
class IteratorBoxHelper<I: IteratorProtocol>: IteratorBox<I.Element> {
    var iterator: I
    init(_ iterator: I) {
        self.iterator = iterator
    }

    override func next() -> I.Element? {
        return iterator.next()
    }
}
```

```
    }  
}
```

这么一来，我们就可以创建一个 `IteratorBoxHelper` 类型的值，并且将它当作 `IteratorBox` 来用，这有效地抹消了 `I` 的类型：

```
let iter: IteratorBox<Int> = IteratorBoxHelper(ConstantIterator())
```

在标准库中，`IteratorBox` 和 `IteratorBoxHelper` 都是私有的。`AnyIterator` 封装将这些实现细节隐藏了起来。

带有 `Self` 的协议

带有 `Self` 要求的协议在行为上和那些带有关联类型的协议很相似。最简单的带有 `Self` 的协议是 `Equatable`。它有一个(运算符形式的)方法，用来比较两个元素：

```
protocol Equatable {  
    static func ==(lhs: Self, rhs: Self) -> Bool  
}
```

要为你自己的类型实现 `Equatable` 并不难。比如，我们有两个简单的 `MonetaryAmount` 结构体，我们可以通过比较它们的属性值来比较两个值：

```
struct MonetaryAmount: Equatable {  
    var currency: String  
    var amountInCents: Int  
  
    static func ==(lhs: MonetaryAmount, rhs: MonetaryAmount) -> Bool {  
        return lhs.currency == rhs.currency &&  
               lhs.amountInCents == rhs.amountInCents  
    }  
}
```

我们不能简单地用 `Equatable` 来作为类型进行变量声明：

```
// 错误：因为 'Equatable' 中有 Self 或者关联类型的要求，  
// 所以它只能被用作泛型约束  
let x: Equatable = MonetaryAmount(currency: "EUR", amountInCents: 100)
```

这个关联类型所面临的问题是一样的：在这个（不正确）的声明中，我们并不清楚 `Self` 到底应该是什么。举个例子，如果 `Equatable` 能够被用作独立的类型，那我们就能够写这样的代码：

```
let x: Equatable = MonetaryAmount(currency: "EUR", amountInCents: 100)
let y: Equatable = "hello"
x == y
```

而 `==` 并不能接受一个货币金额和一个字符串作为输入。你要怎么比较它们两者呢？虽然不能用作独立类型，不过我们可以将 `Equatable` 用作泛型约束。比如，我们可以写一个自由函数 `allEqual`，它会检查一个数组中是否所有的元素都相等：

```
func allEqual<E: Equatable>(x: [E]) -> Bool {
    guard let firstElement = x.first else { return true }
    for element in x {
        guard element == firstElement else { return false }
    }
    return true
}
```

或者我们可以在写 `Collection` 扩展的时候对其进行约束：

```
extension Collection where Iterator.Element: Equatable {
    func allEqual() -> Bool {
        guard let firstElement = first else { return true }
        for element in self {
            guard element == firstElement else { return false }
        }
        return true
    }
}
```

`==` 运算符被定义为了类型的静态函数。换句话说，它不是成员函数，对该函数的调用将被静态派发。与成员函数不同，我们不能对它进行重写。如果你有一个实现了 `Equatable` 的类（比如 `NSObject`），你可能会在创建子类时遇到预想之外的行为。举个例子，比如下面的类：

```
class IntegerRef: NSObject {
    let int: Int
    init(_ int: Int) {
        self.int = int
    }
}
```

```
    }
}
```

我们可以嗯定义一个 `==` 运算符，用来比较两个 `IntegerRef` 的 `int` 属性是否一致：

```
func ==(lhs: IntegerRef, rhs: IntegerRef) -> Bool {
    return lhs.int == rhs.int
}
```

如果我们创建两个 `IntegerRef` 对象，我们可以对它们进行比较，结果正如我们所想：

```
let one = IntegerRef(1)
let otherOne = IntegerRef(1)
one == otherOne // true
```

然而，如果我们将它们声明为 `NSObject`，那么 `NSObject` 的 `==` 就将被使用，而这个运算符在底层使用的是 `==` 来检查引用是否指向同一个对象。除非你留意到静态派发的行为，否则结果看起来就会让人大吃一惊：

