objc.io | objc 中国

Swift 进阶

Chris Eidhof, Airspeed Velocity 著
王巍 译

英文版本 1.0 (2016 年 3 月)，中文版本 1.0 (2016 年 6 月)

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

前言

1

本书介绍

我从 2005 年就开始从事 OS X 的开发工作，2008 年的时候我转到了 iOS。在 2011 年我和别人合作写了一本 iOS 进阶书籍的时候，我已经有多年的开发经验，那时候 Objective-C 的最佳实践已经广为人知，如果遇到问题，我也能够向更有经验的同事和前辈进行请教。

现在 Swift 已经两岁了，并且还在飞速发展着。半年以前的最佳实践可能放到现在已然是明日黄花。那么，真的有人能写出一本关于 Swift 进阶方面的书籍吗？

答案是肯定的，但是这对书的作者有一定的要求：他必须得是从 Swift 诞生起就开始学习的，他必须积极参与到社区中去，他必须对 Swift 知其**然**更要知其**所以然**，他必须能够回答艰深刁钻的问题，同时，他也需要能**提出**艰深刁钻的问题。

Swift 远比它的文档所涉及的内容要广泛。书的作者还需要时刻关注论坛和社交媒体上大家都在讨论什么。他们最好还能够深入到标准库中，去了解 Swift 是如何工作的。除此之外，他们需要紧跟 Apple 的步伐，来区别什么是 “非 Swift 风格”，什么是 “Swift 还未实现” 的东西。这些正是 Airspeed Velocity 和 Chris 所做的事情，正因如此，我十分推荐你读读看他们在本书里说了些什么。

如果你看过我关于 Swift 的演讲的话，你会知道 Airspeed Velocity 是我最喜欢的关于 Swift 的博客。我非常欣赏他对细节一探究竟的那份执着，而这本书里也包含了同样的执着。和他的博客一样，这本书并不忌惮于批评 Swift 的不足之处：对一件事情，有时候在 Swift 中没有很好的办法，有时候 Swift 只能给出差强人意的解决方式。

在探寻 Swift 函数式的时候，Chris 的博客给了我很大帮助。对于函数式的话题，另一本书《函数式 Swift》给出了更完整的阐释，我十分推荐你也看看那本书。通过阅读该书，你会知道 Swift 并不是 Haskell 那样的函数式语言，在 Swift 中我们有很多的高阶函数和不可变的数据结构，但是我们也有循环以及 **var** 变量这样针对特定场景的工具。那本书可以帮助你在使用 Swift 时扬长避短，并让你能够找到一个问题在 Swift 中的解决方式。

Swift 为我们开启了崭新的世界，它充满了新的规则，一切都欣欣向荣。与此同时，这门语言还有很多发展的空间，你的贡献可能会对语言本身和社区产生巨大的影响。我们仍然在探寻如何写好 Swift，以及哪种模式能够更好地工作。现在就来看看 Airspeed Velocity 和 Chris 的观点吧，我希望你也能参与到这场讨论中来。

Rob Napier (@cocoaphony)

2015 年 11 月

译序

Swift 问世两年以来，这门语言带给了我们很多惊喜。在 WWDC 14 上的横空出世，简洁的语法和安全的天性令人称赞。WWDC 15 中面向协议编程的思想的提出和 Swift 开源的消息，让这门语言充满希望。今年，Swift 3.0 所带来的变革，在开源社区推动下的日渐完善，更使得这门语言魅力无穷，它的未来充满了无数可能。

Swift 语言本身并没有太多难点，先进的标准库更是集各门语言之所长，兼顾了语法简洁优雅和书写安全便捷的优点。作为一门 app 开发语言，在发布两年中，Swift 已经被证实了可以很好地完成它的工作，Swift 的开发者也纷纷表达了使用 Swift 来创造程序时的愉悦。而在跨平台和服务器方向，Swift 似乎也具有很大潜力 — 社区里已经有非常多的开发者在进行相关方向的工作。看起来，作为一门推广如此迅速的语言，Swift 的简单是毋庸置疑的。那么，对于一门“简单”的语言，像本书这样的以进阶内容为主的书籍真的值得一看吗？

答案显然是肯定的，否则我们也不会花费力气来写作和翻译这本书。表面上使用的简单并不意味着 Swift 是一门没有内涵的语言，恰恰相反，Swift 的背后蕴藏了很多先进的设计思想，这些思想都是当代软件工程中无数睿智的先行者所不断总结和实践的结晶。本书会告诉你 Swift 中安全和先进特性的背后考量和实现细节：可选值到底是什么？值类型和不变性真的如此重要吗？为什么说 Swift 的字符串设计非常漂亮？要怎么从更深层次理解泛型和接口的思想？同时，本书也会向你推荐一些在这些思想的指导下最佳实践。理解这些内容将有助于你写出更加优秀的 Swift 代码，从而构建出更加稳定和可靠的项目。

想必你已经迫不及待要徜徉于 Swift 进阶话题中了，就让我们从对 Swift 的概述和宏观理解开始吧。

王巍

2016 年 6 月

介绍

2

《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经常在变化，它常常抛弃一切约定，又去遵守另一些约定。

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，很多使用 Swift 来实现 Java 或者 C# 中的设计模式的书籍也纷纷出版，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。你可能很长时间都不会需要去调用 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 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 OS X 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，但是我们希望这本书也可以对非 Apple 平台的程序员有所帮助。

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，有一些主题是 Swift 所特有的：

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 这样类似于 Ruby 和 Python 的方式来编码，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，因为你可以实际上会变成使用 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，也更合适用来的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 发布并引入协议接口的时候，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：

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

Swift 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。这些模板代码往往会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得以避免重复，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 最近在他关于创造 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：

```
let uhOh = someArray.withUnsafePointer { 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...
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不太熟悉，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” 某个变量的值。两个引用可能会指向同一个变量，这引入了一种可能性，那就是这个变量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CommonMark 和互用性的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。

这里我们会遇到另一件复杂的事情。虽然 struct 是值类型，但是它的成员可以由其他多种类型组成，也包含那些引用类型。这就是说，当把一个值类型赋值给另一个值类型的时候，对其中的引用类型所做的复制是一个**浅复制 (shallow copy)**。这种复制会对引用进行复制，但是不会对引用所指向的内容进行复制。

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)**。比如，如果一个 Swift 数组的元素只包含结构体的话，那么它就具有值语义。如果一个数组包含有对象，那么实际上数组中的元素是对这些对象的引用。因此当复制包含对象的数组的时候，只有对象的引用被复制了，而这些对象本身是不会被复制的。这是很取巧的一个定义，我们将在结构体和类中深入研究这种行为。

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。这意味着虽然它们是类，但是它们依然具有值语义 (因为它们即使被到处使用也从不改变)。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。

Swift 标准库中的集合类型都是结构体，也拥有值语义。在底层，它们是通过引用来实现的。在下一章中，我们就将解释这和 Objective-C 里 Foundation 的对应类的不同之处。Swift 结构体使用了名为 “写时复制” (copy-on-write) 的技术来保证操作的高效，我们会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，对于含有引用的结构体，写时复制的特性并不是直接具有的；结构体的作者必须进行对应的实现。写时复制也不是创建值语义的唯一方式，不过它确实是最常用的方式。

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 (比如把这个函数被作为其他函数的返回值返回时)，它将能够 “捕获” 局部变量。这些局部变量将存在于函数中，不会随着局部作用域的结束而消亡，函数也将持有它们的状态。这种行为的变量被称为 “闭合变量”，我们把这样的函数叫做**闭包 (closure)**。

函数可以通过 **func** 关键字来定义，也可以通过 { } 这样的简短的**闭包表达式 (closure expression)** 来定义。有时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实际执行的代码。

类或者协议上的方法可能是**动态派发 (dynamically dispatched)** 的。编译器在编译时不需要知道哪个函数将被调用。在 Swift 中，这种动态特性要么由 vtable 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：

→ 可读性是最重要的。保持简短可以让你的代码更容易被理解。

→ 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，函数和变量使用小写字母开头，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。

→ 试着去使用不可变值：除非你需要改变某个值，否则都应该使用 **let** 来声明变量。不过如果能让代码更加清晰高效的话，也可以选择使用可变的版本。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。

→ 最后可能是程序员中争吵最厉害的问题：我们在大括号换行的处理上选择和 Swift 官方风格一致，那就是大括号不换行：} **else** {。

集合类型

3

数组和可变性

在 Swift 中最常用的集合类型非数组莫属。数组是一个简单的列表，举个例子，要创建一个数字的数组，我们可以这么写：

```
let fibs = [0, 1, 1, 2, 3, 5]
```

我们可以在数组上执行像是 isEmpty 或是 count 这样的常见操作。想要获取一个数组的第一个元素和最后一个元素，可以使用 first 和 last，如果是空数组的话，两者会返回 **nil**。你还可以使用一个指定的索引 (index) 通过下标的方式直接访问数组中的某个元素。下标访问不是安全的操作，在获取元素前，你需要确认或者检验这个索引是否在数组的范围之内。如果下标越界，你的程序就将崩溃。

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.appendContentsOf([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。注意类的实例这个特殊场景，因为类的对象值是一个引用，使用 **let** 定义的对象时，这个引用值永远不会发生变化，但是这个引用所指的内存 (也就是说实例变量本身) 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。

Swift 数组具有值语义，它们从不会在不同的地方被共享使用。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，这个数组会被复制。这在创建变量、将数组传递给函数、或者更多的一些场景中都会发生。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)
```

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]。

对比一下 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 NSArry，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.insertObject(4, atIndex: 3)
b // 现在包括元素 4
```

正确的方式是在赋值 b 时，对 a 先进行复制：

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

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

a.insertObject(4, atIndex: 3)
b // 仍然是 [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。每次都进行复制自然很安全，但是如果这么做的话，又可能造成很多不必要的浪费。

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，并告诉你如何为你自己的类型实现写时复制特性。

数组变形

2014 年 Swift 发布的时候引起了一阵解释使用 map， filter 和 reduce 好处的风潮。对此我们还有一些想要补充的内容，我们会在下面进行简单描述。

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)
}
```

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 squared = fibs.map { fib in fib * fib }
```

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

其次，squared 将由 map 的结果得到，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 SequenceType 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 SequenceType 的话题)：

```
extension Array {
    func map<U>(transform: Element->U) -> [U] {
        var result: [U] = []
        result.reserveCapacity(self.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，U 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 U 究竟是什么，它们可以是任意类型。它们的实际类型会留给调用者来决定。

> 实际上，这个函数的签名应该是
>
> ```
> func map<U>(@noescape transform: Element throws -> U) rethrows -> [U]
> ```
>
> 我们会在函数一章里谈到 **@noescape**，在错误处理一章里谈到 **rethrows**。这两个关键词并不是必须的，它们的存在是为了让调用者使用这个函数的时候更加方便。

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。标准库中总共有 13 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
- → **sort** 和 **lexicographicCompare** — 两个元素应该以怎样的顺序进行排列
- → **indexOf** 和 **contains** — 元素是否符合某个条件
- → **minElement** 和 **maxElement** — 两个元素中的最小/最大值是哪个
- → **elementsEqual** 和 **startsWith** — 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.indexOf("foo") 这样的表达方式，也要比 array.indexOf { $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.uppercaseString < $1.name.uppercaseString } 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。
- → **allMatch** 和 **noneMatch** — 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** — 计算满足条件的元素的个数，和 filter 相似，但是不会构建数组。

- → **indicesOf** — 返回一个包含满足某个标准的所有元素的索引的列表，和 indexOf 类似，但是不会在遇到首个元素时就停止。
- → **takeWhile** — 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。
- → **dropWhile** — 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 takeWhile 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。

有时候你可能会发现你写了好多次同样模式的代码，比如这样：

```
let someArray: [SomeObject] = []

var object: SomeObject?
for oneObject in someArray where oneObject.passesTest() {
    object = oneObject
    break
}
```

在这种情况下，你可以考虑为 SequenceType 添加一个小扩展，来将这个逻辑封装到 findElement 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 SequenceType {
    func findElement (match: Generator.Element->Bool) -> Generator.Element? {
        for element in self where match(element) {
            return element
        }
        return nil
    }
}
```

现在我们就能把代码中的 **for** 循环换成 findElement 了：

```
let object = someArray.findElement { $0.passesTest() }
```

这么做的好处和我们在介绍 map 时所描述的是一样的，相较 **for** 循环，findElement 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 findElement 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 object。

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 object = someSequence.findElement({ $0.passesTest() })
    else { return }
// 在函数中使用找到的 object
```

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 **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<U>(initial: U, combine: (U, Element) -> U) -> [U] {
        var running = initial
        return self.map { next in
            running = combine(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, combine: +)
```

```
[1, 3, 6, 10]
```

Filter

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

```
fibs. filter { num in num % 2 == 0 }
```

为了少写一些代码，我们可以使用 Swift 内建的用来代表参数的简写 $0。这让我们可以不用写出 num 参数，而将上面的代码重写为：

```
fibs. filter { $0 % 2 == 0 }
```

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 (includeElement: Element -> Bool) -> [Element] {
```

```
        var result: [Element] = []
        for x in self where includeElement(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 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 SequenceType {
    public func allMatch(predicate: Generator.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 !self.contains { !predicate($0) }
    }
}
```

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
}
```

reduce 函数对应这种模式，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：

```
fibs.reduce(0) { total, num in total + num }
```

在 Swift 中，所有的运算符都是函数，所以我们也可以把上面的例子写成这样：

```
fibs.reduce(0, combine: +)
```

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，这个字符串每行是原来数组中的一个元素：

```
fibs.reduce("") { str, num in str + "\(num)\n" }
```

reduce 的实现是这样的：

```
extension Array {
    func reduce<U>(initial: U, combine: (U, Element) -> U) -> U {
        var result = initial
        for x in self {
            result = combine(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<U>(transform: Element -> U) -> [U] {
        return reduce([])  {
            $0 + [transform($1)]
        }
    }

    func filter2 (includeElement: Element -> Bool) -> [Element] {
        return reduce([])  {
            includeElement($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 $O(n)$。

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) -> [NSURL]
```

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汇总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 for 循环进行展平 (flatten)，将它归并到一个单一数组中去：

```
let nestedArrays = markdownFiles.map(extractLinks)
var links: [NSURL] = []
for array in nestedArrays {
```

```
    links.appendContentsOf(array)
}
```

这个 **for** 循环看起来和我们在介绍 map 之前所习惯写的循环十分相似，所以也有那些 **for** 循环的种种缺点。我们可以用 flatMap 来替代这段代码。flatMap 所做的事情和 map 十分相似，不过前者还会将结果数组展平。这样一来，markdownFiles.flatMap(links) 就将把所有 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 十分相似，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 appendContentsOf 而不是 append，这样它将能把结果展平，

```
extension Array {
    func flatMap<U>(transform: Element -> [U]) -> [U] {
        var result: [U] = []
        for x in self {
            result.appendContentsOf(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 allCombinations = suits.flatMap { suit in
    ranks.map { rank in
        (suit, rank)
    }
}
```

使用 forEach 进行迭代

在数组上还有一个叫做 forEach 的函数 (它被定义在 SequenceType 上，而 Array 实现了这个协议)。它和 **for** 循环的作为非常类似，不过它们之间有些细微的不同。技术上来说，我们可以不暇思索地将一个 **for** 循环替换为 forEach：

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

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

如果你想要对集合中的每个元素都调用一个函数的话，forEach 就非常有用。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。

因为 forEach 的实现使用了闭包，所以当你在重写包含有 **return** 的 **for** 循环时，可能遇到一些意想不到的情况。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 indexOf(element: Element) -> Int? {
        for idx in self.indices where self[idx] == element {
            return idx
        }
        return nil
    }
}
```

因为 forEach 不能和 **where** 一起直接使用，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 indexOf_foreach(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。有些时候，编译器会为此给出警告，不过有时候又不会。在本书书写的过程中，这样的代码并不会触发警告。

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 }
}
```

在你阅读上面的代码时，你可能没注意到这段代码会将输入数组中的所有数字都打印出来。**return** 并没有停止循环，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。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：

```
fibs[1..<fibs.endIndex]
```

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一个 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])
```

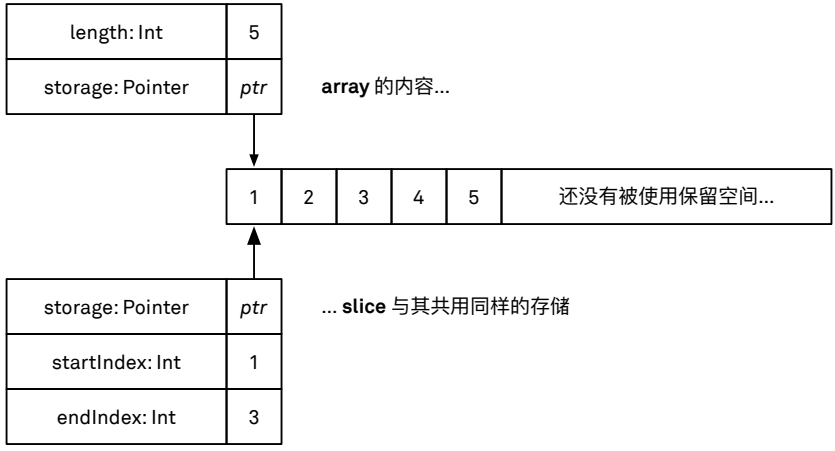

图 3.1: 数组切片

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。桥接时的转换会在两个方向发生：Swift 的数组大多数时候可以被转换为 NSArray，而 NSArray 则一定能被转换为 Swift 数组。

想要从 Swift 数组创建一个 NSArray，前提条件是其中的元素都必须能够转换为 AnyObject。Swift 对象都是可以转换为 AnyObject 的，除此之外，有一部分结构体类型也可以被转换为 AnyObject。比如说 Int 和 Bool 都可以被自动转换为 NSNumber。但其他有一些类型不行，比如 CGPoint 原本应该可以转换为 NSValue，但是现在不行。不过好消息是，编译器会在它不知道如何进行桥接的时候向你求助。

设想我们创建了一个 Swift 的整型数组，接着将它转换为 NSArray，并且检查其中元素的类型。我们可以验证里面的整数被转换为了 NSNumber 类型：

```
var x = [1, 2, 3]
let z: NSArray = x
z[0] is NSNumber

true
```

字典和集合

Swift 中另一个关键的数据结构是字典。字典指的是包含键以及它们所对应的值的数据结构。在一个字典中，每个键都只能出现一次。举个例子，我们现在来为设置界面构建一些基础代码。我们先定义了 Settings 协议，遵守这个协议的类型需要提供一个能够设定它的 UIView 对象。比如对于 String，我们返回 UITextField，而对于布尔值，我们返回 UISwitch：

```
protocol Setting {
    func settingsView() -> UIView
}
```

现在我们可以定义一个由键值对组成的字典了。这个字典中的键是设置项的名字，而值就是这些设定所存储的值。我们使用 **let** 来定义字典，也就是说，我们之后不会再去更改它。

```
let defaultSettings: [String:Setting] = [
    "Airplane Mode": true,
    "Name": "My iPhone",
]
```

想要得到设置字典的值，我们需要使用下标 (比如 defaultSettings["Name"])。字典将会进行查找，并返回一个**可选值**。要是字典中没有要查找的键，字典将会返回 **nil**，而不是直接崩溃。这和数组的行为不一样，你要是用越界下标访问数组的话，程序会直接崩溃。两者接口上的这种区别是因为，字典是一种稀疏结构 (它不包含某个键的可能性要远高于包含它的可能性)，但是数组的序号访问却一般被认为是会成功的。Swift 将可能是 **nil** 的值 (可选值) 和不可能是 **nil** 的值 (普通值) 进行了区分。我们会在有关可选值的章节进一步了解这些内容。

变更

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，可以将它设为 **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

let  oldName = localizedSettings.updateValue("My iPhone", forKey: "Name")
//  "Mein iPhone"
```

Swift 中还有一些其他的内建的集合类型：比如说 Set，它可以被桥接为 NSSet；比如说 Range，它描述的是一个连续不断的索引值范围；比如说 Repeat，它包含多个同样的值；我们甚至可以将可选值 Optional 看作是最多可以容纳一个元素的集合类型。

有用的字典扩展

我们可以自己为字典添加一些扩展，比如增加将两个字典合并的功能。举例来说，当我们将设置展示给用户时，我们希望将它们与用户存储的设置进行合并后再一起展示。假设我们已经有一个 storedSettings 方法来读取用户所存储的设置：

```
func storedSettings() -> [String:Setting]  {
```

现在，我们来为字典增加一个 merge 函数。字典遵守 SequenceType 协议，为了通用性，我们可以让这个方法接受的参数是任意的遵守 SequenceType 协议、并且能产生 (Key,Value) 键值对的类型。这样一来，我们将不仅能够合并字典类型，也能对其他类似的类型进行相同的操作。

```
extension Dictionary {
    mutating func merge<S: SequenceType
        where S.Generator.Element == (Key,Value)>(other: S) {
        for  (k,  v)  in  other {
            self[k]  = v
        }
    }
}
```

现在我们可以将存储的设置与默认设置进行合并了：

```
var settings = defaultSettings
settings.merge(storedSettings())
```

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。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: SequenceType
        where S.Generator.Element == (Key,Value)>(_ sequence: S) {
        self = [:]
        self.merge(sequence)
    }
}

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

最后，为字典添加一个 map 函数来操作并转换字典中的值也会非常有用。因为 Dictionary 已经是一个序列了，它已经有一个 map 函数来产生数组。不过我们在这里想要的是结果保持为字典的版本，它应该只对其中的值进行映射。因此我们把这个函数叫做 mapValues，它将对字典中的值进行转换，并保持对应的键不变，来创建一个新的字典。我们将它作用于保存设置的字典上，用来把保存的值转换为对应的视图。

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

let keysAndViews = settings.mapValues { $0.settingsView() }
```

在闭包中使用集合

在 Swift 标准库中，还包含了 Set 类型。Set 是一系列元素的无序集合，每个元素在 Set 中只会出现一次。Set 遵循 ArrayLiteralConvertible 协议，也就是说，我们可以通过使用和数组一样的字面量语法来初始化一个集合。

```
let mySet: Set<Int> = [1, 2, 2]
mySet
```

```
[2, 1]
```

在你的函数中，无论你是否将它们暴露给调用者，字典和集合都会是非常有用的数据结构。举个例子，如果我们想要为 SequenceType 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 SequenceType where Generator.Element: Hashable {
    func unique() -> [Generator.Element] {
        var seen: Set<Generator.Element> = []
        return filter {
            if seen.contains($0) {
                return false
            } else {
                seen.insert($0)
                return true
            }
        }
    }
}
```

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。

集合协议

留心看看数组、字典以及集合的定义，我们会发现它们都遵守 CollectionType 协议。更进一步，CollectionType 本身是一个遵守 SequenceType 的协议。而序列 (sequence) 使用一个 GeneratorType 来提供它其中的元素。为了理解整件事情，我们会从最底层开始，然后一路向上回到我们的实际例子中。简单说，生成器 (generator) 知道如何产生新的值，序列知道如何创建一个生成器，而集合 (collection) 为序列添加了有意义的随机存取的方式。

生成器

GeneratorType 协议通过两部分来定义。首先，它要求遵守它的类型都需要有一个 Element 关联类型，这个类型也就是生成器产生的值的类型。比如在 String.CharacterView 中，元素类型是 Character。在数组中，生成器的元素类型就是数组的元素类型。在字典中，元素类型是一个同时包含键和值的多元组，也就是 (Key, Value)。

GeneratorType 定义的第二部分是 next 函数，它返回类型为 Element 的可选值。对于遵守 GeneratorType 的值，你可以一直调用它的 next 函数，直到你得到 **nil** 值。结合这两部分，我们就能得到 GeneratorType 协议的定义了：

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

协议中的 **associatedtype** 的意思是，想要遵守这个协议，你需要指定一个**关联的类型**，这个类型可以是通过 **typealias** 显式指定的，也可以通过你实现 next 方法时的返回值来隐式地指出。

文档指出我们可以随意对生成器进行复制，但是实际上生成器的工作是单向的：你只能将生成器的值循环一次。也就是说，生成器并**没有**值语义。所以我们将会使用 **class** 来创建生成器，而不使用 **struct** (想要进一步了解结构体和类的区别，请参看结构体和类的相关章节)。我们能写出的最简单的生成器是一个在被询问下一个值时每次都返回常数的生成器：

```
class ConstantGenerator: GeneratorType {
    typealias Element = Int
    func next() -> Element? {
        return 1
    }
```

```
}
```

我们也可以隐式地为 Element 的赋值，这样一来，ConstantGenerator 的定义就变为：

```
class ConstantGenerator: GeneratorType {
    func next() -> Int? {
        return 1
    }
}
```

要使用这个生成器，我们可以创建一个 ConstantGenerator 的实例，然后用 **while** 循环对它进行枚举，这将会打印无穷的数字 1：

```
var generator = ConstantGenerator()
while let x = generator.next() {
    // 使用 x
}
```

如果我们想要生成一个从 0 开始的斐波那契数列，我们可以为生成器添加一个额外的追踪状态，来记录接下来的两个数字。next 函数做的事情是为接下来的调用更新这个状态，并且返回其中的第一个数。整个生成器也会产生一组 “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)：

```
class FibsGenerator: GeneratorType {
    var state = (0, 1)
    func next() -> Int? {
        let upcomingNumber = state.0
        state = (state.1, state.0 + state.1)
        return upcomingNumber
    }
}
```

我们也可以创造有限序列的生成器，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)。它先将偏移量 offset 设置为 startIndex，然后每次调用 next 的时候移到下一个字符，并且返回从字符串开始到偏移量为止的子字符串：

```
class PrefixGenerator: GeneratorType {
    let string: String
    var offset: String.Index
```

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

    func next() -> String? {
        guard offset < string.endIndex else { return nil }
        offset = offset.successor()
        return string[string.startIndex..<offset]
    }
}
```

(string[string.startIndex..<offset] 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。)

上面这些生成器都能且只能被迭代一次。如果我们想要再次进行迭代，那么就需要创建一个新的生成器。这也通过它的定义有所暗示：我们将它们声明成了类而非结构体。在被不同变量共享的时候，只有引用的传递，而不发生复制。

生成器和值语义

Swift 标准库中的 AnyGenerator 有着曲折的过去。Swift 最开始的时候曾经有个叫做 GeneratorOf 的类型，它是一个结构体，做着与现在 AnyGenerator 类似的事情。在 Swift 2.0 的时候，AnyGenerator 取代了 GeneratorOf，并且它是一个类。不过在 Swift 2.2 里，它又变成了一个结构体。虽然 AnyGenerator 现在是一个结构体，但是它**并不**具有值语义，因为它其实将生成器的实现封装到了一个方法的引用类型中去。作为存在状态的结构，生成器其实在 Swift 标准库中显得有那么一点格格不入，因为标准库里绝大部分类型都是具有值语义的结构体。

为了描述这里的区别，假设我们有一个简单的生成器，比如一个步进器 (每次调用 next 时加一)，然后调用几次 next 函数：

```
let seq = 0.stride(through: 9, by: 1)
var g1 = seq.generate()
g1.next()  // 返回 0
g1.next()  // 返回 1
// g1 准备在下一次 next 时返回 2
```

现在，你对这个生成器进行复制：

```
var g2 = g1
```

现在原有的生成器和新复制的生成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：

```
g1.next()  // 返回 2
g1.next()  // 返回 3
g2.next()  // 返回 2
g2.next()  // 返回 3
```

这是因为 StrideThroughGenerator 是一个简单的结构体，它具有值语义。

但是正如上面提到的，AnyGenerator 虽然是一个结构体，但是它并**不具有**值语义。我们来看看通过封装 g1 所得到的 AnyGenerator 的行为：

```
var g3 = AnyGenerator(g1)
```

上面的构造方法通过引用的方式捕获 g1 (我们会在稍后的章节介绍相关技术)。不论是对 g1 还是 g3 调用 next，它们在底层所操作的生成器实例都是同一个：

```
g3.next()  // 返回 4
g1.next()  // 返回 5
g3.next()  // 返回 6
g3.next()  // 返回 7
```

显然，这可能会造成一些 bug。AnyGenerator 的这种行为也有些出人意料，所以在将来这个行为发生改变的可能性很高。不过，我们只需要遵守一条原则就能避免在这上面纠结，那就是避免去复制生成器。当你需要生成器的时候，总是去创建一个新的。我们接下来要说的序列类型 (SequenceType) 就很好地遵守了这一原则。

序列

进行多次迭代是很常见的行为，因此 SequenceType 应运而生。SequenceType 是构建在 GeneratorType 基础上的一个协议，它需要我们指定一个特定类型的生成器，并且提供一个方法来创建新的生成器：

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

举个例子，想要多次枚举前面我们提到的前缀的话，我们可以把 PrefixGenerator 放到一个序列中去。我们在这里没有显式地指定 GeneratorType 的类型，通过读取 generate 方法的返回类型，编译器可以为我们推断出所需要的 GeneratorType：

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

    func generate() -> PrefixGenerator {
        return PrefixGenerator(string: string)
    }
}
```

现在我们可以为一个特定的字符串创建对应的 PrefixGenerator 了。因为我们实现了 SequenceType 协议，我们现在已经可以用 **for** 循环来迭代给定字符串的所有前缀了：

```
for prefix in PrefixSequence(string: "Hello") {
    print(prefix)
}
```

for 循环的工作原理是这样的：编译器为序列创建一个新的生成器，然后一直调用这个生成器的 next 方法获取其中的值，直到它返回 **nil**。从本质上说，**for** 就是下面这段代码的一种简写：

```
var generator = PrefixSequence(string: "Hello").generate()
while let prefix = generator.next() {
    print(prefix)
}
```

如果我们考虑 Array 的实现，我们可以很轻松地想象出它的生成器和序列会是什么样子。同样，对于字典、集合以及字符串，我们也可以进行一些思考。如果我们要实现自己的数据结构，我们现在知道要是想要让它能和 **for** 一起工作的话，需要让它遵守 SequenceType。

在关于协议一章中，我们将介绍协议和关联类型是如何工作的，我们也会提到使用它们时的一些限制。

基于函数的生成器和序列

还有一种更简单的方法来创建生成器和序列。相比于创建一个新的类，我们可以使用 Swift 内建的 AnyGenerator 和 AnySequence 类型。它们接受一个函数作为参数，并根据这个函数为我们构建相应的类型。比如说，我们可以用下面的代码来定义一个斐波那契数列生成器，而不必使用一个中间的 **class** 类型：

```
func fibGenerator() -> AnyGenerator<Int> {
    var state = (0, 1)
    return AnyGenerator {
        let result = state.0
        state = (state.1, state.0 + state.1)
        return result
    }
}
```

AnyGenerator 可以接受一个函数作为参数。在这里，我们将 state 移到了函数的外面，因为 state 变量是被闭包持有的，所以每次这个函数的被调用时，state 都将被更新。

通过生成器创建序列的更加容易：

```
let fibSequence = AnySequence(fibGenerator)
```

在标准库中，那些遵守 SequenceType 协议的类型可以被 **for** 循环迭代访问。不过，这并没有对操作是否会破坏原来的数据做出任何约束。如果你想要保证操作是非破坏性的话，应该使用 CollectionType 协议。当我们使用遵守 SequenceType 协议的类型时，当想要确认迭代是否具有破坏性时，我们需要去检查文档。举个例子，计算斐波那契数列的操作是非破坏的 (我们随时可以再重新计算它)，但是从标准输入读取输入内容就是破坏性的操作 (一旦我们从键盘读取一行输入内容后，我们就没有办法再次对其进行读取了)。因为 SequenceType 是可能对应着这两种操作的，所以在通过 **for** 循环对它进行访问的时候应当特别小心。

> 如果你觉得生成器和序列以及它们之间的关系很奇怪的话，不要担心，你不是一个人！很多开发者都对这两个类型感到疑惑，为什么我们需要它们。Swift 团队在邮件列表中曾经多次提及其中的原委。生成器和序列之所以像现在这样工作，是因为现有的类型系统的一些限制。可能在 Swift 3.0 或者之后的版本中，SequenceType 和 GeneratorType 的交互方式会发生改变。

集合

集合协议 CollectionType 是在序列上构建的，它为序列添加了可重复进行迭代，以及通过索引访问元素的能力。

为了展示 Swift 的集合类型是如何工作的，我们将实现一个我们自己的集合类型。队列 (queue) 是在 Swift 标准库中不存在，但却又最常用到的容器类型了。Swift 的数组可以很容易地被用来实现栈 (stack) 结构，只需要使用 append 入栈和 popLast 出栈就可以了。但是将数组作为队列来使用的话就不那么理想。你可以通过结合 push 和 removeAtIndex(0) 来实现它，但是要注意从一个数组中移除非末尾元素的时候复杂度是 $O(n)$。因为数组是一段连续的内存，如果你移除一个中间的元素，后面的元素都需要移动并填充中间的空隙。而移除末尾元素的话就没有这个问题，所以它将在常数时间内完成。

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。我们可以通过下面定义的协议来描述队列的特性：

```
/// 一个能够将元素入队和出队的类型
protocol QueueType {
    /// 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，

实现这个协议的**任何**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集合类型 (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，只需要把用 append 来实现 enqueue，然后用结合使用 isEmpty 和 popLast 来实现 dequeue 就行了。

数组和可选值

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**，否则它将移除最后一个元素并返回它。

这和 Array 的 removeLast 方法稍有不同，如果当数组为空的时候，removeLast 会造成一个致命错误并且强制退出。popLast() 等效于 isEmpty ? **nil** : removeLast()。你想用哪个方法依赖于你的使用场景，当你将数组作为栈来使用的时候，你可能会想要将 empty 判定和移除末位元素结合起来使用。另一方面，你可能会想将数组用于比一个简单的栈更复杂的数据结构中。在这些变种里，有可能你已经知道栈是不是已经空了，在这种情况下，再去处理可选值就显得有些麻烦。对于队列来说，我们只提供了一个 dequeue 方法，因为这是队列的最常见的使用方式。

关于这个取舍还有另一个例子，那就是 Array 类型的下标。在使用下标访问数组中的元素时，用户应该确保下标满足预先的条件，不超过下标的最大可能范围。比如要获取下标序号为 9 的元素时，你应该确保数组中至少应该有 10 个元素。

这么设计背后的原因是很多时候我们可以使用数组切片来完成工作。在 Swift 中，我们并不是很经常需要计算下标序号：

- → 想要迭代整个集合?
  ```
  for x in collection
  ```
- → 想要迭代集合中除了第一个元素以外的部分?
  ```
  for x in collection.dropFirst()
  ```
- → 想要迭代集合中除了最后五个元素以外的部分?

```
for x in collection.dropLast(5)
```

→ 想要迭代集合中所有的索引?

```
for idx in collection.indices
```

→ 想要为集合中的所有元素标号?

```
for (num, element) in collection.enumerate()
```

→ 想要找到指定元素在集合中的位置?

```
if let idx = collection.indexOf { someMatchingLogic($0) }
```

→ 想要将集合中的所有元素进行变形?

```
array.map { someTransformation($0) }
```

→ 想要获取符合某个条件的所有元素?

```
collection. filter { someCriteria($0) }
```

传统的 C 风格的 **for** 循环将在 Swift 3.0 被移除。如果你在 Swift 2.0 中写了像是 `for var idx = 0;idx < array.count;++i` 这样的代码，那么你必须将它转变为 **while** 循环。不过一般来说会有更好的方式来重写循环。对我来说，手动操作那些麻烦的索引是一个很容易产生 bug 的地方，所以我尽可能避免写这样的代码。如果确实无法避免，我会选择新写一个可以重用的泛型函数来进行处理，你可以精心测试你的计算下标索引的逻辑，并将它封装到那里。这么做其实很简单，我们会在泛型一章中看到具体的例子。

