Swift 语言快速入门教程

视频教程：http://swift.jikexueyuan.com/?hmsr=wenku_swift

Swift 教程 安卓开发教程 ios 开发教程 cocos2dx 教程 手机应用开发 游戏开发教程

Swift 体会

前言

我不算是一个果粉，但是我很喜欢苹果的产品，甚至可以说是狂热。6 月 2 号晚上我一夜未眠，就是在等苹果的 WWDC，这是开发者的狂欢之夜（或者狂欢之日，因为美国时间是白天）。凌晨 1 点，终于开始了，但是苹果一次又一次的新产品展示让我一次又一次的失望，作为开发者的我表示无感。

我坚持看因为我不相信苹果总是让我失望。最终在 WWDC 的最后一个环节，我真心沸腾了，因为一门新的编程语言 Swift 出现了。要知道在饱受了 Objective-C 语言之苦后看见 Swift 是一种什么心情，你不需要再为引用计数而头疼，不需要再为没有命名空间而不得不添加该死的前缀而烦恼，不需要再为 Objective-C 那难懂的语法而整天晕乎乎的。此外，苹果还为我们提供了 Playground 这个工具，它能够在我们写程序时实时计算出变量的值，甚至还能呈现数字变量的趋势图，让编程富有乐趣及创造性。

由于 Swift 出现得晚，所以她拥有了目前世界几乎所有编程语言的优点，而没有目前世界所有编程语言的缺点，Swift 她真的太漂亮了。下面我们开始一步一步认识她。

运行速度

从苹果官方所给出的数据来看，Objective-C 比 Python 快 2.8 倍，而 Swift 比 Python 快 3.9 倍，可见苹果在 Swift 上下了大量的功夫进行优化。

开发环境

Swift 语言的开发环境是苹果公司提供的集成开发环境 Xcode，可以用来开发 iOS 应用、iOS 游戏、OSX 窗体程序、OSX 游戏、OSX 命令行程序，读者可以直接从 AppStore 中搜索并下载。用 Swift 语言可以做到几乎所有 Objective-C 所做到的事情，所以 Swift 必将取代 Objective-C，如果你还没有学过 Objective-C 语言，那么恭喜你，不用学了，直接学习 Swift 即可。

运行环境

Swift 被强大的 llvm 编译成机器码，直接运行在系统中。由于 Swift 是苹果的产品，所以目前只支持苹果的系统（OSX 和 iOS）。我们期待会有社区开发出跨平台的 Swift 语言，因为这么好的一门编程语言，应该能够让世界上的每一个人享受到才好，就像 mono 让 C#语言跨平台一样。

语言特性

具有所有现代编程语言的特性，包括：面向对象、类扩展、命名空间、闭包、泛型、强类型、函数多个返回值等等。

这些特性能够大大提高程序员的开发效率，从而为企业节约成本，同时让编程工作充满乐趣。

语法简介

通过前文的介绍，相信读者已经迫不及待的想尝试 Swift 了，下面我们就来认识一下 Swift 的语法。

简洁的语法

Swift 抛弃了 Objective-C 那种古板难懂的语法，采用通俗易懂的脚本语言类语法，学过 Python、JavaScript 或者 Lua 语言的读者肯定不会陌生，这大大降低的初学者的学习成本。

变量及常量

如果要定义一个变量 i 等于 1，只需要写 `var i = 1`，可以看出，不需要指定类型，因为她会自动做类型推断

如果要定义一个常量 PI 等于 3.14，只需要写 `let PI = 3.14`，常量只能被赋值一次。

输出语句

在 Swift 中，可以直接使用 println 函数来输出一段任意类型的信息，如下：

```
println("Hello Swift")
```

字符串连接

Swift 语言中的字符串连接同样也非常简单，如果两个值都是字符串，则可直接用加号连接，如下所示：

```
var hello = "Hello"
var world = "World"
var str = hello + world
```

如果将要连接的值中有其它类型，则直接使用\()包括就可连接，如下所示：

```
var hello = "Hello"
var num = 100
var str = "\(hello) \(num)"
```

从上面的示例可以看出，相比 Objective-C 或者 C/C++语言来说，简单太多了。

循环

传统的 C 语言的 for 循环是这么写的：

```
for (int i = 0; i < 100; i++) {
        //TODO
}
```

在 Swift 中，将循环大大简化了，如下所示：

```
for i in 0..100{
    //TODO
}
```

具体内容还有很多，详见本书正文。

条件判断

条件判断与 C 语言并不区别，你可以直接书写 C 语言的语法即可通过，如下：

```
if(count>5){
    //TODO
}
```

在 Swift 中还可以再简洁一些，如下：

```
if count>5 {
    //TODO
}
```

另外，在 Swift 语言中 Switch…case 语句可以不用加 break 关键字。

函数

如果要定义一个函数用来输出一段信息，则代码如下：

```
func sayHello(){
    println("Hello Swift")
}
```

这种写法非常简洁，我曾在 Dart 语言中见过这种写法，func 是个关键字，用来指明所定义的是一个函数，sayHello 是函数名称，()中是该函数的传入参数。如果还想使用传入参数及返回值，如下所示：

```
func max(a:Int,b:Int)->Int{
    if a>b {
        return a
    }else{
        return b
    }
}
```

该函数名称为 max，可传入两个参数，都是整数类型，参数名称分别为 a、b，函数返回值也是整数类型。

Swift 中的函数还可以同时返回多个值，如下所示：

```
func getNum()->(Int,Int){
    return (3,4)
}
```

如果想对该函数调用并获取到这两个返回结果值，则用法如下：

```
let (a,b) = getNum()
```

面向对象

类的定义非常简单，如下所示：

```
class Hello{
    func sayHello(){
        println("Hello Swift")
    }
}
```

该例定义了一个名叫 Hello 的类，其有一个成员函数叫 sayHello，如果想调用该类及相应函数，则用法如下：

```
var h = Hello()
h.sayHello()
```

类的继承的写法也非常简单，这一点继承了 C++语言的优秀传统，如果我们想定义一个名叫 Hi 的类继承自 Hello，则写法如下：

```
class Hi:Hello{
}
```

如果后期我们还想为某类添加功能，有两种方法，第一是直接修改类的源码添加功能，第二是为该类写扩展功能，下面我们来着重介绍第二种，如果我们想为 Hello 类再添加一个 sayHi 的方法，则代码如下所示：

```
extension Hello{
    func sayHi(){
        println("Hi Swift")
```

```
    }
}
```

其中 extension 关键字表示要扩展已经存在的类的功能，如果我们想扩展系统或者第三方的某个类的功能而得不到其源码时，采用这种方式将会是一个绝佳的选择。如下所示：

```
extension String{
    func printSelf(){
        println(self)
    }
}
var str = "Hello Swift"
str.printSelf()
```

我们通过这种方式扩展了系统的 String 类，为她增加了一个 printSelf 的方法，在使用时可直接调用目标对象的 printSelf 方法，非常方便。

关于面向对象的更多特性，请看本书正文。

其实 extension 还有另一个用途，那就是模拟命名空间，请看下一个主题。

命名空间

在 Swift 语言中并没有专门的命名空间的关键字，但是可以模拟命名空间这个面向对象的特性，如下所示：

```
//定义命名空间ime
class ime{
}

//在ime命名空间下定义Hello类
extension ime{
    class Hello{
        func sayHello(){
            println("Hello Swift")
        }
    }
}
```

使用该类及相关方法的代码如下所示：

```
var h = ime.Hello()
h.sayHello()
```

有没让您眼前一亮的感觉呢？

结尾

就写到这里吧，想必大家已经对 Swift 有了一个初步的了解，而且迫不及待的想开始学习了，努力吧，少年，你就是未来的太阳！

目录

第 1 章　欢迎了解 Swift

前言

关于 Swift

Swift 是一款为 iOS 和 OS X 应用编程设计的全新编程语言。它集 C 语言和 Objective-C 编程语言的优点与一身，不受 C 语言的一些兼容性约束。Swift 采用安全编程模式，还添加了更为现代的元素，使编程更为简洁、灵活，也更有趣。界面则基于备受人们喜爱的 Cocoa 和 Cocoa Touch 框架，展示了软件开发的新方向。

Swift 已有多年发展史。 Apple 基于现有的编译器、调试器和框架作为基础架构，为 Swift 奠定了良好的基础。通过自动引用计数 (Automatic Reference Counting (ARC))来简化内存管理程序。在 Foundation 和 Cocoa 框架坚实的基础上，我们的堆栈框架更为现代化和标准化。objective-C 编程语言逐渐发展了支持块、collection literals 和模块的功能，可以使现代语言的框架无需中断即可使用。通过这些基础工作，我们才可以在 Apple 软件开发中引入新的编程语言。

Objective-C 语言的开发人员会感到 Swift 的似曾相识。Swift 采用了 Objective-C 编程语言的命名参数和其功能强大的动态对象模型。提供了通过 Objective-C 编码对现有 Cocoa 框架和 mix-and-match 互操作性进行无缝访问的功能。在此基础上，Swift 引入了许多新的特征，并且将编程语言的程序和部分面向对象进行了统一。

对新的程序员来说，Swift 是一种友好型的程序设计语言。Swift 是首款具有工业级品质的系统编程语言，却又像脚本语言一样具有表现力和趣味性。它支持 playground，还有一个新特征是它允许程序员体验 Swift 代码功能并即刻看到结果，省去了构建和运行一个应用程序的麻烦。

Swift 结合了现代编程语言思维以及 Apple 工程文化的智慧。编译器是根据性能优化的，而编程语言是为开发优化的，无需相互折中。Swift 可以从简单的“hello，world”语言编程学起，然后过渡到整个操作系统。所有这些使得 Swift 成为软件开发者和 Apple 公司未来绝佳的投资选择。

Swift 是一种很好的 iOS 和 OS X 操作系统语言编程方式，它将继续改进，推进新特征和新功能的引入。我们对 Shift 的目标充满期待。我们迫不及待希望你能利用它来创造出什么。

Shift 之旅

按照惯例，新语言学习应从在屏幕上打印“Hello，world”字样开始。在 Shift 语言中，这可以写为简单的一行：

```
println("Hello, world")
```

如果你写过 C 语言或 Objective-C 的代码，那么你对这个句法并不陌生——在 Swift 中，这就代表了一个完整的程序。你无需为诸如输入输出和字符串处理建立一个专门的功能库。全球范围内的代码就是用于是编程的入口，因此你无需编写一个主函数。在每个语句的末尾，你也无需加分号。

这个入门将会为你提供足够的信息，教你编写 Swift 代码以及完成多项编程任务。不用担心自己会有不懂的东西，没有解释清楚的部分会在本书后续中详细阐述。

> 注
>
> 为获得更好的体验效果，可以在 Xcode 的 playground 中打开本章。Playground 允许你编辑代码，且结果会立即显现。

1 简单值

使用 let 声明一个常量，var 声明一个变量。常量值在编译时无需指定，但至少要赋予它一个确切的值。这意味着你可以用常量来命名一个值，只需确定一次就可以在不同的场合使用。

```
var myVariable = 42
myVariable = 50
let myConstant = 42
```

常量和变量的类型必须和赋值时的类型相同。因此你并不需要明确地写下其类型。当创建常量或变量时，提供一个值，并由编译器推断其类型。在上述的例子中，由于 myVariable 的初始值是一个整数，因此编译器推断它是一个整数类型。

如果初始值没有提供足够的信息（或是没有初始值），可以在变量后写出此变量的具体类型，并用冒号隔开。

```
let implicitInteger = 70
let implicitDouble = 70.0
let explicitDouble: Double = 70
```

实验

创立一个常量，确切类型为 Float,值为 4。

值不会隐式转换为另一种类型。如果你需要将一个值转换到不同类型，明确的制作一个所需类型的实例

```
let label = "The width is "
let width = 94
let widthLabel = label + String(width)
```

实验

尝试去掉最后一行 String 转换。 你会得到什么错误？

还有一个更简单的方法将这些值纳入到字符串中： 在括弧中写上值，并且在括弧线中加入反斜杠。例如：

```
let apples = 3

let oranges = 5

let appleSummary = "I have \(apples) apples."

let fruitSummary = "I have \(apples + oranges) pieces of fruit."
```

实验

\()则用来包含字符串中的浮点计算和问候语中的人名。

创建数组和字典使用方括号（[]），并通过在方括号内写索引或键来访问其元素。

```
var shoppingList = ["catfish", "water", "tulips", "blue paint"]

shoppingList[1] = "bottle of water"

var occupations = [

"Malcolm": "Captain",

"Kaylee": "Mechanic",

]

occupations["Jayne"] = "Public Relations"
```

使用初始化语法来创建一个空的数组或字典。

```
let emptyArray = String[]()

let emptyDictionary = Dictionary<String, Float>()
```

如果可以推断类型信息，你可以写一个空的数组为[ ]，一个空的词典为 [:] ——例如，你为一个变量设置新值或为一个函数确立参数。

```
shoppingList = [] // Went shopping and bought everything.
```

2 控制流

使用 if 和 switch 来编写条件句，在条件和规定的循环变量前后使用 for-in 和括弧。

```
let individualScores = [75, 43, 103, 87, 12]
var teamScore = 0
for score in individualScores {
if score > 50 {
teamScore += 3
} else {
teamScore += 1
}
}
teamScore
```

在 if 语句中，条件必须是布尔式表达式，这意味着 if score {...}之类的代码是错误的，而不是一个与 0 进行的隐含比较。你可以一起使用 if 和 let 以避免值的缺失。这些值是可选择的。一个可选值可以包含一个值或包含 nil 来表示该值丢失。在一个值的类型后面写一个问号 (?)代表该值是可选的。

```
var optionalString: String? = "Hello"
optionalString == nil
var optionalName: String? = "John Appleseed"
var greeting = "Hello!"
if let name = optionalName {
greeting = "Hello, \(name)"
}
```

实验

将 optonalName 改为 nil。在问候时你会得到什么？如将 optionalName 改为 nil，可以添加一个 else 子句，设置一个不同的值。

若是可选值为 nil，那么条件是 false，可略过大括号中的代码。否则的话，可选值未包装并赋值于一个常量，该未包装值的变量会转到代码块中。

Switch 支持所有数据以及不同操作系统间的比较——不仅仅限于整数和相等测试。

```
let vegetable = "red pepper"
switch vegetable {
case "celery":
let vegetableComment = "Add some raisins and make ants on a log."
case "cucumber", "watercress":
let vegetableComment = "That would make a good tea sandwich."
case let x where x.hasSuffix("pepper"):
let vegetableComment = "Is it a spicy \(x)?"
default:
vegetableComment = "Everything tastes good in soup."
```

实验

请尝试删除 default。你得到了什么错误？

在执行单值判断的代码匹配后，此程序从 switch 中退出。执行不会顺延到下一个判断中，因此不需要在每个判断代码的结束时退出 switch。

你可以使用 for-in 来迭代字典中的每个元素，同时为每个键值对提供一对名称。

```
let interestingNumbers = [
"Prime": [2, 3, 5, 7, 11, 13],
"Fibonacci": [1, 1, 2, 3, 5, 8],
"Square": [1, 4, 9, 16, 25],
]
var largest = 0
for (kind, numbers) in interestingNumbers {
for number in numbers {
if number > largest {
largest = number
}
}
Largest
```

实验

添加另一个变量来跟踪记录哪一类的数字最大以及数字最大值是什么。

使用 while 来重复执行代码块，直到条件发生改变。循环的条件可以放置到末端，以保证循环至少运行一次。

```
var n = 2
while n < 100 {
n = n * 2
}
n
var m = 2
```

```
do {
m = m * 2
while m < 100
```

你可以在循环中保持一个索引——或者通过使用“..”来表示索引的范围或者声明一个明确的初始值、条件和增量。这两个循环作用相同：

```
var firstForLoop = 0
for i in 0..3 {
firstForLoop += i
}
firstForLoop
var secondForLoop = 0
for var i = 0; i < 3; ++i {
secondForLoop += 1
secondForLoop
```