```
let two: NSObject = IntegerRef(2)
let otherTwo: NSObject = IntegerRef(2)
two == otherTwo // false
```

协议内幕

我们早先提到过，当我们通过协议类型创建一个变量的时候，这个变量会被包装到一个叫做存在容器的盒子中。我们现在来仔细研究下这个行为。

下面有两个函数，它们都接受一个满足 `CustomStringConvertible` 的值作为参数，并且返回值类型的大小。唯一的区别是其中一个函数将协议当作泛型约束来使用，而另一个则当作类型使用：

```
func f<C: CustomStringConvertible>(_ x: C) -> Int {
    return MemoryLayout.size(ofValue: x)
}
func g(_ x: CustomStringConvertible) -> Int {
    return MemoryLayout.size(ofValue: x)
}
```

```
f(5) // 8  
g(5) // 40
```

8 字节和 40 字节的尺寸差别是怎么来的？因为 f 接受的是泛型参数，整数 5 会被直接传递给这个函数，而不需要经过任何包装。所以它的大小是 8 字节，也就是 64 位系统中 Int 的尺寸。对于 g，整数会被封装到一个存在容器中。对于普通的协议，会使用**不透明存在容器** (opaque existential container)。不透明存在容器中含有一个存储值的缓冲区 (大小为三个指针，也就是 24 字节)；一些元数据 (一个指针，8 字节)；以及若干个**目击表** (0 个或者多个指针，每个 8 字节)。如果值无法放在缓冲区里，那么它将被存储到堆上，缓冲区里将变为存储引用，它将指向值在堆上的地址。元数据里包含关于类型的信息 (比如是否能够按条件进行类型转换等)。关于目击表，我们接下来会马上对它进行讨论。

对于只适用于类的协议 (也就是带有 SomeProtocol: class 或者 @objc 声明的协议)，会有一个叫做**类存在容器**的特殊存在容器，这个容器的尺寸只有一个指针的大小，因为它不需要保存值类型 (因为实现协议的类型只能是引用) 或者是元数据 (信息已经保存在类中了)：

```
MemoryLayout<UITableViewDelegate>.size // 16
```

目击表是让动态派发成为可能的关键。它为一个特定的类型将协议的实现进行编码：对于协议中的每个方法，会有一个指向特定类型中的实现的入口。有时候这被称为 vtable。某种意义上来说，在我们前面创建第一版的 AnyIterator 时，我们手动实现了一个目击表。

知道了目击表，在本章一开始的 addCircle 的奇怪行为就很容易解释得通了。因为 addCircle 不是协议定义的一部分，所以它也不在目击表中。因此，编译器除了静态地调用协议的默认实现以外，别无选择。一旦我们将 addCircle 添加为协议必须实现的方法，它就将被添加到目击表中，于是我们就可以通过动态派发对其进行调用了。

不透明存在容器的尺寸取决于目击表个数的多少，每个协议会对应一个目击表。举例来说，Any 是空协议的类型别名，所以它完全没有目击表：

```
typealias Any = protocol<>
```

```
MemoryLayout<Any>.size // 32
```

如果我们合并多个协议，每多加一个协议，就会多 8 字节的数据块。所以合并四个协议将增加 32 字节：

```
protocol Prot {}  
protocol Prot2 {}
```

```
protocol Prot3 {}
protocol Prot4 {}
typealias P = Prot & Prot2 & Prot3 & Prot4

MemoryLayout<P>.size // 64
```

性能影响

存在容器为代码调用添加了一层非直接层，所以相对于泛型参数，一般来说都会造成性能降低。除了可能更慢的方法派发以外，存在容器还扮演了阻止编译器优化的壁垒角色。大多数时候，担忧这里的性能其实是过早优化。但是，如果你想要获取最大化的性能的时候，使用泛型参数确实要比使用协议类型高效得多。通过使用泛型参数，你可以避免隐式的泛型封装。

如果你尝试将一个 [String] (或者其他任何类型) 传递给一个接受 [Any] (或者其他任意接受协议类型，而非具体类型的数组) 的函数时，编译器将会插入代码对数组进行映射，将每个值都包装起来。这将使方法调用本身成为一个 $O(n)$ 的操作 (其中 n 是数组中的元素个数)，这还不包含函数体的复杂度。同样的，大多数情况下这不会导致问题，但是如果你需要写高性能的代码，你可能需要将你的函数写为泛型参数的形式，而不是使用协议类型：

```
// 隐式打包
func printProtocol(array: [CustomStringConvertible]) {
    print(array)
}