但是还是有一些时候你只能使用下标索引。对于数组索引来说，当你使用它时，你必须仔细思考计算索引的逻辑。如果要对使用索引和下标操作取到的值进行解包的话，其实可能有点蠢，因为这么做意味着你并不能信任你的代码。如果你信任你自己的代码的话，你应该可以强制解包结果值，因为你确实**知道**你使用的索引是有效的。但是，这是一件让人很心烦的事情，也并不是什么好习惯。当强制解包变成日常操作的话，你可能会时不时有些小疏忽，然后强制解包了不应该强制解包的东西。因此，为了不让这种习惯成为主流，数组直接没有给你这个选项，而是将下标访问的返回值声明为非可选值。

相比起 popLast 来，removeLast 的意思可能没有那么明显。当把数组当做栈使用时，你经常想要做的是如果数组不为空，就将最后的元素出栈 (或者说弹出 (pop))：

```
while !array.isEmpty {
    // 必须保证数组不为空
    let top = array.removeLast()
    // 处理栈顶元素
}
```

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
}
```

当你知道数组**不会**为空时这么做带来的好处，相比于必须解包所带来的麻烦孰轻孰重，取决于你的决定。

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。

因为我们把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 Queue<Element>: QueueType {
    private var left: [Element]
    private var right: [Element]

    init() {
        left = []
        right = []
    }

    /// 将元素添加到队列最后，复杂度 O(1)
    mutating func enqueue(element: Element) {
        right.append(element)
    }

    /// 从队列前端移除一个元素，复杂度 O(1).
    /// 当队列为空时，返回 nil
```

```
    mutating func dequeue() -> Element? {
        guard !(left.isEmpty && right.isEmpty) else { return nil }

        if left.isEmpty {
            left = right.reverse()
            right.removeAll(keepCapacity: true)
        }
        return left.removeLast()
    }
}
```

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。即使对于单个的操作来说可能耗时会长一些，不过对于非常多的 push 和 pop 操作来说，取出一个元素的**平摊**耗时是一个常数。

理解这个复杂度的关键在于理解反向操作发生的频率以及发生在多少个元素上。我们可以使用 “银行家理论” 来分析平摊复杂度。想象一下，你每次将一个元素放入队列，就相当于你在银行存了一块钱。接下来，你把右侧的栈的内容转移到左侧去，因为对应每个已经入队的元素，你在银行里都相当于有一块钱。你可以用这些钱来支付反转。你的账户永远不会负债，你也从来不会花费比你付出的更多的东西。

这个理论可以用来解释一个操作的消耗在时间上进行**平摊**的情况，即便其中的某次调用可能不是常数，但平摊下来以后这个耗时依然是常数。Swift 中向数组后面添加一个元素的操作是常数时间复杂度，这也可以用同样的理论进行解释。当数组存储空间耗尽时，它需要申请更大的空间，并且把所有已经存在于数组中的元素复制过去。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 CollectionType

有时候在 Swift 中遵守一个协议并不是一件很直接的事情。我们在研究 CollectionType 时，会发现这个协议是对 Indexable 和 SequenceType 的扩展。在写这篇文章时，该协议有两个关联类型和九个方法。不过，这两个关联类型都有默认值，而不少方法也有它们的默认实现。

Generator 这个关联类型的默认值是 IndexingGenerator<Self>。我们选择可以覆盖它，不过如果我们使用这个默认值的话，我们就可以不用去实现 generate() 方法了，因为它已经在协议扩展中被实现了：

```
extension CollectionType where Generator == IndexingGenerator<Self> {
    public func generate() -> IndexingGenerator<Self>
}
```

CollectionType 中的所有其他方法也都有默认的实现。我们要做的是为 Indexable 协议中要求的 startIndex 和 endIndex 提供实现，并且实现一个通过下标索引来获取对应索引的元素的方法。只要我们实现了这三个需求，我们就能让一个类型遵守 CollectionType 了。

我们现在已经有一个能够出队和入队的容器了，要将它转变为一个集合，Queue 需要遵守 CollectionType：

```
extension Queue: CollectionType {
    var startIndex: Int { return 0 }
    var endIndex: Int { return left.count + right.count }

    subscript(idx: Int) -> Element {
        precondition((0..<endIndex).contains(idx), "Index out of bounds")
        if idx < left.endIndex {
            return left[left.count - idx.successor()]
        } else {
            return right[idx - left.count]
        }
    }
}
```

CollectionType 定义了一个 Index 关联类型，不过就和 Element 一样，Swift 可以从方法和属性的定义中推断出这个类型。我们使用了和数组一样的方式，返回一个非可选的值，并且在索引无效的时候让程序崩溃。因为索引从容器的开头进行计算并返回元素，所以 Queue.first 将会返回下一个即将被出队的元素 (所以你可以将它当做 peek 来使用)。

有了这几行代码，Queue 现在已经遵守 CollectionType 了。因为 CollectionType 是基于 SequenceType 的，所以 Queue 也遵守 SequenceType。现在我们的队列已经拥有超过 40 个方法和属性供我们使用了：

```
var q = Queue<String>()
```

```
for x in ["1", "2", "foo", "3"] {
    q.enqueue(x)
}

// 你可以用 for... in 循环访问队列
for s in q { print(s) }          // 打印 1 2 foo 3

// 将其传递给接受序列的方法
q.joinWithSeparator(",")         // "1,2,foo,3"
let a = Array(q)                 // a = ["1", "2", "foo", "3]

// 调用 SequenceType 的扩展方法
q.map { $0.uppercaseString }     // ["1", "2", "FOO", "3"]
q.flatMap { Int($0) }            // [1,2,3]
q.filter {                       // ["foo"]
    $0.characters.count > 1
}

// 调用 CollectionType 的扩展方法
q.isEmpty                        // false
q.count                          // 4
q.first                          // "1"
q.last                           // "3"
```

现在我们能够使用 **for** 循环来访问队列，为其中的元素排序，对元素进行 map 或者 reduce 操作，以及使用任何其他定义在 SequenceType 上的方法了。

遵守 ArrayLiteralConvertible

当创建一个这样的集合类型时，实现 ArrayLiteralConvertible 会是个不错的主意。它能让用户通过他们所熟知的 [value1, value2, etc] 语法来创建一个队列。实现这个协议非常容易：

```
extension Queue: ArrayLiteralConvertible {
    init(arrayLiteral elements: Element...) {
        self.left = elements.reverse()
        self.right = []
    }
}
```

对于我们的队列逻辑来说，我们希望这些元素已经是反转过的，并将它们存储在左侧数组中，这样我们就能直接使用它们了。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，所以直接先逆序将它们复制过去会更高效一些。

现在我们就可以用数组字面量来创建一个队列了：

```
let q: Queue = [1,2,3]
```

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 ArrayLiteralConvertible 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 IntegerLiteralConvertible 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整数的队列类型，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: Queue<Int8> = [1,2,3]
```

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>) {
    // ...
}

// 这个字面量被推断为 Set<Float>，而不是 Array<Int>
takesSetOfFloats([1, 2, 3])
```

遵守 RangeReplaceableCollectionType

队列下一个要支持的是 RangeReplaceableCollectionType。这个协议要求我们做三件事情：

- → reserveCapacity 方法：我们在实现 map 的时候已经使用过这个方法了。因为最后元素的个数事前就已经知道了，因此它能够在数组申请内存时避免不必要的元素复制。一个

集合其实在收到 reserveCapacity 调用时可以什么都不做，忽略掉它并不会造成很大影响。

- → 一个空的初始化方法：在泛型函数中这会很有用，它可以允许我们创建相同类型的空的集合。
- → replaceRange 函数：它接受一个范围和一个集合，并将原来集合中这个范围内的内容用新的集合替换。

RangeReplaceableCollectionType 是展现协议扩展的绝佳的例子。你通过实现一个超级灵活的 replaceRange 函数，就能直接由其得到一组引申出的有用的方法：

- → append 和 appendContentsOf：将 endIndex..<endIndex 范围 (也就是集合末尾的空范围) 用新的元素进行替换。
- → removeAtIndex 和 removeRange：将 i ... i 或者 subRange 的内容用空集合进行替换
- → splice 和 insertAtIndex：将 atIndex..<atIndex (也就是在这个位置的空范围) 用新的值进行替换
- → removeAll：将 startIndex..<endIndex 用一个空集合进行替换

如果一个具体的集合类型能够结合它自己的实现方式，给出这些函数更高效的实现版本的话，它们也可以通过复写自己的版本，而不使用协议接口中默认的实现。

我们在这里选择使用一个非常简单，不那么高效的实现版本。我们在定义这个队列数据结构时就已经指出，左侧的栈以逆序来持有所有的元素。想要简单地实现 RangeReplaceableCollectionType 定义的函数，我们需要把所有元素再逆序，并将其合并到右侧数组中，这样我们就能够一次性地替换整个范围了。这里即使是最高效的实现，也仅只是 $O(n)$ 复杂度，不过它的因子会比较小：

```
extension Queue: RangeReplaceableCollectionType {
    mutating func reserveCapacity(n: Int) {
        return
    }

    mutating func replaceRange
        <C: CollectionType where C.Generator.Element == Element>
        (subRange: Range<Int>, with newElements: C)
    {
        right = left.reverse() + right
        left.removeAll(keepCapacity: true)
```

```
            right.replaceRange(subRange, with: newElements)
        }
    }
```

你也可以自己尝试着实现一个更高效的版本，比如通过确定要替代的范围是不是被分到了左右两个栈中，来减少对于两侧数组的操作。

在上面的例子中，我们没有必要去实现 **init**，因为我们定义的是一个 Queue 结构体，初始化方法已经默认实现了。同时，Queue 也选择忽略掉 reserveCapacity。在这种实现下，appendContentsOf 和 append j将会把元素加到队列尾部，这非常合理。

索引

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 Queue 类型亦是如此。

但是并不是所有的索引都必须满足随机存取特性，也不是所有的随机存取索引都必须是整数。在 Swift 中，一个类型想要成为集合的索引值的话，所需要满足的最小的协议是 ForwardIndexType，它只要求两件事：首先，要有一个 successor 方法来获取下一个索引值；其次，它必须满足 Equatable，也就是说可以用 == 运算符进行判等。

作为示例，我们会实现一个最基础的前向存取集合，那就是单向链表。不过在此之前，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，我们会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：

```
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 l = List<Int>.End.cons(1).cons(2).cons(3)
```

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”)。

这个列表类型有一个很有意思的特性：它是 “持久化的”。列表的节点是不可变的，一旦你创建了一个节点，你就不能再改变它了。将一个元素添加到列表的操作并不会复制整个列表，它只是给了你一个新的节点，并且在之后链上了已经存在的列表。

这意味着两个列表可以共享相同的链尾：

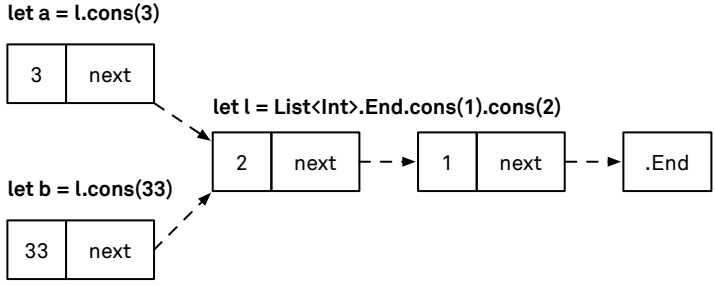

图 3.2: 列表共享

列表的不可变特性在这里非常关键。如果你可以改变列表的话 (比如移除最后一个元素，或者更新某个节点的值之类的)，这样一来共享就会造成问题 — 也许 a 对于改变列表，将影响到 b 中的元素。

可以把这样的链表看做一个栈，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 StackType {
    /// 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 StackType：

```
extension Array: StackType {
    mutating func push(x: Element) {
        append(x)
    }

    mutating func pop() -> Element? {
        return popLast()
    }
}
```

List 也行：

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

    mutating func pop() -> Element? {
        switch self {
        case .End: return nil
        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>.End.cons(1).cons(2).cons(3)
var a = stack
var b = stack

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

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

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

stack.pop() // 4
stack.pop() // 2
stack.pop() // 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。我们会在函数中详细介绍 input，对于可变方法，我们也将在结构体和类再深入探讨。

有了这些知识，我们可以看到，通过使用 **indirect**，我们可以在列表中列举变量。

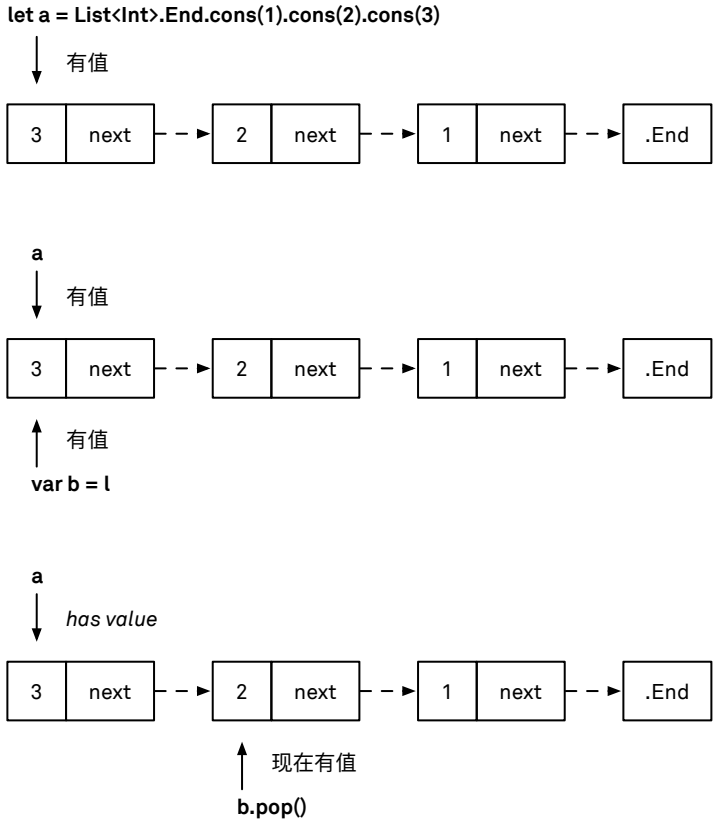

图 3.3: 列举列表

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：

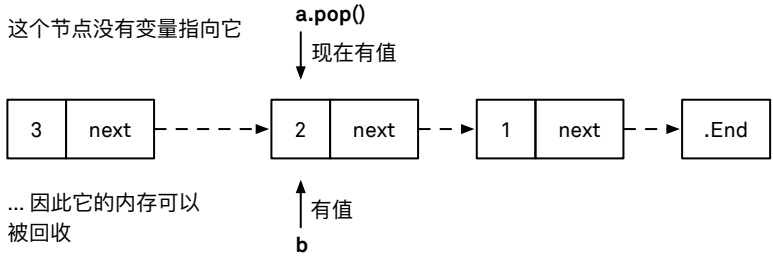

图 3.4: 列表的内存管理

我们会在结构体和类中详细讲解 ARC 的有关内容。

让 List 遵守 SequenceType

因为列表的变量可以进行列举，也就是说，你能够使用它们来让 List 遵守 SequenceType 协议：

```
extension List: SequenceType {
    func generate() -> AnyGenerator<Element> {
        // 声明一个变量来追踪列表当前的列举状态
        var current = self
        return AnyGenerator {
            // next() 将会出栈元素，如果列表为空，则返回 `nil`
            current.pop()
        }
    }
}
```

为了使用起来简单一些，我们也可以让它遵守 ArrayLiteralConvertible 协议：

```
extension List: ArrayLiteralConvertible {
    init(arrayLiteral elements: Element...) {
        self = elements.reverse().reduce(.End) { $0.cons($1) }
    }
```

```
}
```

现在，你能够在列表上使用 **for ... in** 了：

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

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：

```
l.joinWithSeparator(",")          // "1,2,3"
l.contains("2")                   // true
l.flatMap { Int($0) }             // [1, 2, 3]
l.elementsEqual(["1", "2", "3"])  // true
```

让 List 遵守 CollectionType

接下来，我们要让 List 实现 CollectionType。这将可以把像是 first 这样的属性带给我们的数据结构，有了它，我们就能够查看列表的第一个元素，而不用将它出栈了。

更重要的是，通过实现 CollectionType，我们可以保证多次遍历序列不会出现问题。就像 SequenceType 的文档所提到的那样：

> SequenceType 并不要求实现它的类型在列举序列值时要保证非破坏性。想要保证在列举序列元素时不破坏原来的数据，你需要将序列约束为一个 CollectionType。

这也解释了为什么 first 属性只在集合类型上可用，有了 CollectionType 保证，就可以用计算属性来进行这样的实现，而计算属性应该是非破坏的。举一个具有破坏性的序列的例子，比如对 readLine 函数的封装，它可以从标准输入逐行进行读入：

```
let standardIn = AnySequence {
    return AnyGenerator {
        readLine()
    }
}
```

现在你可以在这个序列类型上使用那些扩展方法了。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.enumerate()

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发生在你通过生成器移动到被列举的序列的下一个值时，而不是发生在序列被创建时。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。

enumerate 的一种实现方式可以是这样的：

```
extension SequenceType {
    func enumerate() -> AnySequence<(Int,Generator.Element)> {
        // 现在 Swift 在闭包中还需要一些帮助才能够进行类型推断：
        return AnySequence { _ -> AnyGenerator<(Int, Generator.Element)> in
            // 创建一个新的计数器和生成器并开始列举
            var i = 0
            var g = self.generate()
            // 在闭包中捕获并返回它们，并在新的生成器中进行返回
            return AnyGenerator {
                // 当原来的序列耗尽时，返回 nil
                guard let next = g.next() else { return nil }
                let result = (i,next)
                i += 1
                return result
            }
        }
    }
}
```

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。

在有些情况下，即使序列没有遵守 CollectionType，我们仍旧可以循环多次。比如步进函数 stride 返回的 StrideTo 和 StrideThrough 类型就是这样。它们并不是集合，这是因为事实上你可以将步进的步长设置为浮点数，这让它们很难被描述成一个集合，因此它们仅仅只是序列。不过，只要起始和步长确定以后，你就可以多次复用这个序列。如果你在写一个 SequenceType 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。

对于我们这里的列表类型来说，列举访问是非破坏的，所以可能的话，让它遵守 CollectionType 来说明这一点会是一个不错的选择。在我们实际实现它之前，先来做些准备工作，将节点从列表和索引类型中分离出来。

就像我们在实现遵守 SequenceType 协议时做的那样，我们只需要让枚举类型直接扩展 CollectionType 和 ForwardIndexType 就行了。但是这会导致一个问题。这两个协议所需要的 == 的实现是不太一样的：

- → 索引需要知道从同一个列表得到的两个索引值是否是在同一个位置。它并不需要元素本身遵守 Equatable 协议。
- → 但是，作为集合类型的话，两个列表应该可以被比较，并判定其中的元素是否相等。这需要元素遵守 Equatable 协议。

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，节点枚举不会被外部访问到，我们可以将它的实现标记为私有，这样我们就可以将实现细节隐藏起来。新的 ListNode 类型和我们一开始的 List 类型看上去是完全一样的：

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

    func cons(x: Element) -> ListNode<Element> {
        // 每次 cons 调用将把 tag 加一
        return .Node(x, next: self)
    }
}
```

索引类型封装了 ListNode。索引可以通过访问节点的 next 来创建一个新的索引，由此返回它的之后的一个索引值、但是这只满足了 ForwardIndexType 的一般的要求。我们还需要知道一些额外信息来为索引实现 == 运算符。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)。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> {
    private let  node: ListNode<Element>
    private let  tag:  Int
}
```

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。

ForwardIndexType 要求类型有一个 successor 方法，它应该返回下一个索引：

```
extension ListIndex: ForwardIndexType {
    public func successor() -> ListIndex<Element> {
        switch node {
        case let  .Node(_, next: next):
            return ListIndex(node: next, tag:  tag.predecessor())
        case .End:
            fatalError("cannot increment endIndex")
        }
    }
}
```

ListIndex 也必须实现 Equatable，这样你才能判断两个索引是否相等。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：

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

现在 ListIndex 遵守 ForwardIndexType 了，它能够被 List 用来实现 CollectionType:

```
public struct List<Element>: CollectionType {
```

```
    // 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 var startIndex: Index
    public var endIndex: Index

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

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

```
extension List: ArrayLiteralConvertible {
    public init(arrayLiteral elements: Element...) {
        startIndex = ListIndex(node: elements.reverse().reduce(.End) {
            $0.cons($1)
        }, tag: elements.count)
        endIndex = ListIndex(node: .End, tag: 0)
    }
}
```

现在，我们的列表是一个 CollectionType 的扩展了：

```
let l: List = ["one", "two", "three"]
l.first
l.indexOf("two")
```

除此之外，因为 tag 标记了添加到 .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。而一般的只有前向索引的集合类型的 count 是 $O(n)$ 复杂度的。

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

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 实现一个不同于节点的 == 运算符，用来比较集合中的两个元素：

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

实现自定义切片

和默认的索引生成器一样，集合也有默认实现的切片操作，[Range<Index>]，dorpFirst 就是依赖这个默认实现的：

```
// 等效于 l.dropFirst()
let firstDropped = l[l.startIndex.successor()..<l.endIndex]
```

因为像是 l[somewhere..<l.endIndex] (从某个位置到结尾的切片) 和 l[l.startIndex..<somewhere] (从开头到某个位置的切片) 是非常常见的操作，所以在标准库中用一种更容易理解的方式对它们进行了定义：

```
let firstDropped = l.suffixFrom(l.startIndex.successor())
```

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: CollectionType>: CollectionType {
    let collection: Base
    let bounds: Range<Base.Index>

    var startIndex: Base.Index { return bounds.startIndex }
    var endIndex: Base.Index { return bounds.endIndex }

    subscript(idx: Base.Index) -> Base.Generator.Element { return collection[idx] }

    typealias SubSequence = Slice<Base>
    subscript(bounds: Range<Base.Index>) -> Slice<Base> {
```

```
        return Slice(collection: collection, bounds: bounds)
    }
}
```

默认的实现只不过是返回了一个对原来集合的封装，再加上一个索引的子范围。所以在 List 的场合，它的内存大小将会是原来的两倍：

```
// 一个拥有两个节点 (start 和 end) 的列表的大小：
sizeofValue(l)                  // 返回 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)
sizeofValue(l.dropFirst())  // 返回 64
```

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 {
    private init(subRange: Range<Index>) {
        startIndex = subRange.startIndex
        endIndex = subRange.endIndex
    }
    public subscript(subRange: Range<Index>) -> List<Element> {
        return List(subRange: subRange)
    }
}
```

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：

```
sizeofValue(l.dropFirst())  // 返回 32
```

这么做还为我们的列表结构带来了额外的好处。在包括 Swift 的数组和字符串在内的很多可切片的容器类型中，切片和原集合是共享存储缓冲区的。这会导致一个比较悲剧的副作用：切片在其生命周期里都会持有原来集合的所有缓冲内存，即使原来的容器已经离开它的作用域了。也就是说，你要是读取了 1 GB 的文件到数组或者字符串中，然后取出一个很小的切片，这整个 1 GB 的缓冲区依然会一直留在内存中，直到原来的集合和切片两者都被销毁才能释放。

不过使用这里的 List 的话，情况就要好一些。节点是通过 ARC 管理的：切片后的内容是唯一还存在的复制，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。

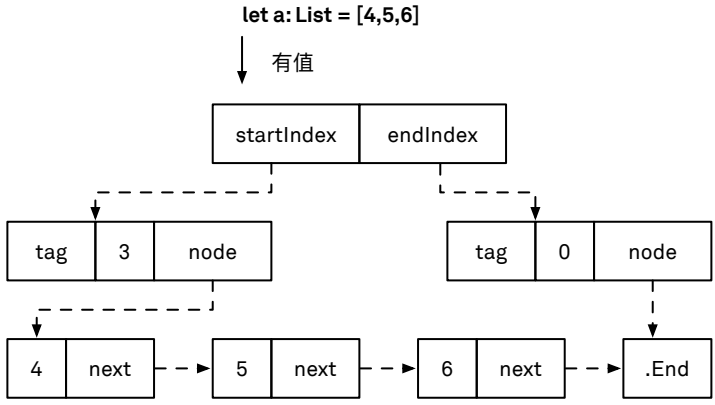

图 3.5: 列表共享和 ARC

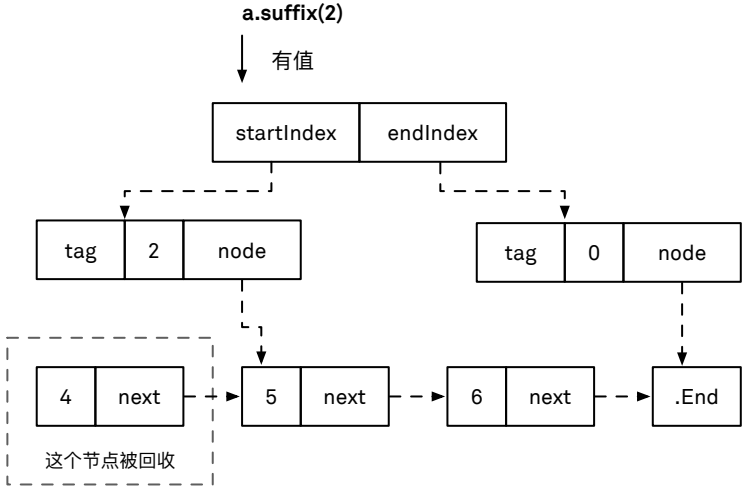

图 3.6: 内存回收

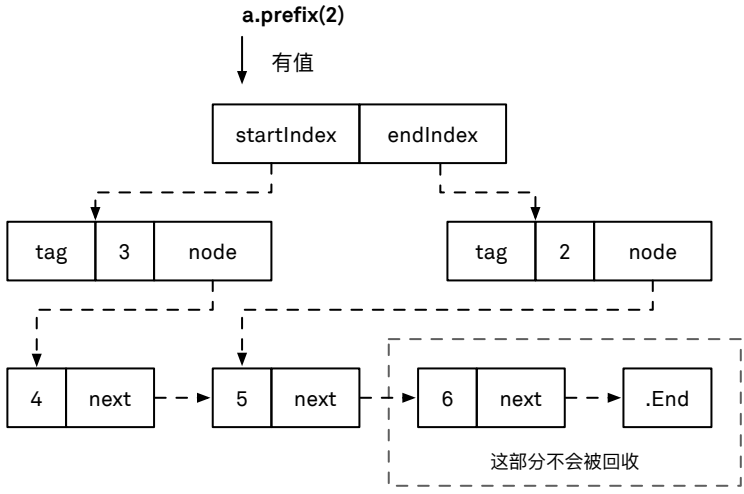

图 3.7: 内存不被回收

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。

默认的切片示例还提醒我们一件事，就算是使用整型作为索引，我们也不能假设集合的索引是从 0 开始的。

举个例子，我们的 Queue 类型使用的是默认的切片下标，所以，当你创建一个队列并且对它进行切片时，你会得到 Slice<Queue<Element>> 类型：

```
let q: Queue = ["a", "b", "c", "d", "e"]
let s = q[2..<5]
```

现在，基于上面 Slice 的实现，这个切片的 startIndex 会是子范围的起始值，而 endIndex 是子范围的终止值。

```
q.startIndex // 0
q.endIndex   // 5
s.startIndex // 2
s.endIndex   // 4
```

这也是为什么我们总是应该用 `for x in collection` 或者 `for index in collection.indices` 来列举集合，而不是用传统的 C 风格 `for` 循环的原因之一。当你需要获取一个集合的索引的时候，也请记得使用 `var i = collection.startIndex`，而不是 `var i = 0`。

任意的集合都可以被切片，有了这个知识，我们可以回顾一下本章早些时候我们写的前缀生成器的代码，并且将它扩展到任意集合类型：

```
class GPrefixGenerator<Base: CollectionType> {
    let base: Base
    var offset: Base.Index

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

    func next() -> Base.SubSequence? {
        guard offset != base.endIndex else { return nil }
        offset = offset.successor()
        return base.prefixUpTo(offset)
    }
}
```

前向索引

我们选择实现单向链表是因为这个数据结构的特性非常有名：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，那这个操作应该是显而易见地不可能在常数时间内完成的”。

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，不过像是 "hello".uppercaseString 这样的操作，显然不可能在常数时间内完成，不过将它作为一个计算属性，也还算情理之中。

函数的话就不一样了。我们的 List **包含**一个 reverse 操作：

```
let reversed = l.reverse()
```

在这里，reverse 是由标准库中 SequenceType 的协议扩展所提供的：

```
extension SequenceType {
    /// Return an `Array` containing the elements of `self` in reverse
    /// order.
    func reverse() -> [Self.Generator.Element]
}
```

它返回的是一个数组。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 reverse() -> 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 reverse 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 reverse 要比定义在 SequenceType 上的通用同名函数更加专用。

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 reverse 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] = l.reverse()
```

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 reverse 所得到的结果和数组的结果是一样的：

```
l.reverse().elementsEqual(l.reverse as [String])
```

一个小提示：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 reverse 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：

```
l.reverse() as Any is List<Int>
```

双向索引

BidirectionalIndexType 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 predecessor。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 CollectionType where Index: BidirectionalIndexType {
    var last: Generator.Element? {
        guard !isEmpty else { return nil }
        return self[endIndex.predecessor()]
    }
}
```

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。

这个协议还为我们带来了一个更高效的 reverse 操作，它不会立即去将集合中的内容反向，而是返回一个延时加载集合：

```
extension CollectionType where Index : BidirectionalIndexType {
```

```
    /// 返回一个包含 `self` 中的元素逆序后的延迟加载的集合
    func reverse() -> ReverseCollection<Self>
}
```

和上面 SequenceType 上的 enumerate 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，并且返回一个原来集合的逆序的索引。在这个索引中，successor 和 predecessor 被互换了。我们可以通过一个相当简单的函数来操作原集合和它的索引，从而达到我们的目标。

```
struct ReverseIndex<Base: BidirectionalIndexType> : BidirectionalIndexType {
    let idx: Base

    func successor() -> ReverseIndex<Base> {
        return ReverseIndex(idx: idx.predecessor())
    }
    func predecessor() -> ReverseIndex<Base> {
        return ReverseIndex(idx: idx.successor())
    }
}

func ==<T>(lhs: ReverseIndex<T>, rhs: ReverseIndex<T>) -> Bool {
    return lhs.idx == rhs.idx
}

struct ReverseCollection
    <Base: CollectionType where Base.Index: BidirectionalIndexType>
    : CollectionType
{
    let base: Base

    var startIndex: ReverseIndex<Base.Index> {
      return ReverseIndex(idx: base.endIndex)
    }
    var endIndex: ReverseIndex<Base.Index> {
      return ReverseIndex(idx: base.startIndex)
    }
    subscript(idx: ReverseIndex<Base.Index>) -> Base.Generator.Element {
        return base[idx.idx.predecessor()]
    }
```

}

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，这个封装将原来的集合“复制”到了 base 变量中。不过，对于像是 Array 这样的写时复制的类型 (或者像是 List 这样的持久化结构体，或者像是 Queue 这样由两个写时复制类型组成的类型)，这个操作是很高效的。但是当 base 的源数据被替换的时候，这个类型中持有的 base 的副本并不会改变。也就是说，ReverseCollection 可观测到的行为和通过 reverse 返回的数组的行为是一致的。

随机存取索引

索引中最高层的一环是随机存取索引 (random access index)。随机存取索引可以让你在常数时间内跳到索引的任意点上。为了实现这个，随机索引在双向索引 BidirectionalIndexType 的基础上添加了两个函数：distanceTo 和 advancedBy，它们两者都需要是 $O(1)$ 复杂度的。

一开始，你可能觉得这没什么大不了的。如果你观察任意索引类型，包括像是 List 类型中使用的前向索引，你会发现它们都拥有和随机索引一样的 advancedBy 函数。但是它们从本质上来说是不同的。对于 ForwardIndexType 和 BidirectionalIndexType 的 advancedBy 函数和 distanceTo 函数来说，它们是以**扩展**的方式被定义的。它们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。

我们已经看过很多集合类型了，它们全部都是结构体。在结构体和类中我们会更深入地介绍值类型和引用类型的区别。通过实现像是 SequenceType 或是 CollectionType 这样的协议，我们的自定义类型可以自动获取很多额外的功能。我们还关注了集合类型上的很多操作，比如 map、filter 和返回一个表示集合的延迟加载的 reverse 等。在泛型中我们将研究如何扩展已有集合，并为它们提供额外功能的方法。在协议里，我们会展示让我们自己的协议拥有额外功能的方式。

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<T> {
    case None
    case Some(T)
}
```

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 indexOf 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 CollectionType where Generator.Element: Equatable {
    func indexOf(element: Generator.Element) -> Optional<Index> {
        for idx in self.indices where self[idx] == element {
            return .Some(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 NilLiteralConvertible 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.indexOf("four")
// 编译错误：removeIndex takes an Int, not an Optional<Int>
array.removeAtIndex(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”：

```
switch array.indexOf("four") {
case .Some(let idx):
    array.removeAtIndex(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.indexOf("four") {
case let  idx?:
    array.removeAtIndex(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，因为我们会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：

```
if let idx = array.indexOf("four") {
    array.removeAtIndex(idx)
}
```

和 **switch** 语句一样，使用 **if** 的可选绑定也可以跟一个 **where** 语句。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.indexOf("four") where idx != array.startIndex {
    array.removeAtIndex(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 NSURL、NSData 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**。而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 = NSURL(string: urlString),
    data = NSData(contentsOfURL: url),
    image = UIImage(data: data)
{
    let view = UIImageView(image: image)
    XCPlaygroundPage.currentPage.liveView = view
}
```

多个 **let** 的每一部分也能拥有一个 **where** 语句：

```
if let url = NSURL(string: urlString) where url.pathExtension == "png",
    let data = NSData(contentsOfURL: 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.destinationViewController
    as? UserDetailViewController
{
    userDetailVC.screenName = userScreenNameLabel.text
}
```

另外一个例子是你可以使用 NSScanner 来进行扫描，它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 stringScanner = NSScanner(string: "myUserName123")
var username: NSString?
let alphas = NSCharacterSet.alphanumericCharacterSet()

if stringScanner.scanCharactersFromSet(alphas, intoString: &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 where 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() where !line.isEmpty {
    print(line)
}
```

我们在集合一章中提到，for x in sequence 这样的语句需要 sequence 遵守 SequenceType 协议。该协议提供 generate 方法，这个方法返回一个遵守 GeneratorType 的类型。GeneratorType 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 generator = array.generate()
while let i = generator.next() {
    print(i)
}
```

所以，一个 for 循环其实就是 while 循环，这样一来，for 循环也支持 where 语句就是情理之中了：

```
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 where 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 generator = (0..<10).generate()
> while let i = generator.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却是**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。

使用 Python 中的 list comprehensions 实现的版本同样会打印三个 3：

```
a = [lambda: i for i in xrange(1, 4)]
for f in a:
    print f()
```

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("\(f ())  ")
}
```

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，我们使用生成器来实现时可能是这样的：

```
var g = (1...3).generate()
var o: Optional<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：

```
do {
    var g = (1...3).generate()
    var i: Int
    var o: Optional<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，它们都将返回 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", "3", "foo"]
let maybeInts = stringNumbers.map { Int($0) }
```

你现在得到了一个元素类型为 Optional<Int> 的数组，也就是说，maybeInts 是 [Int?] 类型。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 "foo" 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
    // 得到三个整数值和一个 `nil`
}
```