使用“..”表示忽略最高值的范围，使用“...”表示的范围则包括两个值。

3 函数和闭包

使用 func 来声明一个函数。通过用括号中的参数列表加上它的名字调用一个函数。使用 ->隔开参数名称和函数返回值类型。

```
func greet(name: String, day: String) -> String {
return "Hello \(name), today is \(day)."
}
greet("Bob", "Tuesday")
```

实验

删除 day 的参数。添加一个包含今天午餐选择的参数。

使用元组（tuple）从一个函数中返回多个值。

```
func getGasPrices() -> (Double, Double, Double) {
return (3.59, 3.69, 3.79)
}
getGasPrices()
```

函数也可以可变参数个数，并将他们收集到一个数组中。

```
func sumOf(numbers: Int...) -> Int {
var sum = 0
for number in numbers {
sum += number
}
return sum
}
sumOf()
sumOf(42, 597, 12)
```

实验

编写一个函数,计算其参数的平均值。

函数可以嵌套。嵌套函数可以访问在外部函数中声明的变量。你可以使用嵌套函数来组织代码,避免过长或者很复杂的函数。

```
func returnFifteen() -> Int {
var y = 10
func add() {
y += 5
}
add()
return y
}
returnFifteen()
```

函数是一级类型的。这意味着函数可以返回另一个函数作为它的值。

```
func makeIncrementer() -> (Int -> Int) {
func addOne(number: Int) -> Int {
return 1 + number
}
return addOne
}
var increment = makeIncrementer()
increment(7)
```

一个函数可以把另一个函数作为它的一个参数。

```
func hasAnyMatches(list: Int[], condition: Int -> Bool) -> Bool {
for item in list {
if condition(item) {
```

```
return true
}
}
return false
}
func lessThanTen(number: Int) -> Bool {
return number < 10
numbers = [20, 19, 7, 12]
hasAnyMatches(numbers, lessThanTen)
```

函数实际上是闭包的一种特殊情况。你可以通过大括号中的代码编写一个没有名字的闭包。使用 in 来区分参数和其主体的返回值类型。

```
numbers.map({
(number: Int) -> Int in
let result = 3 * number
return result
})
```

实验

重写闭包使所有奇数返回零值。

编写闭包时有几种简洁可选项。当一个闭包的类型是已知,比如代表回调,你可以省略的参数类型或者返回值,或两者都可以省略。单个语句隐式闭包可以返回他们唯一语句的声明值。

```
numbers.map({ number in 3 * number })
```

你可以引用一个参数的数字而不是名字，这种方法对于很短的闭包是特别有用的。一个闭包通过传递

括号后的其最后一个参数给函数来作为返回值。

```
sort([1, 5, 3, 12, 2]) { $0 > $1 }
```

4 对象和类

使用 class 名称后面的 class 可以创建一个类。类中一个属性的声明则是和常量和变量属性声明采用同样的方式，除非是在类的上下文中。同样，方法和函数声明采用相同的书写方式。

```
class Shape {
var numberOfSides = 0
func simpleDescription() -> String {
return "A shape with \(numberOfSides) sides."
}
}
```

实验

用 let 添加一个常量属性，并添加另一个方法接受一个参数。

通过在类名称后加小括号的方式创建一个类的实例。使用点语法访问实例的属性和方法。

```
var shape = Shape()
shape.numberOfSides = 7
var shapeDescription = shape.simpleDescription()
```

这种版本的 Shape 类会丢失一些重要的数据。在创建一个实例时，初始化器会设置类。使用 init 创建一个。

```
class NamedShape {
var numberOfSides: Int = 0
```

```
var name: String

init(name: String) {

self.name = name

}

func simpleDescription() -> String {

return "A shape with \(numberOfSides) sides."
```

注，对初始化器而言，self 用于区分 name 属性和 name 参数初始化。当你创建该类的实例时，初始化器参数设定类似于函数调令。 Every property needs a value assigned—either in its declaration (as with numberOfSides) or in the initializer (as with name).每个参数都需要赋予一个值——无论是在声明里还是在初始化器里。

如果需要执行一些需要对象销毁时的清理工作，可以使用 deinit 创建一个析构器。

子类包括其父类的名称，用冒号隔开。在继承标准根类时不需要任何要求,因此你可以根据需要包括或省略父类。

对于父类，可以通过标记 override 的方式重写父类的实现，若没有 override，则其会被编译器检测是错误的。编译器也检测父类中那些实际上没有被重写的任何方法。

```
class Square: NamedShape {

var sideLength: Double

init(sideLength: Double, name: String) {

self.sideLength = sideLength

super.init(name: name)

numberOfSides = 4

}

func area() -> Double {

return sideLength * sideLength
```

```
override func simpleDescription() -> String {
return "A square with sides of length \(sideLength)."
test = Square(sideLength: 5.2, name: "my test square")
area()
simpleDescription()
```

实验

编写另一个 NameShape 的子类，即 Circle，接受一个半径和一个名称作为初始化器中的参数。在 Circle 类中实现 area 和 describe 方法。

除了存储一些简单的属性，属性还有 getter 和 setter。

```
class EquilateralTriangle: NamedShape {
var sideLength: Double = 0.0
init(sideLength: Double, name: String) {
self.sideLength = sideLength
super.init(name: name)
numberOfSides = 3
}
var perimeter: Double {
get {
return 3.0 * sideLength
set {
sideLength = newValue / 3.0
```

```
override func simpleDescription() -> String {
return "An equilateral triagle with sides of length \(sideLength)."
triangle = EquilateralTriangle(sideLength: 3.1, name: "a triangle")
triangle.perimeter
triangle.perimeter = 9.9
triangle.sideLength
```

在 perimeter 的 setter 中，新值具有一个隐式名，即 newValue。在 set 后，你可以在括号内提供一个明确的名称。

请注，EquilateralTriangle 的初始化器有三个不同的步骤：

1. 设置子类声明属性的值。
2. 调用父类的初始化器。
3. 更改父类中定义的属性值。任何额外的设置工作,如使用方法、getter 或 setter 也可以在这里完成。

如果您不需要计算该属性，但仍然需要提供在运行之前和设置新值后的代码，可以使用 willSet 和 didSet。例如，下面的类要确保其三角的边长和正方形的边长一样。

```
class TriangleAndSquare {
var triangle: EquilateralTriangle {
willSet {
square.sideLength = newValue.sideLength
}
}
var square: Square {
willSet {
triangle.sideLength = newValue.sideLength
```

```
(size: Double, name: String) {
square = Square(sideLength: size, name: name)
triangle = EquilateralTriangle(sideLength: size, name: name)

triangleAndSquare = TriangleAndSquare(size: 10, name: "another test shape")
triangleAndSquare.square.sideLength
triangleAndSquare.triangle.sideLength
triangleAndSquare.square = Square(sideLength: 50, name: "larger square")
triangleAndSquare.triangle.sideLength
```

类的方法和函数相比有一个重要的区别。函数参数名只能在函数中使用，但当你调用方法时，方法的参数名（除了第一个参数）也可以被使用。在缺失的情况下，一个方法有一个相同名称的参数，调用时它就是参数本身。在方法内部使用时，可以指定第二个名字。

```
class Counter {
var count: Int = 0
func incrementBy(amount: Int, numberOfTimes times: Int) {
count += amount * times
}
}
var counter = Counter()
counter.incrementBy(2, numberOfTimes: 7)
```

当与可选值一起工作时，你可以在操作前写“？”，类似于方法、属性和下标。如果在“？”之前该值已

经是 nil，所有“?”之后的东西都会自动忽略，整个表达式的值就是 nil。否则，可选值是未包装的，所有“？”之后的均视为未包装值。在这两种情况下,整个表达式的值是一个可选值。

```
let optionalSquare: Square? = Square(sideLength: 2.5, name: "optional square")

let sideLength = optionalSquare?.sideLength
```

5 枚举和结构

使用 enum 来创建枚举。像类和所有其他命名类型，枚举可以有与之关联的方法。

```
enum Rank: Int {

case Ace = 1

case Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten

case Jack, Queen, King

func simpleDescription() -> String {

switch self {

case .Ace:

return "ace"

case .Jack:

return "jack"

case .Queen:

return "queen"

case .King:

return "king"

default:

return String(self.toRaw())
```

```
}

ace = Rank.Ace

aceRawValue = ace.toRaw()
```

实验

编写一个函数，通过比较它们的原始值，比较两个 Rank 值。

在上面的例子中，枚举的原始值类型是 int，所以你只需要指定第一个原始值。按顺序指定原始值的后续部分。你还可以使用字符串或浮点数作为枚举的原始类型。

使用 toRaw 和 fromRaw 函数对原始值和枚举值进行转换。

```
if let convertedRank = Rank.fromRaw(3) {

let threeDescription = convertedRank.simpleDescription()

}
```

枚举的成员值是实际值，不仅仅是原始值的另一种书写方式。事实上，在情况下，你不会提供一个没有意义的原始值。

```
enum Suit {

case Spades, Hearts, Diamonds, Clubs

func simpleDescription() -> String {

switch self {

case .Spades:

return "spades"

case .Hearts:

return "hearts"
```

```
case .Diamonds:
return "diamonds"
case .Clubs:
return "clubs"
}
hearts = Suit.Hearts
heartsDescription = hearts.simpleDescription()
```

实验

添加一个 color 方法到 Suit，并在 spades 和 clubs 时返回“black”,并给 hearts 和 diamonds 返回“red”。

注，枚举的 Hearts 成员的方法可以参考上面两种方式：当赋予一个值到 hearts 常量时,由于常量没有明确的类型，因此枚举成员 Suit.Hearts 会引用其全名。在 switch 中，枚举通过缩写形式引用.Hearts，因为 self 的值是一个已知套系。当值的类型是已知的时，你可以在任何时候使用其缩写形式。

使用 struct 来创建一个结构体。结构体支持许多与类相同的行为，包括方法和初始化器。其中结构体和类之间最重要的区别在于结构体代码之间的传递总是用复制（值传递），但类是通过引用传递。

```
struct Card {
var rank: Rank
var suit: Suit
func simpleDescription() -> String {
return "The \(rank.simpleDescription()) of \(suit.simpleDescription())"
}
```

```
}

let threeOfSpades = Card(rank: .Three, suit: .Spades)

let threeOfSpadesDescription = threeOfSpades.simpleDescription()
```

实验

添加一个方法到 Card 类来创建一整桌的纸牌，每个纸牌都有 rank 和 suit 的组合。

枚举成员的实例可以有与该实例相关联的值。同样的枚举成员的实例可以有与之关联的不同的值。再创建实例时，你可以提供与实例关联值。关联值和原始值是不同的：枚举成员的原始值与其实例相同，当定义枚举时你需要提供原始值。

例如：假设以需要从服务器中获得日出和日落时间为例。服务器可以回应其信息或者反馈一些错误信息。

```
enum ServerResponse {

case Result(String, String)

case Error(String)

}

let success = ServerResponse.Result("6:00 am", "8:09 pm")

let failure = ServerResponse.Error("Out of cheese.")

switch success {

let .Result(sunrise, sunset):

serverResponse = "Sunrise is at \(sunrise) and sunset is at \(sunset)."

let .Error(error):

serverResponse = "Failure... \(error)"
```

实验

给 ServerResponse 和 switch 添加第三种选择情况。

注，日出和日落时间是通过来源于 ServerResponse 的部分匹配值来选择的。

5.1 协议及扩展

使用 protocol 来声明一个协议。

```
protocol ExampleProtocol {
var simpleDescription: String { get }
mutating func adjust()
}
```

类、枚举和结构体都可以采用协议。

```
class SimpleClass: ExampleProtocol {
var simpleDescription: String = "A very simple class."
var anotherProperty: Int = 69105
func adjust() {
simpleDescription += " Now 100% adjusted."
}
}
var a = SimpleClass()
a.adjust()
```

```
aDescription = a.simpleDescription

SimpleStructure: ExampleProtocol {

var simpleDescription: String = "A simple structure"

mutating func adjust() {

simpleDescription += " (adjusted)"

= SimpleStructure()

adjust()

bDescription = b.simpleDescription
```

实验

编写符合该协议的枚举。

注，在更改结构体时，需使用 mutating 关键字来描述 SimpleStructure 以标记实现协议的方法。SimpleClass 的声明并不需要它的任何方法来标记 mutating，因为在一个类中的方法可以随时此类。可以使用 extension 为一个现有的类型添加函数，比如新的方法和计算属性。你可以使用一种 extension 对一个已经声明的类型添加其协议一致性，或者甚至可以从程序库或框架中引进的一种类型。

```
extension Int: ExampleProtocol {

var simpleDescription: String {

return "The number \(self)"

}

mutating func adjust() {

self += 42

}
```

```
}

7.simpleDescription
```

实验

为 Double 类型编写一个拓展,添加一个 absoluteValue 属性。

你可以使用一个协议名字，就像任何其他命名类型——例如，创建一个对象集合,其拥有不同的类型,但所有类型均符合一个协议。当你和一个协议类型的值一起工作时，使用协议定义以外的方法并不可行。

```
let protocolValue: ExampleProtocol = a

protocolValue.simpleDescription

// protocolValue.anotherProperty // Uncomment to see the error
```

尽管变量 protocolValue 有一个 SimpleClass 运行时间模式,编译器将它的类型视为 ExampleProtocol 给定协议类型。这表示你不能意外访问类在协议一致之外进行的方法或属性。

5.2 泛型

在尖括号内写一个名字来创建一个泛型函数或类型。

```
func repeat<ItemType>(item: ItemType, times: Int) -> ItemType[] {

var result = ItemType[]()

for i in 0..times {

result += item

}

return result

}

repeat("knock", 4)
```

你可以创建函数和方法的泛型形式,以及类、枚举和结构体的泛型形式。

```
// Reimplement the Swift standard library's optional type
enum OptionalValue<T> {
case None
case Some(T)
}
var possibleInteger: OptionalValue<Int> = .None
possibleInteger = .Some(100)
```

在泛型名称后面使用一个需求列表——例如,要限定实现一个协议的类型,需要相同的两种类型,或者需要有一个类有一个特定的父类。

```
func anyCommonElements <T, U where T: Sequence, U: Sequence, T.GeneratorType.Element: Equatable,
T.GeneratorType.Element == U.GeneratorType.Element> (lhs: T, rhs: U) -> Bool {
for lhsItem in lhs {
for rhsItem in rhs {
if lhsItem == rhsItem {
return true
}
}
}
return false
anyCommonElements([1, 2, 3], [3])
```

实验

修改 anyCommonElements 函数来创建一个函数，返回一个数组,其中任何两个序列须有共有元素。

在简单的情况下,你可以省略或简写冒号后面的协议或类名。编写 <T：Equatabe>在 T: Equatable>的情况下与 writing <T 是完全相同的。 Equatable>.