// 没有打包
func printGeneric<A: CustomStringConvertible>(array: [A]) {
    print(array)
}
```

总结

在 Swift 中，协议是非常重要的构建单元。使用协议，我们可以写出灵活的代码，而不必拘泥于接口和实现的耦合。我们看到了两种类型的协议以及它们是如何被实现的。我们还使用了类型抹消，也研究了协议类型和带有协议约束的泛型类型之间的性能差异。由于静态和动态派发的不同，有些代码会导致意料之外的行为，我们对此也做了解释说明。

Swift 的协议对 Swift 社区产生了巨大的影响。不过我们还是想要说，请不要过度使用协议。有时候，一个独立的像是结构体或者类的类型要比定义一个协议容易理解得多，简单的类型系统也有利于增加代码的可读性。当然，在一些场景下使用协议可能会大幅提升你的代码可读性，特别是当你在处理一些很古老的 API 的时候，将这些 API 中的类型包装到协议中，往往会让你带来惊喜。

使用协议最大的好处在于它们提供了一种**最小的实现接口**。协议只对某个接口究竟应该做什么进行定义，而把使用多种类型来具体实现这个接口的工作留给我们。同样，这也让测试变得更加容易。我们只需要创建一个满足协议的简单的测试类型就可以开始测试工作了，而不必引入和建立一串复杂的依赖关系。

互用性

11

Swift 最大的一个优点是它在于 C 或者 Objective-C 混合使用时，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，我们将创建一个对 `CommonMark` 库的封装。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，它是一种很流行的用纯文本进行格式化的语法。在这个实践例子之后，我们会研究一下标准库中所提供的操作内存的工具，以及我们如何使用它们来于 C 代码进行交互。

实践：封装 CommonMark

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。用 Swift 来对一个库的接口进行封装，一般来说要比重新发明轮子简单得多，工作量也少得多。同时，封装得当的话，我们的用户将不会看到这个封装和原生实现有什么区别。我们只需要一个动态库和它的头文件，就可以开始进行封装工作了。

我们的例子中会用到 C 语言的 CommonMark 库，它是一个 CommonMark 标准的参考实现，这个实现非常高效，而且测试也很齐全。我们采用层层递进的方式进行封装，让我们可以通过 Swift 访问它。首先，我们围绕库所暴露给外界的不透明类型 (`opaque type`) 来创建一个简易的 Swift 类。然后，我们会将这个类封装到 Swift `enum` 中，并提供更符合 Swift 风格的 API。

封装 C 代码库

让我们从封装一个单独的函数开始，这个函数接受 Markdown 格式的文本，并且将它转换为一个 HTML 字符串。C 接口看起来是这样的：

```
/** 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。  
 */  
char *cmark_markdown_to_html(const char *text, int len, int options);
```

第一个参数的 C 字符串在被导入到 Swift 中时，会变为指向一系列 `Int8` 值的 `UnsafePointer` 指针。通过文档，我们知道这些值是 UTF-8 的编码单元。`len` 参数是字符串的长度：

```
func cmark_markdown_to_html
```

```
(_ text: UnsafePointer<Int8>!, _ len: Int, _ options: Int32)
-> UnsafeMutablePointer<Int8>!
```

当然了，我们想要封装的函数能接受 Swift 字符串，你可能会想到我们需要将 Swift 字符串转换为一个 Int8 指针。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 为我们自动做了这件事情。对于 len 参数我们需要特别小心，因为这里函数需要的是 UTF-8 编码的字符串的字节数，并不是字符串中的字符数。我们可以通过 Swift 字符串的 utf8 形式的 count 来获取正确的值，对于选项值 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)!
    return String(cString: outString)
}
```

在上面的实现中，我们对初始化的字符串进行了强制解包。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不必在调用 markdownToHTML 的时候关心可选值的问题了，返回的结果一定不会为 nil。

注意，在 Swift 自动将 String 原生字符串和 C 字符串之间桥接转换时，假设了你所调用的 C 函数希望的是 UTF-8 编码的字符串。这在绝大多数情况下是正确的，但是也有一些需要不同编码字符串的 C API，这时你就不能用自动桥接了。不过，通常来说转换一下也很简单。比如，如果你需要一个 UTF-16 编码点的数组的话，可以使用 Array(string.utf16)。