前面我们已经知道 for ... in 是 while 循环加上一个生成器的简写方式，生成器的 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 generator = maybeInts.generate()
while let maybeInt = generator.next() {
    // maybeInt 是一个 Int? 值
    // 得到三个整数值和一个 `nil`
}
```

当循环到达最后一个值，也就是从 “foo” 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?
    // 1, 2, 和 3
}

// 或者只对 nil 值进行循环
for case nil in maybeInts {
    // 将对每个 nil 执行一次
}
```

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) within range")
}
```

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.rangeOfString(pattern.s) != nil
}

let s = "bar"
if case Substring("foo") = s {
    print("has substring \"foo\"")
}
```

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 }
```

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 and while var

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

```
if var i = Int(s) {
    i += 1
    print(i)  // 打印 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：

```
if !a.isEmpty {
    // 使用 a[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 {
    // 使用 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代码。看看下面这个例子，它是 pathExtension 方法的一部分实现：

```
func doStuffWithFileExtension(fileName: String) {
    let period: String.Index
    if let idx = fileName.characters.indexOf(".") {
        period = idx
    } else {
        return
    }

    let extensionRange = period.successor()..<fileName.endIndex
    let fileExtension = fileName[extensionRange]
    print(fileExtension)
}
```

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。

```
func doStuffWithFileExtension(fileName: String) {
    guard let period = fileName.characters.indexOf(".") else { return }

    let extensionRange = period.successor()..<fileName.endIndex
    let fileExtension = fileName[extensionRange]
    print(fileExtension)
}
```

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 **@noreturn** 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**。

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。

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

和 Objective-C 不同的是，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：

```
// 假设我们有一个 `Int?` 的变量 i，我们想要找到它的下一个值
let j: Int
if i != nil {
    j = i!.successor()
} else {
   // 这里没有合理的处理方法
   fatalError("no idea what to do now...")
}
```

如果 i 是非 **nil**，那么 j 会有下一个值。但是如果 i 是 **nil** 的话，那么 j 就没有办法设置一个值了。因此使用可选链的时候，j **只能**是可选值，因为它需要考虑 i 可能为 **nil** 的情况：

```
let j = i?.successor()
```

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 j = Int("1")?.successor().successor()
```

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 successor() 后面不需要加上问号？这是因为可选链是一个 “展平” 操作。Int("1")?.successor() 返回了一个可选值，如果你再对它调用 ?.successor() 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 successor 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 {
    func 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)
```

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, 4, 5, 6, 7]]
let sevenOfNine = dictOfArrays["nine"]?[7] // 返回 .Some(7)
```

另外，还有这样的情况：

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

```
if splitViewController != nil {
    splitViewController!.delegate = myDegelate
}
```

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 i = Int(stringteger) ?? 0
```

这里，如果字符串代表一个整数的话，i 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 i。也就是说 lhs ?? rhs 做的事情类似于这样的代码 lhs != **nil** ? lhs! : rhs。

“真是壮观！”Objective-C 程序员可能会有点反讽地说道，“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 i 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 i = !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：

```
let i = array. first ?? 0
```

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：

```
let i = array.count > 5 ? a[5] : 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：

```
let i = array[safe: 5] ?? 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

let n = i ?? j ?? k ?? 0
```

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 // m 的类型是 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
```

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, 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
(s1 ?? "inner") ?? "outer"
let s2: String?? = .Some(nil)
(s2 ?? "inner") ?? "outer"
```

可选值 map

在之前你看到过这个例子：

```
func doStuffWithFileExtension(fileName: String) {
    guard let period = fileName.characters.indexOf(".") else { return }

    let extensionRange = period.successor()..<fileName.endIndex
    let fileExtension = fileName[extensionRange]
    print(fileExtension)
}
```

我们可以稍作改变，现在不在 **else** 块中从函数返回，而是将 fileExtension 声明为可选值，并且在 **else** 中将它设置为 **nil**：

```
func doStuffWithFileExtension(fileName: String) {
    let fileExtension: String?
    if let idx = fileName.characters.indexOf(".") {
        let extensionRange = idx.successor()..<fileName.endIndex
        fileExtension = fileName[extensionRange]
    } else {
        fileExtension = nil
    }
```

```
    print(fileExtension ?? "No extension")
}
```

这样一来，如果文件名中含有一个句点符号，那么 fileExtension 将包括点之后的内容。但是如果没有的话，fileExtension 将会是 **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：

```
func doStuffWithFileExtension(fileName: String) {
    let fileExtension: String? = fileName.characters.indexOf(".").map { idx in
        let extensionRange = idx.successor()..<fileName.endIndex
        return fileName[extensionRange]
    }

    print(fileExtension ?? "No extension")
}
```

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。

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)：

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

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(combine: (Element, Element) -> Element) -> Element? {
        // 如果数组为空，self.first 将是 nil
        guard let fst = first else { return nil }
```

```
            return self.dropFirst().reduce(fst, combine: combine)
        }
    }
```

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(combine: (Element, Element) -> Element) -> Element? {
        return first.map {
            self.dropFirst().reduce($0, combine: combine)
        }
    }
}
```

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
    }
}
```

可选值 flatMap

我们在集合中已经看到，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 x = stringNumbers.first.map { Int($0) }
```

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) }
```

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, b = Int(a) {
    print(b)
}
```

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 view = NSURL(string: urlString)
    .flatMap { NSData(contentsOfURL: $0) }
    .flatMap { UIImage(data: $0) }
    .map     { UIImageView(image: $0) }

if let view = view {
    XCPlaygroundPage.currentPage.liveView = view
}
```

可选链也和 flatMap 很相似：i?.successor() 实际上和 i.flatMap { $0.successor() } 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, 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
}
```

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, combine: +)
```

```
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 filterNil 函数：

```
func filterNil<S: SequenceType, T where S.Generator.Element == T?>
    (source: S) -> [T]
```

```
{
    return source.lazy.filter { $0 != nil }.map { $0! }
}
```

诶？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 SequenceType 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 SequenceType {
    func flatMap<U>(transform: Generator.Element->U?) -> [U] {
        return filterNil (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。

可选值判等和比较

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：

```
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
```

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["three"]? = nil
dictWithNils.indexForKey("three") // 未找到

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 T: 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 ~=<I: IntervalType>(pattern: I，value: I.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，对于可选值中持有的类型遵守 Comparable 协议的可选值来说，也有像是 < 运算符的实现。**nil** 永远比任何的非 **nil** 值小。需要注意，这也就是说 **nil** 在用 < 进行比较时，它要比**所有的**负值都还小。如果你用 < 运算符进行排序操作的话，你会发现所有的 **nil** 将集中在一边：

```
let temps = ["-459.67", "98.6", "0", "warm"]

temps.sort { Double($0) < Double($1) }

["warm", "-459.67", "0", "98.6"]
```

你必须额外小心，这样才能避免预期之外的结果：

```
let belowFreezing = temps.filter { Double($0) < 0 }
```

这个陷阱在可选值的 == 操作中也会出现。下面的代码得到的结果将为 **true**：

```
let anyOne: Any = "1"
let anyTwo: Any = "99"
(anyOne as? Int) == (anyTwo as? Int)
```

这是因为左右两边都是字符串，而非整数，在类型转换后，判等符号两边都是 **nil**，所以它们相等了。

强制解包的时机

上面提到的例子都用了很利索的方式来解包可选值，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”，“当可以让代码便清晰时使用”，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，看看这个 flatten 的实现：

```
func flatten<S: SequenceType, T where S.Generator.Element == T?>
    (source: S) -> [T]
{
    return Array(source.lazy.filter { $0 != nil }.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,
]

let people = ages
    .keys
    .filter { name in ages[name]! < 50 }
    .sort()
```

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。

```
let people = ages
    .filter { (_, age) in age < 50 }
    .map { (name, _) in name }
    .sort()
```

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

func !?<T: StringLiteralConvertible>
    (wrapped: T?, @autoclosure failureText: ()->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?,
    @autoclosure nilDefault: () -> (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: ()?, @autoclosure failureText: ()->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

因为隐式可选值会尽可能地隐藏它们的可选值特性，所以它们在行为上也有一些不一样。

有时候它们会以编译错误的形式出现，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
}

// 普通的 Int
var i = 1
// 将 i 增加为 2
increment(&i)

// 隐式解包的 Int
var j: Int! = 1
// 错误：cannot invoke 'increment' with an argument list of type '(inout Int !)'
increment(&j)
```

其他时候问题可能更隐蔽一些，特别是在嵌套的隐式可选值中会特别奇怪。在隐式解包可选值表现得和非可选值类似的同时，你依然能够使用像是可选链，**nil** 合并，**if let** 和 map 这样的解包技术来安全地处理它们：

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

但是双重嵌套的可选值就有点棘手了。如果将一个双重嵌套可选值传递到函数中，Swift 会进入到内部值中，并把这个内部值传递给函数。这样一来，要是可选值内部是 **nil** 的话，下面的代码就将崩溃：

```
func useString(s: String) {
    print(s)
}
let s: String!! = nil
useString(s)
```

可选值链的情况却又相反，它会操作**外层**值。所以下面的代码返回 **nil**：

```
let s: String!! = nil
s?.isEmpty  // nil
```

但是，这样就将崩溃：

```
let s: String!! = .Some(nil)
s?.isEmpty // 运行时错误： unexpectedly found nil
// 使用两个 ?? 来进入到内部可选值
s??.isEmpty // nil
```

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，我们有三种不同的选择：结构体、枚举和类。其实还有第四种选择，那就是使用捕获变量的闭包，我们也会在本章中提及这个内容。在 Swift 标准库中，大约百分之九十的公开的类型都是结构体，而枚举和类大概各占百分之五。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来间接地访问。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。
- → 除非一个类被标记为 final，否则它都是可以被继承的。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，以及 **let** 和 **var** 在类和值类型上表现的不同。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。我们还会详细谈谈两者之间内存管理上的差异，特别是引用类型的内存管理的方式。在有了这些的基础上，我们会分别使用引用类型和值类型来解决一个同样的问题。最后，我们会讨论继承的话题，然后将一个基于类的代码设计重构为基于结构体的设计。

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着同样的内存地址。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 NSFileHandle 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。但是，如果它们是用原生 Swift 些的话，它们应该被实现为结构体。

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) 是一个好事请，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。不过两者是互补的，编译器移除那些不必要的 “无效” 浅复制，以及像是数组这样的类型中的代码执行 “智能的” 写时复制，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，通常要比一个含有对象的数组高得多。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，我们可能总是需要类。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，CIImage 对象是不可变的：它们代表了一个永不变化的图像。我们甚至可以将类封装到结构体里，但是我们需要特别小心地来维护这种类型的值语义。

有些时候，决定你的新类型应该是结构体还是类并不容易。如果你将一个表示地址的 Address 类型定义为对象的话，因为对象是通过引用进行传递的，所有对这个地址的更改都会影响所有指向相同对象的变量。作为对比，如果你将 Address 定义为结构体的话，所有的变更都是本地的。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

在包括 Objective-C 的很多面向对象编程语言种，数据默认是可变的。比如，我们创建一个类的新的实例后，就可以对它进行改变：

```
let fileHandle = NSFileHandle(forReadingAtPath: "test.txt")
fileHandle?.seekToFileOffset(10) // 改变
```

通常，当处理一个对象时，你会假设在一个方法的这段时间内一些属性是不会改变的。比如，我们想让代码只在文件句柄的位置是零的时候执行。在下面的代码中，我们通过一个简单的条件进行了检查：

```
if fileHandle.offsetInFile == 0 {
```

```
    prepareHandle(fileHandle)
    // 继续处理
}
```

但是可能 prepareHandle 会改变文件句柄的偏移值，在这个方法调用后，文件句柄可能会有一个不同的 offset。如果我们要确保文件句柄不被改变，我们要么向 prepareHandle 传递一个复制，要么再次进行检查。对于文件句柄的这个例子来说，我们无法对其创建复制。

当处理多线程的代码时，这个问题更加突出。看看下面的代码：

```
let  stream: NSInputStream = ...
dispatch_async(...)
while stream.hasBytesAvailable {
    // 处理输入流
}
```

如果这个输入流在线程中共享的话，在 **while** 执行的过程中，hasBytesAvailable 的值有可能会改变。这将导致不可预期的结果。比如，另一个线程在条件检查后开始处理这个输入流，而此时 **while** 中的语句还没有执行，我们的代码将可能发生崩溃。

在近些年，大家在关于不可变性和值类型能够对解决多线程编程时带来的问题这一观点上迅速达成了一致。不可变性和值类型能够让代码更容易被理解，它们增加了开发者对代码能够正确工作的信心，也使代码能够更安全地运行在多线程环境中。

许多框架，包括 Cocoa 在内，都有不可变对象。这些类的实例没有公开的接口来改变对象的内部状态。比如 Cocoa 拥有 NSData 类，它没有方法改变其所代表的数据。相应地，继承了 NSData 的 NSMutableData 类则添加了一些可变的方法。

然而，这种不可变性是没有强制保证的。在下面的函数中，我们假设了 x 是不可变的，也就是说我们希望在函数调用的过程中，它不会发生改变：

```
func processData(x: NSData) {
    // 对 x 进行操作
}
```

但是假设这种 x 确实不变是不正确的。下面的代码里，我们创建了一个可变的数据对象。因为 NSMutableData 是 NSData 的子类，我们可以将它传递给需要的参数类型为 NSData 的函数。因为实际上这个对象是一个 NSMutableData 的实例，它可能会在函数外部被改变，这将造成 processData 函数中的 x 也发生变化：

```
processAsync {
    let y = NSMutableData() //...
    onMainThread {
        processData(y)
    }
    // 对 y 进行改变
}
```

上面的例子里，我们同时控制调用者和被调用的方法，所以我们有两种方法解决这个问题。比如可以传递一个数据的不可变**复制**给 processData。不过更可靠的做法是在 processData **内部**进行复制，因为这个方法可能会被不是由我们所控制的代码调用。所以，我们先进行复制，然后再对它操作：

```
func processData(x: NSData) {
    let data = x.copy() as! NSData
    // 操作 data
}
```

这种方法有两个缺点。首先，我们可能进行了不必要的复制。对于那些已经是不可变的数据 (比如一个 NSData 对象)，Foundation 框架足够智能，会返回同样的对象。对于那些可以改变的数据 (NSMutableData 对象)，即使在复制后不再会改变，但我们仍然需要进行复制操作。其次，我们需要在每个可能被使用可变值进行调用的地方手动进行复制。要是我们忘记了进行复制，就会意外地引入 bug。更糟糕的是，我们可能在运行的时候注意不到这个问题的存在，因为非同步的 bug 通常并不会稳定再现。

在 Swift 中，我们可以让编译器帮助我们确保不可变性。这个保证既对结构体有效，也可以对类进行使用。首先，让我们来看看一个带有可变变量的类：

```
class Person {
    var name: String
    init(name: String) {
        self.name = name
    }
}
```

我们可以创建一个类的实例，并且改变名字属性：

```
let p = Person(name: "John")
p.name = "Dave"
```

```
p.name.appendContentsOf(" Smith")
```

然而，如果我们使用 **let** 来定义名字的话，它将只能在初始化时被赋值一次。在这之后，就永远不会再发生改变了：

```
class ImmutablePerson {
    let  name: String

    init (name: String) {
        self.name = name
    }
}
```

下面的代码无法编译，因为编译器确保了 name 属性的不可变性。我们既不能改变属性本身 (比如为它赋一个新值)，也不能改变它的值 (比如在字符串后面添加其他东西)：

```
let  person = ImmutablePerson(name: "John")
person.name = "Dave" // 无法编译
person.name.appendContentsOf(" Smith") // 无法编译
```

然而，将属性用 **let** 定义并不意味着保证了不可变性。比如，下面这个类的例子中：

```
class File  {
    let  data: NSMutableData

    init (data: NSMutableData) {
        self.data = data
    }
}
```

虽然 File 类的实例中我们用了 **let** 来定义 data，但是我们还是能改变 data。确实，我们不能改变这个属性指向**哪个对象**，但是我们**可以**改变这个被指向对象本身的内容：

```
let  data = NSMutableData()
let  file  = File(data: data)
file .data = NSMutableData() // 无法编译
file .data.appendData(someOtherData) // 可以正确运行
```

这会导致我们意料之外的结果：就算一个属性被定义为 **let**，看起来是不可变的，但是它的值的内部其实是可以改变的。和上面的例子相比，不同之处在于 Swift 的 String 类型是一个结构体，而 NSMutableData 是类。将一个结构体变量标记为 **let** 可以确保它是不可变的，但是将一个对象标记为 **let** 仅只是避免了引用变量被改变。编译器不允许我们改变那些被定义为 **let** 的结构体属性，但是我们依然能改变使用 **let** 定义的对象。在下一节中，我们会详细讨论它们的工作方式。

值类型

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
```

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。

但是可以和一个简单的类的例子比较一下。Apple 的 Core Data 框架中的 CIFilter 类提供了一个输出图像，该图像通常由输入图像和一些其他参数进行配置。在这个例子中，我们创建了一个高斯模糊滤镜：

```
let inputParameters = [
    kCIInputRadiusKey: 10,
    kCIInputImageKey: image
]
let blurFilter = CIFilter(name: "CIGaussianBlur",
    withInputParameters: inputParameters)!

let secondBlurFilter = blurFilter
secondBlurFilter.setValue(20, forKey: kCIInputRadiusKey)
```

对象是通过引用进行传递的，因此，secondBlurFilter 变量所指向的是同一个 CIFilter 实例：它们都是同一个对象的**引用**。改变其中一个的 inputRadius 属性也会改变另一个变量上的 inputRadius 值，因为它们所指向的对象就是同一个。因为两个变量都引用了同一个对象，所以现在它们的半径值都被更新到了 20。

如果我们想要第二个滤镜和第一个不同的话，我们需要在把它赋值给新的变量之前，手动进行复制：

```
let otherBlurFilter = blurFilter.copy() as! CIFilter
otherBlurFilter.setValue(20, forKey: kCIInputRadiusKey)
```

正如上面所指出的，在 Swift 中结构体和枚举都是值类型。我们可以创建我们自己的 GaussianBlur 结构体，来使用变量存储输入图像和半径：

```
struct GaussianBlur {
    var inputImage: CIImage
    var radius: Double
}

var blur1 = GaussianBlur(inputImage: image, radius: 10)
blur1.radius = 20
var blur2 = blur1
blur2.radius = 30
```

因为这里 GaussianBlur 是一个值类型，将 blur1 赋值给新的变量 blur2 将进行一个复制。接下来对这个复制所进行的改变不会影响到原来的值：当我们设置 blur2.radius 时，blur1 的值不会改变。虽然这种复制听起来有点浪费，但是编译器通常可以把复制操作给优化掉。另外，要是结构体是用写时复制的方式来实现的话，实际对数据的复制只会在其中某个值实际改变的时候才会发生。其实这也是 Swift 数组的工作方式；它们是以写时复制的方式实现的。我们之后会仔细研究如何使用这个技术。

我们可以使用扩展的方式在 GaussianBlur 结构体上添加一个 outputImage 属性。这里，我们创建了一个 CIFilter，设置了默认属性，然后用结构体中的值对它进行了配置。下面的代码在每次 outputImage 被访问时都会创建一个新的滤镜。这并不是很高效的做法，如果我们多次访问这个属性，会有好多新的滤镜被创建出来。

```
extension GaussianBlur {
    var outputImage: CIImage {
        let filter = CIFilter(name: "CIGaussianBlur", withInputParameters: [
            kCIInputImageKey: inputImage,
            kCIInputRadiusKey: radius
        ])!
      return filter.outputImage!
    }
```

```
}
```

一种更高效的尝试

想要提高性能的话，我们可以尝试使用一种进行了些许修改的解决方案。这种方式其实**并不正确**，但是我们会把它作为一个基础，在下一节中会由此引出正确的实现。相比于将值存储在结构体中，我们选择存储一个 CIFilter 实例，然后通过自定义属性来对其进行修改。我们将 filter 声明为 **var**，因为我们稍后会需要更改它：

```
struct GaussianBlur {
    private var filter: CIFilter

    init(inputImage: CIImage, radius: Double) {
        filter = CIFilter(name: "CIGaussianBlur", withInputParameters: [
            kCIInputImageKey: inputImage,
            kCIInputRadiusKey: radius
        ])!
    }
}
```

接下来，我们实现 inputImage 和 radius 属性，来直接访问和修改滤镜的属性：

```
extension GaussianBlur {
    var inputImage: CIImage {
        get { return filter.valueForKey(kCIInputImageKey) as! CIImage }
        set { filter.setValue(newValue, forKey: kCIInputImageKey) }
    }

    var radius: Double {
        get { return filter.valueForKey(kCIInputRadiusKey) as! Double }
        set { filter.setValue(newValue, forKey: kCIInputRadiusKey) }
    }
}
```

最后，要取出输出图像，我们可以直接使用滤镜上的 outputImage 属性：

```
extension GaussianBlur {
    var outputImage: CIImage {
```

```
        return filter.outputImage!
    }
}
```

我们还是有同样的基于结构体的 API，要是我们只有一个滤镜的话，所有事情都将按预想工作：

```
var blur = GaussianBlur(inputImage: image, radius: 25)
blur.outputImage
```

不过，一旦我们使用结构体的复制的话，就会遇到一个问题：

```
var otherBlur = blur
otherBlur.radius = 10
```

当创建变量 otherBlur 时，blur 结构体将被复制。Swift 结构体的工作方式是它其中的所有值类型将被复制，而引用类型却只有对于对象的引用会被复制，对象本身不被复制。在这里，虽然 otherBlur 和 blur 是不同的值，但是它们中的滤镜却指向了相同的 CIFilter 实例。现在，两个值的半径都会是 10，因为对于半径的 setter 方法也该变了原来的对象的值。

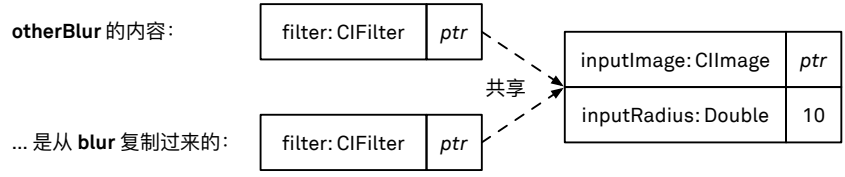

图 5.1: 引用在 blur 和 otherBlur 之间被共享了

写时复制

Swift 在实现值语义的结构体时，其底层使用了可变的对象，这正是我们在上节中所做的事情，这也正是 Swift 中最强大的特性之一。我们从值语义里获益良多，同时又可以保持代码高效。然而，正如我们所看到的，这种做法可能导致意外的数据共享。在本节里，我们将采取策略，在每次结构体被改变时，去复制一个封装后的对象，以此来避免共享。这种技术就叫做写时复制 (copy-on-write)，Swift 的很多数据结构都是以这种方式工作的。要实现写时复制，我们可以利用一个小技巧，来为滤镜自定义存取方法，在返回 CIFilter 之前先对它进行复制：

```
extension GaussianBlur {
    private var filterForWriting: CIFilter {
        mutating get {
            filter = filter.copy() as! CIFilter
            return filter
        }
    }
}
```

这让我们得以更改滤镜的设置方法，在改变值之前先进行复制：

```
extension GaussianBlur {
    var inputImage: CIImage {
        get { return filter.valueForKey(kCIInputImageKey) as! CIImage }
        set {
            filterForWriting.setValue(newValue, forKey: kCIInputImageKey)
        }
    }

    var radius: Double {
        get { return filter.valueForKey(kCIInputRadiusKey) as! Double }
        set {
            filterForWriting.setValue(newValue, forKey: kCIInputRadiusKey)
        }
    }
}
```

上面这种方式可以按照预想工作，但是却带来了性能上的代价：每次我们改变滤镜值的时候，滤镜都会被复制，即便是我们只是做一些本地修改，而没有共享结构体时，这个复制也会发生：

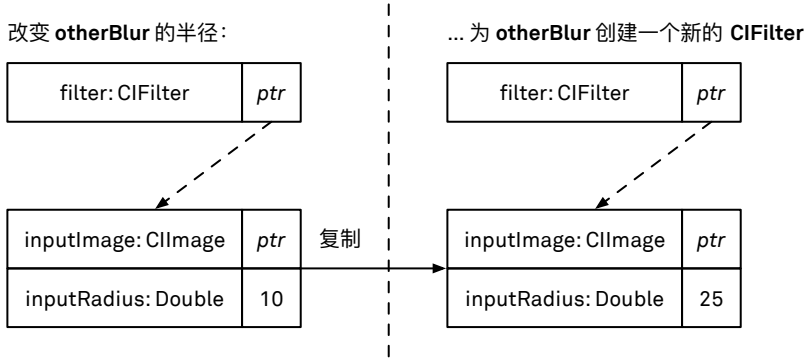

图 5.2: 改变滤镜总会触发复制

高效的写时复制

在 Swift 中有一个方法，isUniquelyReferencedNonObjC，它会检查一个类的实例是不是唯一的引用。我们可以使用它来在保证结构体的值语义的同时，避免不必要的复制。我们可以只在对象被不同结构体共享，并且我们要对它进行更改的时候才进行复制。不幸的是，isUniquelyReferencedNonObjC 函数只对 Swift 对象有用，而 CIFilter 是一个 Objective-C 类。想要绕过这个限制，我们可以创建一个简单的 Box 封装类型，来把任意的 Swift 类封装进去：

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

这个函数让我们可以更改我们的实现。我们将 filter 转换为一个计算属性，它会负责对 Box 进行打包和解包：

```
    private var boxedFilter: Box<CIFilter> = {
        var filter = CIFilter(name: "CIGaussianBlur",
            withInputParameters: [:])!
        filter.setDefaults()
        return Box(filter)
```

```
    }()

    var filter: CIFilter {
        get { return boxedFilter.unbox }
        set { boxedFilter = Box(newValue) }
    }
```

> 其实还有一个 isUniquelyReferenced 函数，不过它也不能作用于 Objective-C 类型。虽然它确实可以接受 Objective-C 作为输入，但是返回值一直是 **false**。实际上这两个函数的实现是一样的，只是它们所接受的输入类型的约束有所不同。

有了这个以后，我们就能够修改 filterForWriting 的实现了。现在我们可以检查滤镜是不是被共享的，并且只在需要的时候创建复制。要是这个实例是滤镜唯一的持有者，那么我们就可以安全地更改已经存在的滤镜：

```
private var filterForWriting: CIFilter {
    mutating get {
        if !isUniquelyReferencedNonObjC(&boxedFilter) {
            filter = filter.copy() as! CIFilter
        }
        return filter
    }
}
```

这正是 Swift 数组内部的工作方式。当你创建一个新的数组的复制时，它背后的数据是一样的。只有当你要修改这个数组时，才会进行复制，这样一来，对该数组的更改不会影响到其他的数组。

我们可以通过编写一些小的测试来验证这个行为。首先，如果我们创建两个共享同一属性的结构体，不会进行复制：

```
let blur = GaussianBlur(inputImage: image, radius: 10)
var blur2 = blur
assert(blur.filter === blur2.filter)
```

当我们改变 blue2 的半径时，filterForWriting 的实现会检测到这个滤镜对象是被共享的，因此它将进行复制：

```
blur2.radius = 25
assert(blur. filter  !== blur2. filter )
```

最后，如果我们再次更改 blue2，因为现在滤镜已经是唯一的了，所以再次对它进行改变并不会导致再次复制：

```
let  existingFilter  = blur2. filter
blur2.radius = 100
assert(existingFilter  === blur2.filter)
```

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。你不需要操心去手动复制那些结构体，因为编译器已经帮你做了这件事，同时，复制也只在绝对必要的时候才会进行。

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。当我们将数组传递给函数时，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。

在当创建不可变类型时，类也还是有其用武之地的。有时候你会想要定义一个从不会被共享的不可变类型，比如代表一个唯一的资源的单例。通过限制接口的定义，这是可以做到的。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。

将已经存在的类型封装到结构体 (或者枚举) 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。就算是 Foundation 中像是 NSData 这样的数据类型，也可以被封装到结构体中，这将帮助你写出更加安全和高效的代码。

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。

回顾一下在集合中的生成器的例子吧，其实我们已经看到过这种行为了。当我们使用生成器时，生成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生成器，我们必须实现 SequenceType，来提供一个返回生成器的函数。

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
    }
}
```

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]
print(john)
```

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。要是 Person 是一个类的话，我们就会引入一个循环引用。但是在这里的结构体版本中，john 只有一个持有者，那就是原来的变量值 john。

> Swift 的结构体一般被存储在栈上，而非堆上。不过这也有例外，如果结构体的尺寸是动态的，或者结构体太大的话，它还是会被存储到堆上。另外，如果结构体值被函数所持有 (就像使用闭包的例子中那样)，那么为了持久化，即使程序已经离开了定义该值的作用域，这个值也会被存储在堆上。

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，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。

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。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：

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

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** 引用表示当被引用的对象被释放时，将引用设置为 **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**。

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

```
    }
}

class Window {
    weak var rootView: View?
}
```

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
```

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
    }
}

class Window {
    var rootView: View?
}
```

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：

```
var window: Window? = Window()
var view: View? = View(window: window!)
window?.rootView = view
view = nil
window = nil
```

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。

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，结果将是不确定的。

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**。

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，这样它们就能够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易于理解。

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

除了类以外，还有另一种引用类型：闭包。我们在可变性一节中已经看到，闭包可以捕获变量。如果这些变量是引用类型的话，闭包将会持有指向它们的强引用。为了捕获变量，这是必要的，但是它也有不好的地方：这很容易引入引用循环。

为了表明对象的初始化和释放，我们创建了一个简单的类来打印这两个阶段：

```
class Example {
    init () { print("init") }
    deinit { print("deinit") }
```

```
}
```

为了测试 **deinit** 是否被调用，我们创建了一个新的函数。在函数内部，我们初始化一个新的实例。在里面我们调用初始化方法，此时 example 的引用计数为 1。接下来函数内部的打印语句将被执行。最后，因为 example 超出了作用域，引用计数也将被减 1。这时 example 的引用计数为 0，所以它将被释放，**deinit** 会被调用：

```
func newScope() {
    let example = Example()
    print("About to leave the scope")
}
```

我们可以通过调用这个函数来进行验证。不过，当我们创建一个引用了 example 的闭包时，这个闭包将持有 example 的强引用，直到它超出作用范围。在这个例子里，我们在函数中返回了一个闭包：

```
func capturingScope() -> () -> () {
    let example = Example()
    return { print(example) }
}
```

如果我们把返回的值赋值给一个变量，我们就可以把闭包保存下来。这时，我们就可以看到 example 是不会被释放的，这是因为闭包仍然对其有一个强引用：

```
let z = capturingScope()
```

不过，如果我们忽略掉这个返回值，闭包就不会有强引用指向它，所以这个闭包将被立即销毁。这会导致 example 的引用计数也掉到 0，它也将被销毁。

```
let _ = capturingScope()
```

当操作闭包时，我们需要留意闭包引用了些什么东西。特别要是闭包引用了自己的时候，非常容易就会造成引用循环，我们将在下一节看到这种情况。

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。举个例子，view controller 引

用了一个 XML 解析器。这个 view controller 通过设置 XML 解析器的回调函数对其进行配置。当回调函数引用了这个 view controller 时，引用循环就出现了。

用图表来看的话，就是这样的：

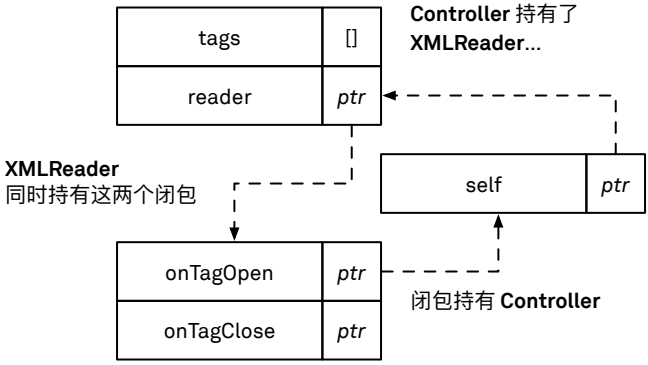

图 5.3: 读取器，控制器和闭包之间的引用循环

把这个翻译成代码，XMLReader 是这样的：

```
class XMLReader {
    var onTagOpen: (tagName: String) -> ()
    var onTagClose: (tagName: String) -> ()

    init(url: NSURL) {
        // ...
        onTagOpen = { _ in }
        onTagClose = { _ in }
    }

    deinit {
        print("reader deinit")
    }
}
```

控制器初始化一个 XMLReader，它还有一个空数组，我们会在其中存储那些我们读到的 tag 名字：

```
class Controller {
    let reader: XMLReader = XMLReader(url: NSURL())
    var tags: [String] = []

    deinit { print("controller deinit") }
}
```

引用循环是由 viewDidLoad 中的代码造成的。这里，我们为 XMLReader 的 onTagOpen 回调设置了一个新的闭包。这个闭包向 view controller 中的 tags 数组中添加一个 tag。这意味着闭包需要持有一个指向 view controller 的引用，XMLReader 将持有这个闭包的引用，而 view controller 又持有了这个 XMLReader，这样就造成了引用循环：

```
func viewDidLoad() {
    reader.onTagOpen = {
        self.tags.append($0)
    }
}
```

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 XMLReader 的引用变为 weak。不过不幸的是，这会导致 XMLReader 消失，因为没有其他指向它的强引用了。
- → 我们可以将 XMLReader 的 onTagOpen 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 XMLReader 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 controller。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。

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 controller。我们可以通过使用**捕获列表**并将捕获变量 **self** 标记为 **weak** 或者 **unowned**。在这个例子中，我们知道 controller 的生命周期会比 XMLReader 长 (因为读取器是被 controller 拥有的)，所以我们可以使用 **unowned**：

```
func viewDidLoad() {
    reader.onTagOpen = { [unowned self] in
        self.tags.append($0)
    }
}
```

如果我们选择了 **weak** 引用，那么 **self** 就会是一个可选值，调用看起来就是这样的：

```
self?.tags.append($0)
```

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 myReader 来引用读取器，我们可以将它在捕获列表中进行初始化：

```
reader.onTagOpen = { [unowned self, weak myReader = self.reader] tag in
    self.tags.append(tag)
    if  tag == "stop" {
        myReader?.stop()
    }
}
```

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。这个变量的作用域只在闭包内部，在闭包外面它是不能使用的。

在下一章有关函数的章节里，我们会探讨 **@noescape**。这个关键字在避免处理闭包时的引用循环很有帮助。

案例学习：使用结构体进行游戏设计

让我们来考虑制作一个角色扮演游戏，游戏里玩家能够探索不同的关卡，并寻找各种物品。我们会从一个非常简单的版本开始。这个初始版本中有一个 Player 类，一个 Health 类，以及每个物品所对应的类。

我们可以用 NSUserDefaults 来保存游戏，它为 iOS 和 OS X 程序提供了一种简单的键值存储方法。我们使用依赖注入的方式让持久化代码可以被测试：也就是说，不直接使用 NSUserDefaults.standardUserDefaults，而是每个类都会在初始化方法中接受一个 NSUserDefaults 实例。这样，在产品代码和测试代码中，我们就能够传入不同的对象来进行存储了。

我们可以创建一个 Storage 协议作为具体的持久化 API 的抽象，这样我们能让测试更加容易一些。我们的游戏类型只关心定义在协议中的那些 API，而不用去管它们的实际的实现。这让我们能够在测试里很容易地将 NSUserDefaults 替换为一个其他的 mock 对象。同时，这也让我们可以在今后可以很灵活地把 NSUserDefaults 替换为其他的存储技术。在这个简化版本里，协议提供了一个下标来获取和设置整数。因为持久化值可能会不存在 (比如在我们第一次运行游戏时)，所以这个下标方法返回可选值。当然，我们让 NSUserDefaults 遵守这个协议：

```
protocol Storage {
    subscript(name: String) -> Int? { get set }
}

extension NSUserDefaults: Storage {
    subscript(name: String) -> Int? {
        get {
            return (objectForKey(name) as? NSNumber)?.integerValue
        }
        set {
            setObject(newValue, forKey: name)
        }
    }
}
```