第 2 章　语言指南

1 基础知识

Swift 是为 iOS 和 OSX 应用程序开发设计的一门新编程语言。然而，Swift 的开发体验与 C 语言和 Objective-C 编程有很多相似之处。

Swift 提供了 C 语言和 Objective-C 编程中所有基础类型，包括整数的 Int,表示浮点值的 Double 和 Float，表示布尔数值的 Bool 以及表示纯文本数据的 String。Swift 还为 Array 和 Dictionary 两个主要集合类型提供了强大版本,正如（集合类型）中所述。

像 C 语言一样,Swift 采用变量存储数据，并通过识别名来引用变量值。Swift 还扩充使用其值不被改变的变量。这些被称为常量,比 C 语言中的常量更强大。当在整个 Swift 操作中不需要改变数值的数据时，使用常量可使代码更安全，作用更明显。

除了一些常见类型外,Swift 还引入了 Objective-C 语言中所没有高级类型。其中包括元组（tuple）,能帮你创建并传递组值。元组可以从函数返回多值来作为单个复合值。

Swift 还引入了可选值（optional types）,处理一些不存在的数值。可选值可以显示为“存在一个值 X”,或者为“不存在任何值”。可选值类似于 Objective-C 中使用指针 nil,但在 Swift 操作中，他们可以为任何类型使用,并不仅仅是类。可选值比 Objective-C 中带有指针的 nil 更安全、语义更生动,并在 Swift 诸多强大功能中得到了深入的应用。

可选值是 Swift 作为一种安全语言类型的一个具体体现。Swift 能帮助你清楚地了解代码可以处理的数据类型。如果你的部分代码希望是 String 类型,类型安全的特性会阻止你错误地把 Int 类型传递过去。如此，在开发过程中能尽早发现并修改错误。

1.1 常量和变量

常量和变量将一个名称(如 maximumNumberOfLoginAttempts 或 welcomeMessage)与特定类型的值(比如数字 10 或字符串“Hello”) 相关联。常量的值一旦设置则不能改变,而变量值可以根据需要设置为不同的值。

1.2 常量和变量的声明

常量和变量必须在使用之前声明。在使用中，用 let 关键词声明来常量，用 ver 关键词来声明变量。下面是一个示例说明了常量和变量如何来跟踪用户登录尝试次数的:

```
let maximumNumberOfLoginAttempts = 10

var currentLoginAttempt = 0
```

此代码可以解读为：

“声明一个 maximumNumberOfLoginAttempts,并赋值为 10。然后,声明一个 currentLoginAttempt 的新变量,并赋予其初始值为 0。”

在这个例子中,允许登录尝试次数的最大值声明为一个常量,因为最大值是不变的。即时登录次数声明为一个变量,因为这个值是每次登录尝试失败次数的累加。

你可以在一行中声明多个常量或多个变量,用逗号分隔：

```
var x = 0.0, y = 0.0, z = 0.0
```

注

如果在代码中存储的值不会改变,则用 let 关键字将它声明为一个常量。只使用变量来存储会更改的值。

1.3 类型注释

在声明常量或变量时，可以提供一个类型注释来注明该常量或变量可以存储值的类型。编写一个类型注释，可以在常量或变量名后面加上一个冒号，接着加一个空格,再加上使用类型的名称。这个示例声明了一个变量为 welcomeMessage，类型注释为字符串 String:

```
var welcomeMessage: String
```

在声明中的冒号意味着“是…类型”,所以上面的代码可以解读为: “声明一个 welcomeMessage 变量，类型是 String。”

“String 类型”的意思是“可以存储任何 String 值。”它的意思是可以存储的“东西的类型”(或“东西的种类”)。

wlecomeMessage 变量可以无误地设置为任何字符串类型的值:

```
welcomeMessage = "Hello"
```

注

在实践中很少需要编写类型注释。如果你在定义常量或变量时提供了初始值,Swift 几乎都可以根据这些初始值推断出用于常量或变量应使用的类型,正如类型安全和类型推断中描述的那样。在上述 welcomeMessage 示例中,因为没有赋予其初始值,所以 welcomeMessage 变量的类型用一个类型注释指明,而不是用一个初始值来推断。

1.4 常量和变量的命名

可以选用包括 Unicode 字符在内的任何字符作为常量和变量的名称:

```
let π = 3.14159
let 你好 = "你好世界"
let = "dogcow"
```

常量名和变量名中不应包括数学符号、箭头、私有（或无效的）Unicode 代码点，或线以及绘图框等符号。不能以数字开头，数字可以包含在名称中。

一旦声明了某一类型的常量或变量，不能再使用同一名称对其重新声明，也不能对其更改以存储不同类型的值。不能将常量转换为变量，或将变量转换为常量。

注

如果需要将常量或变量同一命名为 Swift 保留字，可以通过在关键字周围加上反勾号（`）实现。然而，你应尽量避免使用保留字作为常量或变量的名称。

可将已有变量值更改为另一个相同类型的值。

在该示例中，friendlyWelcome 值由“Hello!”更改为“Bonjour!”:

```
var friendlyWelcome = "Hello!"
friendlyWelcome = "Bonjour!"
// friendlyWelcome is now "Bonjour!"
```

与变量不同，常量一旦设定后就无法对其进行更改。尝试更改的话，编译代码会提示该错误:

```
let languageName = "Swift"
languageName = "Swift++"
```

```
// this is a compile-time error - languageName cannot be changed
```

1.5 常量和变量的输出

可以通过 println 函数显示常量或变量的当前值：

```
println(friendlyWelcome)
// prints "Bonjour!"
```

println 是一个全局函数，用来输出一个值，接着是换行符，来输出适当的输出值。在 Xcode 开发环境中，比如，println 会将输出内容输至 Xcode 中的“控制台”中。（另一个函数 print 执行一样的任务，只不过在数值末端不会输出换行。）

println 函数会输出你传递给它的任何 String：

```
println("This is a string")
// prints "This is a string"
```

同 Cocoa 中 NSLog 函数的操作类似， println 函数可以输出更为复杂的日志信息。此类信息可包括常量和变量的当前值。

Swift 语言采用字符串插值涵盖常量或变量名称作为较长字符串中的占位符。这使得 Swift 语言借助其常量或变量的当前值对其进行替换。将名称用括号括起来并在左括号前附上反斜杠：

```
println("The current value of friendlyWelcome is \(friendlyWelcome)")
// prints "The current value of friendlyWelcome is Bonjour!"
```

> 注
>
> 字符串插值中所有可选项已在“字符串插值”进行描述。

1.6 注释

在编码的非执行文本中添加注释，作为备注或提示。在编码时，此类注释则被 Swift 编译器忽略。

Swift 语言中的注释与 C 语言中的注释极其相似。单行注释以双斜杠（//）开始：

```
// this is a comment
```

也可以编写多行注释，用一个斜杠加上一个星号（/*）开始，以一个星号加上一个斜杠（*/）结束：

```
/* this is also a comment,
but written over multiple lines */
```

不同于 C 语言中的多行注释，Swift 语言中的多行注释可被嵌入到其他多行注释中。可通过先开始一个多行注释块，再在第一个多行注释快中开始第二个多行注释块的方式编写嵌套注释。结束时，先关闭第一个注释块，然后再关闭第二个多行注释块：

```
/* this is the start of the first multiline comment
/* this is the second, nested multiline comment */
this is the end of the first multiline comment */
```

通过嵌入式多行注释可以快速且便捷地在已注释的代码中继续添加注释，尽管代码中已包含了多行注释。

分号

不同于其他许多程序语言，Swift 言语中不需要在每一个代码的末尾处加上分号（;），当然，如果你想你也可以选择加上分号。然而，若在单行中编写多个语句，则需要加上分号：

```
let cat = "      "; println(cat)
// prints "      "
```

2 整数

整数是指像 42 和-23 一样不带小数部分的整数。整数分为有符号整数（正数、零或负数）或无符号整数（正数或零）。

Swift 语言涉及到 8 位、16 位、32 位以及 64 位的有符号整数和无符号整数形式。此类整数的命名规范与 C 语言中的命名规范相类似，8 位的无符号整数为 UInt8 类型，32 位有符号整数为 Int32 类型。与 Swift 语言中的所有类型一致，此类整数类型首字母大写。

2.1 整数边界

可以用每一整数类型的 min 和 max 属性获取其最大值和最小值：

```
let minValue = UInt8.min // minValue is equal to 0, and is of type UInt8
let maxValue = UInt8.max // maxValue is equal to 255, and is of type UInt8
```

此类属性值为适当的数字类型（比如上述示例中的 UInt8），可与同一类型的其他值一样用于表达式中。

2.2 整数

多数情况下，无需在代码中选取具体的某一整数。Swift 语言中提供了另一整数类型- Int 型，其与当前运行平台的字长度相同：

在代码中通常采用 Int 型作为整数值，除非需要一个特定字长的整数。这有助于代码的一致性和互操作性。对于 32 位运行平台，Int 型可存储-2147483648 至 2147483647 范围内的任一值，这对于大部分的整数而言已经足够了。

2.3 UInt

Swift 语言还中提供了一种无符号整数类型- UInt 型，其字长与当前运行平台的字长一致：

> 注
>
> 仅在需与运行平台字长相同的某一特定无符号整数类型的情况下才可采用 UInt 型。否则，应使用 Int 型，即使存储数值为非负数。对于整数值，连续采用 Int 型可增强代码的互操作性，避免了不同数字类型间的转换，且与整数型推断相匹配，正如类型安全和类型推断中所述。

2.4 浮点数

浮点数是指像 3.14159、 0.1 以及 -273.15 一样带有小数部分的数字。

比起整型数字，浮点数可表示更广范围的数值，可以存储比在 Int 型中更大或更小的数值。Swift 提供两个有符浮数类型：

注

Double 至少精确到 15 位十进制数，Float 则为 6 位十进制数。可根据代码中所需处理的数值类型和范围，选用合适的浮点数字。

2.5 类型安全和类型推断

Shift 是一种类型安全语言。S 类型安全语言可以帮助你弄清楚代码使用值的类型。如果你的部分代码需要一个字符串 String,那么你不可以通过错误的传递一个 Int 类型。

Swift 语言符合类型安全，编码时会进行类型检查并标记出任何出错的不匹配类型。这使你能够在开发过程中尽早发现并修改错误。

可在不同类型的数值运行时，通过类型检查避免出现错误。但是，这并不意味着你需要指定所声明的常量以及变量的类型。未指定所需的数值类型时，Swift 语言可通过类型推断确定合适类型的数值。通过类型推断，编码器在编码时仅通过检测所给出的数值，便可自动推断出特定表达式的类型。

由于类型推断这一功能的存在，比起 C 语言或者 Objective-C 语言，Swift 语言无需类型声明。对于常量和变量，仍需明确其类型，但是多数类型已给出。

当声明常量或变量并给出其初始值时，类型推断的作用则特别突出。当定义常量或变量时，可通过向其设定字面值（或常量）完成类型的推断。（字面值是指直接出现在源代码中的数值，比如以下示例中的 42 和 3.14159。）

例如，在未说明类型的前提下，给新常量设定字面值 42，Swift 语言则推断你想要获得一个 Int 型常量，因为采用了整数值对其进行初始化：

```
let meaningOfLife = 42

// meaningOfLife is inferred to be of type Int
```

同样，未指明浮点型常量的类型时，Swift 语言则推断为 Double 类型：

```
let pi = 3.14159

// pi is inferred to be of type Double
```

在推断浮点型数字的类型时，Swift 语言通常选择 Double 类型（而非 Float 类型）

当在同一表达式中将整型常量和浮点常量组合在一起时，可根据以下内容推断出一个 Double 类型：

```
let anotherPi = 3 + 0.14159

// anotherPi is also inferred to be of type Double
```

字面值 3 自身不存在显式类型，因此，可从浮点常量中推断出合适的输出类型-Double 类型。

2.6 数值常量

整型常量可以写成：

所有这些整型常量都一个十进制值 17：

```
let decimalInteger = 17

let binaryInteger = 0b10001 // 17 in binary notation

let octalInteger = 0o21 // 17 in octal notation

let hexadecimalInteger = 0x11 // 17 in hexadecimal notation
```

浮点常量可以是十进制（无前缀）或十六进制（前缀为 0x）。小数点前后须有数字（或十六进制数）。可采用可选指数，其中，用大写或小写 e 表示十进制浮点数，用大写或小写 p 表示十六进制浮点数。

对于带有 exp 指数的十进制数，实际数值是基数乘以 10^{exp}：

对于带有 exp 指数的十六进制数，实际数值是基数乘以 2^{exp}：

所有这些浮点常量都一个十进制数 12.1875：

```
let decimalDouble = 12.1875

let exponentDouble = 1.21875e1

let hexadecimalDouble = 0xC.3p0
```

数值常量可包括额外的格式以方便阅读。可向整数和浮点数中添加 0，也可加入下划线使阅读更方便。格式的类型对常量值没有任何影响：

```
let paddedDouble = 000123.456
```

```
let oneMillion = 1_000_000

let justOverOneMillion = 1_000_000.000_000_1
```

2.7 数值类型转换

2.8 整数转换

存储于整型常量或变量的数值范围根据每一数值类型而存在差异。一个 Int8 型常量或变量中，可存储 -128 至 127 范围内的数值；一个 UInt8 型常量或变量中，可存储 0 至 255 范围内的数值。编码时，对于某一指定整数类型对应的常量或变量中无法匹配的数字，编译器会提示错误：

```
let cannotBeNegative: UInt8 = -1

// UInt8 cannot store negative numbers, and so this will report an error

let tooBig: Int8 = Int8.max + 1

// Int8 cannot store a number larger than its maximum value,

// and so this will also report an error
```

由于每一数字类型可存储不同范围的数值，可根据不同情况在基础类型上转换数值类型。可通过 opt-in 防止隐藏的转换错误，并帮助代码中的类型转换中明确其目的。

为了将一个特定的数字类型转换到另一种，你要初始化现有值所需类型的一个新的数值。在下面的例子中，常量 twoThousand 属于 UInt16 类型，而常量 one 是 UInt8 类型。它们不能被直接相加，因为它们属于不同的类型。相反，该示例调用 UInt16（one）来创建一个初始值为 one 的 UInt16 新类型，并且使用这个值来代替原来的值：

```
let twoThousand: UInt16 = 2_000

let one: UInt8 = 1

let twoThousandAndOne = twoThousand + UInt16(one)
```

由于相加的两边均为 UInt16 类型，因而是允许相加的。输出常量（twoThousandAndOne）推断为 UInt16 类型，因为它是两个 UInt16 值的总和。

SomeType(ofInitialValue)是调用并传递 Swift 类型初始值的一种默认方式。实际上， UInt16 有一个接受 UINT8 值的初始化器，因此这个初始化器用于从现有的 UINT8 创建一个新的 Uint16 的变量 。你不能传入任何类型的数值，但是——它必须是 UInt16 类型初始化器能接受的一个类型。为初始化器提供扩展现有的类型并接受新的类型（包括你自己的类型定义），这些在扩展中有详细说明。

2.9 整数和浮点转换

整数和浮点数类型之间的转换必须明确：

```
let three = 3
let pointOneFourOneFiveNine = 0.14159
let pi = Double(three) + pointOneFourOneFiveNine
// pi equals 3.14159, and is inferred to be of type Double
```

这里，常量 three 的值用于创建 Double 类型的新值，从而使相加的两端是相同类型的。如果没有这个转换，那么将不会允许加法操作。

反过来，浮点到整数的转换也是如此，整数类型可以用 Double 或 Float 值进行初始化：

```
let integerPi = Int(pi)
// integerPi equals 3, and is inferred to be of type Int
```

当用于初始化一个新的整数值时，这种方式的浮点值总是被缩短。这意味着，4.75 会变为 4，-3.9 变为-3。

注

结合数值常量和变量的转换规则与数字类型常值的转换规则不同。常值 3 可直接与常值 0.14159 相加，因为数字常值本身没有明确的类型。其类型仅在它们由编译器计算出时被推断出来。

2.10 类型别名

类型别名为现有类型定义一个替代名称。你可以使用 typealias 关键字来定义类型别名。

当你想用根据上下文指代一个比较合适的名称引用现有类型时，可以用到类型别名，如处理一个来自于外部的指定长度的数据时：

```
typealias AudioSample = UInt16
```

一旦你定义了一个类型别名，你可以在使用原来名称的任何地方使用别名：

```
var maxAmplitudeFound = AudioSample.min
// maxAmplitudeFound is now 0
```

这里，AudoSample 是被定义为 Uint16 的一个别名。因为它是一个别名，调用 AudioSample.min 实际上是调用 UInt16.min，这为 maxAmplitudeFound 变量赋予初始值 0。

2.11布尔值

Shift 具有一种基本的布尔类型，称为 Bool。布尔值被称为逻辑类型，因为他们的可选值只能是 true 或 false。Shift 提供了两个布尔常量值，即 true 和 false：

```
let orangesAreOrange = true
let turnipsAreDelicious = false
```

因为采用布尔类型的常值初始化，所以 orangesAreOrange 和 turnipsAreDelicous 的类型被推断为 Bool。就像上面的 Int 和 Double 一样，你在将他们设置为 true 或 false 的时候并不需要声明常量或变量为 Bool。当它用已知类型初始化其他值的常量或变量时，类型推断有助于使 Swift 代码更简洁、更易读。