封装 cmark_node 类型

除了直接输出 HTML 之外，cmark 库同样提供了将 Markdown 文本解析为一个结构化的节点树的使用方式。举例来说，一串简单的文本可以被转换为一系列文本块层级节点，比如段落，引用，列表，代码块，标题等等。有些元素层级可以包含其他的元素层级，例如引用可以包含多个段落等；而有些元素层级只能包含内联元素，例如标题只能包含被斜体强调的部分。一个节点不能同时包括文本块和内联元素，比如说列表里的某个条目中的内联元素一定是被包装在其中的段落节点里的，而不会直接出现在列表节点中。

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。我们在头文件中所能看到的只有操作或者是返回 cmark_node 指针的函数。Swift 将这些指针导入为 COpaquePointer。(我们会在本章后面的

部分仔细研究标准库中像是 `OpaquePointer` 和 `UnsafeMutablePointer` 这样的很多指针类型之间到底有什么区别。)

在将来，我们可能可以使用 Swift 的“作为成员导入”这一特性来讲这些函数导入为 `cmark_node` 的方法。Apple 使用了这种方式来在 Swift 中提供了一组更加“面向对象的”Core Graphics 和 GCD (Grand Central Dispatch) 的 API。通过在 C 源码中使用 `swift_name` 进行标注，就可以将其导入为合适的 Swift 成员。不过，这个特性现在还不能用在 `cmark` 库中，它可能只能适用于你自己的 C 库。现在作为成员导入还有一些 bug，比如对于不透明指针，有时候会不好用 (而 `cmark` 使用的正是不透明指针)。

现在让我们来将一个节点封装成 Swift 类型，这样使用起来会更简单一些。我们在结构体和类中提到过，当我们创建一个自定义类型时，我们需要考虑值语义是否适用：这个类型是否应该是一个值，还是说把它当作具有同一性的实例会更合适？如果前者更合适的话，我们应该使用结构体或者枚举，而如果后者更合适，那么使用类。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。通过这样的方式封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 函数一样：一个字符串，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 COpaquePointer，它代表了这个文档节点：

```
func cmark_parse_document
(_ buffer: UnsafePointer<Int8>!, _ len: Int, _ options: Int32)
-> OpaquePointer!
```

我们将函数转换为类的初始化方法。注意 cmark_parse_document 这个 C 方法会在解析失败的时候返回 nil。在这个上下文中，nil 不代表一个可选值，而是代表了 C 的 null 指针。我们的初始化方法在遇到解析失败的时候也应该返回 nil (这里的 nil 是可选值)，所以它应该是一个可失败的初始化方法：

```
public init?(markdown: String) {
    let parsed = cmark_parse_document(markdown, markdown.utf8.count, 0)
    guard let node = parsed else { return nil }
    self.node = node
}
```

上面提到过，有很多有意思的函数可以用来操作节点。比如说，有一个函数可以返回节点的类型，用它可以来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，同时它也包含了一个成员来表示错误：

```
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 `Int32` 值的结构体。除此之外，对原来枚举中的每个成员，Swift 还会为它生成一个顶层的变量。只有那些 Apple 的 Objective-C 框架中用 `NS_ENUM` 宏来标记的枚举会被导入为原生的 Swift 枚举。另外，你可以将你的 `enum case` 标注为 `swift_name` 来让它们成为成员变量：

```
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 }
```

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` 函数转变为一个我们的类中的计算属性：

```
var type: cmark_node_type {
    return cmark_node_get_type(node)
}
```

现在，我们可以用 `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。如果一个节点是列表，那么它的列表属性将会是“无序列表”和“有序列表”之一。其他所有非列表的节点的列表类型为“无列表”。再一次，Swift 将对应的 C 枚举映射为一个结构体，其中每种情况都是一个顶层变量，我们可以用类似的方法来将它封装到 Swift 属性中。在这里，我们为这个属性提供了 `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) }  
}
```

对于其他所有的节点属性，都有相似的函数，比如标题的层级，代码块的信息，以及链接的 URL 和文字等。这些属性通常只对特定类型的节点有意义，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) 来解决这个问题。这在 C 代码库的 API 中是一个弱点，而在 Swift 中我们可以进行更好的建模。我们会在下面继续讨论这个话题。