我们的 Player 类含有三个属性。首先是一个 health 对象。它存储玩家的食物点数和经验点数。其次是 chocolates 属性，它是巧克力盒 (BoxOfChocolates) 这个物品的可选值 (在实际的游戏中，可能会有好多不同的物品，不过现在，我们的游戏中就只有这一件物品)。最后我们将会初始化一个 Storage 对象 (它实际上就是一个 NSUserDefaults 对象)。Storage 对象会在初始化过程中被用来加载和存储游戏状态。我们将这个 Storage 对象作为属性保存起来，这样我们之后就能用它来保存游戏的状态。我们在声明时提供了一个默认值，所以我们不需要在每次初始化 Player 时再去提供这个值。不过我们还是提供了一种传入自定义 Storage 值的初始化方式，我们会在书写测试的时候用到它们：

```
class Player {
    let health: Health
```

```
    var chocolates: BoxOfChocolates?
    let storage: Storage

    init(storage: Storage = NSUserDefaults.standardUserDefaults()) {
        self.storage = storage
        health = Health(storage: storage)
    }
}
```

玩家对象里还有两个我们没有提到过的方法：study 和 save。study 方法使用生命值来获取经验，save 方法会调用玩家对象中的属性的 save 方法，来通过 Storage 对象将游戏状态进行持久化。

对于 Health 类，我们也可以先从提供一个非常简单的实现开始。我们将 foodPoints 和 experiencePoints 存储为整数。在初始化方法中，我们要求传入一个满足 Storage 协议的值。接下来，我们将这个 Storage 对象存储在 storage 变量中，这样在之后有人调用 save 时，我们就可以使用它。对于 foodPoints 和 experiencePoints，我们添加了一个 **didSet** 方法，用来在属性改变时对玩家状态进行存储：

```
class Health {
    var foodPoints: Int = 100 {
        didSet { save() }
    }
    var experiencePoints: Int = 0 {
        didSet { save() }
    }

    var storage: Storage

    init(storage: Storage) {
        self.storage = storage
        foodPoints = storage["player.health"] ?? foodPoints
        experiencePoints = storage["player.experience"] ?? experiencePoints
    }

    func save() {
        storage["player.health"] = foodPoints
        storage["player.experience"] = experiencePoints
    }
```

```
}
```

BoxOfChocolates 类和 Health 类非常相似。它将巧克力的数量用 Int 进行存储，save 和 **init** 的实现也是一样的：

```
class BoxOfChocolates {
    private var numberOfChocolates: Int = 10 {
        didSet { save() }
    }
    var storage: Storage

    init(storage: Storage) {
        self.storage = storage
        numberOfChocolates = storage["player.chocolates"]
            ?? numberOfChocolates
    }
}
```

比较有意思的是 BoxOfChocolates 的 eat 方法：它从巧克力盒子中取出并消耗一块巧克力，然后增加玩家的 foodPoints。这个方法还将保证 foodPoints 的上限是 100。相比于在 BoxOfChocolates 中存储一个 Player 的引用 (这很可能导致引用循环)，我们选择在 eat 方法中用参数的形式传递玩家对象：

```
func eat(player: Player) {
    numberOfChocolates -= 1
    player.health.foodPoints = min(100, player.health.foodPoints + 10)
}
```

目前为止都很好，我们在各个独立对象里定义了所有需要的部分。因为我们重写了状态属性的 **didSet** 方法，所以各个属性都将被安全地存储。比方说，我们在代码里调用了 player.health.foodPoints += 1 这样的代码的话，foodPoints 的 **didSet** 就将被触发。

由于我们在处理 user defaults 时使用了依赖注入 (其实我们甚至还用了协议)，我们就让依赖变得非常明显了。现在我们能够很容易地改变这个依赖的类型，比如可以将状态存储到某个 JSON 文件中。同样，我们也能很容易为测试提供另外一种实现方式。

不过，这种设计也有一些不好的地方。首先，当我们的状态改变时，我们必须重写所有的 setter，这样才能对状态进行持久化。其次，更糟糕的是我们在所有的代码中都引入了一个很大的依赖：Storage。就算我们已经让这个依赖非常明显了，想要测试和重构它还是会比较困

难。最后，可能会出现意外的数据共享。虽然 Health 对象的唯一持有者是 Player 对象，然而任意代码都能够保存一个 Health 对象的引用。这也让重构变得更难。

想要改善这个设计，我们可以先把 Storage 依赖给重构掉。我们不再对它进行注入，而是为每个对象提供一个 serialize 方法，它会将对象序列化为一个 plist (一个简单的键值字典)。除此之外，我们还将提供一个初始化方法，来从 plist 中初始化这个值。这让我们可以把 Storage 完全去除掉，这样我们的代码将更容易测试，因为我们不再需要为测试再进行一次实现。不过，我们在属性里放一个 Storage 对象的原因是为了随时可以调用 save。现在没有这个对象了，我们就还需要另外一种方法来在每次变更时自动调用 save。

如果我们将 Player，Health 和 BoxOfChocolates 从类改为结构体，我们就可以将所有的属性上的 **didSet** 移除了。取而代之，我们只需要一个 player 变量上的 **didSet**。更新一个嵌套属性 (比如 player.health.foodPoints) 的操作是一个可变函数，它不仅会改变 health 结构体，也会更改 player 结构体。从语义上来说，改变 player 的一个值，和重新给 player 变量赋一个新的值，除了前者在语法上更自然以外，里面所发生的事情是一样的。

现在，Player 结构体看起来是这样的：

```
struct Player {
    var health: Health
    var chocolates: BoxOfChocolates?

    init(properties: PropertyList = [:]) {
        let healthProperties = properties["health"] as? PropertyList
        health = Health(properties: healthProperties ?? [:])
        if let chocolateProperties = properties["chocolates"] as? PropertyList {
            chocolates = BoxOfChocolates(properties: chocolateProperties)
        }

    }

    mutating func study() {
        health.foodPoints -= 2
        health.experiencePoints += 1
    }

    func serialize() -> PropertyList {
        var result: PropertyList = [
            "health": health.serialize()
```

```
        ]
        result["chocolates"] = chocolates?.serialize()
        return result
    }
}

typealias PropertyList = [String:AnyObject]
```

Health 结构体就变得非常简单了。它仅存储了两个值，并且知道要如何序列化和反序列化它们。BoxOfChocolates 也和此相似，因为看上去几乎一样，我们这里就不把它列出了：

```
struct Health {
    var foodPoints: Int = 100
    var experiencePoints: Int = 0

    init(properties: PropertyList) {
        foodPoints = properties["food"] as? Int ?? foodPoints
        experiencePoints = properties["experience"] as? Int ?? experiencePoints
    }

    func serialize() -> PropertyList {
        return [
            "food": foodPoints,
            "experience": experiencePoints
        ]
    }
}
```

和之前的设计相比，我们可以把 eat 方法从 BoxOfChocolates 结构体中移出来，写到一个 Player 结构体的一个扩展里：

```
extension Player {
    mutating func eat() {
        guard let count = self.chocolates?.numberOfChocolates where count > 0
            else { return }
        self.chocolates?.numberOfChocolates -= 1
        health.foodPoints = min(100, health.foodPoints + 10)
    }
}
```

相比起原来的基于类的实现，这三个结构体都要简单很多。它们不再有任何依赖，因此更容易被测试。得益于结构体的工作方式，我们现在只用监听到 player 值的改变，就可以得知它其中的某个属性发生了改变，这个属性既可以是直接存储在 player 中的，也可以是进行了多层嵌套的结构体。要重新创建自动保存的功能，我们可以新建一个 GameState 类，在这个类中我们存储和观察 player 属性，并像我们在之前的 Player 中所做的那样来进行初始化。注意我们这里使用的是 Serializer 上的另一个下标读取方法，这个下标不会返回整数值，而是返回一个 PropertyList。

```
class GameState {
    var player: Player {
        didSet { save() }
    }
    var serializer: Serializer

    init(serializer: Serializer = NSUserDefaults.standardUserDefaults()) {
        self.serializer = serializer
        player = Player(properties: serializer["player"] ?? [:])
    }

    func save() {
        serializer["player"] = player.serialize()
    }
}
```

使用结构体的设计有一系列优点。代码要比原来更加简短，直击要害。不需要再对每个结构体添加 Storage 的依赖，相反，它现在被移动到了统一的地方进行处理。这也让重构变得更加容易，除了 user defaults 外，结构体可以很简单地由 JSON 或者 Core Data 来初始化。在基于类的实现中，我们必须对每个属性都重写 **didSet**，但是在结构体的实现里，我们只需要在 GameState 类里做一次这件事情就可以了。在基于类的实现中，有可能意外地在不同类中为两个属性使用了同样的键来进行键值存储。而在基于结构体的实现中，因为每个结构体对应一个字典，最终我们存储的是一个嵌套字典的字典，所以这样的错误较难出现。

不过这个基于结构体的实现也有一个缺点，那就是即使一个很小的部分发生了变化，它还是会保存所有的数据。然而，基于类的策略只会对改变的那部分内容进行持久化。我们其实也可以用结构体实现同样的行为，不过其实一次把所有东西都进行保存也没有糟糕透顶。保持简单，可以让我们在通读结构体的代码时，更容易验证代码的正确性。

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，我们往往需要额外的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的问题，也可能可以从使用值类型中受益。

参考

- → ObjCCN - 结构体和值类型
- → 函数式 Swift
- → Mike Ash

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” 弄混淆了，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)")
}

// 这将你刚才声明的函数赋值给了一个变量
// 注意函数名之后没有括号
let funVar = printInt

// 现在，你可以使用该变量来调用你的函数。
```

```
// 注意在函数名后面的括号
funVar(2)  // 将打印 "you passed 2"

// 你也能够写一个接受函数作为参数的函数
func useFunction(funParam: (Int) -> () ) {
    // 调用输入的函数
    funParam(3)
}

// 你可以将使用原来的函数作为参数调用这个新函数
useFunction(printInt)
// 也可以使用赋值的变量来调用
useFunction(funVar)
```

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：

```
// 这是一个返回函数的函数。
// 返回的是一个接受 Int 作为参数，并返回 String 的函数
func returnFunc() -> (Int) -> String {
    func innerFunc(i: Int) -> String {
        return "you passed \(i) to the returned function"
    }
    return innerFunc
}

let myFunc = returnFunc()
myFunc(3)
```

```
you passed 3 to the returned function
```

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。

为了研究者一点，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 returnFunc() -> (Int) -> () {
    var counter = 0 // 局部变量声明
    func innerFunc(i: Int) {
        counter += i  // counter 被“捕获”了
        print("running total is now \(counter)")
    }
    return innerFunc
    // 一般来说，因为 counter 是一个局部变量，它在此时应该离开作用域并被摧毁。
    // 但是这个因为 innerFunc 使用了它，它将继续存在。
}

let f = returnFunc()
f(3)  // 将打印 "running total is now 3"
f(4)  // 将打印 "running total is now 7"

// 如果我们再次调用 returnFunc() 函数，将会生成并“捕获”新的 counter 变量
let g = returnFunc()
g(2)  // 将打印 "running total is now 2"
g(2)  // 将打印 "running total is now 4"

// 这不影响我们的第一个函数，它拥有它自己的 counter。
f(2)  // 将打印 "running total is now 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 }
```

```
// 下面的代码对每个数字都调用了 doubler，然后返回新的结果数组
let a = [1, 2, 3, 4].map(doubler)

// `a` 现在是 [2, 4, 6, 8].
```

使用闭包表达式的语法来写相同的函数，结果是：

```
let doubler = { (i: Int) -> Int in return i*2 }
// 可以像之前那样使用 doubler
[1, 2, 3].map(doubler)
```

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" 是字符串字面量那样。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。假设你想要一组随机数的数组，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：

```
// 生成 100 个随机数
(0..<100).map { _ in arc4random() }
```

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 IntegerLiteralConvertible {
>     associatedtype IntegerLiteralType
>
>     /// Create an instance initialized to `value`.
>     init(integerLiteral value: Self.IntegerLiteralType)
> }
>
> /// The default type for an otherwise unconstrained integer literal.
> 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 isEven = { (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：

```
var isEven: Int8 -> Bool = { $0 % 2 == 0 }
```

```
// 或者
isEven = { $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 的泛用版本会更好：

```
// 作为协议扩展定义在所有的整数类型上
extension IntegerType {
    func isEven() -> Bool {
        return self % 2 == 0
    }
}

// 或者定义为一个顶层函数
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：

```
// 将泛型函数 `isEven` 的一个指定版本赋值给变量：
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：

```
var myArray = [3, 1, 2]

myArray.sort()

[1, 2, 3]
```

实际上一共有四个排序的方法：sort 和 sortInPlace，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 sort() 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.sort(>)

[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：

```
let numberStrings = ["3", "1", "2"]
numberStrings.map { Int($0) }.sort(<)

[Optional(1), Optional(2), Optional(3)]
```

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.sort { lhs, rhs in
    let l = lhs.characters.reverse()
    let r = rhs.characters.reverse()
    return l.lexicographicalCompare(r)
}
```

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够在很大程度上代替 Foundation 框架中各种不同的排序方法的功能。

为了展示这一点，我门先根据 sortedArrayUsingDescriptors 的文档来重新创建一个很复杂的例子。这个排序方法非常灵活，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。

假设你有一个数组，其中的元素是包含了姓和名的字典：

```
let last = "lastName", first = "firstName"

let people = [
    [first: "Jo", last: "Smith"],
    [first: "Joe", last: "Smith"],
    [first: "Joe", last: "Smyth"],
    [first: "Joanne", last: "Smith"],
    [first: "Robert", last: "Jones"],
]
```

(以免你用不赞成的眼光盯着我们一通看，我们先声明，这里确实使用一个结构体来代表姓名要比用字典好很多，但是我们的目的是举例演示，所以这里使用了字典。)

如果你想要先用姓来排序，然后用名排序，在排序的时候你还想忽略大小写，并且使用用户的区域设置来决定顺序，你应该怎么做？使用 sortedArrayUsingDescriptors 的话，代码差不多是这样的：

```
let lastDescriptor = NSSortDescriptor(key: last, ascending: true,
    selector: #selector(NSString.localizedCaseInsensitiveCompare(_:)))

let firstDescriptor = NSSortDescriptor(key: first, ascending: true,
    selector: #selector(NSString.localizedCaseInsensitiveCompare(_:)))


let descriptors = [lastDescriptor, firstDescriptor]

let sortedArray = (people as NSArray) .sortedArrayUsingDescriptors(descriptors)
```

使用 selector 看起来很棒，你可以用它来在运行时来构建 descriptors 这个排序描述符的数组，这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.sortInPlace {
    return $0.localizedCaseInsensitiveCompare($1) == .OrderedAscending
}
```

如果只是想从字典中取出一个键来进行排序，实现起来也很简单。使用为可选值定义的比较操作符就可以完成了 (在字典中查找的结果是可选值，因为姓氏有可能并不存在)：

```
let sortedArray = people.sort { $0[last] < $1[last] }
```

不过，当你要把这个与像是 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：

```
let sortedArray = people.sort { lhs, rhs in
    return rhs[first].flatMap {
        lhs[first]?.localizedCaseInsensitiveCompare($0)
    } == .OrderedAscending
}
```

而且这么做还没有把先通过姓氏排序，再通过名字排序这件事考虑进去。我们可以用标准库的 lexicographicalCompare 方法来实现它。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。另外，我们这次为了代码整洁一些，使用了 **guard** 关键字：

```
let sortedArray = people.sort { p0, p1 in
    let left = [p0[last], p0[first]]
    let right = [p1[last], p1[first]]

    return left.lexicographicalCompare(right) {
```

```
        // 与可选值的 < 操作相同
        guard let l = $0 else { return false }
        guard let r = $1 else { return true }
        return l.localizedCaseInsensitiveCompare(r) == .OrderedAscending
    }
}
```

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，对可选值的处理也比较混乱。

首先，我们来看看怎么处理可选值吧。如果能**使用一个方法**来简单地比较两个可选值的话，情况就会好得多。和可选值的 < 操作符类似，我们认为 **nil** 要比所有非 **nil** 的值要小。如果比较的两侧都不是 **nil** 的话，我们则对可选值的实际内容进行比较：

```
extension Optional {
    func compare(rhs: Wrapped?,
        _ comparator: Wrapped -> Wrapped -> NSComparisonResult)
        -> Bool
    {
        switch (self, rhs) {
        case (nil, nil), (_?, nil): return false
        case (nil, _?): return true
        case let (l?, r?): return comparator(l)(r) == .OrderedAscending
        }
    }
}
```

这里的逻辑和我们在可选值中看到过得 == 很相似，它表达了 “较小” 这个语义。

柯里化函数

Optional.compare 利用了一系列方法的特性来让调用变得简单。你可以将一个实现了所要求的类型的函数传递给它，compare 将依据输入进行比较，比如：

```
let a: String? = "Fred"
let b: String? = "Bob"
a.compare(b, String.localizedCaseInsensitiveCompare)
```

这里，String.localizedCaseInsensitiveCompare 是一个类型为
String -> String -> NSComparisonResult 的函数。这个函数接受字符串作为输入，然后返回另一个函数。你可以将一个字符串**实例**传递给它，这样你就可以得到一个新的方法，这个方法可以对该实例和另一个字符串进行比较操作。

这也是我们在 compare 里处理两个非 nil 值时，我们必须先用左侧值调用传进来的函数，然后再将右侧值传递给**这个**得到的函数，最终可以得到结果的原因。

```
return comparator(l)(r) == .OrderedAscending
```

这种形式的函数被称为**柯里化** (curried) 函数。当你想要创建一系列的函数，并把它们传递到一个像 map 那样的高阶函数时，这个特性会很有用。

假设你需要经常检查一个数 i 是不是另一个数的倍数 (也就是 i 是否能被这个数整除)。这个数**本身**是可以变化的，但是无论数如何改变，决定是否是倍数的规则是不变的：i % n == 0。

你可能会写出这样的函数：

```
func isMultipleOf(n n: Int, i: Int) -> Bool {
    return i % n == 0
}

isMultipleOf(n: 2, i: 3)  // false
isMultipleOf(n: 2, i: 4)  // true
```

也许很多时候你想要在一个高阶函数中使用它，比如在 map 和 filter 里：

```
let nums = 1...10
// 使用 isMultipleOf 来滤出偶数：
let evens = nums.filter { isMultipleOf(n: 2, i: $0) }
// `evens`是 [2, 6, 8, 10]
```

这种使用 isMultipleOf 的方法看起来有点笨拙，也不容易阅读。所以，相比于 isMultipleOf，可能我们定义一个新的函数 isEven 来判断是否是偶数，看起来会更加清晰：

```
let isEven = { isMultipleOf(n: 2, i: $0) }
isEven(2)  // true
isEven(3)  // false
let evens = r. filter (isEven)
```

现在，我们假设你稍微更改一下 isMultipleOf 的定义，将它写为一个柯里化函数：

```
func isMultipleOf(n n: Int) -> Int -> Bool {
    return { i in i % n == 0 }
}
```

isMultipleOf 现在接受一个数字 n，并且返回一个新的函数。这个新函数接受另一个数字，并且检查它是否是 n 的倍数。

你可以使用这个函数来定义 isEven 函数：

```
let isEven = isMultipleOf(n: 2)
```

或者，你也可以把它直接用在 filter 里：

```
let evens = r.filter(isMultipleOf(n: 2))
// 和之前一样，evens 是 [2, 4, 6, 8, 10].
```

函数作为数据

我们的排序方法现在看起来漂亮而紧凑了，但是它如何比较不同的键这件事情仍然是写死的逻辑。我们要怎么做才能更接近原来的基于排序描述符的排序所能具备的灵活性呢？

问题实际上在于 lexicographicalCompare。这个函数作用于两个序列，这在你有两个序列并且你想要用一个单独的比较函数来对它们比较时非常好用。

但是我们现在的问题正好与此相反。我们有两个**值**，并且我们想要用一系列的比较函数来对它们进行比较。我们需要把 lexicographicalCompare 反转过来，传递一个函数的序列给它，来比较两个值，实现大概是这样的：

```
/// Return true if `self` precedes `other` in a lexicographical ("dictionary")
/// ordering, based on applying each element of `isOrderedBeforeSequence` as
/// the comparison between elements until the first element of
/// `isOrderedBefore` detects inequality.
///
/// - Requires: each element of `isOrderedBeforeSequence` is a
///   strict weak ordering over the elements of `self` and `other`.
///   (http://en.wikipedia.org/wiki/Strict_weak_order#Strict_weak_orderings)
```

```
func lexicographicalCompare<T>
    (comparators: [(T,T) -> Bool]) -> (T, T) -> Bool
{
    return { lhs, rhs in
        for isOrderedBefore in comparators {
            if isOrderedBefore(lhs,rhs) { return true }
            if isOrderedBefore(rhs,lhs) { return false }
        }
        return false
    }
}
```

这个版本的 lexicographicalCompare 将对比较函数的序列进行迭代，对每个比较函数，检查左侧的参数是否小于右侧的参数。通过调换参数的顺序，我们也可以确定左侧的参数是不是大于右侧参数。

如果左右两边的元素相互都不小于另一个的话，那么它们只能是相等的。这时，我们将使用比较函数序列中的下一个比较函数再次进行比较。如果一个值和另一个比较，它既不小于那个值，也不大于那个值的话，这两个值将相等，这个规则是通过严格的弱序化 (strict weak ordering) 来保证的，我们在注释中有所提及。在协议一章里我们会讲到更多的细节。

通过将它用柯里化的方式进行定义，我们可以将一系列比较函数传递给它，并得到一个单一类型为 (T, T) -> Bool 的函数，这正是 sort 函数所需要的类型。

现在，我们可以定义一个比较函数的数组了，它们中的每一个都将对字典中的不同键的值进行比较：

```
let comparators: [([String: String], [String: String]) -> Bool] = [
    { $0[last].compare($1[last], String.localizedCaseInsensitiveCompare) },
    { $0[first].compare($1[first], String.localizedCaseInsensitiveCompare) },
]
```

在排序中使用这个结果：

```
let sortedArray = people.sort(lexicographicalCompare(comparators))
```

我们需要为 comparators 变量指明类型，因为它是被单独声明的。如果对你来说将排序的顺序写死也没什么问题的话，这个类型也可以通过周围的上下文来进行推断：

```
let sortedArray = people.sort(lexicographicalCompare([
    { $0[last].compare($1[last], String.localizedCaseInsensitiveCompare) },
    { $0[first].compare($1[first], String.localizedCaseInsensitiveCompare) }
]))
```

你可以将用这种方式内联声明的函数看作是函数的 “字面量” — 就像你在代码中写 1 这个整数字面量或者 "foo" 这个字符串字面量是一样的。

现在，我们实现了通过代码来对比较的操作重新排序。和比较描述符的实现方式一样，你现在可以在运行时根据用户输入来构建一个比较函数的数组，排序函数的行为也将随之改变。

这也给了我们一种对不同元素使用不同排序顺序的能力。比如我们可以按照升序排序姓氏，然后按照降序排序名字：

```
let sortedArray = people.sort(lexicographicalCompare([
    { $0[last] < $1[last] },
    { $0[first] > $1[first] },
]))
```

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。

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的排序的话，将排序方法从比较方法中分离出来的话，你就可以很简单地交换排序的顺序了。

```
let sortedArray = people.stableSort { lhs, rhs in
    lexicographicalCompare(lhs, rhs, comparators)
}
```

局部函数和变量捕获

如果你想要一个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.appendContentsOf(self[i..<mi])
        tmp.appendContentsOf(self[j..<hi])
        replaceRange(lo..<hi, with: tmp)
    }

    mutating func mergeSortInPlace() {
        let n = count
        var size = 1

        while size < n {
            for lo in 0.stride(to: n-size, by: size*2) {
                merge(lo, (lo+size), min(lo+size*2,n))
            }
            size *= 2
        }
    }
}
```

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，这让事情变得更复杂。可能你会需要自己申请你的缓冲区。

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.appendContentsOf(self[i..<mi])
            tmp.appendContentsOf(self[j..<hi])
            replaceRange(lo..<hi, with: tmp)
        }

        let n = count
        var size = 1

        while size < n {
            for lo in 0.stride(to: n-size, by: size*2) {
```

```
                merge(lo, (lo+size), 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。

一种实现回调的更通用的方法是，使用一个观察者接口 (Observer) 和一个可以接收观察者并在某个事件发生的时候进行回调的可观察接口 (Observable)，像下面这样：

```
protocol Observable {
    mutating func register(observer: Observer)
}

protocol Observer {
    func receive(event: Any)
}

// 接下来这样来进行实现：
struct EventGenerator: Observable {
    var observers: [Observer] = []

    mutating func register(observer: Observer) {
        observers.append(observer)
    }

    func fireEvents(event: Any) {
```

```
            for observer in observers {
                observer.receive(event)
            }
        }
}

struct EventReceiver: Observer {
    func receive(event: Any) {
        print("Received: \(event)")
    }
}

var g = EventGenerator()
let r = EventReceiver()
g.register(r)

var gen = EventGenerator()
let receiver = EventReceiver()
gen.register(receiver)
gen.fireEvents("hi!")
gen.fireEvents(42)

Received: hi!
Received: 42
()
```

上面的代码可以正常工作，但是等等，Any？不好吧，没有类型的东西就是罪恶。应该用泛型来搞定，对吧？下面这种实现如何：

```
protocol ObserverType {
    associatedtype Event
    func receive(event: Event)
}

struct StringEventReceiver: ObserverType {
    func receive(event: String) {
        print("Received: \(event)")
    }
}
```

这修正了类型的问题。当实现 Observable 时，你可以设定事件的类型，那些观察者的 receive 方法也必须匹配这个类型。看起来不错。

不过你会在实现 Observable 时遇到问题。因为 ObserverType 是一个带有关联类型的协议，所以你现在不再能这样来定义 Observable 了：

```
protocol Observable {
    // 错误：protocol 'ObserverType' can only be used as a generic constraint
    // because it has Self or associated type requirements
    mutating func register(observer: ObserverType)
}
```

当你要处理的协议带有 **Self** 或者关联类型的时候，你就不再能将协议的名字当作一个单独的类型来使用了 (我们会在协议中探讨细节)。相反，你需要定义一个泛型方法，它的参数可以是**实现**了 ObserverType 的任意类型：

```
protocol Observable {
    mutating func register<O: ObserverType>(observer: O)
}
```

Observable 协议现在可以编译了，但是它仍然是类型不定的。任意类型的 ObserverType 都可以被注册，不论它们接收的是 String 还是 Int 甚至或者是 Foo。事实上，我们希望 Observable 产生特定类型的事件，所以我们可以也为它添加一个 Event 关联类型：

```
protocol Observable {
    associatedtype Event
    mutating func register
        <O: ObserverType where O.Event == Event>(observer: O)
}
```

使用闭包来消除类型

现在，我们**几乎**可以实现一个将 Event 指定为 String 类型的 StringEventGenerator 了：

```
struct StringEventGenerator: Observable {
    var observers: [???] = []

    // 因为 Event 只在内部被作为占位符使用，Swift 将不会
```

```
    // 推断它的类型，所以我们需要显示地为这个关联类型指定类型别名
    typealias Event = String

    mutating func register
        <O: ObserverType where O.Event == String>(observer: O) {
        observers.append(observer)
    }
}
```

但是 observers 数组的类型应该是什么？在 register 中，O 可以是**任意**类型，但是 Swift 的数组只能包含某一种特定的类型。

一种解决方法是将传递给 register 的类型封装到一个闭包内，这个闭包可以捕获观察者，并且调用它上面的 receive。之后，当我们想要调用 receive 时，我们只需要调用这个闭包。这样一来，observers 数组的类型就应该 [String -> ()]：

```
struct StringEventGenerator: Observable {
    var observers: [String -> ()] = []

    typealias Event = String

    mutating func register
        <O: ObserverType where O.Event == String>(observer: O)
    {
        observers.append { observer.receive($0) }
    }

    func fireEvents(event: String) {
        for observer in observers {
            observer(event)
        }
    }
}
```

现在我们就有一对类型安全的 observable/observer 了：

```
var gen1 = StringEventGenerator()
let rec1 = StringEventReceiver()
gen1.register(rec1)
```

```
gen1.fireEvents("hi!")
```

用函数代替协议回调

我们将观察者存储到数组里的方式为我们指明了实现 Observable 协议的另一条道路，那就是不再使用 ObserverType，而是直接让 register 接受一个函数作为回调：

```
protocol Observable {
    associatedtype Event
    // 移除所有 ObserverType 的引用，只注册一个函数
    mutating func register(observer: Event -> ())
}
```

因为 register 的参数现在已经是闭包了，所以我们不再需要在将它进行封装。除此之外，其他的实现都基本相同：

```
struct StringEventGenerator: Observable {
    var observers: [String -> ()] = []

    mutating func register(observer: String -> ()) {
        observers.append(observer)
    }

    func fireEvents(event: String) {
      for observer in observers {
        observer(event)
      }
    }
}
```

现在，为了把 receiver 也添加进去，我们可以使用之前所看到的柯里化方法的调用：

```
// StringEventReceiver 不再需要满足 ObserverType:
struct StringEventReceiver {
    func receive(event: String) {
        print("Received: \(event)")
    }
```

```
}

var g = StringEventGenerator()
let r0 = StringEventReceiver()

// StringEventReceiver.receive(r0) 返回一个函数，receive 函数将会在 r0 上被调用
let callback = StringEventReceiver.receive(r0)

// 现在，我们注册这个函数，以对 g 进行观察：
g.register(callback)
```

还有一种更简短的方法，那就是使用 instance.method，不添加参数括号，也可以达到同样的效果：

```
let r = StringEventReceiver()
g.register(r.receive)
```

并没有要求说 callback 必须是在一个对象上的调用。独立的闭包也能够被注册：

```
g.register { print("Closure received \($0)") }
```

因为注册所做的不过是传递一个闭包，所以你可以把一个相同的对象的不同的方法多次进行注册：

```
extension StringEventReceiver {
    func receiveDifferently(event: String) {
        print("Received \(event) differently")
    }
}

let receiver = StringEventReceiver()
g.register(receiver.receive)
g.register(receiver.receiveDifferently)
```

当一个协议上只定义了一个函数的时候，将它用闭包的回调代替可以让代码变得简单得多。不过，要是一个协议定义了多个紧密相关的函数的话 (举个例子，比如为 table view 提供数据)，相比于多个单独的回调，将它们分组到一起会更好。我们可以通过使用一个单独的实现了所有相关函数的对象或者结构体来确保这一点。

函数回调和代理协议的另一个不同在于注销的时候。如果我们提供了 API 来让函数注册回调的话，我们也需要一种方式来注销它们。我们可以使用一个 token，或者是返回一个用来注销的回调。而在基于对象的代理中，因为对象具有同一性，所以事情就简单得多，我们只需要简单地从代理的列表中将这个对象移除就可以了。

使用闭包为值类型赋予引用语义

让我们假设现在有简单的结构体，它其中有一个 **mutating** 的方法：

```
struct StringStoringReceiver {
    var str = ""
    mutating func receive(event: String) {
        str += str.isEmpty ? event : ", \(event)"
    }
}

var stringR = StringStoringReceiver()
```

如果我们尝试将这个部分实现的 receive 方法传递给 register 函数的话，我们会得到一个编译错误：

```
// 错误：partial application of 'mutating' method is not allowed
g.register(r.receive)
```

不过，要绕开这个限制很简单。你只需要创建一个闭包就可以了，而不要直接把函数传递给它。闭包将会捕获 stringR，另外由于它是被 **var** 定义的，我们也可以更新它。闭包的类型是 String -> ()，而不像直接传递函数那样是被标记为 **mutating** 的：

```
g.register { stringR.receive($0) }
```

现在我们能够触发事件了，stringR 将得到更新：

```
g.fireEvents("hi!")
g.fireEvents("one")
g.fireEvents("two")
stringR.str

hi !, one, two
```

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。

> **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。

这带来了不少的影响。最好的地方在于，使用 **inout** 比使用引用要安全得多，我们马上来说明这一点。先来看看一段非常直接的代码，它通过 **inout** 参数来曾加一个数字：

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

var x = 0
inc(&x)
x

1
```

输出结果是 1，这是因为 x 通过 **inout** 传递给了 inc 函数，并在函数内部进行了加一。当 inc 函数返回时，i 的值将被复制给 x。

下面是我们熟悉的闭包捕获变量的用法。这种实现方式在 inc 函数里声明了一个变量。这个变量将在方法中被持有，就和对象的实例变量一样，在不同的对 h 的调用中，这个变量是被共享的。

```
func inc() -> () -> Int {
    // 声明变量 i
    var i = 0
    // 并且将它在闭包内捕获，每次调用时加一
    return {
      i += 1
      return i
    }
```

```
}

let h = inc()
print(h()) // 打印 1
print(h()) // 打印 2

1
2
```

现在，让我们尝试将两种方式结合起来。也就是说，创建一个 inc，它接受 **inout** 参数，然后将**这个参数**用一个对它加一的闭包进行捕获：

```
// 接受 inout 参数
func inc(inout i: Int) -> () -> Int {
    // 并将它捕获在返回的函数中
    return { i += 1; return i }
}
```

如果 **inout** 做的是传递引用的话，那么传递给 inc 的变量的值将随着对捕获闭包进行调用次数的增加而增大。但是这并**没有**发生：

```
var x = 0
let fx = inc(&x)
print(fx()) // 打印 1
print(x)    // 仍然是 0

1
0
```

相反，在 inc 中，i 被捕获了，通过值传递并复制回来的这个过程只完成了一半。闭包捕获到了值为 0 的变量 i。当 inc 结束时，被复制回来的实际上是那个**还没有被改变**的值 0。接下来对 fx 的调用所增加的只是被捕获的那个复制。

最后，我们可以通过在 inc 返回前调用一次闭包来将捕获的变量 i 加一，以此验证函数结束时的复制行为：

```
func inc1(inout i: Int) -> () -> Int {
    // 在闭包中存储一份复制
    let f: () -> Int = {
```

```
        i += 1
        return i
    }
    // 在退出前对其进行调用，i 加一
    f()
    // 返回函数
    return f
}

var x = 0
let f = inc1(&x)
x

1
```

现在，因为在返回前闭包对 i 进行了一次操作，然后 i 被复制回了 x，所以 x 也增加了。

可能这些例子看起来都是特殊情况，也一点儿都不重要，但是其实它们是语言安全性的关键。inc 不知道关于传递给它的变量作用范围的任何事情。如果 **inout** 是传递引用的话，想想看下面的情况下会发生什么：

```
// 声明一个变量，稍后会将一个闭包赋值给它
let f: () -> Int

// 开启一个局部作用域
do {
    // 声明局部变量
    var x = 0

    // 调用 inc 并传入局部变量。将返回的闭包存储在外部变量中
    f = inc(&x)

// 局部作用域结束，x 被摧毁
}

// 现在，调用 f，它将访问 x
print(f())   // 打印什么？
```

没错，除了崩溃你将别无所获。Swift 要做的就是将这种无意义的操作彻底从语言中移除出去。除非你使用了名字里含有 “unsafe” 字样的方法或者类型，否则你应该能够免于像这段程序这样的意外崩溃。

事实上，因为闭包实际将捕获的通过 **inout** 传递给 inc 的值是通过值来传递的，所以上面的代码是完全安全的。在 inc 返回之前，内部的 i 的值已经被写回 x 了。

& 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 inc 函数的例子：

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

现在，假设你在和上面相似的作用域情况下，传入一个数组。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：