当你使用如 if 语句的条件语句时，布尔值会特别有用：

```
if turnipsAreDelicious {
println("Mmm, tasty turnips!")
} else {
println("Eww, turnips are horrible.")
}
```

```
// prints "Eww, turnips are horrible."
```

如 if 语句的条件语句在控制流程中会有详细讲述。

转 Swift 类型安全会防止非布尔值被 Bool 代替。下面的例子描述了在编译时的报错：

```
let i = 1

if i {

// this example will not compile, and will report an error

}
```

然而，下面的替代案例是有效的：

```
let i = 1

if i == 1 {

// this example will compile successfully

}
```

i == 1 比较的结果为 Bool 类型，所以第二个例子通过了类型检查。在这样像 i==1 的比较会在基本运算符中详细讨论。

对于 Swift 语言中其他类型安全的例子，这种方法避免了意外的错误并确保代码的特定部分的目的是明确的。

3 元组

元组可以将多个数值组织成一个单一复合值。元组中的值可以是任何类型，并且彼此不必是相同的类型。

在这个例子中，（404，“Not Found”）是一个描述 HTTP 状态代码的元组。HTTP 状态代码是你请求一个网页时，Web 服务器返回的一种特殊值。如果你要求的网页不存在，的网页就会返回 404 状态代码 NotFound。

```
let http404Error = (404, "Not Found")
```

```
// http404Error is of type (Int, String), and equals (404, "Not Found")
```

（404， “NotFound”）元组由一个 Int 与和一个 String 一起为 HTTP 状态代码赋予两个单独的值：即一个数字和一个人们认知的字符串描述。它可以描述为“类型的元组（Int,String）”。

你可以从任何类型的排列中创建元组，根据自己的意愿，他们可以包含许多不同的类型。创建一种类型的元组（Int，Int，Int）或（String，Bool），或者你需要的任何其他的排列,不会受到任何限制。

你可以将一个元组分解为单独的常量或变量，然后再像通过以往方式进行访问：

```
let (statusCode, statusMessage) = http404Error
println("The status code is \(statusCode)")
// prints "The status code is 404"
println("The status message is \(statusMessage)")
// prints "The status message is Not Found"
```

如果你只需要元组的部分值，当你分解元组时，可以通过使用下划线（_）来忽略的部分元组：

```
let http200Status = (statusCode: 200, description: "OK")
```

另外，可以使用从零开始的索引号对元组中的各个元素值进行访问：

```
println("The status code is \(http200Status.statusCode)")
// prints "The status code is 200"
println("The status message is \(http200Status.description)")
// prints "The status message is OK"
```

元组在作为函数的返回值时作用比较明显。尝试检索网页的函数可能会返回（Int，String）元组类型来描述页面检索的成功或失败。通过返回由两个不同值（两个为不同的类型）组成的元组，可以比只返回单个类型的单个值提供更多的信息。欲了解更多信息，请参阅有多个不同返回值得函数。

注

元组对相关值的临时组是有用的。他们不使用于创造复杂的数据结构体。如果数据结构

体可能会持续超过一个临时的范围，那么它就是一个类或结构体构型，而不是一个元组。欲了解更多信息，请参阅类和结构。

3.1 可选类型

可以在一个值缺失的情况下使用可选类型。一个可选类型是：

这里有一个值，并且等于 x

或

这里没有值

注

optonals 的概念并在 C 语言或 Objective-C 中并不存在。但在 Objective-C 中有一个最相似的类型，即从一种方法返回 nil，否则将返回一个对象，nil 意为“缺乏一个有效的对象。然而，这仅适用于对象，它并不适用于结构体、 C 语言的基本类型或枚举数值。对于这些类型，Objective-C 的方法通常是返回一个特殊值（例如 NSNotFound），以指示不存在的值。这种方法假定该方法的调用者知道有一个特殊的值用于测试并检验。Swift 的可选类型可以帮助你指出任何值的不存在情况，而不需要特殊的常量。

这里有一个例子。Swift 中 String 类型有一个叫做 toInt 的方法，可以尝试将 String 值转换为一个 Int 值。然而，并不是每个字符串都可以转换成整数。字符串“ 123”可以转换成数字值 123，而字符串“ hello，world”不能用于转换。

下面的示例使有 toInt 方法来将 String 转换成 Int：

```
let possibleNumber = "123"
let convertedNumber = possibleNumber.toInt()
// convertedNumber is inferred to be of type "Int?", or "optional Int"
```

由于 toInt 方法可能会失败，因此它返回一个可选的 Int 类型，而不是一个 Int 类型 。一个可选 Int 被

写为 Int?，而不是 Int。问号表示它所包含的值是可选的，这意味着它可能包含一些 Int 值，也可能不包含任何数值。（它不包含其他任何数值，如 Bool 值或 String 值。这可能是一个 Int，或者什么都不是。）

3.2 if 语句和强制解包

你可以使用一个 if 语句找出一个可选类型是否包含一个值。如果可选类型包含一个值，它的计算结果为 true，如果没有值，它的计算结果为 false。

一旦你确定可选类型确实包含一个值，在输出的时候，你可以在可选类型的名称末尾添加一个感叹号（!）来完成。感叹号可以明确的表示“我已经清楚的知道了可选类型的值;可以使用。” 这就是所谓的可选类型的强制解包：

```
if convertedNumber {
println("\(possibleNumber) has an integer value of \(convertedNumber!)")
} else {
println("\(possibleNumber) could not be converted to an integer")
}
// prints "123 has an integer value of 123"
```

欲了解更多 if 语句相关问题，请参阅控制流程。

> 注
>
> 尝试使用“!”访问不存在的可选类型值会导致运行错误。使用“!”对数值进行强制解包前，要确保可选类型包含一个非 0 值。

3.3 选择绑定

可以使用可选绑定来确定一个可选类型是否包含一个值，如果是，将该值作为一个临时常量或变量。选择绑定可以用于 if 和 while 语句中，用来检查可选类型的值，并提取该值转换成一个常量或变量，作为单一操作的一部分。if 和 while 语句更详细的描述见控制流程。

编写 if 语句的可选绑定，如下所述：

```
if let constantName = someOptional {
statements
}
```

你可以使用选择绑定重写上面的 possibleNumber 例子，而不是进行强制解包：

```
if let actualNumber = possibleNumber.toInt() {
println("\(possibleNumber) has an integer value of \(actualNumber)")
} else {
println("\(possibleNumber) could not be converted to an integer")
}
// prints "123 has an integer value of 123"
```

这可以理解为：

“如果可选 Int 类型通过 possibleNumber.toInt 返回一个包含值，设置一个名为 actualNumber 的新的常量并纳入可选类型的值中。”

如果转换成功，则 actualNumber 常量在 if 语句的第一个分支内可用。它已经被初始化为一个可选类型的值，因此访问它的值时没有必要使用“!”为后缀。在这个例子中， actualNumber 被简单地用于输出转换的结果。

可以同时使用常量和变量来进行选择绑定。如果你想在 if 语句的第一个分支内操作 actualNumber 的值，你可以写成 if var actualNumber，如此，actualNumber 就成为一个变量而不是常量。

3.4 nil

你可以通过指定一个特定 nil 值设置可选变量为无值的状态：

```
var serverResponseCode: Int? = 404
// serverResponseCode contains an actual Int value of 404
serverResponseCode = nil
// serverResponseCode now contains no value
```

注

nil 不能用于非可选常量和变量。如果代码中的常量或变量需要能够应付一定条件下缺失值的情况，则需要将它声明为相应类型的可选类型值。

如果你定义一个可选类型常量或变量而没有提供默认值，这个常量或变量会自动设置为 nil：

```
var surveyAnswer: String?
// surveyAnswer is automatically set to nil
```

注

Swift 中的 nil 与 Objective – C 中的 nil 并不一样。在 Objective - C 中，nil 是一个指向不存在对象的指示器。而在 Swift 中，nil 不是一个指示器——它是一个特定类型的空值。任何可选类型都可以设置为 nil，而不仅仅是对象类型。

3.5 隐式解包可选类型

如上所述，可选类型表示一个常量或变量可以视“空值”。可以在 if 语句中检查可选类型是否存在一个值，并且可以用选择绑定有条件地解包来访问可选类型的值，如果它确实存在。

有时从程序的结构体中可以清楚地看出，一个可选类型的值在设定之后会一直存在。在这些情况下，有必要去掉每次检查和解包被访问的可选类型值，因为它可以一直保证有一个假设值存在。

这些类型的可选类型被定义为隐式解包可选类型。你可以在需要的可选类型后加上一个感叹号（String!）而不是一个问号（String？）来编写一个隐式解包可选类型。

当一个可选类型值在第一次定义后被确定存在并在每一点上确定假定存在后，隐式解包可选类型是必要的。在 Swift 中隐式解包可选类型主要用于类的初始化过程中，在无主引用和隐式展开可选属性中有详细描述。

隐式解包可选类型实际上是一个正常的可选类型，但它也可以用作一个非选择值，而无需在每次访问的时候来解包可选类型值。下面的例子显示了一个可选类型 String 和一个隐式解包可选类型 String 之间的

区别：

```
let possibleString: String? = "An optional string."
println(possibleString!) // requires an exclamation mark to access its value
// prints "An optional string."
let assumedString: String! = "An implicitly unwrapped optional string."
println(assumedString) // no exclamation mark is needed to access its value
// prints "An implicitly unwrapped optional string."
```

在每次使用时，可以把隐式解包可选类型当作自动解包的可选类型。你只需在声明一个值时在可选类型后加上一个感叹号，而不用在每次使用时都要在可选类型后面加上一个感叹号。

注

当隐式解包可选类型不包含一个值时，在你访问时，就会导致运行错误。如果你在一个不包含值的正常的可选类型后加上一个感叹号，其结果是完全一样的。

你还可以将一个隐式解包可选类型作为一个正常的可选类型对待，检查它是否包含一个值：

```
if assumedString {
println(assumedString)
}
// prints "An implicitly unwrapped optional string."
```

你也可以使用带有可选绑定的隐式解包可选类型来检查在单个语句中的值：

```
if let definiteString = assumedString {
println(definiteString)
}
```

```
// prints "An implicitly unwrapped optional string."
```

注

当在后来的点中有一个变量可能为 nil 时，不能使用隐式解包可选类型。如果你需要在一个变量使用周期中检查是否有 nil 值，可以使用一个正常的可选类型。

3.6 断言

可选类型帮你检查运行中一个值是否存在，并使用代码来处理数值缺失时的情况。但是在某些情况下，如果值不存在，或者值不符合一定的条件,那么这将会影响代码的执行。在这种情况下，可以在代码中加入断言来结束代码的执行，并提供一个机会来调试引起缺失或无效值的原因。

3.7 断言调试

断言是一种运行检查合乎逻辑的条件是否为 true 的方法。从字面上说，断言“asserts”条件为 true。可以使用断言以确保后续执行的任何代码必须满足成立的条件。如果条件计算结果为 true，那么继续执行代码; 如果条件的计算结果为 false，那么代码执行结束，应用程序被终止。

如果在调试环境中，如果代码中断，例如，在 Xcode 中构建并运行一个应用程序时，可以通过查看调试语句来查看断言中断时应用程序的状态。可以提供一个合适的调试信息当作一个合适的断言。

您可以通过调用全局 assert 函数编写一个断言。可以将 assert 函数传递为一种表达式，计算结果表示为 true 或 false，如果条件的结果为 false，则应显示一条消息。

```
let age = -3

assert(age >= 0, "A person's age cannot be less than zero")

// this causes the assertion to trigger, because age is not >= 0
```

在这个例子中，只有当 age> = 0 的时候，计算条件为 true，代码执行继续，也就是说，age 的值需是非负数。如果 age 的值是负的，如上面的代码，那么 age> = 0 计算结果为 false，断言会中断，应用程序终止。

断言消息不能使用字符串插值。如果需要的话，可以省略断言消息，如下面的例子：

```
assert(age >= 0)
```

3.8 何时使用断言

只要条件可能是 false，就可以使用断言，但为了保证代码继续执行，其条件必须是 true。断言检查的合适脚本包括：

一个整形下标索引被传递到一个自定义下标实现，但该下标索引值不能太低或太高。

一个值被传递到一个函数内，但一个有效值意味着该函数不能完成其任务。

An optional value 一个可选值目前为 nil，但是一个非 nil 值对于后续编码是必须的。

参阅下标和函数。

> 注
>
> 断言会导致应用程序终止，在不可能出现无效情况时，断言不能成为指定代码的代替品。然而，在无效情况可能发生的情况下，断言在应用程序发布前能确保发展过程中突出显示该条件。

3.9 基本运算符

运算符是用来检查，更改或组合值的特殊符号或短语。例如，加法运算符（+）将两个数字相加（如设 i = 1 + 2）。更复杂的例子包括逻辑 AND 运算符&&（如 if 中 enteredDoorCode && passedRetinaScan）和递增运算符++ i，是由 1 增加 i 值的快捷方式。

Swift 支持大部分标准 C 语言中的运算符并改进了几个功能以消除常见的编码错误。赋值运算符（=）不返回一个值，以防止和运算符（==）相等时被误用。算术运算符（+ ， - ，* ，/，％等等）检测和禁止值溢出，以避免在使用比存储他们的类型所允许的值的范围更大或更小的数值出现意外的结果。您可以通过使用 Swift 中的溢出运算符来选择值的溢出范围，详见上溢运算符。

不同于 C 语言，Swift 可以在浮点数上执行余数（％）计算。Swift 还提供了 C 语言中没有的两个范围运算符（a..b 和 A... B），作为表达范围值的快捷方式。

本章介绍了 Swift 的常见运算符。高级运算符包括了 Swift 中的高级运算符，并介绍了如何为自定义类型自定义运算符以及执行标准运算符。

3.10 术语

运算符分为一元，二元或三元运算符：

一元运算符有一个操作数（如-a）。一元前缀运算符会出现在该操作数前（如！b），一元后缀运算符会在操作数后出现（如 i + + ）。

二元运算符有两个操作数（如 2 + 3 ），并且是中缀，在它们出现在两个操作数之间。

三元运算符有三个操作数。与 C 语言类似，Swift 只有一个三元运算符，即三元条件运算符（ a ？b ：c）

数值运算符影响操作数。在表达式 1 + 2， +号是一个二元运算符并且它的两个操作数值为 1 和 2 。

3.11 赋值运算符

赋值运算符（ a = b ）用 b 初始化或更新 a 值：

```
let b = 10

var a = 5

a = b

// a is now equal to 10
```

如果右边的赋值是具有多个值的元组，其元素可以一次分解成多个常量或变量：

```
let (x, y) = (1, 2)

// x is equal to 1, and y is equal to 2
```

不同于 C 语言和 Objective-C 中的赋值运算符，Swift 中的赋值运算符本身并不返回一个值。下面的语句是无效的：

```
if x = y {

// this is not valid, because x = y does not return a value

}
```

这一特征可以防止使用相等的运算符（==）时，不小心使用赋值运算符（ = ）。通过使 if x = y 无效，Swift 可以帮助你避免代码中出现这些类型的错误。

3.12 算术运算符

Swift 支持所有数字类型的四则算术运算符：

```
1 + 2 // equals 3

5 - 3 // equals 2

2 * 3 // equals 6

10.0 / 2.5 // equals 4.0
```

不像在 C 语言和 Objective-C 中的算术运算符，Swift 算术运算符不允许值在默认情况下溢出。您可以通过使用 Swift 的溢出运算符（比如 a& ＋b 的）选择值溢出范围。详见上溢运算符。

加法运算符还支持 String 串联：

```
"hello, " + "world" // equals "hello, world"
```

两个 Character 值，或者一个 Character 值和一个 String 值，可以加在一起创建一个新的 String 值：

```
let dog: Character = "        "

let cow: Character = "        "

let dogCow = dog + cow

// dogCow is equal to "                "
```