有些节点可以拥有子节点，为了对这些子节点进行枚举，CommonMark 库提供了 `cmark_node_first_child` 和 `cmark_node_next` 函数。我们想要在我们的 `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，不断使用 `cmark_node_first_child` 或者 `cmark_node_next`，来将子节点加入到数组中，直到这两个函数返回代表列表结尾的 `nil`。再一次提醒，这里的 `nil` 不是可选值的 `nil`，它是 C 的 `null` 指针在 Swift 中的对应。因为这个原因，我们并不能使用像是 `while let` 这样的可选值绑定语法，而需要在循环中手动地检查 `nil`：

```
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` 函数或者 `AnySequence` 类型) 而不是数组。不过，这里存在一个问题，因为子节点的元素们是延迟返回的，在创造这个序列和最后使用这个序列的过程中，节点的结构可能已经发生了改变。这种情况下，使用序列的代码可能会返回错误的值，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的话，返回一个数组会是更安全的选择：

```
var childrenS: AnySequence<Node> {  
    return AnySequence { () -> AnyGenerator<Node> in  
        var child = cmark_node_first_child(self.node)  
        return AnyGenerator {  
            let result: Node? = child == nil ? nil : Node(node: child)  
            child = cmark_node_next(child)  
        }  
    }  
}
```

```
        return result
    }
}
}
```

有了这个对节点进行了封装的类，现在通过 Swift 来访问 CommonMark 库生成的抽象语法树就方便多了。现在我们不再需要调用像是 `cmark_node_get_list_type` 这样的函数，而只需要通过使用 `node.listType` 就可以了，这带给我们自动补全和类型安全等诸多好处。不过，我们还有改进的余地。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。

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。使用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。我们将分别为所有允许的内联元素以及所有的文本块层级元素创建枚举。通过这么做，我们能将 CommonMark 文档结构化。举例来说，一个纯文本元素 `Text` 将只存储一个 `String`，而表示强调的 `Emphasis` 节点则包含了一个其他内联元素的数组。这些枚举将会是我们的库的公开接口，这可以将 `Node` 类转换为只在库的内部进行使用的实现细节：

```
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?)
}
```

类似地，段落和标题只能包含内联元素，而引用则可以包含其他的文本块层级元素。列表被定义为一个 `Block` 元素的数组，每个 `Block` 代表了一个列表中的条目：

```
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
}
```

ListType 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，我们通过将 Node 节点类的数据转换为枚举的表示形式后，也就可以将它们看作是值了。我们遵循 [API 设计准则](#) 的建议，通过初始化的方式进行类型转换。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 InlineElement 枚举值，另一对则依据枚举值重新构建对应的 Node。这让我们可以用任意的 InlineElement 或者 Block 值来重新构建一个 CommonMark 文档。之后我们就可以把文档传递给 CommonMark，让它来渲染 HTML, man 页面，或者是转回 Markdown 文本。

我们先来写将 Node 转换为 InlineElement 的初始化方法。我们通过使用 switch 语句可以对节点的类型进行选择，然后构建出对应的 InlineElement 值。比如说，对于一个文本节点，我们将节点的字符串内容取出，它是 cmark 库中的节点上的 literal 属性。因为我们知道文本节点一定会有值，所以我们可以对 literal 进行安全的强制解包，而不必考虑其他类型的节点上这个属性可能为 nil 的情况。而对于斜体强调和粗体的节点来说，它们只有子节点，而没有 literal 值。要解析像是强调和粗体的节点，我们需要对这类节点的子节点进行枚举映射，递归地调用初始化方法将它们进行转换。为了避免重复的代码，我们创建了一个 inlineChildren 内联函数，在必要的时候可以进行调用。switch 的 default 语句应该永远不会被执行，如果发生了这种情况，我们选择杀死程序。不使用可选值返回或者 throws 是因为按照 Swift 的约定，它们是用来表达期望中的错误的，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_SOFTBREAK:
    self = .softBreak
case CMARK_NODE_LINEBREAK:
    self = .lineBreak
case CMARK_NODE_CODE:
    self = .code(text: node.literal!)
case CMARK_NODE_HTML_INLINE:
    self = .html(text: node.literal!)
case CMARK_NODE_CUSTOM_INLINE:
    self = .custom(literal: node.literal!)
case CMARK_NODE_EMPH:
    self = .emphasis(children: inlineChildren())
case CMARK_NODE_STRONG:
    self = .strong(children: inlineChildren())
case CMARK_NODE_LINK:
    self = .link(children: inlineChildren(), title: node.title, url: node.urlString)
case CMARK_NODE_IMAGE:
    self = .image(children: inlineChildren(), title: node.title, url: node.urlString)
default:
    fatalError("Unrecognized node: \(node.typeString)")
}
}
}