```
let fun: () -> Int
do {
    var array = [0]
    fun = incref(&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。我们以代表文件系统中一个文件的结构体作为开始，它有一个 computeSize 方法，该方法将会去询问 file manager 该结构体本身所代表的文件的大小。

```
struct File {
    let path: String

    func computeSize() -> Int? {
        let fm = NSFileManager.defaultManager()
        guard let dict = try? fm.attributesOfItemAtPath(self.path),
            let size = dict["NSFileSize"] as? Int
            else { return nil }
        return size
    }
}
```

如果我们觉得 computeSize 是一个比较昂贵的操作的话 (在后面的版本中，我们将会对文件夹中的所有内容用递归的方式计算大小)，我们可以考虑将计算得到的大小缓存在一个私有属性里。需要注意的是，我们处理的是 **struct**。如果我们选择缓存的做法，那么我们将需要把这个方法标记为 **mutating**，否则我们将无法写出这个属性：

```
    private var cachedSize: Int?

    mutating func cachedComputeSize() -> Int? {
        guard cachedSize == nil else { return cachedSize! }
        let fm = NSFileManager.defaultManager()
        guard let dict = try? fm.attributesOfItemAtPath(self.path),
```

```
            let size = dict["NSFileSize"] as? Int
            else { return nil }
        cachedSize = size
        return size
    }
```

因为这个方法现在是 **mutating** 的了，它的调用者现在也需要使用 **var** 来修饰这个存储了文件的变量，否则它将无法调用这个方法：

```
var file = File("/Users/chris/Desktop")
print(file.cachedComputeSize())
```

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** (它和上面的例子工作的方式完全一致)。接下来，我们想要访问 File 上的这个属性的时候，也只能将它定义为 **var**，这也和之前的例子一样：

```
lazy var size: Int? = {
    let fm = NSFileManager.defaultManager()
    guard let dict = try? fm.attributesOfItemAtPath(self.path),
        let size = dict["NSFileSize"] as? Int
        else { return nil }
    return size
}()
```

我们定义这个延迟属性的方式比较特别，我们使用了一个闭包表达式来进行定义，调用这个表达式返回的是我们想要存储的值，在我们的例子中，是一个可选整数值。当这个属性第一次被访问时，闭包将会被执行 (注意最后的圆括号)，它的返回值被存储到属性中。对于需要超过一行来初始化的延迟属性，这是常见的初始化方式。

如果我们不想缓存文件的尺寸，但是还是想要通过属性而不是方法来获取它的话，我们可以将它转换为一个计算属性。不过要注意，这样一来的话，每次我们访问这个属性时，它都将被重新计算：

```
var size: Int? {
    let fm = NSFileManager.defaultManager()
    guard let dict = try? fm.attributesOfItemAtPath(self.path),
        let size = dict["NSFileSize"] as? Int
        else { return nil }
```

```
    return size
}
```

默认情况下，如果我们像上面那样声明一个变量，只有 getter 会被生成。如果我们想要同时提供 getter 和 setter，就必须分别对它们进行指定。比如，我们可以添加一个获取和设置文件内容的属性：

```
extension File {
    var data: NSData? {
        get {
            return NSData(contentsOfFile: path)
        }
        set {
            let  theData = newValue ?? NSData()
            theData.writeToFile(path, atomically: true)
        }
    }
}
```

设置文件的内容现在就和设置一个属性一样简单了：

```
var  file  = File(path: "test.txt")
file .data = someData
```

对于属性来说，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 = .blackColor()
        }
    }
}
```

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]
```

我们也可以为我们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。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 CollectionType 类型中) 寻找一个子序列。现在，我们将会扩展 CollectionType，让它支持半有界区间，也就是那些只指定了 startIndex 或者 endIndex 其中一个的范围。我们先创建两个新的 **struct** 来表示这类范围：

```
struct RangeStart<I: ForwardIndexType> { let start: I }
struct RangeEnd<I: ForwardIndexType> { 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: ForwardIndexType>(lhs: I) -> RangeStart<I> {
    return RangeStart(start: lhs)
}

prefix operator ..< { }
prefix func ..<<I: ForwardIndexType>(rhs: I) -> RangeEnd<I> {
    return RangeEnd(end: rhs)
}
```

我们可以将 CollectionType 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 CollectionType {
    subscript(r: RangeStart<Self.Index>) -> SubSequence {
        return self[r.start..<self.endIndex]
    }
    subscript(r: RangeEnd<Self.Index>) -> SubSequence {
        return self[self.startIndex..<r.end]
    }
}
```

有了这些，我们就可以通过这样的下标来读取了：

```
fibs [2..<]
```

```
[1, 2, 3, 5]
```

我们可以用它来为集合实现一个按模式进行查找的函数。比如遍历字符串中的字符并寻找首个匹配某个模式的索引：

```
extension CollectionType where Generator.Element: Equatable,
    SubSequence.Generator.Element == Generator.Element {
    func search
        <S: SequenceType where S.Generator.Element == Generator.Element>
        (pat: S) -> Index?
    {
        return self.indices.indexOf {
            self[$0..<].startsWith(pat)
        }
    }
}
```

我们可以在字符串中搜索第一个 ", "，并且基于得到的索引，返回直到这个逗号的索引为止的字符串了：

```
let greeting = "Hello, world"
if let idx = greeting.characters.search(", ".characters) {
    // Print everything that comes before ", "
    print(String(greeting.characters[..<idx]))
```

```
}
```

Hello

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，我们将返回则个默认值。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 SequenceType where Generator.Element: Hashable {
    func frequencies() -> [Generator.Element: Int] {
        var result: [Generator.Element: Int] = [:]
        for x in self {
            result[x, or: 0] += 1
        }
        return result
    }
}
```

自动闭包和内存

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：

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

在 Swift 中，除了将短路操作直接构建到 and 函数中去，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, @autoclosure _ r: () -> Bool) -> Bool {
```

```
    guard l else { return false }
    return r()
}
```

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

这让我们可以使用短路操作的行为定义我们自己的函数和运算符。比方说，像是 ?? 和 !? (我们在可选值一章中定义过这个运算符) 这样的运算符现在可以直接编码实现。在 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。autoclosure 在写 log 函数的时候也会很有用，在 log 功能被关闭的时候我们并不需要对输出的字符串进行求值。

noescape 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，为了不在闭包中持有对 controller 的强引用，我们将 **self** 标记为了 **weak**：

```
reader.onTagOpen = { [unowned self, weak myReader = self.reader] tag in
    self.tags.append(tag)
    if tag == "stop" {
        myReader?.stop()
    }
}
```

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。如果我们看一看传递给 map 的闭包的类型，我们会看到它是被 **@noescape** 所标注的：

```
@noescape transform: (Self.Generator.Element) throws -> T
```

这对编译器和方法的调用者来说都是一个信号，说明这个闭包不会逃离 map 的作用范围。换句话说，一旦 map 操作完成，闭包就不会再被引用。对于 transform 闭包，既没有非同步的调用，也没有任何全局变量或者属性会持有这个闭包。这些都将由编译器来进行保证。

对编译器来说，这意味着可以对代码进行一些小的优化。对调用者来说，这意味着我们不再需要关心内存管理，可以像写普通的代码那样，不需要 **weak** 或者 **unowned** 修饰，也不需要在闭包表达式内访问调用者的属性或者方法时添加 **self**.。

当你在写一个接受闭包的函数时，你应该确保在尽可能的情况下添加 **@noescape** 标注。这将会使这个函数的调用者使用起来简单很多。他们不再需要写 **self**.，也不再需要考虑使用 **weak** 或者 **unowned** 标注。

默认情况下，所有被标注为 **@autoclosure** 的参数都会自动是 **@noescape** 的。对一个 **@autoclosure** 来说，如果出现你需要它能够逃出作用域的罕见场景的时候，你可以将这个参数标记为 **@autoclosure**(escaping)。这在操作异步代码的时候会很有用。

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的函数来替代一些常见的面向对象的模式。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)。最后我们还介绍了 **@autoclosure** 和 **@noescape** 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，那你就不怎么走运了。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

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。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 == double.characters.count
```

```
true
```

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：

```
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)
```

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 ó 和一个点；通过组合 ọ 和一个尖音符；或者是通过组合 o 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！所以下列语句不会造成断言失败：

```
let chars: [Character] = [
    "\u{1ECD}\u{300}",
    "\u{F2}\u{323}",
    "\u{6F}\u{323}\u{300}",
    "\u{6F}\u{300}\u{323}",
]

assert(chars.dropFirst().all { $0 == chars.first },
    "All the elements in chars should be equal")
```

(all 方法对序列中的所有元素进行条件检查，判断是否为真。此方法的定义见集合。)

实际上，某些变音符号可以被无限地添加：

```
let zalgo = "s̼̐͗o̟̽̋o̟͔̓ñ̻͈"
```

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}👦" == "👩‍👩‍👦‍👦"
```

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。

这意味着 String 符合遵循 CollectionType 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 CollectionType 或 SequenceType 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: CollectionType {
    // 什么都不用做，需要的实现已经有了
}
```

实际上，你可以更进一步：

```
extension String: RangeReplaceableCollectionType {}

// 现在你能够以集合的方式来使用字符串了
var greeting = "Hello, world!"
if let comma = greeting.indexOf(",") {
    print(greeting[greeting.startIndex..<comma])
    greeting.replaceRange(greeting.indices,
        with: "How about some original example strings?")
}
```

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 String.Index 只遵循 BidirectionalIndexType 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 advancedBy 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.startIndex.advancedBy(5)
s[idx]    // 字符 "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：

```
let safeIdx = s.startIndex.advancedBy(400, limit: s.endIndex)
assert(safeIdx == s.endIndex)
```

如果你已经读过泛型这章，你也许会好奇为什么越界的 advancedBy(_:limit) 返回的是 endIndex 的值而不是一个可选值，这意味着你无法区分这个方法是正好**触及**尾端还是越过了尾端。不光是你，我们也为此感到好奇。

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 {
        let strIdx = self.startIndex.advancedBy(idx, limit: endIndex)
        guard strIdx != 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？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数种有用的技术。许多操作 Array 的函数一样可以操作 String.characters。

从简单的开始，遍历字符串中的字符不用整数索引也很容易。如果你想要依次为每个字符编号，你只需要用一个 **for ... in** 循环：

```
for (i, c) in "hello".characters.enumerate() {
    print("\(i): \(c)")
}
```

```
0: h
1: e
2: l
3: l
4: o
```

或许你要找到某个特定的字符。这种情况你可以用 indexOf：

```
var greeting = "Hello!"
if let idx = greeting.characters.indexOf("!") {
```

```
    greeting.insertContentsOf(", world".characters, at: idx)
}
```

和 Array 一样，String 支持 RangeReplaceableCollectionType 协议中的所有方法。但是再一次地，它并不遵循该协议。insertContentsOf 函数将同种元素类型 (比如说，字符串的 Character 类型) 的另一集合插入到一个给定的索引位置。注意这个被插入的集合并不需要是另一个 String；你也可以很容易地将一个字符数组插入到字符串中。

和集合类型的特性相比，字符串所**没有**提供的 MutableCollectionType 协议中的方法。这个协议为集合在单个元素的下标的 **get** 之外增加了 **set** 的特性。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 SubSlice。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 SubSlice 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()
```

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 CollectionType {
    public func split(maxSplit: Int  = default,
        allowEmptySlices: Bool = default,
        @noescape isSeparator: (Self.Generator.Element) -> Bool)
        ->  [Self.SubSequence]
}
```

其最简单的使用方式如下：

```
let commaSeparatedArray = "a,b,c".characters.split { $0 == "," }
```

这个函数和 NSString 的 componentsSeparatedByCharactersInSet 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 wrap(after: Int = 70) -> String {
        var i = 0
        let lines = self.characters.split(allowEmptySlices: true) { character in
            switch character {
            case "\n", " " where i >= after:
                i = 0
                return true
            default:
                i += 1
                return false
            }
        }.map(String.init)
        return lines.joinWithSeparator("\n")
    }
}
```

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 CollectionType where Generator.Element : Equatable {
    public func split(separator: Self.Generator.Element,
        maxSplit: Int = default,
        allowEmptySlices: Bool = default)
        -> [Self.SubSequence]
}
```

如此，你就可以写出 "1,2,3".characters.split(",").map(String.init)。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 CollectionType where Generator.Element: Equatable {
    func split
```

```
        <S: SequenceType where Generator.Element == S.Generator.Element>
        (separators: S) -> [SubSequence]
    {
        return split { separators.contains($0) }
    }
}
```

现在，你就可以写出下列语句了：

```
"Hello, world!".characters.split(",! ".characters).map(String.init)
```

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 {
    private 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.characters.dropFirst(),
                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(regexp.characters,
                text.characters.suffixFrom(idx))
            {
                return true
            }
            guard idx != text.endIndex else { break }
            idx = idx.successor()
        }

        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
    private 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 where regexp.dropFirst().first == "*" {
        return matchStar(c, regexp.dropFirst(2), 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, rc = regexp.first where rc == "." || tc == rc {
        return matchHere(regexp.dropFirst(), 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
private static func matchStar
    (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, text.suffixFrom(idx)) {
            return true
        }
        if idx == text.endIndex || (text[idx] != c && c != ".") {
            return false
        }
        idx = idx.successor()
    }
```

```
    }
}
```

匹配器用起来很简单：

```
Regex("^h..lo*!$").match("hellooooo!")
```

```
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**。

另外的地方，因为表达式只在 regexp.first 不为空时才会执行，这样就防止了在空字符串上调用 dropFirst 而导致的崩溃：

```
if let c = regexp.first where regexp.dropFirst().first == "*" { }

if regexp.first == "$" && regexp.dropFirst().isEmpty
```

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, 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？尽管你既可以通过半开操作符 ..< (就是 “直到**但不**包括”)，也可以通过闭区间操作符 ... (就是 “直到**且**包括”) 来创建用来索引的区间，但其实只有一种 Range 类型，

就是半开区间。a ... b 中的 ... 实际上返回了 Range(start: a, end: b.successor())，也就是说，它把标记的结束位置从**指定的**位置变成了指定位置**之后**的那个位置。

这对数组来说没什么关系，因为数组是用整数来索引的。没人会介意你是否将标记数组尾端的索引整数值加一，反正我们也不会实际去访问它。但字符串的索引要复杂得多。它们实际上记录着其所指向的底层存储的位置。我们已经看到，它们必须如此，因为只有通过查看字符串的内容你才能知道要向前移动多少个字节。

所以当你执行 "hello".endIndex.successor() 时会发生运行时错误: “fatal error: can not increment endIndex.”。同样地，当 ... 尝试将尾部索引变成封闭区间时也会发生同样的错误。因而我们只能被迫使用 C 风格的循环方式并使用 **break** 来终止循环。

StringLiteralConvertible

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。就如在集合一章中涉及的数组字面量一样，"" 是字符串字面量。

字符串字面量隶属于 StringLiteralConvertible、ExtendedGraphemeClusterLiteralConvertible 和 UnicodeScalarLiteralConvertible 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 标量还是从一个字位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: StringLiteralConvertible {
    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：

```
func findMatches(strings: [String], regex: Regex) -> [String] {
    return strings.filter { regex.match($0) }
}

findMatches(["foo","bar","baz"], regex: "^b..")
```

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 2.0 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。)

在 Swift 2.0 中，sizeof(String) 语句在 64 位平台上会返回 24，在 32 位平台上会返回 12。字符串的内部存储是由下列内容组成：

```
struct String {
    var _core: _StringCore
}
```

```
struct _StringCore {
    var _baseAddress: COpaquePointer
    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, _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: COpaquePointer
    var _countAndFlags: UInt
    var _owner: AnyObject?
}

let clone = unsafeBitCast(bits, StringCoreClone.self)
```

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：

```
// 额外的 UnsafePointer 初始化，可以将被指向的类型由 Void 转换为 UInt8：
puts(UnsafePointer(clone._baseAddress))

hello
10
```

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
```

假设 count 属性不可用，我们也可以在之前克隆的结构体中通过屏蔽旗标的比特位来提取这个值：

```
extension StringCoreClone {
    var count: Int  {
        let  mask = 0b11 << UInt(sizeof(UInt)*8 - 2)
        return Int(_countAndFlags & ~mask)
    }
}
```

第二处不同是 _baseAddress 现在指向的字符是 16 位的了。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。

为了证明即便字符串含有 4 字节长的 emoji 字符时其缓冲区依然是 UTF-16 形式的，我们使用 UTF16.decode 方法来解码该缓冲区：

```
let  buf = UnsafeBufferPointer(
    start:  UnsafePointer<UInt16>(emojiBits._baseAddress),
    count: emojiBits.count)
```

```
var gen = buf.generate()
var utf16 = UTF16()
while case let .Result(scalar) = utf16.decode(&gen) {
    print(scalar, terminator: "")
}
```

这会输出

"Hello, 🌎" .

_StringCore 的最后一个属性 _owner 是一个空指针。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.appendContentsOf(" world")
let greetingBits = unsafeBitCast(greeting, StringCoreClone.self)
```

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 isUniquelyReferenced 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：

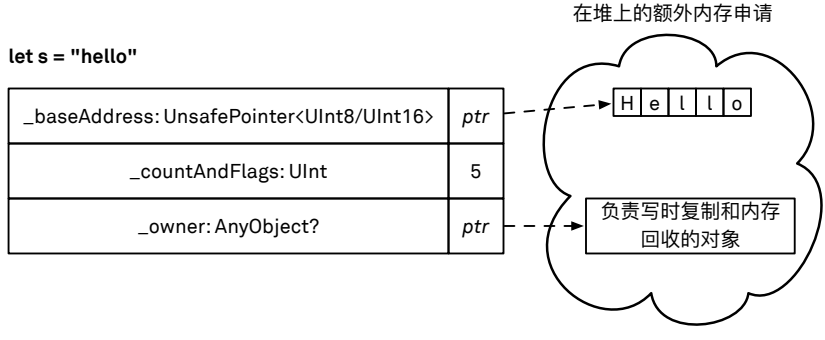

图 7.1: String 值的内存

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：

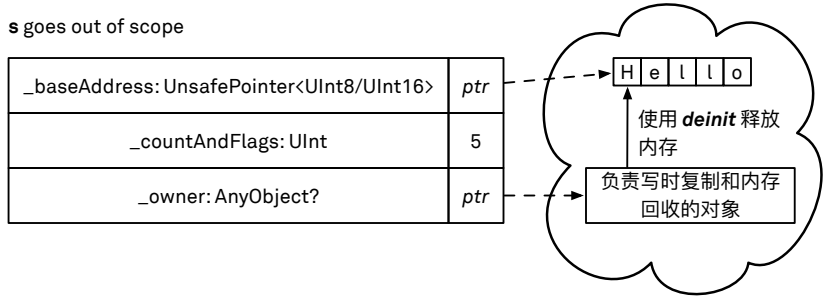

图 7.2: 当 String 离开作用域

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：

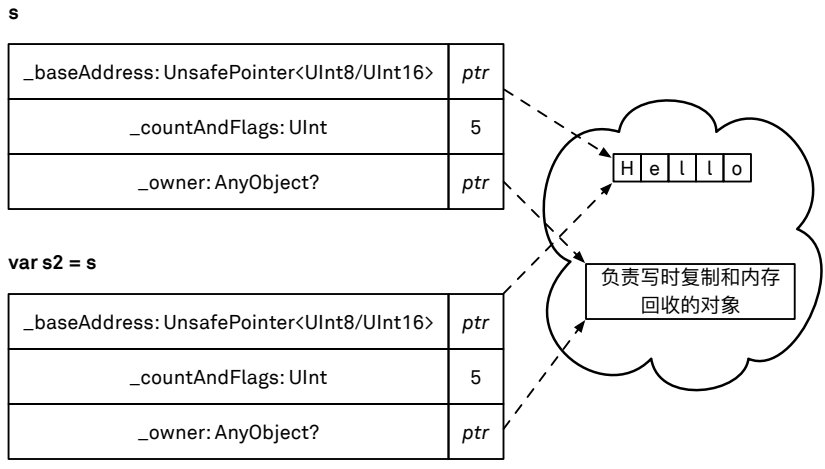

图 7.3: 两个 String 共享同样的内存

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：

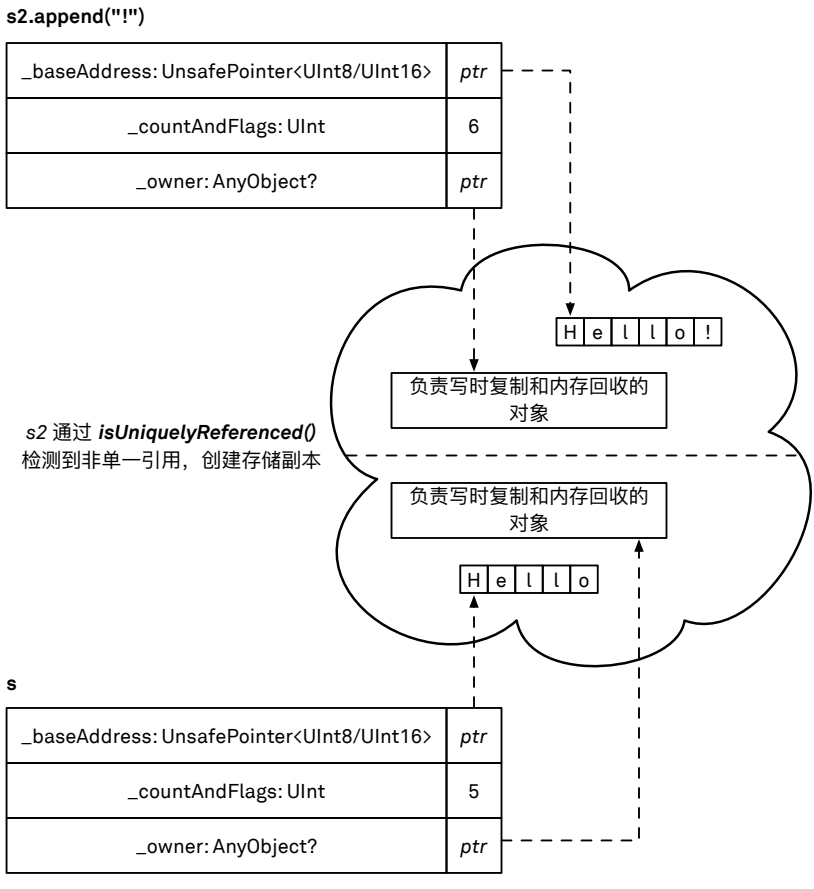

图 7.4: 改变字符串

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：

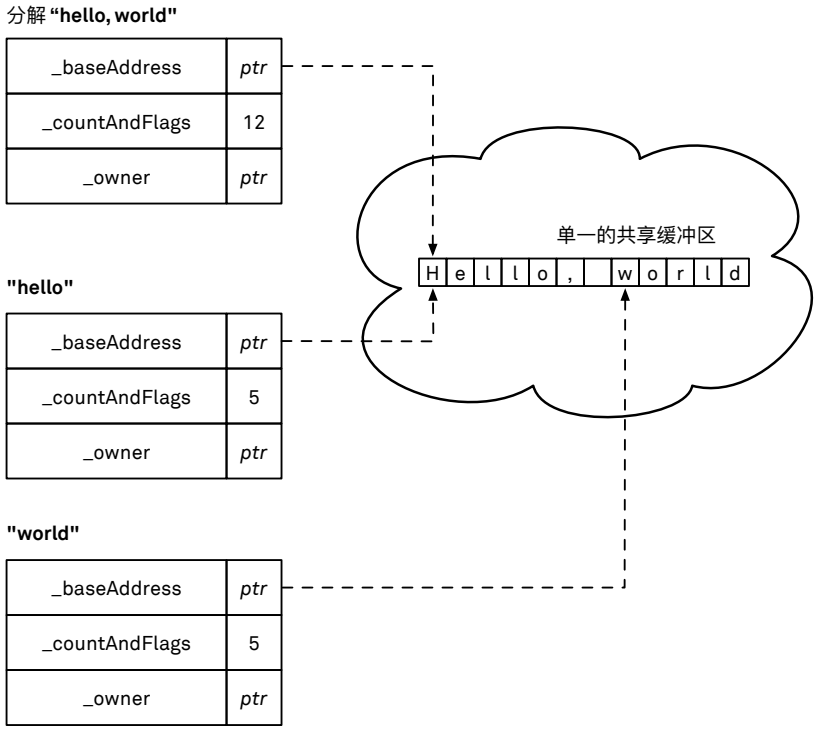

图 7.5: 字符串切片

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 (_, _, owner) = unsafeBitCast(s, (UInt, UInt, NSString).self)
print(owner) // ref 将是一个引用
owner === ns // 它就是原来的那个 NSString
```

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？如果你执行 sizeof(Character) 语句，你会看到其占用 9 个字节。奇数字节往往是枚举值的一个迹象：一个字节用来标明枚举项，其余用来存储关联值。

所以字符内部有点像是这样：

```
enum Character {
    case Large(Buffer)
    case Small(Int64)
}
```

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。

当然，字符实际上不是枚举值；它们是结构体，而枚举是其内部的私有实现。实际上你可以通过 Swift 2.0 的 REPL 来查看这些信息。只需要在命令行中输入如下命令：

```
$ echo ":type lookup Character" | swift |  less
```

之后就能看到 Character 的内部实现滚过你的屏幕。你可以使用该技术来探索标准库中的任意类型，以此来深入了解其内部实现 (再次提醒：这样做只是为了加深理解，绝不要就这些实现细节做出假设；应该只依赖公开的接口)。

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 NSCharacterSet 的结合使用。NSString 和其相关类都操作 UTF-16 的编码点。如果你要用 NSCharacterSet 来拆分一个字符串，可以通过 utf16 方式来操作：

```
extension String {
    func words (splitBy: NSCharacterSet = .alphanumericCharacterSet()) -> [String] {
        return self.utf16.split  {
            !splitBy.characterIsMember($0)
        }.flatMap(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.UTF16View 切片的数组。这些切片可以通过 map 函数调用 Stirng 接收 UTF16View 类型参数的构造方法来转回成字符串。这个构造方法是一个可失败的构造方法 (以防索引到字符边界内的位置上)，因此我们使用 flatMap (在可选值 这章中描述) 来过滤掉结果为 **nil** 的元素。

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 UTF16View 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，必需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。

最后，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)。

在之前的 Swift 版本中，Foundation 框架扩展了 String.UTF16View.Index 类型以遵循 RandomAccessIndexType 协议。不过在 Swift 2.0 中不再如此。然而因为其依然实现了遵循该协议所需的所有方法，你可以自己加上声明来遵循这个协议：

```
extension String.UTF16Index: RandomAccessIndexType {
```

```
    /* 不需要任何实现 */
}
```

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("world".utf16)?
    .samePositionIn(helloWorld)
{
    print(helloWorld[idx..<helloWorld.endIndex])
}
```

```
world!
```

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，下列断言为真：

```
let text = "Look up your Pok\u{0065}\u{0301}mon in a Pokédex."
assert(text.utf16.search("Pokémon".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 = NSCharacterSet.alphanumericCharacterSet().invertedSet
text.utf16.split(isSeparator: nonAlphas.characterIsMember)
```

```
[StringUTF16("Look"), StringUTF16("up"), StringUTF16("your"),
StringUTF16("Pokémon"), StringUTF16("in"), StringUTF16("a"),
StringUTF16("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可以被接受。

```
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一分钟：

```
extension Regex: CustomStringConvertible {
    public var description: String {
        return "/\(regexp)/"
    }
}
```

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.init 的参数，又或者用在某个字符串插值中使用)，都会得到 /expression/。

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(reflecting:) 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。我们可以把集合一章中的 Queue 例子扩展一下：

```
extension Queue: CustomStringConvertible, CustomDebugStringConvertible {
```

```
    var description: String {
        // 使用 String.init 对元素进行打印，它将使用 CustomStringConvertible
        let elements = map { String($0) }.joinWithSeparator(", ")
        return "[\(elements)]"
    }

    var debugDescription: String {
        // 使用 String.init(reflecting:) 对元素进行打印，它将使用 CustomDebugStringConvertible
        let elements = map { String(reflecting: $0) }.joinWithSeparator(", ")
        return "[\(elements)]"
    }
}
```

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
}
```

然而如果你这么做了，你很快就会懊恼地发现 String 居然不遵循 CustomStringConvertible！这到底是为什么？字符串显然是最适合 print 的东西了。这是 Swift 开发团队在告诉你不要这么使用 CustomStringConvertible。无论如何都应该使用 String.**init** 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。

Streamable 和 OutputStreamType

如果你的类型很复杂，那么比起 CustomStringConvertible，实现 Streamable 协议会更容易一些。后者要求实现一个泛型方法 writeTo。该方法接受任意遵循 OutputStreamType 协议的类型作为参数，并将 **self** 写入其中。最容易的实现方式就是用接受一个输出流作为参数的 print 函数的重载。这个输出流参数很容易被忘记。除非你测试的时候传的 target 参数不是标准输出，不然你都注意不到：

```
extension Queue: Streamable {
    func writeTo<Target: OutputStreamType>(inout target: Target) {
        print("[", terminator: "", toStream: &target)
```

```
            print(self.map { String($0) }.joinWithSeparator(","),
                terminator: "", toStream: &target)
            print("]", terminator: "", toStream: &target)
        }
}
```

实现了 Streamable 协议后能输出到哪里呢？字符串就是一种输出流：

```
var s = ""
let q: Queue = [1,2,3]
q.writeTo(&s)

print(s)
```

```
[1,2,3]
```

String 类型是标准库中唯一一个输出流 (即遵循 OutputStreamType 协议)。不过我们可以自己创建输出流，比如这个将输入缓冲到数组的例子：

```
struct ArrayStream: OutputStreamType {
    var buf: [String] = []
    mutating func write(string: String) {
        buf.append(string)
    }
}
```

或者扩展 NSMutableData 以使其支持接受一个流并以 UTF-8 编码方式将流的内容写入自身：

```
extension NSMutableData: OutputStreamType {
    public func write(string: String) {
        string.nulTerminatedUTF8.dropLast().withUnsafeBufferPointer {
            self.appendBytes($0.baseAddress, length: $0.count)
        }
    }
}
```

尽管 String 是唯一遵循 OutputStreamType 的类型，其它像是 print 这样的函数也都使用 Streamable 类型。看看下面这个很傻的例子：

```
struct SlowStreamer: Streamable, ArrayLiteralConvertible {
    let contents: [String]
    init(arrayLiteral elements: String...) {
        contents = elements
    }
    func writeTo<Target: OutputStreamType>(inout target: Target) {
        for x in contents {
            print(x, toStream: &target)
            sleep(1)
        }
    }
}

let slow: SlowStreamer = [
    "You'll see that this gets",
    "written slowly line by line",
    "to the standard output",
]

print(slow)
```

当有新行被 print 到 target 时，输出就出现了，并不会等待调用完成。在内部，print 函数大概使用了某种遵循 OutputStreamType 协议的标准输出所进行的封装。你也可以为标准错误写个类似的东西：

```
struct StdErr: OutputStreamType {
    mutating func write(string: String) {
        // 字符串可以直接传递给接受 `const char*` 的 C 函数，
        // 查看协同工作一章来了解更多内容
        fputs(string, stderr)
    }
}

var standarderror = StdErr()

print("oops!", &standarderror)
```

流也可以持有状态，也可以对输出进行转换。另外，你还可以把几个流串在一起：

```
struct ReplacingStream<T: OutputStreamType>: OutputStreamType {
    var outputStream: T
    let toReplace: DictionaryLiteral<String, String>
    init(replacing: DictionaryLiteral<String, String>, output: T) {
        outputStream = output
        toReplace = replacing
    }
    mutating func write(string: String) {
        let toWrite = toReplace.reduce(string) {
            $0.stringByReplacingOccurrencesOfString($1.0, withString: $1.1)
        }
        print(toWrite, &outputStream, appendNewline: false)
    }
}

var replacer = ReplacingStream(
    replacing: ["in the cloud": "on someone else's computer"],
    output: standarderror)

let source = "More and more people are finding it convenient " +
    "to store their data in the cloud."

print(source, &replacer)
```

上面的代码使用了 DictionaryLiteral 而不是普通的字典。如果你想使用 [key: value] 字面量语法，又不想要 Dictionary 会导致的消除重复键以及重新排序键这两个副作用，那用 DictionaryLiteral 就很方便。在这种情况下，DictionaryLiteral 是键值对数组 (即 [(key, value)]) 的另一种不错的替代，它还允许调用者使用更便捷的 [:] 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 ViewType: CollectionType

    static var caret: ViewType.Generator.Element { get }
    static var asterisk: ViewType.Generator.Element { get }
    static var period: ViewType.Generator.Element { get }
    static var dollar: ViewType.Generator.Element { get }

    static func viewFrom(s: String) -> ViewType
}
```

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: "^") }
    static var asterisk: UInt8 { return UInt8(ascii: "*") }
    static var period: UInt8 { return UInt8(ascii: ".") }
    static var dollar: UInt8 { return UInt8(ascii: "$") }

    static func viewFrom(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 viewFrom(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 sizeof(CharacterViewSelector) 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.ViewType.Generator.Element: Equatable,
        V.ViewType.SubSequence == V.ViewType>
{
    let regexp: String
    /// 从正则表达式字符串中构建
    init(_ regexp: String) {
        self.regexp = regexp
    }

    /// 当表达式匹配字符串时返回 true
    func match(text: String) -> Bool {
        let text = V.viewFrom(text)
        let regexp = V.viewFrom(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.dropFirst(), 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(regexp, text.suffixFrom(idx)) {
                return true
            }
            guard idx != text.endIndex else { break }
            idx = idx.successor()
        }
        return false
```

```
    }


    /// 从文本开头匹配正则表达式字符串
    static func matchHere(regexp: V.ViewType, _ text: V.ViewType) -> Bool {
        // 更多匹配代码
        return true
    }
    // 其他代码
}
```

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 where V.ViewType.Generator.Element: Equatable,
        V.ViewType.SubSequence == V.ViewType>
    (_: V.Type) {
    let r = Regex<V>("h..a*")
    var count = 0

    let startTime = CFAbsoluteTimeGetCurrent()
    while let line = readLine() {
        if r.match(line) { count = count &+ 1 }
    }
    let totalTime = CFAbsoluteTimeGetCurrent() - startTime
    print("\(V.self): \(totalTime)")
}

func ~=<T: Equatable>(lhs: T, rhs: T?) -> Bool {
    return lhs == rhs
}

switch Process.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")
```

}

结果显示不同的视图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	0.5 秒
UnicodeScalar	0.8 秒
UTF8	1.1 秒
Characters	9.1 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 视图将会给你带来相当不错的性能提升。

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。

如果我们仔细看看已有的 Swift 类型的话，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。可选值的存在是为了安全。比如说，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**。

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 CollectionType 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(ErrorType)
    case Success(A)
}
```

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 contentsOfFile1(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: ErrorType {
    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 contentsOfFile(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 = contentsOfFile("input.txt")
switch result {
    case let .Success(contents):
        print(contents)
    case let .Failure(error):
        if let fileError = error as? FileError
            where fileError == .FileDoesNotExist
        {
            print("Empty file")
        } else {
            // 处理错误
        }
}
```

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 contentsOfFile 的情况，包含 **throws** 的时候函数是这样的：

```
func contentsOfFile(filename: String) throws -> String
```

现在，我们需要将所有对 contentsOfFile 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。