详情可参阅连接字符串和字符。

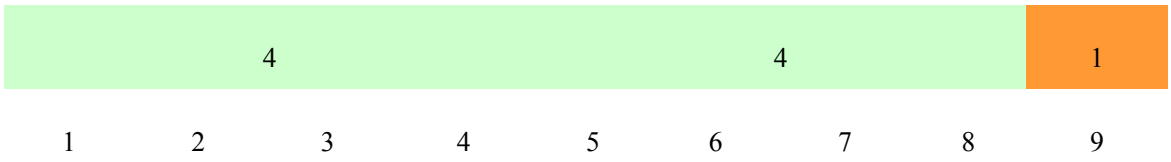
3.13 余数运算符

余数运算符（a%b）计算 a 包含 b 的多少倍后，返回剩下的值（即余数）。

注

余数运算符（%）在其他语言中也被称为模运算符。然而，严格地说，其在 Swift 中的负数范围意味着它是一个余数，而不是一个模运算符。

下述为余数运算符的运算原理。若要计算 9%4，你首先计算出



9 包括两个 4，余下的数是 1(橙色所示)。

在 Swift 内，可以编写为：

```
9 % 4 // equals 1
```

要确定 a%b 的答案，%运算符按照下面的等式计算，并返回余数作为其结果：

a=（b × 某乘数）+余数

乘数是 a 中所包含多少个 b 的最大数。

在方程量内，插入 9 和 4：

9 =（4×2）+1

同样的方法也适用于 a 为负值时余数的计算

```
-9 % 4 // equals -1
```

在方程量内，插入-9 和 4：

-9 =（4×-2）-1

余数为-1。

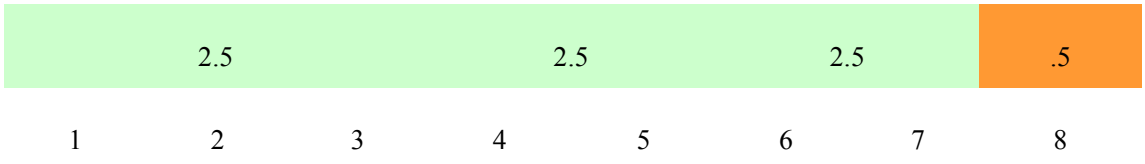
b 的负值，负号被忽略。这意味着 a % b 和 a%-b 结果一样。

3.14 浮点余数计算

和 C 语言和 Objective-C 中的余数运算符不同，Swift 的余数运算符还可以操作浮点数：

```
8 % 2.5 // equals 0.5
```

在此示例中，8 除以 2.5 等于 3，余数为 0.5，所以余数运算符返回一个 Double 值 0.5。



3.15 递增和递减运算符

与 C 语言相似，Swift 提供了增量运算符(++)和减量运算符(--)，作为增加或减少数值变量的一个快捷方式（每次增加或减少 1）。你可以使用这些运算符调整任何整数或浮点类型的变量。

```
var i = 0

++i // i now equals 1
```

每次调用++i，i 的值增加 1。从根本上讲，++是 i=i+1 的简写。同样，--可作为 i=i-1 的简写。

++和--符号可以用作前缀运算符或后缀运算符。++i 和 i++是 i 值增加 1 的两种有效方法。同样地，--i 和 i--是把 i 的值降低 1 的两种有效途径。

请注，这些运算符可以修改 i 也可返回一个值。如果只想递增或递减存储值，你可以忽略返回值。然而，如果你使用返回值，它将根据你使用运算符的前缀或后缀版本而有所不同，根据以下规则：

如果运算符被编写于变量之前, 返回其值时将对变量产生增量。

如果运算符被编写于变量之后，返回其值后将对变量产生增量。

例如：

```
var a = 0

let b = ++a

// a and b are now both equal to 1

let c = a++

// a is now equal to 2, but c has been set to the pre-increment value of 1
```

在上面的例子中，b=++a，返回数值前 a 增量。这就是 a 和 b 都等于为新值 1 的原因。

然而，返回其值后，let c = a + + 需增加一个 a 量。这意味着把旧值 1 赋予 c，而 a 将更新为等于 2。

除非你需要 i++的具体范围，否则建议你在所有的情况下使用++i 和--i，因为这些是对 i 进行修改和返回结果的有代表性的范围。

3.16 一元减号运算符

使用一个“-”前缀来表示数值的切换符号，称为一元减号运算符：

```
let three = 3
let minusThree = -three // minusThree equals -3
let plusThree = -minusThree // plusThree equals 3, or "minus minus three"
```

一元减号运算符直接位于它所操作的数值前面，没有任何空格

3.17 一元加号运算符

一元加号运算符（+）直接返回它所操作的数值，没有其它改变

```
let minusSix = -6
let alsoMinusSix = +minusSix // alsoMinusSix equals -6
```

虽然一元加号运算符实际上不做任何操作，但是当你使用一元减号运算符操作负数时，你可以用一元加号运算符保证正数代码的对称性。

3.18 复合赋值运算符

与 C 语言类似，Swift 提供复合赋值运算符将赋值（=）与另一个操作结合起来。例如赋值加法运算符（+=）：

```
var a = 1
a += 2
// a is now equal to 3
```

a += 2 是 a = a + 2 的快捷表达方式。实际上，加法与赋值会组合到一个运算符中，并在同一时间执行任务。

注

复合赋值运算符不返回值。例如，不可以编写为 let b = a + = 2。

这种方式与上面提到的递增和递减运算符不同。

完整的复合赋值运算符列表详见“表达式”。

3.19 比较运算符

Swift 支持所有标准 C 语言中的比较操作符：

等于 (a == b)

不等于 (a != b)

大于 (a > b)

小于 (a < b)

大于或等于 (a >= b)

小于或等于 (a <= b)

注

Swift 还提供了两个恒等运算符(=== and !==)，可以用它来测试两个对象引址是否都指向同一个对象实例。欲了解更多信息，请参阅类和结构。

每个比较运算符返回一个 Bool 值，显示该语句是否为 true：

```
1 == 1 // true, because 1 is equal to 1

2 != 1 // true, because 2 is not equal to 1

2 > 1 // true, because 2 is greater than 1

1 < 2 // true, because 1 is less than 2
```

```
1 >= 1 // true, because 1 is greater than or equal to 1

2 <= 1 // false, because 2 is not less than or equal to 1
```

比较运算符通常用在条件语句，如 if 语句：

```
let name = "world"

if name == "world" {

println("hello, world")

} else {

println("I'm sorry \(name), but I don't recognize you")

}

// prints "hello, world", because name is indeed equal to "world"
```

欲了解更多 if 语句相关问题，请参阅控制流。

3.20 三元条件运算符

三元条件算子是有三个部分的特殊运算符，其形式为：question？answer1：answer2。它是基于 question 是否属实对两个表达式之一进行判定的快捷方式。如果 question 是真实的，它计算 answer1 的结果，并返回其值；否则它计算 answer2 并返回它的值。

三元条件运算符是下面的代码的快捷方式：

```
if question {

answer1

} else {

answer2

}
```

下面是一个例子，它计算表格行的像素高度。如果行有页眉，行高应该是比内容高 50 像素；如果行没

有页眉，行高应该是比内容高 20 像素：

```
let contentHeight = 40
let hasHeader = true
let rowHeight = contentHeight + (hasHeader ? 50 : 20)
// rowHeight is equal to 90
```

上述例子是下面代码的简写：

```
let contentHeight = 40
let hasHeader = true
var rowHeight = contentHeight
if hasHeader {
rowHeight = rowHeight + 50
} else {
rowHeight = rowHeight + 20
}
// rowHeight is equal to 90
```

使用三元条件运算符的第一个例子的意味着使用一行代码就可以正确设定 rowHeight。这比第二个例子更简洁，并删除 rowHeight 设定为变量，因为它的值不需要使用 if 语句修改。

三元条件运算符提供了一种有效的简写，以决定考虑哪两个表达式。然而，使用三元条件运算符时应谨慎。如果使用过度，其简洁性可以导致代码难以阅读。避免使用三元条件运算符将多个实例结合成一个复合语句。

3.21 范围运算符

Swift 包括两个范围运算符，这是表达值的范围的快捷方式。

3.22 闭区间运算符

闭区间运算符（a...b）定义了一个从 a 到 b 的范围，包括值 a 和 b。

当你想迭代一个范围以用到所有的值时，闭区间运算符是非常有用的，例如“for-in”循环：

```
for index in 1...5 {
println("\(index) times 5 is \(index * 5)")
}
// 1 times 5 is 5
// 2 times 5 is 10
// 3 times 5 is 15
// 4 times 5 is 20
// 5 times 5 is 25
```

欲了解更多 for-in 循环内容，请参阅控制流。

3.23 半开区间运算符

半开区间运算符（a..b）定义了一个从 a 到 b 的范围，但不包括 b。称其为半开是因为它包含第一个值，而不包含最后一个值。

当你处理从零开始的列表（如数组）时，半开区间是特别有用的，可以一直数到列表的长度（但不包括此长度）：

```
let names = ["Anna", "Alex", "Brian", "Jack"]
let count = names.count
for i in 0..count {
println("Person \(i + 1) is called \(names[i])")
}
// Person 1 is called Anna
```

```
// Person 2 is called Alex

// Person 3 is called Brian

// Person 4 is called Jack
```

请注，这个数组包含四个项目，但 0..count 只统计到 3（数组中最后一项的索引），因为它是一个半开区域。欲了解更多数组相关信息，请参阅数组。

3.24 逻辑运算符

逻辑运算符修改或综合布尔逻辑值，结果是 true 或 false。Swift 支持在 C 语言中出现的三个标准的逻辑运算符：

逻辑 NOT (!a)

逻辑 AND (a && b)

逻辑 OR (a || b)

3.25 逻辑 NOT 运算符

逻辑运算符（！a）反转一个布尔值，将 true 变为 false，false 改为 true。

逻辑 NOT 运算符是一个前缀操作符，直接显示在它操作的值之前，之间没有任何空格。该运算符可以被理解为“not a”，如下例所示：

```
let allowedEntry = false

if !allowedEntry {

println("ACCESS DENIED")

}

// prints "ACCESS DENIED"
```

短语“if !allowedEntry ”可以读作“if not allowed entry.”随后的行只在“not allowed entry”是 true 时执行；也就是说，如果 alowed Entry 是 false。

和在这个例子中一样，谨慎选择布尔常量和变量的名字可以使代码简洁易读，同时避免双重否定或混乱的逻辑语句。

3.26 逻辑 AND 运算符

逻辑 AND 运算符（a && b）创建逻辑表达式，要求每个值都是 true，整个表达式才是 true。

如果其中任一个值为 false，整个表达式也将是 false。事实上，如果第一个值为 false，第二个值将不会被测算，因为它不可能使整个表达式成立。该运算符被称之为短路求值。

这个例子考虑两个 Bool 值，只在这两个值都是 true 的情况下允许访问：

```
let enteredDoorCode = true

let passedRetinaScan = false

if enteredDoorCode && passedRetinaScan {

println("Welcome!")

} else {

println("ACCESS DENIED")

}

// prints "ACCESS DENIED"
```

3.27 逻辑 OR 运算符

逻辑 OR 运算符 (a || b) 是一个由两个相邻的字符组成的中缀运算符。你用它来创建逻辑表达式，其中两个值只要有一个是 true，整个表达式就是 true。

类似于上面的逻辑 AND 运算符，逻辑 OR 运算符使用短路求值测算其表达式。如果逻辑 OR 运算符的左边是 true，那么其右边将不被计算，因为它不能改变整个表达式的结果。

在下面的示例中，第一个 Bool 值（hasDoorkey）是 false，但第二值（knowsOverridePassword）是 true。因为一个值是 true，整个表达式也测算为 true，并允许访问：

```
let enteredDoorCode = true

let passedRetinaScan = false

if enteredDoorCode && passedRetinaScan {

println("Welcome!")
```

```
} else {
println("ACCESS DENIED")
}
// prints "ACCESS DENIED"
```

3.28 组合逻辑运算符

可以综合运用多个逻辑运算符来创建较长的复合表达式：

```
if enteredDoorCode && passedRetinaScan || hasDoorKey || knowsOverridePassword {
println("Welcome!")
} else {
println("ACCESS DENIED")
}
// prints "Welcome!"
```

此示例使用多个&& 和 || 运算符创建较长的复合表达式。然而，&&和||运算符仍然只运行两个值，所以这实际上是三个小表达式链接在一起。这可以理解为：

如果我们输入了正确的门的代码并通过了视网膜扫描；或者如果我们有一个有用的门钥匙；或者如果我们知道紧急替代密码，然后允许访问。

在 enteredDoorCode，passedRetinaScan 和 hasDoorKey 中，前两个小表达式是 false。然而，紧急替代密码是已知的，所以整体复合表达式仍然为 true。

3.29 显式括号

当没有严格需要时，可以使用括号使一个复杂表达式更容易阅读。在上面门的例子中，在复合表达式的第一部分添加括号使复合表达式的目的变得明确：

```
if (enteredDoorCode && passedRetinaScan) || hasDoorKey || knowsOverridePassword {
println("Welcome!")
} else {
```

```
println("ACCESS DENIED")
}
// prints "Welcome!"
```

括号清晰地显示出，在整体逻辑中，前两个值都被视为独立的可能发生的情形的一部分。复合表达式不改变输出结果，但对读着而言，其总体意图是较为明确的。可读性总是较为简洁；使用括号有助于明确你的意图。

3.30 字符串和字符

字符串是多个字符的有序集合，如 "hello, world" 或者 "albatross"。Swift 的字符串主要是 String 类型，这反过来又代表了 Character 类型值的集合。

Swift 中 string 和 Charater 两种类型提供了一种快速、符合 Unicode 标准的方式来处理代码中的文本。字符串创建和操作的语法是简单可读的，这和 C 语言中的字符串语法相似。字符串串联和将两个字符串用+运算符加在一起一样简单，和字符串的可变性由在选择常量或变量决定，就像 Swift 中的其他值一样。

尽管是这种简单的语法，Swift 的 String 类型是一个快速、现代化的字符串。每个字符串是由独立编码的 Unicode 字符组成，并提供各种 Unicode 表示方式支持访问这些字符。

字符串也可以用来将各种常量、变量、常值以及表达式嵌入到较长的字符串中，这一过程称为字符串插值。这让它更容易地来创建显示、存储以及输出自定义字符串值。

注

Swift 的 String 类型被无缝桥接成为 Foundation 的 NSString 类。如果你处理在 Cocoa 或 Cocoa Touch 中的 Foundation 框架时，整个 NSString API 可以调用你所创建的任何 String 值，除了本章所描述的 String 特征外，你也可以根据 API 需要 NSString 实例使用 String。

有关综合使用 Foundation、Cocoa 和字符串的更多信息，参见 Using Swift 和 Cocoa 以及 Objective- C.