```

我们可以使用同样的方式将文本块层级元素进行转换。我们需要注意，根据节点类型的不同，一个文本块层级元素可以包含内联元素，列表条目或者是其他文本块层级元素。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层级元素组成的数组：

```

extension Block {
init(_ node: Node) {
    let parseInlineChildren = { node.children.map(Inline.init) }
    let parseBlockChildren = { node.children.map(Block.init) }
    switch node.type {
        case CMARK_NODE_PARAGRAPH:
            self = .paragraph(text: parseInlineChildren())
        case CMARK_NODE_BLOCK_QUOTE:

```

```

    self = .blockQuote(items: parseBlockChildren())
case CMARK_NODE_LIST:
    let type = node.listType == CMARK_BULLET_LIST ?
        ListType.unordered : ListType.ordered
    self = .list(items: node.children.map { $0.listItem }, type: type)
case CMARK_NODE_CODE_BLOCK:
    self = .codeBlock(text: node.literal!, language: node.fenceInfo)
case CMARK_NODE_HTML_BLOCK:
    self = .html(text: node.literal!)
case CMARK_NODE_CUSTOM_BLOCK:
    self = .custom(literal: node.literal!)
case CMARK_NODE_HEADING:
    self = .heading(text: parseInlineChildren(), level: node.headerLevel)
case CMARK_NODE_THEMATIC_BREAK:
    self = .thematicBreak
default:
    fatalError("Unrecognized node: \(node.typeString)")
}
}
}

```

现在，只要有一个文档级别的 Node，我们就可以很容易地将它转换为一个由 Block 元素组成的数组了。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。因为按照定义，值并不在意它们是如何被创建的，我们可以通过代码直接从头开始创建 Markdown 的语法树，而完全不必通过使用 CommonMark 库。节点的类型现在也更加清晰了，你很难搞错节点类型和它们对应的值的有效性，你不再会意外地访问到列表的标题这样并不存在的属性，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
    }
}
}
```

在我们继续更多操作之前，让我们现在实现逆向转换，也就是把一个 Block 转换回 Node。我们之所以要实现这个逆向转换，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 语法规树中生成 HTML 或者是其他的文本格式时，它只接受 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，依据指定的类型和子节点来从头开始创建一个新的 cmark_node。还记得我们写过一个_deinit 并在其中释放了以该节点为根节点的树（包括其下的子节点们）的内存么。这个 _deinit 将会保证我们在这里初始化的内容可以被正确地释放：

```
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)
        }
    }
}
```

我们经常会需要创建只有文本的节点，或者是有一系列子节点的节点。所以，我们写了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，我们可以来写这两个 node 转换函数了。使用我们刚才定义的初始化方法，事情就会变得很直接。我们只需要对元素类型进行判断，然后根据类型来创建节点就行了。这里是内联元素的转换方法：

```
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 .code(let text):
                self.init(type: CMARK_NODE_CODE, literal: text)
            case .strong(let children):
                self.init(type: CMARK_NODE_STRONG, 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
            case let .image(children, title, url):
                self.init(type: CMARK_NODE_IMAGE, elements: children)
                self.title = title
                urlString = url
            case .softBreak:
                self.init(type: CMARK_NODE_SOFTBREAK)
            case .lineBreak:
                self.init(type: CMARK_NODE_LINEBREAK)
        }
    }
}
```

为文本块层级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
            case .html(let text):
                self.init(type: CMARK_NODE_HTML_BLOCK, literal: text)
            case .custom(let literal):
                self.init(type: CMARK_NODE_CUSTOM_BLOCK, literal: literal)
            case let .heading(text, level):
                self.init(type: CMARK_NODE_HEADING, elements: text)
                headerLevel = level
            case .thematicBreak:
                self.init(type: CMARK_NODE_THEMATIC_BREAK)
        }
    }
}
```

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)，我们将转换从底层的 C API 中抽象出来。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。

低层级类型概览

在标准库中有不少类型提供了低层方式来访问内存。这些类型的数量可能会有很多，但是它们的命名都很统一。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** 的类型不提供自动的内存管理 (这个 **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** 类型表示作用于一连串的多个元素，而非一个单独的元素上。
- **raw** 类型包含无类型的原始数据，它和 C 的 **void*** 是等价的。在类型名字中不包含 **raw** 的类型的数据是具有类型的。
- **mutable** 类型允许它指向的内存发生改变。