调用一个可抛出的函数时，我们也需要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：

```
do {
    let result = try contentsOfFile("input.txt")
    print(result)
} catch FileError.FileDoesNotExist {
    print("File does not exist")
} 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 ErrorType 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: ErrorType {
```

```
    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 contents = try parseFile(contents)
    print(contents)
} catch ParseError.WrongEncoding {
    print("This file uses the 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。

> 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。
>
> 因为 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: `FileError` if the file does not exist or
///           the process doesn't have read permissions.
func contentsOfFile(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。

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, Error> {
    case Failure(Error)
    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 parseFile(contents: 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 _ = parseFile(contents)
```

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。

上面的例子里，可能忽略掉错误并没有什么大不了的。但是有时候提醒你不要忘记处理错误确实会很有意义，特别是在你调用的是那些不会返回普通值的函数。比如，下面这个函数：

```
func setupServerConnection() throws {
    // 实现
}
```

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。

错误和 Objective-C 接口

在 Objective-C 里，并没有像 **throws** 和 **try** 这样的机制。Cocoa 的通用模式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 contentsOfFile(fileName: String) throws -> String
```

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(contents: 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) 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 isPrime(number) else { return false }
    }
    return true
}
```

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 CollectionType 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 SequenceType {
    func all(@noescape check: Generator.Element -> Bool) -> Bool {
        for el in self {
            guard check(el) 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 checkPrimes(numbers: [Int]) -> Bool {
    return numbers.all(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。

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 SequenceType {
    func all(@noescape check: Generator.Element throws -> Bool) rethrows
        -> Bool
    {
        for el in self {
            guard try check(el) 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(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 contentsOfFile 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：

```
let file = open("test.txt", O_RDONLY)
defer { close(file) }
let string = try processFile(file)
```

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。

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。比如，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 contents = try? parseFile(filename) {
```

```
    print(contents)
}
```

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：

```
func optional<A>(value: A?, orError e: ErrorType) throws -> A {
    guard let x = value else { throw e }
    return x
}
```

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：

```
let int = try optional(Int("42"), orError: ReadIntError.CouldNotRead)
print(int)
```

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，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。而且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。

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(checkFile)
        let contents = try contentsOfFile("Pidfile")
        return try optional(Int(contents),
```

```
            orError: ReadIntError.CouldNotRead)
    } catch {
        return 42 // 默认值
    }
}
```

如果我们选择使用 Result，实现一个返回 Result 的 all 函数的变形并不会很难。链式调用多个返回 Result 的函数实际上是一系列 flatMap 操作。对于一个 Result 来说，它的 flatMap 函数的变形可能会与它的可选值重载的版本以相同的方式工作：如果一个输入值是成功的情况，则将它解包并且用传入的变形函数进行操作。如果输入值是失败的情况，那么直接不加改变地将其返回，而不对它使用变形函数。这实现起来也很优雅：

```
func checkFilesAndFetchProcessID(filenames: [String]) -> Result<Int> {
    return filenames
        .all(checkFile)
        .flatMap { _ in contentsOfFile("Pidfile") }
        .flatMap { contents in
            Int(contents).map(Result.Success)
                ?? .Failure(ReadIntError.CouldNotRead)
        }
}
```

高阶函数和错误

在写这本书的时候，Swift 的错误和回调函数并不能很好地协同工作。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 compute(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：

```
compute { 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？我们首先可能会尝试下面这种类型写法。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：

```
func compute(callback: Int throws -> ())
```

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 来重写上面的 (错误的) 尝试时，这个区别带来的问题就会比较明显了：

```
func compute(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：

```
func compute(callback: Result<Int> -> ())
```

不过坏消息是，现在没有方法可以很清晰地用 **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(f: () 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 { (theResult: () throws -> Int) in
    do {
        let result = try theResult()
        print(result)
    } catch {
        print("An error happened: \(error)")
```

```
    }
}
```

所以，对于异步的错误处理来说，Result 可能会是更好的选择。然而，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，这可能让你的 API 使用起来更加困难。当你有很多异步函数时，使用 Result 带来的好处可能会很明显，但是如果你只有一个回调的话，那么上面的那种嵌套函数的方式可能会更好一些。

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其实褒贬不一。无类型错误可以让我们的类型签名保持简单，比如要是我们需要指定错误类型时，多余一种错误会发生的情况下，类型签名很快就会变得特别复杂：

```
func checkFilesAndFetchProcessID(filenames: [String])
    throws ReadFileError, CheckFileError, MiscellaneousError -> Int
```

但是，这么设计的不足也显而易见：没有机制可以指定只有一种错误会发生，这导致了额外的模板代码。另外，**throws** 只能与函数协同工作，而不能作用于其他类型上，这为我们使用它增添了不必要的复杂度 (就像我们在异步回调的例子中看到的那样)。和其他很多方面一样，Swift 的错误处理设计对于 80% 的情况来说是务实和优秀的，但是它的不足使你在将它用到更高阶的情景时，事情会很快变糟：仅仅只是为了使用内建的错误处理，而将结果封装到一个额外的函数中，这增加了不必要的复杂度。另外，我们要提醒你的是，我们所面临的不是一个罕见情况，可以失败的异步 API 是非常常见的。

如果你想要指定错误的类型，你可以使用 Result 加上一个代表错误类型的泛型参数的组合。但是，这会向你的代码库引入另一组结构。你需要基于你所构建的内容，来决定是否值得将它添加到你的项目中。

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** 语句可以让我们在作用域结束时对代码进行清理，而不必关心作用域的退出方式，不论是正常退出，还是被 **throws**，又或者是由于条件不满足而提早 **return**，**defer** 语句所定义的代码块都将被执行。

泛型

9

和大多数现在语言一样，Swift 拥有不少能被归类于**泛型编程**下的特性。比如，我们可以写出泛型的数据类型。我们已经看到的像是 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。我们也可以创造出同样名字但是不同类型的函数。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。GeneratorType 协议就是一个这样的例子：它所生成的 Element 就是一个泛型。在本章中，我们将会研究如何书写泛型代码。

泛型编程的目的是通过最小的假设来表达算法。比如，考虑集合一章中的 findElement 函数。我们原本可以将它写成 Array 的一个扩展，但是这是一个较大的假设。我们最后将它作为 SequenceType 的扩展来使用，这相对是一个较小的假设。SequenceType 的假设只涉及到可以迭代，它没有其他更多的要求。

在本章中，我们将用泛型编程的思想来创建一个算法，它可以对两个序列做减法。我们会提供这个算法的多种实现，每个都基于不同的假设集合。最后，我们会教你如何使用泛型数据来将通用代码进行提取。我们会重构一段网络代码，它很复杂，而且处处都是异步。在重构后，我们会得到只有一个地方进行异步操作，其他都是基于泛型的简单片段的实现。如果你对泛型背后的理论细节感兴趣的话，可以去阅读这篇“泛型编程语言支持的深入比较学习”的论文。

库代码

假设你现在有一组进行过更新的数据，你想要高亮那些新加的和变更过的行，这就需要验证一个数组中的元素并不是全被包含在另一个数组中。Set 类型为你提供了一种非常简单的方法来达到这个目的：

```
let old = [1, 2, 3]
let new = [1, 2, 4, 5]

Set(new).subtract(old)

[5, 4]
```

可能这就是你所需要的了。但是，这种方法有一些不足。首先，Set 会把多个同样的元素合并为一个。所以当 old 允许重复的元素的时候，就不能使用了。另外，由于 Set 是无序集合，结果中的元素的顺序会合 new 中的元素不一致。如果你想用上面的代码来更新一个 table view 的话，你需要解决这两个问题。

保持条目顺序不变比较容易，只要在 new 上使用 `filter` 就可以了：

```
let oldSet = Set(old)
new.filter { !oldSet.contains($0) }
```

[4, 5]

另一个问题解决起来就困难一些。Set 中的元素类型是满足 Hashable 协议的，如果你想要将上面的代码封装到一个泛型函数中，可能你需要这样来写：

```
extension CollectionType where Generator.Element: Hashable {
    /// 返回一个新数组，该数组中的元素存在于`self`中，但是不存在于`toRemove`里
    func subtract(toRemove: [Generator.Element]) -> [Generator.Element] {
        let removeSet = Set(toRemove)
        return self.filter {
            !removeSet.contains($0)
        }
    }
}
```

这样，你可以在数组上调用这个求差操作了：

```
new.subtract(old)
```

[4, 5]

这里 subtract 被定义为了 SequenceType 的协议扩展，其中要求序列的元素满足 Hashable。它接受一个同样类型的数组作为输入的移除项，并返回一个从当前数组中移除输入数组中元素的数组。subtract 只要求元素满足 Hashable 协议，而不关心它们到底是什么类型，这是因为只有 Hashable 的类型可以被用在 Set 里。元素类型可以是 Int，可以是 String，也可以是我们自己的用户定义类型，对于它们唯一的要求是满足 Hashable。

但是要是你的类型不满足 Hashable 呢？并不是所有类型都满足这个协议的，特别是那些用户定义类型往往不会去刻意实现 Hashable。我们可以把门槛降低一点，让这个方法对满足 Equatable 协议的元素也适用呢？这样一来，我们就不能使用 Set 了，但是我们可以用标准的 contains 函数：

```
extension SequenceType where Generator.Element: Equatable {
    func subtract(toRemove: [Generator.Element]) -> [Generator.Element] {
        return self.filter {
            !toRemove.contains($0)
```

```
        }
    }
}
```

现在，你可以对任何满足了 Equatable 的类型调用 subtract 了。比如 Range 就实现了 Equatable，但是它并不遵守 Hashable：

```
let ranges1 = [0..<1，1..<4]
let ranges2 = [0..<1，1..<4，5..<10]

ranges2.subtract(ranges1)

[Range(5..<10)]
```

这种方式更加泛用，它适用于更广阔范围的类型。不过，它也有很大的缺点，那就是这个泛型化处理将更消耗我们的性能。这个算法的复杂度特性是 $O(n^2)$，也就是说，随着输入尺寸的增长，最坏情况的时间将会呈二次方增长。这是因为数组的 contains 时间复杂度是 $O(n)$ 的。这很容易理解，想象一下 contains 做的事情，它对序列进行循环，检查它是否满足一个给定的元素。但是我们的 subtract 算法是在一个 filter 循环里调用 contains 的，和之前类似，这个 filter **同样**是一个线性时间复杂度的函数。在一个 $O(n)$ 循环里再进行一次 $O(n)$ 循环，得到的结果就是一个 $O(n^2)$ 的函数。

对于小输入集来说问题不大，如果你最多只会在一两百个元素的数组上调用这个方法的话，可能不会察觉到性能的问题。但是如果你的数组中有成千上万的元素的话，那不好意思了，性能将大幅恶化至无法忍受。基于 Set 的版本就没有这个问题，因为基于哈希的 contains 查找可以在常数时间内完成；Set 版本中的 filter 也是 $O(n)$ 的，所以整个函数的复杂度只有 $O(n)$。

不过好消息是你**不是**只能在这些选项中择一而从。你可以使用重载的方式**同时**实现两个版本的 subtract。你可能已经对那些接受不同种类的参数的重载很熟悉了，比如对 String.**init** 的重载，让它既能接受 String 也能接受 Int 作为参数。Swift 的重载非常灵活，你不仅可以通过提供不同的输入类型进行重载，也可以通过不同输出类型重载。另外，你还可以基于不同的泛型约束进行重载，就像我们上面看到的那样。

Swift 中对于选取哪个重载函数有着一套复杂的规则，它基于函数是否是泛型以及输入的类型是什么来决定使用的重载版本。在这里列举这些规则的话会篇幅太长，总结下来其实就是 “使用最具体的版本”。比方说，一个指定了类型的非泛型版本的使用优先级要比泛型版本要高。

这里，我们有两个 subtract 版本，它们都是泛型。但是要求元素是 Hashable 的版本更加具体，因为 Hashable 是对 Equatable 的扩展，所以它带来了更多的约束。正如 subtract 例子中所表现的那样，这些约束可能会使算法更加高效，因此更具体版本的函数可能会是更好的选择。

还有另一种方法可以让 subtract 更加泛用。直到现在，这个函数都接受的是一个数组参数。但是 Array 是一个具体类型。实际上 subtract 并不需要这个具体的类型。现有的两个版本中，只有三个函数被调用：两个函数中的 filter，Hashable 版本中的 Set.init，以及 Equatable 版本中的 contains。在这三个地方，这些函数其实只需要输入类型满足 SequenceType 协议就足够了：

```
extension SequenceType {
    /// Return an `Array` containing the elements of `self`, in order,
    /// that satisfy the predicate `includeElement`.
    func filter(includeElement: (Self.Generator.Element) throws -> Bool)
        rethrows -> [Self.Generator.Element]
}

extension SequenceType where Generator.Element: Equatable {
    /// Return `true` iff `x` is in `self`.
    func contains(element: Self.Generator.Element) -> Bool
}

// And inside `Set`:
struct Set<Element: Hashable>:
    Hashable, Equatable, CollectionType, Indexable, SequenceType,
    ArrayLiteralConvertible
{
    // ...
    /// Create a `Set` from a finite sequence of items.
    init<S: SequenceType where S.Generator.Element == Element>
        (_ sequence: S)
}
```

所以，subtract 中 toRemove 所需要的仅仅是一个满足 SequenceType 的类型。除此之外，这两个序列类型甚至**不需要是同样的类型**。它们只需要满足序列中的元素的类型相同就行了。这样一来，我们可以将 Hashable 的版本重写为对于任意两种序列进行的操作：

```
extension SequenceType where Generator.Element: Hashable {
    func subtract
```

```
        // 为 toRemove 序列类型定义的占位符。
        // 序列的元素类型必须和本身相同。
         <S: SequenceType where S.Generator.Element == Generator.Element>
        (toRemove: S) -> [Generator.Element]
    {
        let removeSet = Set(toRemove)
        return self.filter { !removeSet.contains($0) }
    }
}
```

现在，两个序列的类型可以不必相同了，这开启了很多的可能性。比如，现在你可以传递一个数字的 Range 来进行移除了：

```
[2, 4, 8].subtract(0..<3)
```

```
[4, 8]
```

你可以对满足 Equatable 的版本进行同样的变更。

使用闭包对行为进行参数化

但是这依然留给我们一个问题，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。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 ==<T : Equatable>(lhs: [T], rhs: [T]) -> Bool
```

但是这并不意味着你可以将 subtract 用在元素是数组类型的序列上：

```
// 错误：cannot invoke 'subtract' with an argument list of type '([Int])
[[1, 2], [3], [4]].subtract([3])
```

这是因为 Array 并不遵守 Equatable。因为如果数组包含的类型不遵守 Equatable 的话，数组本身也没办法遵守 Equatable。所以，它只能为那些所包含的元素是 Equatable 的实现提供 == 操作符的实现，而不能让自己本身遵守这个协议。

那么，要怎么才能让 subtract 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。比如第二个版本中的 contains 实际的定义是：

```
extension SequenceType where Generator.Element : Equatable {
    /// Return `true` iff an element in `self` satisfies `predicate`.
    func contains
        (@noescape predicate: (Self.Generator.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 }
print ((0..<5). contains(isEven))
print ([1,3,99]. contains(isEven))

true
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 subtract：

```
extension SequenceType {
    func subtract<S: SequenceType>
        (toRemove: S,
            predicate: (Generator.Element, S.Generator.Element) -> Bool)
        -> [Generator.Element]
    {
        return self. filter { sourceElement in
            !toRemove.contains { removeElement in
                predicate(sourceElement, removeElement)
            }
        }
    }
}
```

现在，我们可以把一个数组序列从另一个中去除了，只需要通过闭包表达式来用 == 将两个数组进行比较就可以了：

```
[[1, 2], [3], [4]].subtract([[1, 2], [3]]) { $0 == $1 } as [[Int]]
```

```
[[4]]
```

只要你提供的闭包能够处理，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, 3]
let strings = ["1", "2"]

ints.subtract(strings) { $0 == Int($1) }
```

```
[3]
```

对集合采用泛型操作

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 `count`)
    func binarySearch
        (value: Element, isOrderedBefore: (Element, Element) -> Bool)
        -> Int?
    {
        var left = 0
        var right = count - 1

        while left <= right {
            let mid = (left + right) / 2
```

```
            let candidate = self[mid]

            if isOrderedBefore(candidate,value) {
                left = mid + 1
            } else if isOrderedBefore(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(value: Element) -> Int? {
        return self.binarySearch(value, isOrderedBefore: <)
    }
}
```

> 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 indexOf 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 对数组也是有效的，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 CollectionType **where** Index: RandomAccessIndexType 上的，这里 **where** 语句是需要存在

的，否则我们无法保证对数时间复杂度。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 <= 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 CollectionType where Index == Int {
    public func binarySearch(value: Generator.Element,
        isOrderedBefore: (Generator.Element, Generator.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 String 的各种表现方式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 ReverseRandomAccessCollection。我们在集合一章中看到过它，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。

如果你把 Int 索引的要求去掉，你将得到一些编译错误。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 CollectionType where Index: RandomAccessIndexType {
    public func binarySearch(value: Generator.Element,
        isOrderedBefore: (Generator.Element, Generator.Element) -> Bool)
        -> Index?
    {
        guard !isEmpty else { return nil }
        var left = startIndex
        var right = endIndex - 1

        while left <= right {
            let mid = left.advancedBy(left.distanceTo(right)/2)
            let candidate = self[mid]

            if isOrderedBefore(candidate, value) {
```

```
                left = mid + 1
            } else if isOrderedBefore(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 CollectionType
    where Index: RandomAccessIndexType, Generator.Element: Comparable
{
    func binarySearch(value: Generator.Element) -> Index? {
        return binarySearch(value, isOrderedBefore: <)
    }
}
```

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，(**left** + **right**) / 2 被用稍微丑陋一点的 **left**.advancedBy(**left**.distanceTo(**right**)/2) 替代了，为什么?

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 Index.Distance。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。

successor 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 successor 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 successor 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)。这就是 ForwardIndexType 所定义的约束条件：

```
public protocol ForwardIndexType : _Incrementable {
```

```
    associatedtype Distance : _SignedIntegerType = Int
}
```

_SignedIntegerType 要求类型满足 IntegerLiteralConvertible，这也是我们可以写 endIndex - 1 而不必须每次都写成 endIndex.predecessor() 的原因。也正由于这一点，我们需要一个额外的 **guard** 来确保集合不为空。当你只是在做整数操作的时候，生成一个 -1 的 **right** 值，并检查它是否小于零是没什么不妥的。但是要是处理的是索引的话，你就需要确保不会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。)

_SignedIntegerType 还满足 IntegerArithmeticType，这样你可以将距离相加，或者找到两个距离相除的余数。_SignedIntegerType 同样属于 SignedNumberType，你可以通过它来计算两个索引之间距离的绝对值。

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 advancedBy(distance:) 来将一个距离加到索引上。

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 = startIndex.advancedBy(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 CollectionType 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 ReverseRandomAccessCollection 了：

```
let a = ["a", "b", "c", "d", "e", "f", "g"]
let r = a.reverse()

assert(r.binarySearch("g", isOrderedBefore: >) == r.startIndex)
```

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]
assert(s.startIndex != 0)
assert(s.binarySearch("c") == s.startIndex)
```

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 shuffleInPlace() {
        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])
        }
    }

    func shuffle() -> [Element] {
        var clone = self
        clone.shuffleInPlace()
        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 Index.Distance 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。

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 _SignedIntegerType {
    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。

现在我们可以写泛型代码来在任意索引区间内生成随机数了：

```
extension Range {
    var arc4random: Element {
        return startIndex.advancedBy(
          Index.Distance.arc4random_uniform(count)
        )
    }
}
```

最后，我们把这个用在我们的泛型洗牌函数的实现中：

```
extension MutableCollectionType where Index: RandomAccessIndexType {
    mutating func shuffleInPlace() {
        for i in indices.dropLast() {
            let j = (i ..< endIndex).arc4random
            guard i != j else { continue }
            swap(&self[i], &self[j])
        }
    }
}

extension SequenceType {
    func shuffle() -> [Generator.Element] {
        var clone = Array(self)
        clone.shuffleInPlace()
        return clone
    }
}
```

可以看到，作为 count - 1 和范围操作符的替代，我们使用了 .indices 来获取一个完整的区间。之后我们用 dropLast 来去掉最后一项，然后用 i 对这个范围进行循环。

接下来，我们使用新定义的基于区间的 arc4random 函数来获取要进行交换的随机索引，并且保持其他索引不变。

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 MutableCollectionType。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 MutableCollectionType 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 shuffle 和 shuffleInPlace 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.shuffleInPlace 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 RangeReplaceableCollectionType，就让它返回和它所随机的内容同样类型的集合：

```
extension MutableCollectionType
    where Self: RangeReplaceableCollectionType,
        Index: RandomAccessIndexType
{
    func shuffle() -> Self {
        var clone = Self()
        clone.appendContentsOf(self)
        clone.shuffleInPlace()
        return clone
    }
}
```

这个实现依赖了 RangeReplaceableCollectionType 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 indexOf 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。然而，如果你尝试这么做的话，会得到一个编译错误：

```
extension CollectionType
    where Generator.Element: Equatable,
        SubSequence.Generator.Element == Generator.Element
{
    func search
        <Other: SequenceType
            where Other.Generator.Element == Generator.Element>
        (pattern: Other) -> Index?
    {
```

```
        for idx in self.indices {
            // 错误：cannot convert value of type 'Other'
            // to expected argument type
            if suffixFrom(idx).startsWith(pattern) {
                return idx
            }
        }
        return nil
    }
}
```

这看起来很奇怪，我们通过 Other.Generator.Element == Generator.Element 对 Other 进行过约束，让它和我们的元素是相同类型。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.Generator.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 2.0 中，语言还没有强大到能自己推断出这个约束。要绕过这一点，我们需要将 SubSequence 声明为关联类型，这种实现方式看上去应该是这样的：

```
protocol CollectionType {
    associatedtype SubSequence: Indexable, SequenceType
        // 这将无法编译！
        // 因为你不能把 where 语句加入关联类型中
        where SubSequence.Generator.Element == Generator.Element
}
```

要修正这个问题，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 CollectionType
where Generator.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，这么做是没问题的。比如字符串的情况下：

```
// 可以编译，并返回 true：
"hello".characters.search("ell".characters)
```

但是我们在集合中已经看到过，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 CollectionType
    where Generator.Element: Equatable,
        SubSequence.Generator.Element == Generator.Element
{
    // 和之前的 search 实现一样
}
```

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**.Index 和 Other.Index 都遵守 RandomAccessIndexType。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 CollectionType
    where Generator.Element: Equatable, Index: RandomAccessIndexType,
        SubSequence.Generator.Element == Generator.Element
{
    func search
        <Other: CollectionType where
            Other.Index: RandomAccessIndexType,
            Other.Index.Distance == Index.Distance,
            Other.Generator.Element == Generator.Element>
        (pat: Other) -> Index?
    {
        // 如果匹配模式比集合长，那么就不可能会比配，提前退出
        guard !isEmpty && pat.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
        for i in startIndex...endIndex.advancedBy(-pat.count) {
            // 检查从当前位置开始的切片是否以待匹配模式开头
```

```
            if self.suffixFrom(i).startsWith(pat) {
                // 如果是，我们发现了这个子序列
                return i
            }
        }
        // 否则，没有找到
        return nil
    }
}
```

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。即使是 CollectionOfOne 类型的索引 Bit，也使用 Int 作为 Distance。想要替换这个约束的话，我们可以用少量 numericCast 来进行保证，比如 **guard** numericCast(pat.count) <= count **else** { **return nil** }。

由于 Range 本身也是一个集合，你可以将内层 **for** 循环用一个 indexOf 调用来代替，并传给它用来匹配子集合模式的闭包。因为 indexOf 返回的已经是一个可选值了，在没有找到目标时将返回 **nil**，所以你可以直接将 indexOf 调用的结果进行返回。这可以使代码的目的变得更加明确：

```
  return (startIndex...endIndex.advancedBy(-pat.count))
            .indexOf { self.suffixFrom($0).startsWith(pat) }
```

需要注意的是，编译器**不会**强制你将索引指定为随机存取类型。如果你将上面的两个随机存取索引的约束注释掉，这段代码仍然可以编译和运行。但是，因为即使最初级的 ForwardIndexType 也有 advancedBy 成员函数，所以你所重载的 search 可能在索引**不是**随机存取的状况下，算法的效率也会**比较低**。重载方法会不会比原方法更快，是依据被搜索的集合以及作为模式的子序列的相对长度所决定的。这是因为计算它们的长度和对它们进行索引步进的操作要多次调用 successor，它们由常数时间复杂度变为了线性时间复杂度，在我们的例子里，因为我们只调用这个函数一次，所以随机存取的要求并不是必须的，不过它依然会是一个很好的优化。

但是想想看在一个循环里多次调用 advancedBy 的情况吧。比方说，在我们上面的二分查找的例子中，要是你忘了 RandomAccessIndexType 约束，代码依然可以编译，但是你在循环中的时间复杂度将变为线性，这让整个算法退化到了平方复杂度 — 这要比直接遍历集合进行线性搜索还要慢得多。

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。我们不仅可以创建泛型的方法，我们也可以创建泛型数据类型。

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。我们创造了 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。如果 URL 加载失败的话，data 将会是 **nil**，回调的参数也将会是 **nil**。如果 JSON 反序列化失败，json 将会为 **nil**。最后，如果用户解析的过程中发生了错误，那么 users 将会为 **nil**：

```
func loadUsers(callback: [User]? -> ())  {
    let  usersURL = webserviceURL.URLByAppendingPathComponent("/users")
    let  data = NSData(contentsOfURL: usersURL)
    let  json = data.flatMap {
        try? NSJSONSerialization.JSONObjectWithData($0,
            options: NSJSONReadingOptions())
    }
    let  users = (json as? [AnyObject]).flatMap { jsonObject in
      jsonObject.flatMap(User.init)
    }
    callback(users)
}
```

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。两个方法都很难测试，我们需要确保网络服务可以在测试是被访问到，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。

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>(pathComponent: String,
    parse: AnyObject -> A?,
    callback: A? -> ()) {
    let resourceURL = webserviceURL
        .URLByAppendingPathComponent(pathComponent)
    let data = NSData(contentsOfURL: resourceURL)
    let json = data.flatMap {
        try? NSJSONSerialization.JSONObjectWithData($0,
            options: NSJSONReadingOptions())
    }
    callback(json.flatMap(parse))
}
```

现在，我们可以将 loadUsers 函数重写为下面的形式：

```
func loadUsers2(callback: [User]? -> ()) {
    loadResource("/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 AnyObject 转换为一个 AnyObject 的数组，接着对每个元素用提供的解析函数进行解析：

```
func jsonArray<A>(f: AnyObject -> A?) -> AnyObject -> [A]? {
    return { arr in
        (arr as? [AnyObject]).map { $0.flatMap(f) }
    }
}
```

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("/posts", parse: jsonArray(BlogPost.init), callback: callback)
}
```

这让我们能少写很多重复的代码，现在我们可以花一些时间来将我们的同步 URL 处理重构为异步加载。由于我们有了统一的请求函数，所以不再需要分别更新 loadUsers 或者 loadBlogPosts 了。虽然 loadBlogPosts 现在很短，但是想测试它并不容易：它是基于回调的，并且需要网络服务处于可用状态。

在我们添加异步加载之前，我们可以更进一步。在 loadResource 函数中，pathComponent 和 parse 联系非常紧密。我们可以将它们打包进一个结构体中，而不是分别进行传递。当然，和函数一样，这个结构体也可以是泛型的：

```
struct Resource<A> {
    let pathComponent: String
    let parse: AnyObject -> A?
}
```

现在，我们可以定义一个新的 loadResource，它是 Resource 上的一个函数，并使用它对所要加载的内容进行描述：

```
extension Resource {
    func loadSynchronous(callback: A? -> ()) {
        let resourceURL = webserviceURL
            .URLByAppendingPathComponent(self.pathComponent)
        let data = NSData(contentsOfURL: resourceURL)
        let json = data.flatMap {
            try? NSJSONSerialization.JSONObjectWithData($0,
                options: NSJSONReadingOptions())
        }
        callback(json.flatMap(self.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(pathComponent: "/users", parse: jsonArray(User.init))
let postsResource: Resource<[BlogPost]> =
```

```
    Resource(pathComponent: "/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。也就是说，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：

```
extension Resource {
  func loadAsynchronous(callback: A? -> ()) {
      let session = NSURLSession.sharedSession()
      let resourceURL = webserviceURL
          .URLByAppendingPathComponent(pathComponent)

      session.dataTaskWithURL(resourceURL) { data, response, error in
          let json = data.flatMap {
              try? NSJSONSerialization.JSONObjectWithData($0,
                  options: NSJSONReadingOptions())
          }
          callback(json.flatMap(self.parse))
      }.resume()
  }
}
```

现在，我们就能够很容易地进行测试了。测试 Resource 是否配置正确要比异步地测试 loadUsers 函数要简单得多。通常，网络请求都是异步的，网络服务也不一定可用，所以网络代码的测试一般都很困难。在我们现在的实现方式中，我们只需要异步测试 loadAsynchronous，而代码的其他部分都很简单，也没有受到异步代码的影响。当然，我们可以继续扩展 Resource 数据类型，让它有更多的配置选项，比如设定 HTTP 方法，让它可以进行 POST 方式请求等等。

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。它让我们可以重写一个完全不同的加载资源的函数，而不需要改变资源本身的其他任何代码。

总结

在本章一开始，我们将泛型编程定义为使用最小的假设来描述算法的过程。回顾一下，我们将 subtract 函数在数据抽象的最小假设下进行了表述。我们同时添加了精确版本和泛型版本的变种，籍此保证了函数的性能。对于 Hashable 的元素，对于 Equatable 的元素，以及对于接受

一个闭包来判断相等的情况，我们都给出了实现。在使用时，编译器会自动按照我们的问题为我们选择最精确的形式。

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 结构体中进行了移除。在最后的 Resource 中没有根域名的信息，也没有关于如何加载数据的假设。我们在 Resource 中只对 API 接入点进行描述。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。

在下一章里，我们会对协议进行探索，它是泛型编程中的关键构建模块。它可以约束泛型类型，并让我们能够明确地指定对于类型的假设。

协议

10

WWDC 2015 上有一个非常具有影响力的专题，叫做 “面向协议编程” (Protocol-Oriented Programming)。在本书这部分，我们将研究一个面向协议编程的例子，并看看为什么要这么做。

让我们假设我们要写一个日历应用。要求很简单：我们需要将所有日历中的事件在一个列表中显示，并使用日期进行排序。当用户点击一个事件的时候，我们显示这个实现的详细信息。为了将事件显示在列表里，我们创建一个自定义视图来显示这个列表。(注意我们只会展示重要的部分，这可以让解释保持简洁。)

事件的列表在屏幕上所显示出来的的只会是所有事件的一个子集。在我们的应用中，我们想要显示一段时间范围内的所有事件。所以，我们需要定义一个能存储时间范围的结构体：

```
struct DateRange {
    var startDate: NSDate
    var endDate: NSDate
}
```

我们还创建了一个自定义的初始化方法，它接受一个日期，并创建一个日期范围。我们将输入的日期作为起始日期，然后计算出终止日期：

```
extension DateRange {
    init (startDate: NSDate = NSDate(), durationInDays days: Int = 1) {
```

另外，我们还定义了一个简单的 Event 结构体，它会持有事件的标题和日期。稍后，我们将用更复杂一些的结构体或者是从数据库中取出的对象来替代它：

```
struct Event {
    let  title :  String
    let  date: NSDate
}
```

我们已经准备好编写我们的 CalendarView 了。因为这是一个 iOS 应用，所以我们创建了一个 UIView 子类，它有三个属性：

- → displayRange 获取和设置日期范围。
- → delegate 代理，我们通过它来对用户导致的变化作出响应。
- → events，存储事件的数组。在真实应用中，它可能是一个数据源对象，可以根据需要加载事件。

```
class MyCalendarView: UIView {
    var displayRange: DateRange = DateRange()
    var delegate: CalendarViewDelegate?
    var events: [Event] = []
}
```

要将每个单独的事件添加到视图中，我们只需要循环遍历在数据源数组中的所有事件，然后将落在我们想要显示的日期范围内的事件筛选出来。对于每一个满足条件的事件，我们添加一个视图：

```
func setupViews() {
    let displayedEvents = events.filter {
      displayRange.contains($0.date)
    }
    for event in displayedEvents {
        addEventView(event)
    }
}
```

在类之间共用代码

现在假设我们有了新的需求：我们想要再添加一种类型的视图。也许是一个月份的视图，或者是一个只列举即将到来的预约的日程表视图。我们如何重用我们已经写过的代码呢？

很多习惯 Objective-C 的程序员的第一反应可能是使用子类。他们会创建一个 AbstractCalendarView 抽象类，然后再创建像是 WeekView 或者 AgendaView 这样的东西。在我们研究解决方案之前，我们先来看看子类的一些共通的问题。

通过子类共享代码不具有灵活性。类只能有一个父类，这在有时会成为问题。比如，要是你想让 WeekView 同时是 AbstractCalendarView 和 UICollectionView 的子类就是不可能的。在 Cocoa 中还有更多的例子，比如 NSMutableAttributedString 的设计师就需要选择到底是用 NSAttributedString 还是 NSMutableString 作为基类。

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。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 SequenceType where Generator.Element: Comparable {
    public func sort() -> [Self.Generator.Element]
}
```

想要实现 sortInPlace 进行原地排序的话，我们需要更多的构建条件。你需要能够随机存取元素，而不仅仅是进行线性迭代。CollectionType 满足这点，而 MutableCollectionType 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。RandomAccessIndexType 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 MutableCollectionType
    where Index: RandomAccessIndexType, Generator.Element: Comparable
{
    public mutating func sortInPlace()
}
```

最常见的方法是，通过协议来描述最小的功能，然后将它们组合起来使用。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。我们已经在集合这章中一开始使用单个 cons 方法构建 List 类型的例子中看到过这样的应用场景了。我们使得 List 实现了 SequenceType 协议，而不用改变原来 List 结构体的实现。实际上，即使我们不是这个类型的原作者，也可以完成这件事情。通过添加 SequenceType 的支持，我们直接获得了 SequenceType 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。

让我们回头来看看 CalendarView 的例子。我们不使用共享父类的方式，而是选择定义一个协议，让两种视图都去遵守这个协议。这么做有不少优点：

- → 我们可以不被强制要求指定一个父类。MyCalendarView 既可以是 UIStackView 的子类，或者也可以是 UITableView 的子类。与此同时 MonthCalendarView 可以是 UICollectionView 的子类。子类视图的类型由它们自己确定，而不需要一个 CalendarView 父类。
- → 视图不需要担心重写 CalendarView 的事情，不过在重写 UIView 方法的时候，它们仍然要格外小心，但是这是不可避免的。
- → 即使我们不打算实现协议中的方法，我们也不需要在类型中将它写出并留空。我们马上会看到如何用协议扩展的方法提供默认的实现。

第一步，让我们来创建一个 CalendarView 协议。这个协议包含三个属性：displayRange，delegate 以及 events。在协议中，我们需要指明这些属性能做什么，这里我们将这三个属性都标记为 **get** 和 **set**：

```
protocol CalendarView {
    var displayRange: DateRange { get set }
    var delegate: CalendarViewDelegate? { get set }
    var events: [Event] {  get set }
}
```

要让两种视图都满足我们的协议非常简单。在 MyCalendarView 中，我们已经有这三个属性了，所以我们可以直接用一行来表明 MyCalendarView 遵守这个协议。这里我们通过追加的方式将协议添加到了类型上，这个特性非常强大。就算接口不一致，我们也可以通过为 MyCalendarView 写一个接口，并提供所需要的方法，来让 MyCalendarView 遵守 CalendarView。