3.31 字符串常值

你可以在代码中计入预定义的 String 值作为字符串常值。一个字符串常值是用双引号（“”）将文本字

符引起来的固定序列。

字符串常值可以用来提供一个常量或变量的初始值：

```
let someString = "Some string literal value"
```

注，Swift 可以推断 String 类型为一个 someString 常量，因为它是用一个字符串常值初始化的。

字符串常量可以包含以下特殊字符：

逃逸的特殊字符 \0 (空字符), \\ (反斜杠), \t (水平移动), \n (换行), \r (回车), \" (双引号) and \' (单引号) 单字节Unicode标量, 被写作 \xnn, nn 是两个十六进制位两字节Unicode 标量, 被写作 \unnnn, nnnn 是四个十六进制位四字节Unicode 标量, 被写作 \Unnnnnnnn, nnnnnnnn 是八个十六进制位

下面的代码显示了每一种特殊字符的一个例子。该 wiseWords 常量包含两个去掉的双引号字符。dollarSign，blackHeart,和 sparklingHeart 常量展示了三种不同的 Unicode 标量字符格式：

```
let wiseWords = "\"Imagination is more important than knowledge\" - Einstein"
// "Imagination is more important than knowledge" - Einstein
let dollarSign = "\x24" // $, Unicode scalar U+0024
let blackHeart = "\u2665" // ♥, Unicode scalar U+2665
let sparklingHeart = "\U0001F496" //  , Unicode scalar U+1F496
```

3.32 初始化空字符串

创建一个空的字符串值作为建立一个较长的字符串的出发点，或者指定一个空字符串常值为变量，或使用初始化器语法初始化一个新的 String 实例：

```
var emptyString = "" // empty string literal
var anotherEmptyString = String() // initializer syntax
// these two strings are both empty, and are equivalent to each other
```

你可以通过检查布尔 isEmpty 性来检测一个 String 值是否为空：

```
if emptyString.isEmpty {
```

```
println("Nothing to see here")
}
// prints "Nothing to see here"
```

3.33 字符串易变性

可以通过将一个特定的 String 指定给一个变量（在这种情况下，它可以被修改），或一个常量（在这种情况下，它不能被修改）来显示它是否可以被修改（或被突变）

```
var variableString = "Horse"
variableString += " and carriage"
// variableString is now "Horse and carriage"
let constantString = "Highlander"
constantString += " and another Highlander"
// this reports a compile-time error - a constant string cannot be modified
```

注

这种方法不同于 Objective-C 和 Cocoa 中的字符串突变，在这两种语言中你选择两个类之一（NSString 和 NSMutableString）来表示一个字符串是否可以突变。

3.34 字符串为数值类型

Swift 的 String 类型为数值类型。如果你创建一个新 String 值，当它通过一个函数或方法，或当它被指定给一个常量或变量时，这个 String 值会被复制。在每一种情况下，现有 String 值的一个新的复制会被创建，这个新的复制会被传递或指定，而不是采用原来的版本。数值类型的详细内容参阅结构和枚举是值类型。

注

该与 Cocca 中的 NSString 不同。当你在 Cocoa 中创建个 NSString 实例时，并把它传递给一个函数或方法或将其指定给一个变量，你总会在给同一个单一 NSString 传递或指定一

个引用。除非你有特别要求，否则不会发生复制字符串。

Swift 的默认复制 String 的行为可以确保当一个函数或方法传递你的 String 值，不论是谁赋予的，你都会拥有一个确切的 String 值。你可以确信你传递的字符串将不被修改，除非你自己修改它。

实际上，Swift 的编译器优化字符串使用，只有在必要的时候才会复制。这意味着你在使用字符串的数值类型时，这些数值类型总是处在较好的性能状态。

3.35 使用字符运算

Swift 的 String 类型代表以指定的顺序集合的 Character 值。每个 Character 代表一个单一 Unicode 字符。你可以使用 for-in 循环迭代某个字符串来访问字符串中的单个 Character 值：

```
for character in "Dog!        " {
println(character)
}
// D
// o
// g
// !
//
```

在 for-in 循环语句在循环语句中有详细描述。

另外，通过提供一个 Character 类型的注释从一个单一字符字符串常值来创建一个独立的字符常量或变量：

```
let yenSign: Character = "¥"
```

3.36 计算字符

可以通过调用全局 countElements 函数以及传递字符串作为函数的唯一参数来检索字符串中的字符

数：

```
let unusualMenagerie = "Koala 	, Snail 	, Penguin 	, Dromedary 	"

println("unusualMenagerie has \(countElements(unusualMenagerie)) characters")

// prints "unusualMenagerie has 40 characters"
```

注

不同的 Unicode 字符和同一 Unicode 字符的不同表示方式可以要求不同数量的内存来存储。因此，Swift 中的字符和字符串表示形式并不占据相同数量的内存。所以，一个字符串的长度只能在迭代字符串、考虑它的每个字符之后来计算。如果你在使用特别长的字符串值，要知道，countelements 函数必须迭代字符串中的每个字符，来计算字符串的准确数值。

此外还需注由 countElements 返回的字符数并不总是与包含相同字符的 NSString 的长度属性相同。NSString 的长度基于字符串内 UTF-16 的 16 位代码数，不是字符串内的 Unicode 字符数。为了反映这一事实，当访问一个 Swift 的 String 值时， NSString 的长度属性被称为 utf16count 。

3.37 字符串和字符串联

使用加法运算符(+)可将 String 值和 Chracter 值相加（或串联）来创建一个新的 String 值：

```
let string1 = "hello"

let string2 = " there"

let character1: Character = "!"

let character2: Character = "?"


let stringPlusCharacter = string1 + character1 // equals "hello!"

let stringPlusString = string1 + string2 // equals "hello there"
```

```
let characterPlusString = character1 + string1 // equals "!hello"

let characterPlusCharacter = character1 + character2 // equals "!?"
```

此外可以使用加法赋值运算符(+=)将一个 String 值或 Character 值附加到现有的 String 变量上：

```
var instruction = "look over"

instruction += string2

// instruction now equals "look over there"

var welcome = "good morning"

welcome += character1

// welcome now equals "good morning!"
```

> 注
>
> 不能将一个 String 或 Character 附加到现有的 Character 变量上,，因为一个 Character 值必须只包括单个字符。

3.38 字符串插值

字符串插值是从常量、变量、文本和表达式中，通过把它们的值插入字符串常值内而构造出新的 String 值的一种方式。你插入字符串常值的每个项目外面需要加上括号，并在括号前面加一个反斜杠：

```
let multiplier = 3

let message = "\(multiplier) times 2.5 is \(Double(multiplier) * 2.5)"

// message is "3 times 2.5 is 7.5"
```

在上面的示例中，multiplier 值（ \ （multiplier））插入到一个字符串常值中。当通过计算字符串插值来创建实际的字符串时，要用 multiplier 的实际值来取代占位符。

multiplier 值也是字符串中后来的较大表达式的一部分。此表达式计算 Double(multiplier) * 2.5 的值，并将结果 （7.5） 插入字符串。在这种情况下，当计入字符串常值时，其表达式写成 \(Double(multiplier) * 2.5)。

注

你编写的在括号内插入的字符串的表达式不能包含非转义的双引号 (") 或反斜杠 (\)，并且不能包含返回或换行。

3.39 比较字符串

Swift 提供了三种方法来比较 String 值：字符串等式、前缀等式和后缀等式。

3.40 字符串等式

如果以相同的顺序包含完全相同的字符，那么两个字符串值相等：

```
let quotation = "We're a lot alike, you and I."
let sameQuotation = "We're a lot alike, you and I."
if quotation == sameQuotation {
println("These two strings are considered equal")
}
// prints "These two strings are considered equal"
```

3.41 前缀和后缀等式

若要检查字符串是否具有特定字符串的前缀或后缀，可以调用字符串 hasPrefx 和 hasSuffix 方法，这两个方法将使用 String 类型的参数并返回布尔值。这两种方法执行基本字符串和前缀或后缀字符串之间逐个字符进行比较。

下面的示例考虑表示从 Shakepeare 的 Romeo and Juliet 中的第一幕两个场景的位置的字符串数组：

```
let romeoAndJuliet = [
"Act 1 Scene 1: Verona, A public place",
```

```
"Act 1 Scene 2: Capulet's mansion",
"Act 1 Scene 3: A room in Capulet's mansion",
"Act 1 Scene 4: A street outside Capulet's mansion",
"Act 1 Scene 5: The Great Hall in Capulet's mansion",
"Act 2 Scene 1: Outside Capulet's mansion",
"Act 2 Scene 2: Capulet's orchard",
"Act 2 Scene 3: Outside Friar Lawrence's cell",
Act 2 Scene 4: A street in Verona",
Act 2 Scene 5: Capulet's mansion",
Act 2 Scene 6: Friar Lawrence's cell"
```

你可以在 romeoAndJuliet 数组中使用 hasPrefx 方法来计算这出戏第一幕的场景数：

```
var act1SceneCount = 0
for scene in romeoAndJuliet {
if scene.hasPrefix("Act 1 ") {
++act1SceneCount
}
}
println("There are \(act1SceneCount) scenes in Act 1")
// prints "There are 5 scenes in Act 1"
```

同样，使用 hasSuffix 方法来计算发生在 Capulet 府宅和 Friar Lawrence 牢房或其周围的场景数：

```
var mansionCount = 0
var cellCount = 0
```

```
for scene in romeoAndJuliet {
if scene.hasSuffix("Capulet's mansion") {
++mansionCount
} else if scene.hasSuffix("Friar Lawrence's cell") {
++cellCount
}
}
("\(mansionCount) mansion scenes; \(cellCount) cell scenes")
prints "6 mansion scenes; 2 cell scenes"
```

3.42 大写和小写字符串

您可以使用 uppercaseString 和 lowercaseString 的属性访问字符串的大写或小写版本

```
let normal = "Could you help me, please?"
let shouty = normal.uppercaseString
// shouty is equal to "COULD YOU HELP ME, PLEASE?"
let whispered = normal.lowercaseString
// whispered is equal to "could you help me, please?"
```

3.43 Unicode

Unicode 是编码和文本表达的国际标准。它使你可以用标准化的形式表示几乎任何语言的任何字符，从外部来源（如文本文件或 web 页）中读取和写入这些字符。

Swift 的字符串和字符的类型与 Unicode 完全兼容。他们支持不同的 Unicode 编码，如下所述。

3.44 Unicode 术语

在 Unicode 中的每个字符可以由一个或多个 unicode 标量表示。Unicode 标量是唯一一个有 21 位数字（和名称）的字符或修饰符，如 U + 0061 代表 LOWERCASE LATIN LETTER A （"a"） 或 U + 1F425 为 FRONT-FACING BABY CHICK ("□")。

当一个 Unicode 字符串编写为一个文本文件或一些其他存储时，这些 unicode 标量则被编入其中的一个 Unicode 定义格式中。每个格式会一小块的形式编制字串符，称为代码单元。这些包括 UTF-8 格式 （以 8 位代码单元进行字符串编码） 和 UTF-16 格式 (以 16 位代码单元进行字符串编码)。

3.45 字符串的 Unicode 表达式

Swift 提供几种不同的方式来访问字符串的 Unicode 表达式。

你可以使用 for-in 语句迭代访问字符串，以 Unicode 字符访问字符串中的个别 Character 值。该进程在与字符工作中有详细描述。

或者，以 Unicode 兼容的其他三个表示方式的一种访问字符串值：

下面的每个示例显示下面的字符串的不同表达式，分别由字符 d、 o、 g，!和口字符（DOG FACE 或 Unicode 标量 U + 1F436）组成：

```
let dogString = "Dog!        "
```

3.46 UTF-8

可以通过迭代访问其 utf8 属性来访问字符串的 UTF-8 表达式。此数量值的类型是 UTF8View，这是一个 8 位无符号（UInt8）值的集合，是字符串 UTF-8 表达式中每一位数的属性：

```
for codeUnit in dogString.utf8 {
print("\(codeUnit) ")
}
print("\n")
// 68 111 103 33 240 159 144 182
```

在上面的例子中，第一个四位十进制 codeUnit 值（68，111，103，33）分别代表字符 D，o, g，和！，这些字符的 UTF-8 表达式和 ASCII 表达式相同。最后四个 codeUnit 值（240，159，144，182）是一个四字节的 UTF-8 表达式，代表 DOG FACE 字符。

3.47 UTF-16

可以通过迭代访问其 utf16 属性来访问字符串的 UTF-16 表达式。此数量值的类型是 UTF16View，这是一个 16 位无符号（UInt16）值的集合，是字符串 UTF-16 表达式中每一位数的属性：

```
for codeUnit in dogString.utf16 {
print("\(codeUnit) ")
}
print("\n")
// 68 111 103 33 55357 56374
```

再次，前四个 codeUnit 值（68，111，103，33）表示的字符 D，o，g 和！,这些字符 UTF-16 代码单元和字符串 UTF-8 表达式有一样的数值

第五和第六 codeUnit 值（55357 和 56374）是 DOG FACE 字符的 UTF-16 代理对表达式。这些值是 U＋d83d（十进制值 55357）的引导替代值和 U＋dc36（十进制值 56374）的跟踪替代值。

3.48 Unicode 标量

可以通过迭代 unicodeScalars 属性来访问字符串值的 Unicode 标量表达式。此数量值的属性是 UnicodeScalarView 类型，这是一个 UnicodeScalar 集合值。Unicode 标量是任何 21 位 Unicode 代码点，不是引导替代代码点或跟踪替代代码点。

每个 Unicodescalar 数量值都能和 21 位的标量一一对应，可用一个 UInt32 数值表示：

```
for scalar in dogString.unicodeScalars {
print("\(scalar.value) ")
}
print("\n")
// 68 111 103 33 128054
```

在前四个 UnicodeScalar 的值特性（68，111，103，33）代表的仍是 D，o, g，和！。第五个和最后一个用 UnicodeScalar 的值属性为 128054，这是一个的十六进制值 1F436 的十进制数，这相当于 DOG FACE 的

Unicode 标量 U + 1F436 的字符串。

使用另一种方式查询其数值属性，每个 UnicodeScalar 值也可以被用来构造一个新的 String 值，如字符串插值：

```
for scalar in dogString.unicodeScalars {
println("\(scalar) ")
}
// D
// o
// g
// !
//
```

3.49 集合类型

Swift 提供两个集合类型，即数组和字典，用于存储值集合。数组储存在相同类型的数值有序表内。字典存储相同类型值的无序集合，可以通过一个唯一标识符（也称为密钥）进行访问和查阅。

Swift 中的数组和字典明确表明可以存储的值和密钥的类型。这意味着你不能在数组或字典中插入任何一个的错误类型的数值。这也表明你可以确信数组或字典中数值类型的检索。Swift 的使用显式类型集合，确保代码可以清楚地表达它所服务的数值类型，可以帮助你在代码编写的初期获取任何不匹配类型。

> 注
>
> 当 Swift 的数组类型被指定给一个常量或变量，或当传递给一个函数或方法时，其会表现出不同的形式。更多信息可参考集合可变性和集合类型的配置和拷贝行为。

4 数组

一个数组可以在一个有序列表中存储同一类型的多值数值。相同数值可以在一个数组的不同位置多次

出现。

Swift 数组特别指代它可以存储几类数值。Swift 数组不同于 Objective-C 的 NSArray 和 NSMutableArray，后者都可以存储任何类型的对象，但不能提供的处理后的任何信息。在 Swift 中，特别是数组中，无论是通过显式类型注释或通过类型推断，数值的存储都非常明确，不局限于一个类类型。例如，当你创建一个 Int 值的数组时，你不能插入除了 Int 值之外的其他任何值到数组中。Swift 数组迅速、安全而且内容清晰。

4.1 数组类型缩写语法

Swift 数组类型的编写形式是 Array<SomeType>，这是一个允许存储的字节。你也可以用缩写形式 SomeType []来进行数组编写。虽然这两种形式的作用相同，但是在数组编写指南中提到数组类型时主要使用缩写形式。

4.2 数组常值

你可以用数组常值初始化一个数组，这是在编写数组集合时的一个缩写形式。数组常值可以作为列表数值编写，写在括号中。以逗号分隔：

[value 1 , value 2 , value 3]

下面的示例是一个存储 String 数值创建的 shoppingList 数组：

```
var shoppingList:  String[] = ["Eggs", "Milk"]
// shoppingList has been initialized with two initial items
```

该 shoppingList 变量被声明为“String 值数组”，用 String[ ]编写。因为这个特殊的数组具有指定的 String 类型值，所以只允许存储 String 值。在这里，用两个 String 值初始化（“Eggs”和“Milk”）的 shoppingList 数组是以数组常值形式编写的。

> 注
>
> shoppingList 数组被声明为一个变量（用 var 引导器）并不是一个常量（用 let 引导器），因为还有更多项目要添加在下面的 shoppingList 例子中。

在这种情况下，数组常值只包含两个 String 值。这和 shoppingList 变量声明类型（即数组只包含 String 的值）相匹配，并允许数组常值以用两个初始项目初始化 shoppingList 的方式来赋值。