如果你需要单纯的无类型存储，而并不需要与 C 交互，你可以使用 **ManagedBuffer** 来申请内存。这也是 Swift 的集合在低层使用的管理内存的方式。它由一个单独的 **header** 值 (用来存储像是元素个数这样的数据)，以及一连串的存储元素的内存组成。另外，它还有一个 **capacity** 属性，**capacity** 与实际元素个数不同，比如一个 **count** 是 17 的数组可能会有一个 **capacity** 为 32 的缓冲区，也就是说，数组在申请更多内存之前，还可以容纳 15 个元素。这种类型还有一个变种，叫做 **ManagedBufferPointer**，我们在此就不做讨论了。

有时候你需要手动进行内存管理。比如在一个 C API 中，你可以需要将一个对象当作上下文传给某个函数。然后，C 并不知道 Swift 的内存管理机制。如果 API 是同步的，你可以传入这个 Swift 对象，再将它转换回来，一切都没问题 (我们会在下一节看到具体细节)。但是，如果你的 API 是异步的，你就需要手动持有和释放这个对象，因为不这么做的话，在对象离开作用域后

它就会被销毁。我们可以使用 `Unmanaged` 类型来完成这个任务，它有 `retain` 和 `release` 函数，这两个函数和初始化方法一样，可以更改或者保持当前的引用计数。

`UnsafePointer` 是最简单的指针类型，它与 C 中的 `const` 指针类似。这是一个泛型类型，它将所指向的内存的数据类型进行了泛型化。你可以通过其他类型的指针，使用 `UnsafePointer` 的初始化方法来创建一个不安全指针。Swift 还支持一种特殊的函数调用的语法，通过在任意类型正确的可变变量前面加上 `&` 符号，可以将它们转变 **in-out 表达式**：

```
var x = 5
func fetch(p: UnsafePointer<Int>) -> Int {
    return 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值被传回给调用者。Swift 在幕后创建和传递给函数的指针只在函数调用期间是可以保证有效的。切记不要常识返回这个指针，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。我们在之前就指出过，一个 `Unsafe` 类型不提供任何内存管理，所以我们这里需要依赖的事实是在调用 `fetch` 时，`x` 依然在作用范围内。

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，并且改变内存的值，这种改变将通过 **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
```

除了使用 **in-out 表达式**，你也可以通过申请内存的方式来直接使用 `UnsafeMutablePointer`。Swift 中申请内存的方式和 C 中的规则很相似：在申请内存后，你必须对其进行初始化，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(to: 42, count: 2)
```

```
z.pointee // 42
// 指针计算:
(z+1).pointee = 43
// 下标:
z[1] // 43
z.deallocate(capacity: 2)
// 垃圾内存
z.pointee // 42
```

在 C API 中，一个指向未指定元素类型的字节序列的指针 (比如 `void*` 或者 `const void*`) 是很常见的。在 Swift 中与之等价的是 `UnsafeMutableRawPointer` 和 `UnsafeRawPointer` 类型。使用 `void*` 或者 `const void*` 的 C API 将会被导入为这样的类型。通常，你可以将这些类型通过它们的一个实例方法转换为 `Unsafe[Mutable]Pointer` 或者其他确定类型的指针，这类方法有 `assumingMemoryBound(to:)`、`bindMemory(to:)` 或 `load(fromByteOffset:as:)`。

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。你可以通过初始化函数来将不透明指针转换为其他的指针。

在 Swift 中，我们经常会使用 `Array` 类型来存储一系列连续值。在 C 里，一个序列通常被用一个指向首个元素的指针以及元素的个数来表示。如果我们想要将这样的序列作为集合来使用，我们可以将序列转为一个 `Array`，但这会导致元素被复制。通常来说这是一件好事 (因为一旦这些元素存在于数组中，它们内存就将由 Swift 运行时进行管理)。然而，有时候你却不想为每个元素创建复制。对于那种情况，我们可以使用 `Unsafe[Mutable]BufferPointer` 类型。你通过一个指向起始元素的指针和元素个数的数字来初始化这个类型。缓冲区指针让 Swift 与 C 的集合协同工作变得容易很多。

最后，在未来版本的 Swift 中，会加入 `Unsafe[Mutable]RawBufferPointer` 类型。有了这个类型的话，将那些原始数据当作集合来处理就会方便得多 (它们可以在低层提供与 `Data` 和 `NSData` 等价的类型)。

函数指针

我们先来看看如何封装标准 C 库中的 `qsort` 排序函数。这个函数被导入 Swift 的 Darwin (或者如果你在用 Linux 的话，则是 Glibc) 中时，类型是下面这样的：