```
extension MyCalendarView: CalendarView {}
```

在 MyCalendarView 的实现中，我们对 events 数组进行了过滤，从而确保只包含那些在 displayRange 范围内的事件。为了不重复代码，我们将它移动到协议中。我们通过使用 events 和 displayRange 可以为这个功能创建一个协议扩展：

```
extension CalendarView {
    var eventsInRange: [Event] {
        return events.filter { displayRange.contains($0.date) }
    }
}
```

现在，所有遵守 CalendarView 的类型都可以直接获取查找日期范围内的事件的特性了。我们以前已经看到过这项技术了。比如，当一个类型满足 SequenceType 后，它直接免费获得了很多额外的功能。所有这些功能都是构建在少数几个必须的方法上的。

有了协议之后，我们进行实现的思路就完全不同了。比如，我们可以构建一个命令行的日历应用，它可以将日期范围内德所有事件打印出来：

```
struct TextCalendarView: CalendarView {
    var displayRange: DateRange
    var delegate: CalendarViewDelegate?
    var events: [Event] = []

    func display() {
        let formatter = NSDateFormatter()
        formatter.dateStyle = .ShortStyle
        formatter.timeStyle = .ShortStyle
        for event in eventsInRange {
            print("\(formatter.stringFromDate(event.date)): \(event.title)")
        }
    }
}
```

如果 CalendarView 是一个 UIView 子类的话，代码共享就不会如此容易了。

重写协议方法

当我们继续深入 CalenderView 之前，我们先来考虑一些协议接口的细微之处。现在我们想要让日历中的条目能够在社交网络上被共享。我们马上可以创建一个 Shareable 协议，要实现它的类型必须提供一个 socialMediaDescription 属性：

```
protocol Shareable {
    var socialMediaDescription: String { get }
}
```

我们可以在协议扩展中加入 share 方法。它把将要分享到社交网络的的内容打印到标准输出中。我们可以再加入一个方法来对这个输出添加一些格式。

```
extension Shareable {
    func share() {
        print("Sharing: \(self.socialMediaDescription)")
    }

    func linesAndShare() {
      print("----------")
      share()
      print("----------")
    }
}
```

想要让任一类型满足 Shareable，只需要实现 socialMediaDescription 就行了。当我们这么做的时候，就可以免费从协议扩展中获取 share 方法了。另外，如果有需要的话，我们也可以在遵守 Shareable 的类型中实现我们自己的 share 方法。比如，我们可以让 String 实现自己的 Shareable 协议：

```
extension String: Shareable {
    var socialMediaDescription: String { return self }

    func share() {
        print("Special String Sharing: \(self.socialMediaDescription)")
    }
}
```

现在，当我们创建一个字符串并且调用 share 时，它会使用我们自定义的实现：

```
"hello".share()
// 打印 "Special String Sharing: hello"
```

如果我们将 hello 转换为一个 Shareable 的话，调用 share 会得到不同的结果：

```
let hello: Shareable = "hello"
hello.share()
// 打印 "Sharing: hello"
```

更有趣的是，猜猜看如果我们直接在 String 上调用 linesAndShare 时会发生什么？

```
"hello".linesAndShare()
// 打印：
//
// ----------
// Sharing: hello
// ----------
```

后面两个例子的输出可能和你想的不一样。在这里，就算我们为 String 添加了 share 方法，它也不会重写我们的默认实现，而仅仅是 “遮盖” 了默认的实现。这两个 share 实现究竟哪个会被调用，取决于编译器在调用时所看到的类型信息。这个方法的调用有可能是静态派发 (statically dispatched) 的，也可能相反，是在运行时将接收者的类型考虑进去后进行的动态派发 (dynamically dispatched)。

如果我们将 share 作为**协议的要求**写在协议定义中的话，行为就会不一样了。这里，我们将它加入到协议中：

```
protocol Shareable {
    var socialMediaDescription: String { get }
    func share()
}
```

现在，对于 share 的调用将会被动态派发：

```
let hello: Shareable = "hello"
hello.share()
// 打印 "Special String Sharing: hello"
```

在协议扩展中的静态和动态派发的区别有时候会让人很迷惑。要解释为什么之前自定义实现中的扩展方式没有被调用的话，那是因为这个方法没有被列在协议的要求中。

也就是说，根据一个方法是否被写在协议声明的要求中，协议扩展可以包含两种不同的语义。所有在协议要求中写明的方法都是需要在遵守协议的类型中被实现的，它们是协议的**自定义入口**。协议扩展中对于这些要求被实现的方法的实现，为不需要自定义行为的类型提供了默认的行为，但是这些类型也可以提供它们自己的实现版本，这些在类型中的版本一定会被调用，因为这种调用是动态派发的。

另一方面，对于那些协议扩展中进行了实现，但是不存在于协议定义中的方法，将通过静态派发的方式进行调用。你应该使用这样的方法来为所有遵守协议的类型提供共通的功能，这类方法不是自定义行为的入口。

除开性能上的提升以外，静态派发还解决了一个动态派发语言通常会面临的问题。如果你是在写 Objective-C，那么 String 上的 share 方法将会覆盖协议中的同名方法。为了避免这个问题，对于自定义的扩展方法，我们总是需要为它们加上前缀 (比如，oci_share())。通过这么做，我们可以避免错误地覆盖已有的方法。而在 Swift 中，这不再需要了。

添加关联类型

在日历应用最终完成之前，我们可能会决定想要用另一种方法对事件进行存储。现在 Event 是存储在一个结构体中的，我们可能希望使用 Core Data 和数据库来存储这些事件。一个简单的解决方式是把所有出现 Event 的地方用 CoreDataEvent 来替代。不过因为 Event 结构体更容易创建和测试，我们还是希望能在测试中保留它们。

我们不去做粗暴的查找替换，而是在协议中为事件的类型创建一个泛型，让它变成一个关联类型。升级后的 CalendarView 的定义看起来是这样的：

```
protocol CalendarView {
    associatedtype EventType

    var displayRange: DateRange { get set }
    var delegate: CalendarViewDelegate? { get set }
    var events: [EventType] { get set }
}
```

> 在 Swift 2.2 之前，关联类型所使用的关键字并不是 **associatedtype**，而是 **typealias**。所以在 Swift 2.2 之前的代码中，在关联类型和类型别名的地方你看到的都是 **typealias**。

这里，我们马上就遇到问题了。我们添加的 eventsInRange 的方法现在无法编译了。问题的原因在于 event.date。代码仍然假设存储在 events 数组中的是 Event 类型。我们可以将这个假设写出来，使得编译通过。在这个扩展里，我们指出仅在关联类型 EventType 是特定的 Event 类型时，这个扩展有效：

```
extension CalendarView where EventType == Event {
    var eventsInRange: [EventType] {
        return events.filter { displayRange.contains($0.date) }
    }
}
```

现在，eventsInRange 方法只在关联类型是 Event 的时候可以使用。不过我们的 CoreDataEvent 也有一个 date 属性。当然，我们可以为 CoreDataEvent 新建一个扩展。但是，通过添加一个新的协议 HasDate 来指出一个类型拥有 date 属性会是更好的选择：

```
protocol HasDate {
    var date: NSDate { get }
}
```

Event 和 CoreDataEvent 都可以通过添加一行代码来实现这个协议，因为它们都已经有 date 属性了。我们将上面的 eventsInRange 的实现改为仅在 EventType 实现了 HasDate 时有效。这样一来，该扩展就能同时作用于 Event 和 CoreDataEvent。如果我们之后要添加另一个类型，只需要让它也遵守 HasDate 就行了：

```
extension CalendarView where EventType: HasDate {
    var eventsInRange: [EventType] {
        return events.filter { displayRange.contains($0.date) }
    }
}
```

我们可以不加改变地使用 TextCalendarView。因为 events 属性已经是 Event 值的数组了，因此编译器可以推断出关联类型是 Event，就没有必要再去指明它了。如果我们想要把 TextCalendarView 也作为泛型来使用的话，可以为它添加一个泛型参数 E。只要类型 E 满足 HasDate，我们就可以使用命令行来显示这些事件。改变后的代码如下所示：

```
struct TextCalendarView<E: HasDate>: CalendarView {
    var displayRange: DateRange
    var delegate: CalendarViewDelegate?
    var events: [E] = []

    func display() {
        for event in eventsInRange {
            let formatter = NSDateFormatter()
            formatter.dateStyle = .ShortStyle
            formatter.timeStyle = .ShortStyle
            print("\(formatter.stringFromDate(event.date)): \(event)")
        }
    }
}
```

现在的解决方案要比 MyCalendarView 这个类一开始的时候要复杂得多。通过将共享的功能提出到协议中，我们可以共享代码，而不必进行子类处理。现在的方案是泛型化的方案，我们可以使用不同的满足 CalendarView 协议的类来展示数据。同样地，我们也可以改变数据源：只要事件类型上有 date 属性，我们就能将它显示出来。

在你自己的应用中，你可以采取上面的步骤来获取更多的灵活性。但是，不论采取哪种技术，你都要牢记进行取舍。让你的代码更加通用的代价是它将变得更加复杂。这么做确实值得么？我们最好从一个非泛型版本开始，然后一步一步地进行重构。

我们回头看看，为什么基于协议的版本会更加灵活。当我们在使用一个协议时，我们只知道这个类型的**接口**信息。我们知道这个类型上有某些特定的方法和/或变量。但是我们并不知道关于满足这个接口的类型的具体实现细节。另外，有些协议还指明了实现的需求，比如实现应该收敛于哪类复杂度等。

当我们使用类的时候，接口和实现是紧紧绑定在一起的。类暴露了它的实现中的某个方法，让它可以被调用。子类在重写这些方法的时候，需要格外小心。换句话说，子类需要知道很多父类中的实现细节。相比于协议，这种紧耦合使得使用子类的时候非常复杂。

带有 Self 的协议

通常，我们可以把协议类型当作普通类型来使用。举例来说，我们可以创建一个协议来指明一个事件应该具有 date 属性和 title 属性：

```
protocol EventLike {
    var date: NSDate { get }
    var  title :  String {  get }
}
```

通过添加一个空扩展，我们可以让上面的 Event 结构体遵守 EventLike。

现在，我们可以把 EventLike 当作和其他任意类型一样来进行使用。例如，我们可以定义一个类型为 EventLike 的变量。这将会把 Event 中所包含的类型信息全部抛弃。从被声明开始，我们就只能以只读的方式获取 date 和 title 属性了。这个类型的接口被限制在了协议所定义的接口中。虽然实际上这是一个 Event，但是我们不能以 Event 的方式来访问它了：

```
let  sampleEvent: EventLike = Event(title: "My event", date: NSDate())
```

我们还可以将遵守同样协议的不同类型放到同一个集合中。比如，我们可以创建一个 EventLike 对象的数组：

```
let  sampleCDEvent = CoreDataEvent(title: "My CD event", date: NSDate())
var events: [EventLike] = [sampleEvent, sampleCDEvent]
```

对于上面这个数组，我们可以进行 map 操作，并提取出所有的日期。这是因为 date 是定义在协议中的，通过访问数组中 EventLike 的 date 属性就可以获取到它：

```
let  dates = events.map { $0.date }
```

现在，我们来向 EventLike 协议中添加更多功能。比如，我们可以添加一个 overlapsWith 方法，它会检查两个事件在时间上有没有重叠：

```
protocol EventLike {
    var date: NSDate { get }
    var  title :  String {  get }
    func overlapsWith(other: Self) -> Bool
}
```

我们还可以为 Event 结构体添加一个 durationInSeconds 属性，这样我们便能够结合 date 来表达事件的持续时间段。然后对 Event 进行扩展，使它再次满足新定义的 EventLike 协议：

```
struct Event {
    let  title :  String
```

```
    let date: NSDate
    let durationInSeconds: NSTimeInterval
}

extension Event: EventLike {
    func overlapsWith(other: Event) -> Bool {
        return date.timeIntervalSinceDate(other.date) < durationInSeconds ||
            other.date.timeIntervalSinceDate(date) < other.durationInSeconds
    }
}
```

这个变更让新的 EventLike 协议完全不同于原来的版本。因为我们添加了一个对 **Self** 进行了引用的方法，它不再能是一个独立的类型。比如，下面的代码不再能够编译了：

```
let test: EventLike = Event(title: "Chris", date: NSDate(),
    durationInSeconds: 100)
```

我们可以这么来理解为什么当协议中有 **Self** 要求的时候，我们不能够再将它当作独立类型使用：让我们考虑 overlapsWith 方法，当我们使用 Event 时，只有在把它和另一个 Event 参数用 overlapsWith 方法进行比较时，才能得到有效的结果。如果将 test 值的类型设为 EventLike 的时候，我们就把这个类型信息抛弃了。没有了 **Self** 来限制协议，我们将可以用一个别的类型来调用 overlapsWith。

现在我们知道了，我们不能把这个新版本的 EventLike 当作独立类型来使用。也就是说，我们不能将它用作泛型参数了。比如，我们不能定义一个元素类型为 EventLike 的数组：

```
let array: [EventLike] = [] // 这句代码不能编译
```

将 **Self** 引入到协议中，我们限制了灵活性。这是一件好事，因为我们所实现的 overlapsWith 方法并不接受两个不同类型的事件进行比较。除此之外，我们并没有什么损失，因为我们依然可以将 EventLike 元素存储在集合中。只不过我们需要添加一个约束，保证所有的元素都是同样的类型。例如，我们可以创建一个新的 **struct** 来将同样泛型类型的值存储在一个数组里。我们通过创造一个泛型结构体，并将这个泛型约束为遵守 EventLike 协议的类型就可泄了。现在，我们就有有一个所有元素都是同样类型的同质数组了：

```
struct EventStorage<E: EventLike> {
    let events: [E]
}
```

即使有这个约束，我们的类型依然是泛型的。只要目标类型满足 EventLike，我们就能够在这个类型上使用该结构体。确实，我们不能在一个数组中混合存储不同的事件类型，但是我们依然可以写出有用的函数。比如，我们可以创建一个 EventStorage 的扩展来检查数组中的事件是否和另一个事件有重叠：

```
extension EventStorage {
    func containsOverlappingEvent(event: E) -> Bool {
        return !events.lazy.filter (event.overlapsWith).isEmpty
    }
}
```

上面的方法中，我们不能简单地传入 EventType，而是要保证参数和数组中的元素具有同样的类型。

关联类型

带有关联类型的协议和那些带有 **Self** 要求的协议很类似。你不能将它们声明为独立的变量类型。想要理解这个事实，可以考虑一下这样的场景：假设你声明了一个协议用来存储某个类型，并将它的值取回：

```
protocol StoringType {
    associatedtype Stored

    init(_ value: Stored)
    func getStored() -> Stored
}

// 一个存储整数的实现
struct IntStorer: StoringType {
    private let stored: Int
    init(_ value: Int) { stored = value }
    func getStored() -> Int { return stored }
}

// 一个存储字符串的实现
struct StringStorer: StoringType {
    private let stored: String
    init(_ value: String) { stored = value }
```

```
    func getStored() -> String { return stored }
}

let intStorer = IntStorer(5)
intStorer.getStored() // 返回 5

let stringStorer = StringStorer("five")
stringStorer.getStored() // 返回 "five"
```

到目前为止，一切顺利。

让一个变量的类型是协议类型的主要目的是为了获取动态特性。这使你可以将不同的类型赋值给同一个变量。在运行时，根据变量底层的实际类型，这个变量将可以表现出多态 (polymorphic) 特性。

但是如果一个协议有关联类型的话，这就行不通了。下面的代码实际会如何工作?

```
// 如君所见，因为 StoringType 中存在关联类型，
// 这段代码无法编译

// 随机将 someStorer 赋值为 intStorer 或 stringStorer：
var someStorer: StoringType = arc4random()%2 == 0 ? intStorer : stringStorer
let x = someStorer.getStored()
```

上面的代码中，x 的类型应该是什么？是一个 Int 吗？还说是一个 String？在 Swift 中，所有的类型都必须在编译时就被确定。一个函数不能在运行的时候改变类型。

和带有 **Self** 的协议一样，你只能把 StoringType 当作泛型约束使用。假设你想要接受任意 StoringType 类型并打印它的值，你可以这么做：

```
func printStoredValue<S: StoringType>(storer: S) {
    let x = storer.getStored()
    print(x)
}

printStoredValue(intStorer)
printStoredValue(stringStorer)
```

这是可行的，因为在编译的时候，相当于编译器编写了两个版本的 printStoredValue，一个接受 Int 存储，另一个接受 String。在这两个版本中，x 分别都具有具体的类型。

当你将 Swift 中的协议和 Objective-C 里的协议做比较的时候，你也可以看到普通的协议和带有关联类型或者 **Self** 的协议的区别。我们可以用 Swift 的普通协议来替换原来 Objective-C 中所有使用协议的地方。但是那些带有关联类型或者 **Self** 的协议却不一样，它们完全无法在 Objective-C 中表达出来。

即使不和 Objective-C 相比，将带有关联类型或 **Self** 的协议看作是和普通协议完全不同的东西会对代码设计很有帮助。它们不是独立的类型，我们只能将它们作为泛型约束来使用。

协议的本质

协议和泛型

要是一个协议既没有使用 **Self**，也没有关联类型的要求，比如 CustomStringConvertible 或者标准库中其他一些类似的协议，下面的代码有什么区别吗？

这样：

```
func f(x: CustomStringConvertible) { }
```

和这个相比：

```
func g<T: CustomStringConvertible>(x: T) { }
```

让我们来检查看看区别。先要警告的是，下面有些行为是没有在文档中定义的，所以有可能发生改变。我们在写作的时候使用了最新版本的 Swift 进行了测试，但是可能在之后的版本中会发生变化。另外，所有的数字都假设是在 64 位平台上的。

功能上的区别

上面两个例子的区别是什么？第一种写法里，函数接受一个协议类型 CustomStringConvertible 作为参数，它将可以通过协议提供的方法来访问参数中的值。

而在第二种写法里，g 所接受的是一个类型 T，这个类型实现了 CustomStringConvertible。换句话说，T 就是通过参数所实际传入的类型。如果你传入了 Int，那么 T 就是 Int；如果你传入的是整数的数组，那么 T 就是 [Int]。

大多数情况下，这个区别不会对函数的实现带来什么影响。因为就算 T 可能是 Int 或者 [Int]，你还是只能使用由 CustomStringConvertible 所规定的那些属性。因为函数需要能工作于**任意**满足 CustomStringConvertible 协议的类型上，所以你不能调用 x.successor() 或者 x.count，唯一能用的只有 x.description。

功能上来说最大的区别在于当两种形式有返回值的时候。假设你想要自己实现一个标准库中取绝对值的 abs 函数。可能看起来是这样的：

```
func myAbs<T: SignedNumberType>(x: T) -> T {
    if x < 0 {
        return -x
    } else {
        return x
    }
}

// myAbs 对所有种类的有符号数都适用
// (比如 Int，Int64，Float，Double 等等)
let i: Int8 = -4
let j = myAbs(i)
// j 将会是值为 4 的 Int8 整数
```

这个函数依赖于 SignedNumberType 所提供的三个特性：负号运算符，小于号运算符 (因为 SignedNumberType 实现了 Comparable) 以及使用数字 0 来创建相同的类型进行比较 (通过 IntegerLiteralConvertible 实现)。它将输入与 0 进行比较，并返回它本身或是它的相反数。另外，它所返回的值的类型和输入类型是一样的。如果你传入的是 Int8，那么你得到的也是 Int8。如果你传入的是 Double，你得到的也将是 Double。如果你用协议本身作为输入来写这个函数的话，你得到返回会是协议的类型。这不仅会导致类型推断上的不便，而且你很可能需要将结果再用 **as**! 强制转换为你想要的类型，然后才能使用它。这种对于类型信息的丢失被称作**类型抹消** (type erasure)。

从 Swift 2.0 开始，你可以将这个函数写成一个协议扩展。但是它依然是一个泛型函数，这和我们上面所写的全局函数很类似。不过，我们用 **Self** 替代了 T 的位置，相应地，隐式变量 **self** 也取代了参数 x。对协议进行扩展让我们可以将 abs 写成一个计算属性，这种形式使用起来更加自然：

```
extension SignedNumberType {
    var myAbs: Self {
        if self < 0 {
            return -self
        } else {
            return self
        }
    }
}
let k = i.myAbs
// j 将会是值为 4 的 Int8 整数
```

协议的本质

除了功能上的差别，这两个函数还有什么区别呢？你其实还能从一些别的方面对协议类型和实际类型的 T 占位符进行区分。比如，检查一下值的大小：

```
func takesProtocol(x: CustomStringConvertible) {
    // 每次都会打印 40
    print(sizeofValue(x))
}

func takesPlaceholder<T: CustomStringConvertible>(x: T) {
    // 这将打印输入类型的大小 (比如 Int64 是 8，
    // Int8 是 1，类的引用是 8)
    print(sizeofValue(x))
}

takesProtocol(1 as Int16)       // 打印 40
takesPlaceholder(1 as Int16)  // 打印 2

class MyClass: CustomStringConvertible {
    var description: String { return "MyClass" }
}

takesProtocol(MyClass())    // 打印 40
takesPlaceholder(MyClass()) // 打印 8
```

看起来似乎 CustomStringConvertible 是一个固定大小的盒子 (box)，它可以装载所有能被打印的值。这种装箱技术在一些像是 Java 和 C# 的语言中是常见特性。在上面的例子中，Swift 甚至把对类的引用也装到了一个 40 字节的箱子中。如果你习惯将协议想成是对于纯虚基类的引用的话，这种表现可能会让你很惊讶。实际上 Swift 的箱子既可以装载值，也可以装载引用类型。只针对类的协议 (也就是那些使用 **class** 修饰的协议) 会小一些，占用 16 个字节，因为它们肯定不会装载更大尺寸的内容。

我们依然可以将这些泛型函数用协议扩展的方法进行重写：

```
extension CustomStringConvertible {
    func asExtension() {
        print(sizeofValue(self))
    }
}
```

我们可以得到相同的结果：

```
(1 as Int16).asExtension() // 打印 2
MyClass().asExtension() // 打印 8
```

那么，当一个类型被使用协议引用装箱的时候，这 40 个字节里的内容到底是什么呢？我们可以通过 Swift 的按位转换来一探究竟：

```
// 读取一个协议变量内部数据的函数
func dumpCustomStringConvertible(c: CustomStringConvertible) {
    var p = c
    // 你也可以使用 unsafeBitCast
    withUnsafePointer(&p) { ptr -> Void in
        let intPtr = UnsafePointer<Int>(ptr)
        for i in 0.stride(to: sizeof(CustomStringConvertible)/8, by: 1) {
            print("\(i):\t 0x\(String(intPtr[i], radix: 16))")
        }
    }
}

let i = Int(0xb1ab1ab1a)
dumpCustomStringConvertible(i)

// 打印出：
```

```
// 0:       0xb1ab1ab1a
// 1:       0x7fff5ad10f48
// 2:       0x2000000000
// 3:       0x10507bfa8
// 4:       0x105074780
```

对于一个 8 字节的整数，协议首先将这个整数值打包进协议值内。接下来的两个八位看起来是未初始化的内存，只是为了对齐用的 (它们的内容每次运行都不同)。最后两个八位是指向所存储类型的元数据的指针。

如果我们创建一个 16 字节或者是 24 字节的自定义结构体，它依然会比协议中的前三个八位字节持有。一旦我们的结构体超过了 24 字节这个大小，它就将转变为持有一个引用该值的指针：

```
struct FourInts: CustomStringConvertible {
    var a = 0xaaaa
    var b = 0xbbbb
    var c = 0xcccc
    var d = 0xdddd
    var description: String { return String(a,b,c,d) }
}

dumpCustomStringConvertible(FourInts())

// 打印出：
// 0:       0x7f8b5840fb90 // 这是一个指向 FourInts 值的指针
// 1:       0xaaaa         // 未初始化的内存
// 2:       0xbbbb         // 同上
// 3:       0x10dde52a8    // 元数据
// 4:       0x10dde51b8    // 元数据
```

我们怎么知道第一部分就是一个 FourInts 类型的指针呢？这个结论是我们通过解开这个引用并实际观察它的结果所得到的。我们在 dumpCustomStringConvertible 里添加一个函数，用它来告诉这个方法底层的值的真实类型：

```
func dumpCustomStringConvertible
    <T>(var p: CustomStringConvertible, type: T.Type)
{
    withUnsafePointer(&p) { ptr -> Void in
        let intPtr = UnsafePointer<Int>(ptr)
```

```
        for i in 0.stride(to: (sizeof(CustomStringConvertible)/8), by: 1) {
            print("\(i):\t 0x\(String(intPtr[i], radix: 16))")
        }
        // 如果指向的值过大
        if sizeof(T) > 24 {
            // 我们得到的是一个整数，我们在这里将其转换为一个指针，
            // 使用 UnsafePointer<T> 的 bitPattern 进行初始化
            let valPtr = UnsafePointer<T>(bitPattern: Int(intPtr.memory))
            // 现在我们来看看内存中这个位置的内存内容
            print("value at pointer: \(valPtr.memory)")
        }
    }
}

dumpCustomStringConvertible(FourInts(), type: FourInts.self)

// 打印出：
// 0:       0x7f8b5840fb90
// 1:       0x7fff909c5395
// 2:       0xaaaa
// 3:       0x10dde52a8
// 4:       0x10dde51b8
// value at pointer: (43690, 48059, 52428, 56797)
```

对吧！这些值正是我们在结构体中存储的四个整数。

在继续讨论之前，我们想最后强调一点。当你用一个协议引用一个值类型的时候，并不会将它变为一个引用类型：

```
protocol Incrementable {
    var x: Int { get }
    mutating func inc()
}

struct S: Incrementable {
    var x = 0
    mutating func inc() {
        x += 1
    }
```

```
}

var p1: Incrementable = S()
var p2 = p1
p1.inc()
p1.x  // 现在是 1
p2.x  // 依旧是 0
```

性能影响

这样看来，协议似乎会在代码中添加一些非直接的层。这会使得基于协议的实现比泛型占位符的实现更加耗费资源吗？为了测试这一点，我们可以构建一些很小的执行基本操作的结构体，然后通过协议或者是泛型占位符的方式来运行它们。

首先，协议看起来是这样的：

```
protocol NumberGeneratorType {
    mutating func generateNumber() -> Int
}
```

这里我们没有使用关联类型的方式，所以该协议能做的事情就非常有限了，它所做的就是生成一个整数。下面我们提供了该协议的三种不同实现。其中第二种会多次进行迭代，并且将得到的数字进行累加：

```
struct RandomGenerator: NumberGeneratorType {
    func generateNumber() -> Int {
        return Int(arc4random_uniform(10))
    }
}

struct IncrementingGenerator: NumberGeneratorType {
    var n: Int
    init(start: Int) { n = start }
    mutating func generateNumber() -> Int {
        n += 1
        return n
    }
}
```

```
struct ConstantGenerator: NumberGeneratorType {
    let n: Int
    init(constant: Int) { n = constant }
    func generateNumber() -> Int {
        return n
    }
}

func generateUsingProtocol(g: NumberGeneratorType, count: Int) -> Int {
    var generator = g
    return 0.stride(to: count, by: 1).reduce(0) { total, _ in
        total &+ generator.generateNumber()
    }
}

func generateUsingGeneric<T: NumberGeneratorType>(g: T, count: Int) -> Int {
    var generator = g
    return 0.stride(to: count, by: 1).reduce(0) { total, _ in
        total &+ generator.generateNumber()
    }
}
```

如果我们将代码用 -O 进行编译 (这里的 -O 指的是编译优化的级别，默认情况下，应用的 release 版本会使用 -O 优化，这是一种比较强的编译器优化级别)，并将 count 设置为 10,000，我们得到的时间结果如下所示：

方法	时间
泛型随机	261,829μs
协议随机	286,625μs
泛型递增	5,481μs
协议递增	45,094μs
泛型常数	0μs
协议常数	43,666μs

那么这个结果说明了什么？调用 arc4random 是非常昂贵的，所以随机数情况下使用协议所带来的影响可以忽略，不过它确实还是能让人注意到的。但是在递增生成器的例子中，协议带来的用时比例就要比实际执行的时间高得多了。

在常数生成器的例子中，如果使用泛型版本，编译器可以将所有代码都优化掉，将调用转变为一个简单的乘法操作 (迭代次数的时间乘以要生成的常数)。正因如此，它所耗费的是常数级别的时间。而协议所带来的非直接因素在这种情况下就扮演了壁垒的角色，阻止了编译器的优化。

实际上，就算你使用 -Ounchecked 这个比 -O 更强的编译优化，递增生成器中也会发生同样的事情。递增中对于溢出的检查阻止了编译器对其进行优化，而协议的版本也和之前一样保持不变。

对于大部分使用而言，担心协议带来的性能问题属于 “过早优化” 的范畴。使用泛型的更大的优势来自于更加具有表达性的代码，以及避免我们之前提到的类型抹消的问题。不过如果你已经习惯了书写泛型函数，它所带来的性能提升自然是极好的。当要写一个像是 sort 或者 reduce 这样可能会被调用很多次并且与循环紧密写作的库函数时，这一点就很关键了。

当然，这里讨论的内容都有可能会改变。因为协议其实并没有用来定义任何动态行为，所以编译器理论上**也可以**对它们进行优化。不过现在来说，编译器并没有这么做。

动态派发

希望得到动态行为可能是你会去选择使用协议而非泛型的一个场合。即使你要操作的是结构体而非类，协议也可以让你以动态派发的方式进行操作。让我们来看一个简单的例子。

结构体，协议以及动态派发

看看下面的代码：

```
func f(p: CustomStringConvertible) {
    print(p)
}

func g<T: CustomStringConvertible>(t: T) {
    print(t)
}
```

```
let a = [1,2,3]
let i = 1

// 下面的代码将正常工作：两个类型都可已被转换为 CustomStringConvertible
// 并传递到函数中去，接下来，正确的 `description` 将在运行时被调用
f(arc4random()%2 == 0 ? a : i)
```

函数 f 的参数是一个打包后的协议值。因此，我们可以用实际类型不同的值来调用它，而这个值是运行时决定的。协议类型带给了我们一些动态行为，即使我们所使用的是结构体而非类。

下面的代码片段将**无法**编译。T 需要在编译时就决定好是 Int 的数组或者是 Int，但不能两者都是：

```
g(arc4random()%2 == 0 ? a : i)
```

这虽然是一个生造的用例，但是我们通过它可以清晰地看到，通过使用协议，Swift 中的结构体也是可以获得动态特性的。在上面的部分，我们已经看到使用协议的方法对性能来说会有损耗。不过更重要的是 f 相对不那么灵活。它不能直接为我们返回输入类型，而只能返回一个被装箱的协议类型，其中原始的类型信息都被抹消了。

数组协变

如果你尝试将一个 [String] (或者元素为其他任意类型的数组) 传递到一个接受 [Any] (或者其他任何元素为协议，而非具体类型所定义的数组) 的函数时，你将会得到一个编译错误。

虽然所有类型都满足 Any 协议，但是这与 Any 协议是所有类型的隐式父类这件事情是完全不同的。

我们上面看到过，当你将一个类型转换为协议时，你其实是用另外的结构体创建了一个新的值。所以，对于一个类型是 Any 的字符串来说，它需要被从 String 的形式在物理上转换为 Any 的形式：

```
sizeof(String) // 24 字节 (在 64-bit 环境下)，将其转换为 Any 形式的话：
sizeof(Any) // 32 字节，其中包含了一些有关类型到底是什么的元数据
```

因为值类型是直接存储在数组里的，所以这样的数组与普通数据有很大差别。在背后，编译器可能需要等效地进行这样的操作：

```
names.map { $0 as Any } // 创建 Any 版本的新数组
```

Swift 原本也许可以自动帮你完成将特定类型数组转换为协议数组这个流程，只需要你为接受 Any 的函数传入单个变量就可以了。但是我们应该庆幸 Swift 没有这么做。要是这个自动转换会发生的话，在当你的数组很大时，可能会造成在幕后发生大量的处理，而出现性能的问题。

这和你拥有一个存储引用类型的数组是不同的，在引用类型数组中，所有的元素都是指向实际数据的指针，这些指针的尺寸都是相同的，我们可以直接使用它，而不必进行向上的转换操作：

```
class C  { }
class D: C { }

let  d = D()
let  c:  C = d
// 这些值将会是一样的
unsafeBitCast(d, UnsafePointer<Void>.self)
unsafeBitCast(c, UnsafePointer<Void>.self)
```

也就是说，“数组 [D] 其实上是一个 [C] 数组” 这仅仅只是为了编译器高兴，在背后并不会发生任何数据转换：

```
// 这可以良好工作，运行时也不需要转换：
func f(cs:  [C])  { }
let  ds = [D(), D()]
f(ds)
```

但是协议与此不同，你不能把类和父类的引用那一套东西用在协议和遵守这个协议的类型上，它们完全是两回事儿：

```
protocol P { }
extension C: P { }

sizeofValue(C())        // 8 字节 (一个指针)
sizeofValue(C() as P)   // 40 字节

func g(ps:  [P])  { }
g(ds)  // 无法编译，需要转换
```

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。我们在例子中已经看到过这一点，CalendarView 协议就被多个不同类型所实现。在协议上进行扩展，是一种添加新功能的更通用的做法。我们也对两种不同的协议进行了区分：第一种，是像 Objective-C 的协议那样的 “简单” 协议；而第二种，带有关联类型或者 **Self** 的协议要更强大一些。不过这种强大是有代价的，我们不能再将它作为独立的类型来使用。之后，我们深入研究了协议的实现细节。将它们作为独立类型使用，或者通过泛型约束来使用，将会导致不同的结果。通过泛型约束来使用协议可以让我们的返回类型更加灵活，而作为独立类型使用将给我们更多的动态特性。

我们认为，Swift 的协议将会在很长一段时间内对 Swift 社区产生巨大的影响。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，往往会给你带来惊喜。

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。在测试时，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，这让书写测试变得非常容易。

实践：封装 CommonMark

11

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。

CommonMark 还提供了一个用 C 编写的参考实现，这个实现非常高效，而且测试也很齐全。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在类型安全以及易用性上有什么区别。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。

我们采用层层递进的方式进行封装。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 中去。

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);
```

Swift 中，第一个参数的 C 字符串会被导入为指向一系列 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)。我们也需要传递字符串长度，对于选项值 options，我们传入 0 就可以了：

```
func cmark_markdown_to_html
    (text: UnsafePointer<Int8>, len: Int, options: Int32)
    -> UnsafeMutablePointer<Int8>
```

如果想要我们的封装函数能接受 Swift 字符串，当然了，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 length 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：

```
extension String {
    public func markdownToHTML() -> String {
        let  outString = cmark_markdown_to_html(self, self.utf8.count, 0)
        return String(UTF8String: 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。

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: CustomStringConvertible {
    let  node: COpaquePointer

    init (node: COpaquePointer) {
        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)
    -> COpaquePointer
```

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 convenience init?(markdown: String) {
    let  node = cmark_parse_document(markdown, markdown.utf8.count, 0)
    guard node != nil else {  return nil  }
    self. init (node: node)
```

}

上面提到过，有很多很有意思的可以操作节点的函数。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: COpaquePointer) -> 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。对原来枚举中的每个成员，Swift 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：

```
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 child != nil {
        result.append(Node(node: child))
        child = cmark_node_next(child)
    }
    return result
}
```

我们还可以选择返回一个序列而不是数组，就像下面展示的这样。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，而 Node 类则可以只在库的内部进行使用：