由于 Swift 中的类型推断，在你将包含相同类型数值的数组常值对其初始化时，你不必写数组类型。shoppingList 的初始化可以使用简写形式：

```
var shoppingList： String[] = ["Eggs", "Milk"]
```

由于数组常值的所有数值都是同一类型的，Swift 数组推断 String[ ]是可以用于 shoppingList 变量的正确的类型。

4.3 访问和修改数组

可以通过相应的修改方法和属性或通过使用下标语法来访问和修改数组。

如查找数组中的项目数，只需查找其只读计量属性：

```
println("The shopping list contains \(shoppingList.count) items.")
// prints "The shopping list contains 2 items."
```

使用简写布尔值 isEmpaty 来检测计数属性是否等于 0：

```
if shoppingList.isEmpty {
println("The shopping list is empty.")
} else {
println("The shopping list is not empty.")
}
// prints "The shopping list is not empty."
```

可以通过调用数组的数组末端追加方法添加一个新的项目：

```
shoppingList.append("Flour")
// shoppingList now contains 3 items, and someone is making pancakes
```

另外，还可用加法赋值运算(+=)将一个新项目添加到数组末尾：

```
shoppingList += "Baking Powder"

// shoppingList now contains 4 items
```

您还可以用加法赋值运算（+=）符为数组添加兼容项目

```
shoppingList += ["Chocolate Spread", "Cheese", "Butter"]

// shoppingList now contains 7 items
```

可利用下标数组值的方法检索，即把你想检索的数值索引用方括号括起来并放在数组名称后面：

```
var firstItem = shoppingList[0]

// firstItem is equal to "Eggs"
```

请注，数组中的第一项的索引为 0，不是 1。Swift 中数组的索引就是 0。

你可以用下标语法改写给定索引的数值：

```
shoppingList[0] = "Six eggs"

// the first item in the list is now equal to "Six eggs" rather than "Eggs"
```

你也可以用下标语法即刻改变值的范围，即使替换值和原有值数组长度不同。就像用“Cheese”替换“Chocolate Spread”，用“Banana”和“Apples”替换“Butter”：

```
shoppingList[4...6] = ["Bananas", "Apples"]

// shoppingList now contains 6 items
```

注

不能使用下标语法来添加新项目到数组末尾。如果尝试使用下标语法来检索或设置一个索引值（此为数组现有范围以外的值），这将会引发运行错误。然而，你通过对数组的 count 属性比较使用一个有效索引。除了当 count 为 0（即数组为空），该数组中的最大有效索引是 count-1，因为数组索引从零开始。

插入一个项目到指定索引处的数组，称为数组的 insert（atIndex：）方法：

```
shoppingList.insert("Maple Syrup", atIndex： 0)
// shoppingList now contains 7 items
// "Maple Syrup" is now the first item in the list
```

这叫做 insert 方法，即以索引 0 指示用 shopping list 中的 Maple Syrup 来插入一个新项目。

同样，你可以用 removeAtIndex 方法从数组中移除此项。此方法可以应用于移除指定索引和返回移除项（如果你不需要它，你可以忽略返回值）：

```
let mapleSyrup = shoppingList.removeAtIndex(0)
// the item that was at index 0 has just been removed
// shoppingList now contains 6 items, and no Maple Syrup
// the mapleSyrup constant is now equal to the removed "Maple Syrup" string
```

当项目被移除的时候，数组不能进行任何编写，所以索引值 0 此时等同于“Six eggs”：

```
firstItem = shoppingList[0]
// firstItem is now equal to "Six eggs"
```

如果你想从数组中删除最后一个项目，使用 removeLast 方法而不是 removeAtIndex 方法以避免需要查询数组的 count 属性。像 removeAtIndex 方法，removeLast 将返回被删除的项目：

```
let apples = shoppingList.removeLast()
// the last item in the array has just been removed
// shoppingList now contains 5 items, and no cheese
// the apples constant is now equal to the removed "Apples" string
```

4.4 迭代数组

你可以用 for-in 循环方式在数组中迭代整套的数值：

```
for item in shoppingList {
```

```
println(item)
}
// Six eggs
// Milk
// Flour
// Baking Powder
// Bananas
```

如果你需要每一项的整数索引和它的值，使用全局枚举函数将数组进行迭代，在由索引和该项目值所组成的数组中，枚举函数为每一项返回一个元组。你可以将元组分解成临时的常量或者变量，作为迭代的一部分。

```
for (index, value) in enumerate(shoppingList) {
println("Item \(index + 1)：  \(value)")
}
// Item 1：  Six eggs
// Item 2：  Milk
// Item 3：  Flour
// Item 4：  Baking Powder
// Item 5：  Bananas
```

欲了解更多 for-in 循环相关信息，请参阅循环语句。

4.5 创建并初始化数组

你可以用初始化器语法创建一个特定类型（不设置任何原始值）的空数组：

```
var someInts = Int[]()
println("someInts is of type Int[] with \(someInts.count) items.")
```

```
// prints "someInts is of type Int[] with 0 items."
```

注，由于被设置为一种 Int[]初始化器的输出，someInt 变量的类型会被推断为 Int[]。

或者，如果上下文已经提供类型信息，比如一种函数论证或者一种已被确定类型的变量或者常量，你可以用空数组常值创建一个空数组，写为[]（用方括号括起来）。

```
someInts.append(3)
// someInts now contains 1 value of type Int
someInts = []
// someInts is now an empty array, but is still of type Int[]
```

Swift 的数组类型也为创建特定大小的数组提供了一个初始化器，将它所有的值设置成一个既有默认值你将要加入到新数组中的项数（称为 count）和适当类型的默认值（称为 repeatedValue）传输到初始化器：

```
var threeDoubles = Double[](count： 3, repeatedValue： 0.0)
// threeDoubles is of type Double[], and equals [0.0, 0.0, 0.0]
```

由于类型推断，使用该初始化器时，无需详细说明要存储数组的类型，因为它可以从默认值中推断出来。

```
var anotherThreeDoubles = Array(count： 3, repeatedValue： 2.5)
// anotherThreeDoubles is inferred as Double[], and equals [2.5, 2.5, 2.5]
```

最后，你可以用加法运算符（+）通过将两个类型兼容的数组加在一起来创建一个新数组。新数组的类型可以由相加的两个数组类型推断出来。

```
var sixDoubles = threeDoubles + anotherThreeDoubles
// sixDoubles is inferred as Double[], and equals [0.0, 0.0, 0.0, 2.5, 2.5, 2.5]
```

4.6 字典

字典是储存同一类型多个值的容器。每一个值都与一个独一无二的键相关联，在字典中键充当该值的标示符。不像数组中的项，字典中的项没有特定的指令。当需要查以标示符为基础的数值时，可以使用字典，这跟现实世界中用来查特定单词定义的字典大多相同。

Swift 字典特别描述了可以存储的键类型和值类型。它们和 Objective-C 的 NSDictionary 和

NSMutableDictionary 类别不一样，Objective-C 的 NSDictionary 和 NSMutableDictionary 可以用任何种类的对象作为键和值，并且不提供任何有关这些对象本质的信息。在 Swift 里面，不管是通过显式类型注释还是通过类型推断，特定字典都可以清晰地储存键类型和数值类型。

Swift 的字典类型写为字典<KeyType, valueType>，KeyType 可以用作字典键的数值类型，valueType 是字典为那些键储存的数值类型。

唯一的局限是 KeyType 必须是 hashable，也就是说，它必须提供一种描绘自我的独特方式。所有的 Swift 基本类型（比如 String、Int、Double 和 Bool）都默认为是 hashable，并且所有这些类型均可作为字典的键。无关联值的枚举成员值（如枚举中所述）也默认为是 hashable。

4.7 字典常值

你可以用字典常值初始化字典，这和之前见到的数组常值有类似的语法。字典常值是一种将一个或者多个键值对写为 Dictionary 集合的缩写形式。

一个键值对是键和值的组合。在字典常值中，每一个键值对中的键和值都用冒号隔开。在括号中将键值对写成一个列表，并用逗号隔开。

[key 1 ： value 1 , key 2 ： value 2 , key 3 ： value 3]

下面的例子创建了一个存储国际机场名字的字典。该字典中，这些键是三个字母的国际航空运输协会编码，值是机场名字：

```
var airports： Dictionary<String, String> = ["TYO"： "Tokyo", "DUB"： "Dublin"]
```

据称，机场字典可以声明为 Dictionary 类型，即<String，String>，意思是“键为是 String 型，值也是 String 型的字典”

> 注
>
> 该 airports 字典被声明为一个变量（用 var 引导器）并不是一个常量（用 let 引导器），因为还有更多 airports 会出现在下列例子中的字典里。

airports 字典用包含两个键值对的字典常值初始化。第一对为“TYO”键和“Tokyo”值。第二对为“DUB”键和“Dublin”值。

本字典常值包含两个 String：String 对。这和 airports 变量声明的类型相匹配（一部字典只有一个 String 键和一个 String 值），因此字典常值的赋值允许作为用两个初始项初始化 airports 字典的方式。

至于数组，如果你要用键和值类型一致的字典常量初始化的话，就没必要写下字典的类型。airports 的初始值可以使用简写形式：

```
var airports = ["TYO":  "Tokyo", "DUB":  "Dublin"]
```

因为常值中所有的键都有是相同的类型，同样的所有值也是相同的类型，Swift 可以推断 Dictionary<String，String>是用作 airports 字典的正确类型

4.8 访问和修改字典

你相应的方法和属性或通过使用下标语法来访问和修改字典。至于数组，你可以通过核对它的只读 count 属性在 Dictionary 中找出项目数。

```
println("The dictionary of airports contains \(airports.count) items.")
// prints "The dictionary of airports contains 2 items."
```

你可以用下标语法将新项目添加到字典中。将一种新的适当类型的键用作下标语法，并赋一个新的相应类型的值。

```
airports["LHR"] = "London"
// the airports dictionary now contains 3 items
```

你也可以用下标语法来改变特定键所关联的值。

```
airports["LHR"] = "London Heathrow"
// the value for "LHR" has been changed to "London Heathrow"
```

使用字典的 updateValue（forKey：）作为下标的另一种方式，设定或更新特殊键值。如上述下标实例，updateValue（forKey：）方法，如果键不存在就设定一个键值，如果存在一个键就更新它的值。但是不同于下标，updateValue（forKey：）方法，在执行更新操作后会返回旧值。 这让你能够检查更新是否发生。

updateValue（forKey：）方法可以返回字典值类型的可选值。例如，对于存储 String 值的字典来说，该操作返回 String？值或者“可选 String”如果值存在且尚未更新，则该可选值包括该键的旧值；如果值不存在就是 nil。

```
if let oldValue = airports.updateValue("Dublin International", forKey：  "DUB") {
println("The old value for DUB was \(oldValue).")
}
// prints "The old value for DUB was Dublin."
```

你还可以使用下标语法从特定键的字典中来检索值。如果有可能请求值不存在的键的话，字典的下标会返回字典值类型的可选值。如果字典包含请求键的值，下标会返回包含该键已有值的可选值。否则，下标会返回 nil：

```
if let airportName = airports["DUB"] {
println("The name of the airport is \(airportName).")
} else {
println("That airport is not in the airports dictionary.")
}
// prints "The name of the airport is Dublin International."
```

你可以使用下标语法，通过为该键赋一个 nil 值从字典中移除一个键值对。

```
airports["APL"] = "Apple International"
// "Apple International" is not the real airport for APL, so delete it
airports["APL"] = nil
// APL has now been removed from the dictionary
```

或者，通过移除 removeValueForKeyd 的方法来移除一个键值对。如果值存在，该方法会移除键值对；如果该值不存在，则返回移除的值或 nil。

```
if let removedValue = airports.removeValueForKey("DUB") {
println("The removed airport's name is \(removedValue).")
} else {
```

```
println("The airports dictionary does not contain a value for DUB.")
}
// prints "The removed airport's name is Dublin International."
```

4.9 迭代字典

你可以用 for-in 循环在字典中迭代键值对。字典的每个项都被还原成（key，value）元组，并且作为迭代的一部分，你可以将元组成员分解成临时常量或变量

```
for (airportCode, airportName) in airports {
println("\(airportCode)：  \(airportName)")
}
// TYO：  Tokyo
// LHR：  London Heathrow
```

欲了解更多 for-in 循环相关信息，请参阅 For Loops。

你也可以通过访问 keys 和 values 属性来检索字典的键和值的迭代集合。

```
for airportCode in airports.keys {
println("Airport code：  \(airportCode)")
}
// Airport code：  TYO
// Airport code：  LHR
for airportName in airports.values {
println("Airport name：  \(airportName)")
}
Airport name：  Tokyo
Airport name：  London Heathrow
```

如果你需要通过带有数组实例的 API 来使用字典的键和值，用 keys 和 values 属性初始化一个新数组。

```
let airportCodes = Array(airports.keys)

// airportCodes is ["TYO", "LHR"]

let airportNames = Array(airports.values)

// airportNames is ["Tokyo", "London Heathrow"]
```

注

Swift 的 Dictionary 类型为无序集合类型。当迭代没有特别说明的字典时，需要检索键、值和键值对的顺序。

4.10 创建一个空字典

至于数组，可以通过初始化器语法来创建一个指定类型的 Dictionary。

```
var namesOfIntegers = Dictionary<Int, String>()

// namesOfIntegers is an empty Dictionary<Int, String>
```

该例子创建了一部 Int 及 String 型空字典，以存储整型值可读名字。其键为 Int 类型，其值为 String 类型。

如果上下文已经提供了类型信息，可以用空字典常值来创建空字典，写为[：]（方括号中加一个冒号）

```
namesOfIntegers[16] = "sixteen"

// namesOfIntegers now contains 1 key-value pair

namesOfIntegers = [：]

// namesOfIntegers is once again an empty dictionary of type Int, String

N
```

注

实际上，Swift 数组和字典类型常作为泛型集合执行。欲了解更多泛型类型和集合的相关信息，请参见泛型。

4.11 可变性集合

一个单一集合中的数组和字典可以存储多个值。如果你创建一个数组或者一部字典并赋值给变量，创建的集合就会是可变的。这意味着，通过添加更多的项到集合中，或者从它已经包含的项中移除现有的项，创建这个集合之后，可以改变（或突变）它的大小。相反的，如果你赋值一个数组或者一部字典给一个常量，那么该数组或者字典就是不可变的，也不能改变它的大小。

至于字典，不可变也意味着你不能替换字典中现有键的值一部不可变字典的内容一旦设置就不可更改。

但不可变对数组来说略有不同。你不能采取任何有可能改变不可变数组大小的行为，但你可以设置数组中现存索引的新值。这使得数组大小固定的时候，Swift 数组类型可以为数组运算提供最佳功能

Swift 数组类型的可变行为也影响数组实例的赋值和修改。更多信息可参考集合类型的配置与拷贝行为。

注

在不需要改变集合大小的情况下，创建不可变集合是一种很好的做法。这种做法可使 Swift 编译器将你创建的集合性能最优化。

4.12 流量控制

Swift 提供所有类似于 C 语言的流量控制概念。这些包括 for 和 while 循环中运行任务多次，if 和 switch 语句在一定条件下执行代码的不同分支；；以及像 break 和 continue 语句转换执行流量到代码的另外一点。

除了在 C 语言中发现的传统的 for-condition-increment 循环, Swift 增加了 for-in 循环,使它容易迭代 arrays、 dictionaries、 ranges、 strings 和其他序列。

Swift 的 switch 语句也比在 C 语言中的副本更加强大。在 switch 语句中的 case 语句不会“落入”到下一个 Swift case 语句，避免了由于丢失 break 语句而导致的一般 C 语言错误。这些 cases 语句可以与许多不同类型的模式匹配,包括范围匹配、元组和特定类型的投射。在 switch case 语句中匹配的值可以在整个 case 语