```
swift public func qsort( _ __base: UnsafeMutableRawPointer!, _ __nel: Int, _ __width: Int, _ __co
```

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。

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：

```
func qsortStrings(array: inout [String]) {
    qsort(&array, array.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。因为这个指针其实是一个 UnsafeMutableRawPointer 类型 (在 C 声明中，它是 void *base)，所以我们需要添加 & 前缀。如果该函数的参数在 C 中的声明是 const void *base 的话，那么导入到 Swift 时将会是一个不可变值，这种情况下，我们可以不需要 & 前缀。这和使用 Swift 函数的 inout 参数的规则是一样的。
- 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 平台恰好相同，但是这并不是说对于其他类型都会是相同的，比如 MemoryLayout<Character>.size 是 9，而 MemoryLayout<Character>.stride 的值是 16。当你将代码从 C 转换为 Swift 时，对于 C 中的 sizeof，在 Swift 中使用 MemoryLayout.stride 会更加合理。

→ 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 可以自动将 Swift 的函数进行桥接，所以我们只需要传递一个符合类型签名的闭包或者函数就可以了。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，编译器将只允许你提供不捕获任何局部变量的闭包作为最后一个参数。`@convention(c)` 这个参数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，我们知道我们得到的是一个 `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。

为另外的类型创建另一个封装非常容易，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。我们这么做的时候会遇到一个 C 函数指针带来的限制。在我们编写本书的时候，Swift 编译器在遇到下面的代码时会产生一个段错误 (`segfault`)。不过就算没有这个错误，下面的代码也是不可能编译成功的：它的闭包捕获了外界的东西。具体来说，它捕获了比较和判等的操作符，这对于每种泛型类型来说都是不同的。我们对此似乎无能为力，由于 C 的工作方式，我们遇到的是一个无法绕开的限制：

```
// 语法有效，代码无效
extension Array where Element: Comparable {
    mutating func quicksort() {
        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。如果这个函数没有什么上下文依赖的话，那么这个地址将会是静态的，并且在编译的时候就已经确定了。然而，对于泛型函数来说，其实我们传入了泛型类型这个额外的参数。而其实对于泛型函数来说，它是没有确定的地址的。闭包的情况与此类似。就算编译器能够对闭包进行重写，让它可以作为一个

函数指针被传递，其中的内存管理的问题也无法自动完成，我们没有办法知道应该在何时释放这个闭包。

实际使用的时候，这也是很多 C 程序员所面临的问题。在 macOS 上，有一个 `qsort` 的变种，叫做 `qsort_b`。与 `qsort` 那样接受函数指针的方式不同，它接受一个 `block` 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，`qsort_b` 在大多数平台上都是不可用的。另外，除开 `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不安全的 `void` 指针作为参数，并且在调用回调函数时将这个指针再传递回给调用者。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 `UnsafeRawPointer` 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,  
    _: 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` 来在 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` 指针。在我们的例子中，我们想要把比较用的闭包传递过去。在传递参数时，通过在一个 `var` 变量前面加上 `&` 符号，我们就可以自动地将它转换为 `UnsafeRawPointer`，所以我们要做的就是将上面传入 `qsort_b` 的那个用来比较的闭包用一个叫做 `thunk` 的变量存储起来。然后在当我们调用 `qsort_r` 时，把 `thunk` 变量的引用传递进去。在回调中，我们可以将得到的 `void` 指针转换回它原本的类型 `Block`，然后就可以简单地调用闭包了：

```
typealias Block = (UnsafeRawPointer?, UnsafeRawPointer?) -> Int32  
func qsort_block(_ array: UnsafeMutableRawPointer, _ count: Int,  
    _ width: Int, f: @escaping Block)  
{  
    var thunk = f  
    qsort_r(array, count, width, &thunk) { (ctx, p1, p2) -> Int32 in
```

```
let comp = ctx!.assumingMemoryBound(to: Block.self).pointee
return comp(p1, p2)
}
}
```

通过使用 `qsort_block`，我们现在可以重新定义 `qsortWrapper` 函数，并且为 C 标准库中的 `qsort` 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 x = [3,1,2]
x.quicksort()
x // [1, 2, 3]
```

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，因为使用 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。