```
public enum InlineElement {
    case Text(text: String)
    case SoftBreak
    case LineBreak
    case Code(text: String)
    case InlineHtml(text: String)
    case Emphasis(children: [InlineElement])
    case Strong(children: [InlineElement])
    case Link(children: [InlineElement], title: String?, url: String?)
    case Image(children: [InlineElement], 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: [InlineElement])
    case Header(text: [InlineElement], level: Int)
    case HorizontalRule
}
```

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。我们写了一对函数，来从 Node 类型中创建 Block 和 InlineElement 枚举值，以及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 的函数。通过使用 switch 语句可以对节点的类型进行选择，然后构建对应的 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 inlineElement 函数将它们进行转换。为了避免重复的代码，我们创建了一个 parseChildren 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 Node {
    func inlineElement() -> InlineElement {
        let parseChildren = { self.children.map { $0.inlineElement() } }
        switch type {
        case CMARK_NODE_TEXT:
            return .Text(text: literal !)
        case CMARK_NODE_SOFTBREAK:
            return .SoftBreak
        case CMARK_NODE_LINEBREAK:
            return .LineBreak
        case CMARK_NODE_CODE:
            return .Code(text: literal !)
        case CMARK_NODE_INLINE_HTML:
            return .InlineHtml(text: literal !)
        case CMARK_NODE_EMPH:
            return .Emphasis(children: parseChildren())
        case CMARK_NODE_STRONG:
            return .Strong(children: parseChildren())
        case CMARK_NODE_LINK:
            return .Link(children: parseChildren(), title : title ,
                url: urlString)
        case CMARK_NODE_IMAGE:
            return .Image(children: parseChildren(), title: title ,
                url: urlString)
        default:
```

```
                fatalError("Expected inline element, got \(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 parseListItem 函数中，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 Node {
    public func block() -> Block {
        let parseInlineChildren = { self.children.map { $0.inlineElement() } }
        let parseBlockChildren = { self.children.map { $0.block() } }
        switch type {
        case CMARK_NODE_PARAGRAPH:
            return .Paragraph(text: parseInlineChildren())
        case CMARK_NODE_BLOCK_QUOTE:
            return .BlockQuote(items: parseBlockChildren())
        case CMARK_NODE_LIST:
            let type = listType == CMARK_BULLET_LIST ?
                ListType.Unordered : ListType.Ordered
            return .List(items: children.map { $0.parseListItem() }, type: type)
        case CMARK_NODE_CODE_BLOCK:
            return .CodeBlock(text: literal!, language: fenceInfo)
        case CMARK_NODE_HTML:
            return .Html(text: literal!)
        case CMARK_NODE_HEADER:
            return .Header(text: parseInlineChildren(), level: headerLevel)
        case CMARK_NODE_HRULE:
            return .HorizontalRule
        default:
            fatalError("Unrecognized 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，相比于注释文档来说，这是极大的进步。

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 { $0.block() } ?? []
    return blocks.filter {
        switch $0 {
        case .Header(_, 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。

我们要做的是创建两个叫做 node 的函数：一个负责将 InlineElement 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：

```
convenience init(type: cmark_node_type, children: [Node] = []) {
    let node = cmark_node_new(type)
    for child in children {
        cmark_node_append_child(node, child.node)
    }
    self.init(node: 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 { $0.node() })
    }
    convenience init(type: cmark_node_type, elements: [InlineElement]) {
        self.init(type: type, children: elements.map { $0.node() })
    }
}
```

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 InlineElement {
    func node() -> Node {
        let node: Node
        switch self {
        case .Text(let text):
            node = Node(type: CMARK_NODE_TEXT, literal: text)
        case .Emphasis(let children):
            node = Node(type: CMARK_NODE_EMPH, elements: children)
        case .Code(let text):
            node = Node(type: CMARK_NODE_CODE, literal: text)
        case .Strong(let children):
            node = Node(type: CMARK_NODE_STRONG, elements: children)
        case .InlineHtml(let text):
            node = Node(type: CMARK_NODE_INLINE_HTML, literal: text)
        case let .Link(children, title, url):
            node = Node(type: CMARK_NODE_LINK, elements: children)
            node.title = title
```

```
            node.urlString = url
        case let .Image(children, title, url):
            node = Node(type: CMARK_NODE_IMAGE, elements: children)
            node.title = title
            node.urlString = url
        case .SoftBreak: node = Node(type: CMARK_NODE_SOFTBREAK)
        case .LineBreak: node = Node(type: CMARK_NODE_LINEBREAK)
        }
        return node
    }

}
```

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 Block {
    func node() -> Node {
        let node: Node
        switch self {
        case .Paragraph(let children):
          node = Node(type: CMARK_NODE_PARAGRAPH, elements: children)
        case let .List(items, type):
          let listItems = items.map {
              Node(type: CMARK_NODE_ITEM, blocks: $0)
          }
          node = Node(type: CMARK_NODE_LIST, children: listItems)
          node.listType = type == .Unordered
              ? CMARK_BULLET_LIST
              : CMARK_ORDERED_LIST
        case .BlockQuote(let items):
          node = Node(type: CMARK_NODE_BLOCK_QUOTE, blocks: items)
        case let .CodeBlock(text, language):
          node = Node(type: CMARK_NODE_CODE_BLOCK, literal: text)
          node.fenceInfo = language
        case .Html(let text):
          node = Node(type: CMARK_NODE_HTML, literal: text)
        case let .Header(text, level):
```

```
            node = Node(type: CMARK_NODE_HEADER, elements: text)
            node.headerLevel = level
        case .HorizontalRule:
            node = Node(type: CMARK_NODE_HRULE)
        }
        return node
    }
}
```

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。

对节点进行迭代

这本书就使用 Markdown 编写的，在出版的过程中，我们需要将 Markdown 转换为很多种格式。比如说，我们制作了用来在电子设备上阅读的 PDF 格式，其中的链接是可以被点击的。我们同样还制作了适合打印的 PDF，在那里所有的链接都被用纯文本和在页脚上显示的 URL 代替了。要实现这种转换，我们可以先写一个 deepApply 函数，它负责对所有内联元素递归地进行映射操作。一开始，我们可能会尝试定义一个接口，让我们可以在每个内联元素上执行一个函数：

```
func deepApply(elements: [Block], f: InlineElement -> InlineElement) -> [Block]
```

不过，这种方法将操作限制在了一对一的替换或者更改中。即使我们想要做的只是用文本替换链接这样的简单工作，在这种方法下也是无法完成的。因为虽然看起来这个操作像是一对一的替换，但是链接元素的子节点其实是一个由其他内联元素组成的数组。由于对于节点类型，我们没有一个泛型容器可以对应它们，所以也就没有任何元素可以让我们用来替换链接元素。另

外，我们也不能向语法树中添加元素 (比如脚注)，或者完全移除掉某些节点。一种更通用的接口是返回元素为 InlineElement 的数组，这样一来，我们就可以通过返回一个空数组来移除一个元素，返回只有元素本身组成的单个元素的数组来表示不进行改变，如果想要改变的话，可以返回一个不同的数组。这个函数的签名看起来是这样的：

```
public func deepApply(elements: [Block],
    _ f: InlineElement -> [InlineElement]) -> [Block] {
```

你可以在示例代码中找到相关的实现。接下来，我们将写一个把链接元素进行剥离，将它替换为链接中的文本的函数。我们可以对元素的类型进行选择，如果它是链接的话，返回它的 children，也就是链接中的内联元素；如果类型不是链接，返回一个只有原来的元素本身的单元素数组：

```
func stripLink(element: InlineElement) -> [InlineElement] {
    switch element {
    case let .Link(children, _, _):
        return children
    default:
        return [element]
    }
}
```

现在，我们可以而将文档元素数组中的所有链接替换为文本了：

```
deepApply(elements, stripLink)
```

添加脚注的工作稍微困难一些，因为 CommonMark 默认没有对于脚注或者上标的支持，所以我们需要手动添加它们。我们通过上标 tag 可以很容易生成对应的 HTML。但是，为了能够生成递增的数字，我们需要一个额外的计数器。我们通过为 addFootnote 函数添加一个 **inout** 参数来达到这个目的。该函数可以在函数体内改变计数器的数值，新的数值将在函数返回时被复制给调用者。还要注意的是，这里我们需要使用柯里化，因为 deepApply 需要我们传入的是 InlineElement -> [InlineElement] 类型的函数，而不是 (**inout** Int，InlineElement) -> [InlineElement]。通过柯里化，当给定一个 **inout** Int 后，就可以构建出所需要的类型的函数了：

```
func addFootnote(inout counter: Int) -> InlineElement -> [InlineElement] {
    return { element in
        switch element {
        case let .Link(children, _, _):
```

```
            counter += 1
            return children +
                [InlineElement.InlineHtml(text: "<sup>\(counter)</sup>")]
        default:
            return [element]
        }
    }
}
```

要调用这个函数，我们首先定义变量 counter。因为它是一个 **inout** 参数，我们需要在把它传递给函数的时候在变量名字前面加上 & 符号。Swift 要求将它写明，这样我们就不会在没注意的时候犯错误了：

```
var counter = 0
let  newElements = deepApply(elements, addFootnote(&counter))
```

还有另一种实现 addFootnote 函数的方法。我们可以在函数内部定义 counter 变量，然后返回一个捕获这个计数器变量的闭包。通过将创建计数器的工作移动到函数内部，可以让代码更加简单和安全。上一个例子中的 counter 变量可能会在代码的其他地方被改动，而下面的这种写法中 counte 只会在闭包中被共享：

```
func addFootnote2() -> InlineElement -> [InlineElement] {
    var counter = 0
    return {  element in
        switch element {
        case let  .Link(children,  _,  _):
            counter += 1
            return children +
                [InlineElement.InlineHtml(text: "<sup>\(counter)</sup>")]
        default:
            return [element]
        }
    }
}
```

最后一步是生成实际的脚注。我们首先需要从文档中将链接提取出来。这里使用了 deepCollect，它负责迭代文档中的所有层级的全部元素。这个函数对每个元素执行一个函数，在这个函数里我们可以返回任意类型值的数组，在 deepCollect 结束后它们将被组合为由全部结果构成的数组。我们在这里将检查一个元素是否是链接，如果是的话，则返回链接的 URL：

```
func linkURL(element: InlineElement) -> [String?] {
    switch element {
    case let .Link(_, _, url):
        return [url]
    default:
        return []
    }
}
```

deepCollect 有两种重载。一种接受 Block 元素，另一种则与 InlineElement 值配合使用：

```
public func deepCollect<A>(elements: [Block], f: Block -> [A]) -> [A]
public func deepCollect<A>(elements: [Block], f: InlineElement -> [A]) -> [A]
```

现在可以收集所有元素中的链接，并把对应的 URL 全部提取出来了：

```
let links: [String?] = deepCollect(elements, linkURL)
```

使用 deepCollect 和 deepApply，我们就能很容易地提取和操作由 Markdown 文档带来的各种类型的信息了。之后，我们可以将它们再转回 Node，并将其渲染为 HTML，Markdown，或者 man 页面。在实例中，我们创建了一套原生的 Swift 枚举，它提供了易于使用的 API。同时，得益于 Swift 枚举和 C 数据结构之间的转换方法，我们依然可以使用 CommonMark 代码库的完整特性。

通过为 (像是 Node 类这样的) C 代码创建封装，我们可以将底层 C API 进行抽象转变。这让我们能将精力集中在 Swift 风格的 API 接口上。整个项目都可以在 GitHub 上找到。

互用性进阶

12

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。我们在封装 CommonMark 中已经看到过这方面的例子。libuv 是一个异步 I/O 的库。在本章中，我们将通过把这个较大项目中的一部分进行封装，来实现一个小型的 TCP 服务器。

可能有些人对选择这个库的理由有所质疑。libuv 主要是用来管理异步队列和线程池的。在 Apple 平台我们有 Grand Central Dispatch (GCD) 可以使用，为什么还要去封装 libuv 呢？

除了作为混合使用 Swift/C 的很好的示例以外，libuv 的一个特点是它是跨平台的。在 WWDC 2015 上关于 Swift 最大的新闻不是什么语言特性，而是 Swift 将要成为开源项目，并包含有 Linux 版本。这带来了一点疑问：“为什么会有人想要在 Linux 上用 Swift？要知道那里既没有 Foundation 框架也没有 Apple 的 API 和开发生态。” 其实 Foundation 框架的很大部分已经在开源代码一开始公布的时候就包含在内了，而且 Linux 平台上还有很多功能强大的 C 代码库。就像我们在 CommonMark 一章中看到的那样，只需要不多的关于互用性的知识，我们就能将它用不算多的代码有效地封装为 Swift 风格的 API。

函数指针

当我们深入 libuv 之前，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。在导入过程中参数的名字都丢失了，我们在这里将参数名字加了回来：

```
func qsort(base: UnsafeMutablePointer<Void>, nel: Int, width: Int,
    compar: (@convention(c) (UnsafePointer<Void>, UnsafePointer<Void>)
        -> Int32)!)
```

qsort 的 man 页面 (man qsort) 描述了如何使用这个函数：

> qsort() 和 heapsort() 函数可以对一个有 nel 个元素的数组进行排序，base 指针指向数组中第一个成员。数组中每个对象的尺寸由 width 规定。
>
> 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(inout array: [String]) {
    qsort(&array, array.count, strideof(String)) { a, b in
        let l: String = UnsafePointer(a).memory
        let r: String = UnsafePointer(b).memory
        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 UnsafeMutablePointer 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 strideof **而非** sizeof 来获取每个元素的宽度。在 Swift 中，sizeof 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，特别是结构体和枚举，都会是相同的。当你将代码从 C 转换为 Swift 时，对于 C 中的 sizeof，可能在 Swift 中使用 strideof 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。

比较函数接受两个 void 指针。一个 C 的 void 指针可以指向任何东西。我们要做的第一件事就是将这个指针转换为实际的类型。qsort 里，指针所引用的是数组中的元素，在我们的例子中，我们知道它是 Swift 字符串。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 A 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：

```
func qsortWrapper<A: Comparable>(inout array: [A]) {
    qsort(&array, array.count, strideof(A)) { a, b in
        let l: A = UnsafePointer(a).memory
        let r: A = UnsafePointer(b).memory
        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 OS X 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，它是一个不安全的可变 void 指针。注意这个参数也被加入到了比较函数中，因为 qsort_r 会在调用这个比较函数时将 thunk 传递过来：

```
func qsort_r(base: UnsafeMutablePointer<Void>, nel: Int, width: Int,
```

```
    thunk: UnsafeMutablePointer<Void>,
    compar: (@convention(c)
        (UnsafeMutablePointer<Void>, UnsafePointer<Void>,
            UnsafePointer<Void>)
        -> 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不安全的可变 void 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不安全可变指针。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 = (UnsafePointer<Void>, UnsafePointer<Void>) -> Int32
func qsort_block(array: UnsafeMutablePointer<Void>, _ count: Int,
    _ sz: Int, f: Block)
{
    var thunk = f
    qsort_r(array, count, sz, &thunk) { (ctx, p1, p2) -> Int32 in
        return UnsafePointer<Block>(ctx).memory(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：

```
func qsortWrapper<A: Comparable>(inout array: [A]) {
    qsort_block(&array, array.count, strideof(A)) { a, b in
        let l: A = UnsafePointer(a).memory
        let r: A = UnsafePointer(b).memory
        if r > l { return -1 }
        else if r == l { return 0 }
        else { return 1 }
    }
}
```

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。

封装 libuv

libuv 是一个解决异步和事件驱动编程问题的跨平台框架，它是用 C 写的。在 Apple 的平台上，这些任务一般是由 NSRunLoop 和 GCD 来处理，但是如果我们要写跨平台的 Swift 的话，libnv 也是一个选项。这个库在背后支持了像是 Node.js 和很多其他项目。如果你想要了解更多关于 libuv 的内容，可以看一看 Nikhil Marathe 的《libuv 介绍》这本书。

一个简单的例子

想要开始进行 libuv 的 Swift 接口开发的一个很好的起始点是将一些示例代码迁移到 Swift。下面这段代码来自之前提到的书中，它展示了如何用 C 来写一个 libuv 版本的 Hello World：

```
int main() {
    uv_loop_t *loop = malloc(sizeof(uv_loop_t));
    uv_loop_init(loop);

    printf("Now quitting.\n");
    uv_run(loop, UV_RUN_DEFAULT);

    uv_loop_close(loop);
    free(loop);
    return 0;
}
```

在 Swift 中，我们可以用 **defer** 语句来稍微改变一下代码的顺序，这样我们可以将 alloc 和 dealloc，以及 **init** 和 close 分组写在一起。所有的 **defer** 代码块都会在即将离开作用域的时候被逆序地调用。我们通过将 1 传递给 UnsafeMutablePointer.alloc 初始化了一个 uv_loop_t：

```
func loopAndQuit() {
    let loop = UnsafeMutablePointer<uv_loop_t>.alloc(1)
    defer { loop.dealloc(1) }
```

```
    uv_loop_init(loop)
    defer { uv_loop_close(loop) }

    print("Now quitting")
    uv_run(loop, UV_RUN_DEFAULT)
}
```

为了说明我们将要构建的抽象方式，我们会把 loop 封装在一个类里。因为我们要处理的是可变指针，而它只能提供引用语义的特性，所以代表了值语义的结构体在这里并不是合适的选择。这个类的实现非常直接：我们把申请内存空间和初始化的相关内容移动到 **init** 阶段，把关闭和回收内存空间移动到 **deinit** 阶段，然后我们只需要提供一个 run 方法用来启动这个 loop 就可以了：

```
typealias LoopRef = UnsafeMutablePointer<uv_loop_t>

class Loop {
    let loop: LoopRef

    init(loop: LoopRef = LoopRef.alloc(1)) {
        self.loop = loop
        uv_loop_init(loop)
    }

    func run(mode: uv_run_mode) {
        uv_run(loop, mode)
    }

    deinit {
        uv_loop_close(loop)
        loop.dealloc(1)
    }

    static var defaultLoop = Loop(loop: uv_default_loop())
}
```

现在，我们可以把我们的程序重写为三行代码了，所有关于 loop 的细节都被移动到了类里。注意只要 loop 还在作用域中，指针就是有效的。不过，一旦它离开了作用域，**deinit** 就会被调用，相应的内存也会被释放掉：

```
func main() {
    let  loop = Loop()
    print("Now quitting")
    loop.run(UV_RUN_DEFAULT)
}
```

和 CommonMark 的封装一样，在已有库的主要数据类型上提供一层类的简单封装会让事情变得容易很多。这会使得整个封装更易于理解。之后，可以在此基础上进行更高层级的抽象。

TCP 服务器

接下来我们可以开始写 TCP 服务器了，这主要有下面几步：

1. 初始化一个新的 TCP socket。
2. 将这个 socket 绑定到一个地址上。
3. 监听新的连接，并在 C 回调中处理它们。

在回调中，我们需要做的事情有：

1. 为客户端建立一个 socket。
2. 在这个新建的 socket 里接受从服务器过来的输入连接。
3. 开始从客户端 socket 读取数据。
4. 将返回写给客户端。
5. 释放资源。

这里面有些细节是比较乏味的，所以我们将只专注于和本章相关的有意思的部分。你可以在 GitHub 上找到完整的源码。

对流进行封装

流 (Stream) 是 libuv 中的核心数据类型之一。流代表的是一个双向通讯的信道，你可以打开，读取，写入以及关闭一个流。在 libnv 中，有三种类型的流被实现了，它们是：TCP socket，管道 (pipe) 和控制台流 (比如标准输入和标准输出)。libnv 中使用的术语有时候会让人迷惑，比如

listen 函数是对 socket 进行操作的，但是它却在 stream 上被实现。我们在这里跟随了 libuv 的选择，也将它在 stream 的封装中进行实现。

我们从把 stream 数据类型进行简单封装入手。在 libnv 中，一个流” 对象 “是用一个指向 uv_stream_t 的指针来表示的。和之前的处理一样，我们也将它封装到类里：

```
typealias StreamRef = UnsafeMutablePointer<uv_stream_t>

class Stream {
    var stream: StreamRef

    init(_ stream: StreamRef) {
        self.stream = stream
    }
}
```

要开始监听，我们可以封装 uv_listen 函数。它将会开始监听一个流，然后当新的连接抵达的时候调用我们传进来的回调函数。注意这个回调是一个 C 函数指针，我们会在本章稍后的时候介绍如何处理它们。在这个回调里，我们需要接受 socket，所以我们对 uv_accept 函数也进行了封装：

```
func listen(backlog numConnections: Int, callback: uv_connection_cb)
    throws -> ()
{
    let result = uv_listen(stream, Int32(numConnections), callback)
    if result < 0 { throw UVError.Error(code: result) }
}

func accept(client: Stream) throws -> () {
    let result = uv_accept(stream, client.stream)
    if result < 0 { throw UVError.Error(code: result) }
}
```

最后，另一个基础函数是关闭 socket。在我们的例子中，我们在 socket 关闭之后就不再重用它的 handle 了，所以除了关闭 socket，我们还对内存进行了释放。在上面的 Stream 类中，我们也可以在 **deinit** 中执行 free。但是在这里我们不能这么做。这是因为这些回调函数是以异步方式工作的，封装的类可能在异步操作还在执行的时候就离开作用域了。这种情况下，对 stream 指针的内存进行释放将会导致代码崩溃：

```
func closeAndFree() {
    uv_close(UnsafeMutablePointer(stream)) { handle in
        free(handle)
    }
}
```

封装 TCP Socket

解下来我们要封装的是 TCP socket。Socket 在 libnv 中被 uv_tcp_t 类型表示，在我们的封装类里，我们还是希望在类里保存一个指向原来 socket 的指针。libuv 的文档告诉我们 uv_tcp_t 是 uv_stream_t 的一个 “子类”。也就是说，在 C 中，我们可以安全地进行下面这样的向下转换：

```
uv_tcp_t *tcp =  ...;
uv_stream_t *stream = (uv_stream_t*) tcp;
```

这需要在我们的 TCP socket 类型里有所体现。我们创建一个 socket 实例变量，在它的 **init** 函数中，我们调用父类的 **init** 函数，并且进行同样的非安全类型转换：

```
class TCP: Stream {
    let  socket = UnsafeMutablePointer<uv_tcp_t>.alloc(1)

    init (loop:  Loop = Loop.defaultLoop) {
        super.init(UnsafeMutablePointer(self.socket))
        uv_tcp_init(loop.loop, socket)
    }
}
```

现在，当我们想要启动一个 TCP 服务器时，可以为 TCP 添加一个方法，来将 socket 绑定到一个地址上。这里的 Address 类是另一个简单的封装类 (这里我们不再将它写出了，你可以在完整的源代码中看到它)：

```
func bind(address: Address) {
    uv_tcp_bind(socket, address.address, 0)
}
```

现在我们已经具备了构建一个简单的 TCP 服务器的所有组件了。在将它们放在一次的首次尝试中，我们可能会写出下面这样的代码。我们创建一个新的 socket，将它绑定到 8888 端口，然后开始监听到达的连接。当一个连接抵达时，我们不处理任何数据，而是立即将它关闭：

```
func TCPServer(handleRequest: (Stream, NSData, () -> ()) -> ()) throws {
    let server = TCP()
    let addr = Address(host: "0.0.0.0", port: 8888)
    server.bind(addr)
    try server.listen(backlog: numConnections, callback: { stream, status in
        let client = TCP()
        try! server.accept(client)
        client.closeAndFree()
    })
    Loop.defaultLoop.run(UV_RUN_DEFAULT)
}
```

然而，当我们编译上面的代码时，我们立即遇到了问题。C 代码库的回调希望函数是一个 C 函数指针。而闭包外上下文中的 server 变量在闭包内被捕获了，这使得该闭包不能被转换为一个 C 函数指针。要绕过这个问题，可以在回调中使用 stream 参数 (它的类型为 UnsafeMutablePointer<uv_stream_t>)。接下来我们会通过 stream 来重新构建服务器 socket：

```
func TCPServer(handleRequest: (Stream, NSData, () -> ()) -> ()) throws {
    let server = TCP()
    let addr = Address(host: "0.0.0.0", port: 8888)
    server.bind(addr)
    try server.listen(backlog: numConnections, callback: { stream, status in
        let server = Stream(stream)
        let client = TCP()
        try! server.accept(client)
        client.closeAndFree()
    })
    Loop.defaultLoop.run(UV_RUN_DEFAULT)
}
```

不过，只要我们尝试做一些更复杂的事情，我们马上又会遇到同样的问题：传入的函数不能捕获回调代码块之外的任何变量，否则我们就不能将它当作 C 函数指针来使用。在 qsort 的例子中，我们使用了该函数的一个变种，它接受一个 void 指针并且在回调中传回这个指针，libuv 中也有类似的技术。只不过，libuv 的函数中不再是接受一个 void 指针，很多数据类型都有一个叫做 data 的成员，它就是用来解决这个问题的。

比如说，我们从客户端 socket 中读到的所有数据都会通过数据块的方式传递给我们，读取数据的回调将会被多次调用。我们可以使用 stream 的 data 字段来存储这些缓存数据，然后将它们

传递给外部作用域，这种方式虽然比较灵活，但是使用起来会比较困难。在这里，我们将 listen 和 read 重写为直接接受 Swift 闭包作为回调的版本。这样一来，我们的 API 的用户将会得到一个他们所期望的接口，而不必又跳一遍我们刚才踩过的坑。

最后，我们依然需要一种方法来将 Swift 闭包传递给 C 的回调。我们通过将一个同时包含 read 和 listen 回调的对象存储在 Stream 类中的叫做 context 的属性里。Stream 类会在稍后负责将这个对象与一个 void 指针相互转换，我们后面会看到这个过程：

```
typealias ReadBlock = ReadResult -> ()
typealias ListenBlock = (status: Int)  ->  ()

class StreamContext {
    var readBlock: ReadBlock?
    var listenBlock: ListenBlock?
}
```

我们在 stream 对象上添加一个 listen 方法，它将接受一个 Swift 闭包，并将其存储在上下文中。然后，当回调被调用时，我们从类型为 uv_stream_t 的 serverStream 中重新创建 Stream 对象，从上下文中获取 Swift 闭包。并且调用它。注意在 C 的变体中，回调有一个指向服务器 stream 的指针作为参数，它可以持有上下文。这样一来，我们就不需要在 Swift 的回调中带有这个参数，因为我们可以在闭包内部使用变量捕获来引用所需要的两个 socket 了。

```
func listen(numConnections: Int, theCallback: ListenBlock) throws -> () {
    context.listenBlock = theCallback
    try  listen(backlog: numConnections) { serverStream, status in
        let  stream = Stream(serverStream)
        stream.context.listenBlock?(status: Int(status))
    }
}
```

对于 read，我们使用同样的方法。读取的回调接受的参数是一个 stream，读取的字节数，以及一个缓冲对象。这个回调会被调用多次，每个数据块完成时就调用一次。回调中可能有三种情况会发生，我们可以通过 bytesRead 的值来判定调用对应的是哪种情况：如果 bytesRead 是正值，那么代码块是有效的；如果它是 UV_EOF，代表所有数据都已经被读入了；如果它是任意的其他负值，那么就意味着发生了错误。为了区分这些情况，我们创建了一个 ReadResult 枚举：

```
enum ReadResult {
    case Chunk(NSData)
```

```
    case EOF
    case Error(UVError)
}
```

我们可以为 ReadResult 添加一个额外的初始化方法，来由读取的字节数和指向 uv_buf_t 的指针创建一个 ReadResult。在我们遇到 EOF 的时候，我们将 **self** 设置为 EOF；在遇到错误时，我们将 **self** 设置为错误枚举；最后，当我们读到了一些字节时，我们使用刚刚读取到的数据块来创建一个 NSData 值：

```
extension ReadResult {
    init(bytesRead: Int, buffer: UnsafePointer<uv_buf_t>) {
        if bytesRead == Int(UV_EOF.rawValue) {
            self = .EOF
        } else if bytesRead < 0 {
            self = .Error(.Error(code: Int32(bytesRead)))
        } else {
            self = .Chunk(NSData(bytes: buffer.memory.base, length: bytesRead))
        }
    }
}
```

我们现在具备了实现 read 函数的所有部件。首先，我们将 Swift 的回调存储在我们的上下文中。接下来，在读取的回调中，我们通过 serverStream 重建 Stream 对象。然后，我们使用 bytesRead 和 buf 构建了一个 ReadResult，并最终用这个读取结果值来调用 Swift 的回调。

```
func read(callback: ReadBlock) throws {
    context.readBlock = callback
    uv_read_start(stream, alloc_buffer) { serverStream, bytesRead, buf in
        defer { free_buffer(buf) }
        let stream = Stream(serverStream)
        let data = ReadResult(bytesRead: bytesRead, buffer: buf)
        stream.context.readBlock?(data)
    }
}
```

存储 void 指针

要在 stream 的 void 指针中存储上下文对象，我们需要采取和刚才我们看到的稍有不同的方式。到目前为止，我们都使用的是 & 前缀的方法来创建不安全的可变指针。这仅仅只会创建一个指向某个变量内存的指针，而不会持有它的值。然而，因为 libnv 是一个异步库，使用我们的不安全指针的回调有很大可能会比变量本身具有更长的生命周期。解决办法是手动持有这个变量，然后在我们完成工作后再去释放掉它。所以，我们要创建一个 retainedVoidPointer 函数，它会持有指针 (或者当其中值为 **nil** 时返回一个 null 指针)。手动持有也意味着我们需要对它的生命周期特别小心，在使用完毕后，需要再释放一次。所以，我们还写了一个 releaseVoidPointer 来进行释放。因为我们不能对结构体进行持有，所以我们需要将结构体包装到一个 Box 类中。然后，使用 Unmanaged 我们就可以将它通过一个不透明指针转换为不安全的可变指针了：

```
// 如果 A 的值不是 nil，则持有它
func retainedVoidPointer<A>(x: A?) -> UnsafeMutablePointer<Void> {
    guard let value = x else { return nil }
    let unmanaged = Unmanaged.passRetained(Box(value))
    return UnsafeMutablePointer(unmanaged.toOpaque())
}

// 释放指针中的值，并返回这个值
func releaseVoidPointer<A>(x: UnsafeMutablePointer<Void>) -> A? {
    guard x != nil else { return nil }
    return Unmanaged<Box<A>>.fromOpaque(COpaquePointer(x))
        .takeRetainedValue().unbox
}
```

最后，我们还需要一个不会释放指针的 unsafeFromVoidPointer 函数。这个函数做的是将指针中存储的值提取出来 (除非给入的是 **nil** 指针)，同时不改变持有关系。

```
// 返回指针中的值，而不释放它
func unsafeFromVoidPointer<A>(x: UnsafeMutablePointer<Void>) -> A? {
    guard x != nil else { return nil }
    return Unmanaged<Box<A>>.fromOpaque(COpaquePointer(x))
        .takeUnretainedValue().unbox
}
```

现在，我们可以扩展 Stream 类，为它添加一个 _context 属性。我们用这个属性来在 libuv 的 stream 的 data 字段中存储类型为 StreamContext 的可选值对象。将它的类型设置为可选值

是因为在我们第一次访问它的时候，它可能还是 nil。另外，我们也希望可以将它设置为 nil，这将能让我们释放当前的上下文：

```
var _context: StreamContext? {
    get {
        return unsafeFromVoidPointer(stream.memory.data)
    }
    set {
        let _: StreamContext? = releaseVoidPointer(stream.memory.data)
        stream.memory.data = retainedVoidPointer(newValue)
    }
}
```

最后，我们在 _context 属性的基础上创建一个 context 封装，它将在 _context 为 nil 时创建一个空值并对它进行赋值。如果 _context 已经有值了，则直接返回 _context 计算属性的内容：

```
var context: StreamContext {
    if _context == nil {
        _context = StreamContext()
    }
    return _context!
}
```

当然，我们可以将 _context 和 context 合并为一个属性，但是我们现在的做法更清晰一些。最终，要清理上下文时，我们需要对 closeAndFree 的实现稍加更改，添加一行代码将 _context 设置为 nil，这将让我们能够释放当前的上下文对象。

构建 TCP 服务器

现在已经万事俱备了，我们已经准备好了写一个可以与 Swift 回调协同工作的 TCP 服务器的所有部件。首先，我们再次对 Stream 进行扩展，为其添加一个 bufferedRead 方法。它使用我们之前定义的 read 方法来将每次读取的数据块添加到一个 NSMutableData 实例上。当我们读到 stream 的结束时，将整个缓冲区作为参数对回调函数进行调用：

```
func bufferedRead(callback: NSData -> ()) throws -> () {
    let mutableData = NSMutableData()
    try read { [unowned self] result in
        switch result {
```

```
            case .Chunk(let data):
                mutableData.appendData(data)
            case .EOF:
                callback(mutableData)
            case .Error(_):
                self.closeAndFree()
            }
        }
    }
```

我们可以把所有这些部分包装到一个函数里，这个函数将设置 TCP 服务器并且运行它。我们使用一个闭包来对函数进行配置，这个闭包在接受到一些数据时，可以写入响应返回给客户端。在我们的例子中，我们做的仅仅是在数据写出完成后将 socket 关闭：

```
typealias RequestHandler = (data: NSData, sink: NSData -> ()) -> ()
```

有了这些扩展方法，我们最后的封装函数相对就十分简短了。首先，我们创建一个 socket，绑定地址，然后开始监听新连接。当新连接到达时，我们为客户端创建一个 socket。然后我们尝试让服务器接受这个 socket，并用缓冲的方式读取数据。当所有数据都被接收到以后，我们通过回调来处理请求。最后，在接收，读取或者写入数据的任何一步过程中发生错误时，关闭这个 socket：

```
func runTCPServer(handleRequest: RequestHandler) throws {
    let server = TCP()
    let addr = Address(host: "0.0.0.0", port: 8888)
    server.bind(addr)
    try server.listen(numConnections) { status in
        guard status >= 0 else { return }
        let client = TCP()
        do {
            try server.accept(client)
            try client.bufferedRead { data in
                handleRequest(data: data, sink: client.put)
            }
        } catch {
            client.closeAndFree()
        }
    }
    Loop.defaultLoop.run(UV_RUN_DEFAULT)
```

```
}
```

让我们来写一个简单的回音服务器吧，它将会读取一个输入的字符串，并且将它原封不动地作为请求的响应返还给客户端。我们运行服务器，在回调中，我们通过数据创建一个字符串，将其打印到调试控制台中，并将同样的内容返还给客户端：

```
try runTCPServer() { data, sink in
    if let string = String.fromCString(UnsafePointer(data.bytes)),
       let data = string.dataUsingEncoding(NSUTF8StringEncoding) {
        print(string)
        sink(data)
    }
}
```

要对它进行测试，我们可以使用 OS X 的 nc 命令，它可以将数据发送给服务器：

```
echo "Hello" | nc 127.0.0.1 8888
```

到这里为止，我们本章对于互用性的讨论就接近尾声了。我们研究了 C 函数指针，以及它不能捕获状态的原因。如果我们想要在 C 函数指针的地方使用 Swift 闭包的话，我们要么将它改写为不带捕获的形式，要么使用一个提供 (比如 qsort_r 中的 thunk 指针这样的) 额外参数的版本的 C API 来存储必要信息。

我们在前一章中已经看到过，将 C 类型用类来封装是很直接的。我们可以使用 **deinit** 来执行清理工作。另外，我们在本章里也看到了，当要处理的是异步代码时，由于 C 指针可能会超出其封装类的生命周期，所以事情变得有点复杂。因此，我们必须在封装一个异步库的时候谨慎地处理内存的问题：我们不会想要太快释放内存，我们也不会想引入内存泄漏。通过仔细考虑关于内存的问题，我们可以而写出暴露最小 API 接口的良好封装。这样一来，使用 API 的用户就可以不用担心内存管理的问题了。比如 runTCPServer 函数仅仅只是接受一个函数作为参数，而所有对于 libuv 的封装都被隐藏起来了。