句中绑定到临时使用的常量或变量,而复杂的匹配条件可以表示为每个 case 语句的 where 子句。

4.13 For 循环语句

For 循环语句多次执行一组语句。Swift 提供了两种循环语句：

For-in 执行范围、序列、集合或级数等每个项目中的一组语句。

For-condition-increment 在满足一个特定的条件前执行一组语句,通常是通过每一次循环结束时递增一个计数器。

4.14 For-In

使用 for-in 循环来迭代集合中的项目,如数组的范围,在数组中的项目,或字符串中的字符。

下例打印 5 乘法表的前几个条目：

```
for index in 1...5 {
println("\(index) times 5 is \(index * 5)")
}
// 1 times 5 is 5
// 2 times 5 is 10
// 3 times 5 is 15
// 4 times 5 is 20
// 5 times 5 is 25
```

被迭代集合的项一个封闭的范围内 1 到 5 的数字,如使用封闭范围的操作符表示的(...)。索引值被设置为在范围(1)内的第一个数字, 执行循环语句。在这种情况下,循环只包含一种语句,它为索引当前值从 five-times- table 中输出一个条目。语句执行后,索引值被更新以包含范围(2)内的第二个值,并再次调用 println 函数。持续执行本程序，直到该范围的末尾。

在上面的例子中,索引是一个常数,它的值是在每次迭代循环开始时自动设置的。因此，在使用之前，无需声明索引值。在循环声明中，其内容简明扼要隐式的声明了这一点,而不需要 let 声明的关键词。

注

索引常数只存在循环体范围内。如果你想检查循环完成后的索引值,或者如果你想将它的值作为一个变量使用,而不是一个常量,你必须在循环使用之前自己声明。

如果你不需要此范围内的每一个值,您可以通过在变量名位置处用下划线忽略其值:

```
let base = 3
let power = 10
var answer = 1
for _ in 1...power {
answer *= base
}
println("\(base) to the power of \(power) is \(answer)")
// prints "3 to the power of 10 is 59049"
```

这个例子计算一个数字对于另一个数字能量的价值(在这个案例中,3 对 10 的能量)。它用 3 乘以起始值 1(那是 3 对 0 的力量),十倍,使用从 0 到 9 的半封闭循环。这个计算不需要知道每通过一次循环的各个计数器值——它只需要循环执行正确的次数。下划线字符_(用于循环变量的地方)使单个的值被忽视以及不提供访问每个迭代循环的当前值。

使用 for-in 循环一个数组进行逐项迭代:

```
let names = ["Anna", "Alex", "Brian", "Jack"]
for name in names {
println("Hello, \(name)!")
}
// Hello, Anna!
// Hello, Alex!
```

```
// Hello, Brian!

// Hello, Jack!
```

您也可以迭代字典以访问其键值对。当 dictionary 被迭代时，dictionary 中的每一项作为一个元组被返回(键，值),并且在 for-in 循环体中，你可以分解(键，值)元组的成员明确命名使用的常量。在这里,dictionary 的键分解为一个称为 animalName 的常数,并且 dictionary 的值分解为一个称为 legCount 的常数：

```
let numberOfLegs = ["spider"： 8, "ant"： 6, "cat"： 4]

for (animalName, legCount) in numberOfLegs {

println("\(animalName)s have \(legCount) legs")

}

// spiders have 8 legs

// ants have 6 legs

// cats have 4 legs
```

在一个 Dictionary 中的项不一定是按照键入的顺序迭代的。一个 Dictionary 的内容本质 上是无序的,并且迭代它们并不能保证他们将被检索的顺序。欲了解更多关于数组和的详细信息，请参阅集合类型。)

除了数组 和 dictionaries 外,您还可以使用 for-in 循环迭代字符串中的字符值：

```
for character in "Hello" {

println(character)

}

// H

// e

// l

// l
```

```
// o
```

4.15 For-条件-递增

除了 for-in 循环, Swift 还支持带有条件和增量器的传统 C 语言风格的 for 循环：

```
for var index = 0; index < 3; ++index {
println("index is \(index)")
}
// index is 0
// index is 1
// index is 2
```

循环格式的一般形式为：

```
for initialization ; condition ; increment {
statements
}
```

分号将循环定义的三个部分分开,跟 C 语言一样 .然而,与 C 语言不同的是, Swift 不需要括号将整个“初始化;条件;增量”的代码块包起来。

循环按照下面流程执行：

1. 循环第一次进入时,初始化表达式计算一次，设置好循环所需的常量或变量。

2. 计算条件表达式。如果计算结果为假,循环终止,并且继续执行 for 循环尾括号(})后面的代码。如果计算结果为真,则执行循环体大括号内的代码。

3. 在执行完所有语句后，再计算增量表达式。计数器的值可能会增加或减少,或基于语句执行的结果将初始化的变量设定为一个新值。计算完增量表达式,执行返回到步骤 2,并且表达条件再一次被计算。

上面描述的循环格式和执行过程可以简略为以下概述(或相当于)：

```
initialization
```

```
while condition {
statements
increment
}
```

常量和变量在初始化表达式中的声明(比如 var 指数只是在 for 循环本身范围内有效。要检索循环结束后 index 最终的值,必须在循环开始之前声明:

```
var index:  Int
for index = 0; index < 3; ++index {
println("index is \(index)")
}
// index is 0
// index is 1
// index is 2
println("The loop statements were executed \(index) times")
// prints "The loop statements were executed 3 times"
```

注:循环完成后 index 的最终值是 3,而不是 2。最后一次执行增量表达式调用了+ +index,把 index 设为 3,导致 index< 3 等于假,循环结束。

4.16 While 循环

while 循环在条件变为 false 前执行一组语句。这些类型的循环最好使用在第一个迭代开始前并不知道迭代器的数字的时候。Swift 提供了两种 while 循环:

4.17 While

一个 while 循环开始于计算单个的条件。如果条件为真,则一组语句将重复执行直到条件变为假。

while 循环的一般式为:

```
while condition {

statements

}
```

比如，玩这个名为简单游戏（又称）：详见**错误！未找到引用源。**4-1

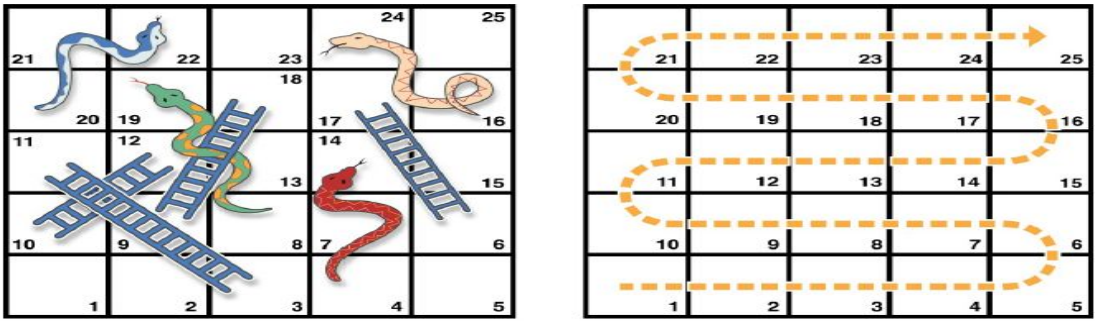


图 4-1 简单游戏

游戏的规则如下：

游戏版上有 25 个方块，其目的是落在 25 个方块上或者之外。

每一轮中，用户摇六面骰子，根据摇出来的数字移动，根据其上带点的箭头沿水平方向。

如果用户最后停留在梯子的底部，可移到前一步。

如果用户最后停留在蛇形的头部，可将其移至底部。

这个游戏板由一个 Int 值数组表示出来。其大小是基于一个称为 finalSquare 的常数,这个常数用来初始化数组并在之后检查胜利条件。该游戏板用 26 个值为零的 Int 初始化,不是 25 个(分别位于 0 到 25 索引)：

```
let finalSquare = 25

var board = Int[](count：  finalSquare + 1, repeatedValue：  0)
```

然后，一些方块给蛇与梯子设有更具体的值。在游戏板里，你能在带有方块和梯子基的地方向上移动正数,而在有蛇头的地方你只能向下移动负数：

```
board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02

board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
```

方块 3 有梯子的底部,让你移动至方块 11。为了表述这个动作,board[03]等于+08,相当于一个整数值 8(即 3 和 11 的差)。一元运算符加运算符(+)与一元减运算符(-i),小于 10 的数字用零替代,这样所有板上的定义就一致了。

(风格调整不是必需的,但会使代码更加简洁。)

玩家开始于方块 0,就在游戏板左下角的外面。首次掷骰子是把玩家带到游戏板里面去:

```
var square = 0
var diceRoll = 0
while square < finalSquare {
// roll the dice
if ++diceRoll == 7 { diceRoll = 1 }
// move by the rolled amount
square += diceRoll
if square < board.count {
// if we're still on the board, move up or down for a snake or a ladder
square += board[square]
("Game over!")
```

本示例使用一个非常简单的方法来掷骰。它始于 diceRoll 0 值,而不是一个随机数字发生器。每一次 while 循环,diceRoll 通过自加运算符(++)递增,然后检查是否过大。++ diceRoll 的返回值等于 diceRoll 自加以后的值。如果这个返回值等于 7,骰子值则过大,将其值重置为 1。这样,diceRoll 的值将总是 1,2,3,4,5,6,1,2 等等。

掷骰后,玩家根据 diceRol 移动前进。有可能骰子的数会让玩家超过方块 25,游戏就结束了。应对这种情况下,代码在将存储在 board 中的值添加到当前的 square 值之前先检查 square 的值是否少于 board 数组 count 属性,如果是,将玩家向上或向下移动到相应梯子或蛇。

这项检查如果没有被执行,board 可能试图取得 board 数组范围以外的值,这将会引发一个错误。如果

square 现在等于 26,代码将试图检查 board[26]的值,这个值超过了数组限制。

当前 while 循环执行结束后，会检查循环的条件以查看循环是否可以再次执行。如果玩家已经移到或移出方块 25,循环的条件会计算结果为假,游戏结束。

在这种情况下使用 while 循环是适当的,因为游戏的长度在 while 循环开始时是不清楚。让循环一直执行直到特定的满足条件出现。

4.18 Do-While 循环

while 循环的另一个形式为 do-while 循环,在考虑循环的条件前，先执行一次整个循环块。然后，继续重复循环，直到条件为假止。

下面是 do-while 循环的一般式为：

```
do {
statements
} while condition
```

再以为例，由 while 循环改写成 do-while 循环。FinalSquare, board, square, and diceRoll 的值与 while 循环以完全相同的方式初始化：

```
let finalSquare = 25
var board = Int[](count:  finalSquare + 1, repeatedValue:  0)
board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
var square = 0
var diceRoll = 0
```

在这个版本的游戏中，循环的第一步操作是检查一个梯子或蛇。在游戏板中没有梯子将玩家直接带到方块 25,所以只移动一个梯子赢得这场比赛是不可能的。因此，在本循环中首先检查蛇或梯子会更安全。

游戏一开始,玩家在“方块 0”。board[0]总是等于 0,没有别的功能：

```
do {
```

```
// move up or down for a snake or ladder
square += board[square]
// roll the dice
if ++diceRoll == 7 { diceRoll = 1 }
// move by the rolled amount
square += diceRoll
} while square < finalSquare
println("Game over!")
```

代码检查蛇和梯子后,开始掷骰子,玩家通过 diceRol 方块前进。然后终止当前循环。

循环的条件(虽然 square< finalSquare)和之前是一样的,但是这一次会在第一次循环的结尾才计算。前一个示例中的 do-while 循环的结构比 while 循环更适合本例游戏。在上面的 do-while 循环中, while 循环条件确认 square 仍在游戏板里面后，立即执行。不需要进行早期版本中游戏的数组越界的检查。

4.19 条件语句

通常，根据特定条件，需要执行不同的代码块。当错误发生时，您可能希望运行一个额外的代码,或当一个值变得过高或过低时，显示一个消息。要做到这一点，你需要编写条件代码。

Swift 提供了两种方法来给代码增加条件分支,称为 if 语句和 switch 语句。通常,使用 if 语句来计算只有只有少量分支的条件。Switch 语句更适合于具有多个可能排列的更复杂的条件,并且在模式匹配的情况下可以帮助选择一个适当的代码分支来执行。

4.20 If

在其最简单的形式中，if 语句仅有一个 if 条件。只有 if 条件为真时，if 语句才可执行所有语句：

```
var temperatureInFahrenheit = 30
if temperatureInFahrenheit <= 32 {
println("It's very cold. Consider wearing a scarf.")
}
```

```
// prints "It's very cold. Consider wearing a scarf."
```

前面的示例检查温度是否小于或等于 32 华氏度(水的冰点)。如果为真，则打印消息。否则不打印消息,并且继续执行 if 语句大括号之后的代码。

if 语句可以提供一套二选一的语句,称为,用于当 if 条件是假的时候。由 else 关键字控制这些语句：

```
temperatureInFahrenheit = 40
if temperatureInFahrenheit <= 32 {
println("It's very cold. Consider wearing a scarf.")
} else {
println("It's not that cold. Wear a t-shirt.")
}
// prints "It's not that cold. Wear a t-shirt."
```

始终执行其中一个分支语句。因为温度已经增加到 40 度,没有冷到要戴着一条围巾,所以 else 分支被触发。

若考虑增加子句，你可以链接多个 if 语句：

```
temperatureInFahrenheit = 90
if temperatureInFahrenheit <= 32 {
println("It's very cold. Consider wearing a scarf.")
} else if temperatureInFahrenheit >= 86 {
println("It's really warm. Don't forget to wear sunscreen.")
} else {
println("It's not that cold. Wear a t-shirt.")
}
// prints "It's really warm. Don't forget to wear sunscreen."
```

这里添加的if语句用以应对极端炎热的情况。最后的else子句保留,打印温度既不太热也不太冷的情况。

然而,最后 else 子句是可选的,并且如果不需要写完整就可以去掉:

```
temperatureInFahrenheit = 72

if temperatureInFahrenheit <= 32 {

println("It's very cold. Consider wearing a scarf.")

} else if temperatureInFahrenheit >= 86 {

println("It's really warm. Don't forget to wear sunscreen.")

}
```

在这个例子中,温度既不太冷也不太热才能引发 if 语句或 else 条件来打印消息。

4.21 Switch

Switch 语句用一个值来匹配相对应的几个匹配模式。然后，基于一开始成功匹配的模式，执行一个相对应的代码块。一个 switch 语句为了应对多个潜在情况，提供给 if 语句一个备用项。

在最简单的形式中，switch 语句将一个值与一个或多个相同类型的值比较：

```
switch some value to consider {

case value 1 ：

respond to value 1

case value 2 ,

value 3 ：

respond to value 2 or 3

default：

otherwise, do something else

}
```

每一个 switch 语句由多个可能的 case(情况)组成,其中都用 case 关键字开头。除了比较特定的值,Swift

为每种 case 提供几种方法来以应对更复杂的匹配模式。这些方法将在本节后面介绍。

每个 switch case 的主体是代码执行的一个独立的分支,与 if 语句的分支方式相似。由 switch 语句决定选择哪个分支语句。被称为所考虑的切换值。

每一个 switch 语句必须详细。也就是说,每一个所考虑类型可能的值必须与 switch 中的一个 case 相匹配。如果不能为每一个可能的值提供一个 switch 中的 case 值,可以定义一个默认选取器来解决。选取器用 dafault 关键字来表示，并且应始终出现在最后。

这个示例使用一个 switch 语句来考虑一个称为 someCharacter 的小写字符：

```
let someCharacter：  Character = "e"

switch someCharacter {

case "a", "e", "i", "o", "u"：

println("\(someCharacter) is a vowel")

case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",

"n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z"：

println("\(someCharacter) is a consonant")

default：

println("\(someCharacter) is not a vowel or a consonant")

prints "e is a vowel"
```

Switch 语句第一条 case 可匹配了五个小写元音字母。同样，其第二个 case 匹配所有小写辅音字母。

Switch 中 Case 中写上所有其他可能的英文字母，使其成为 case 的一部分，这是不实际的,所以这 switch 语句提供了一个 default case 以匹配其他所有不是元音或辅音的字母。这样保证了 switch 语句没有遗漏。

4.22 无隐式贯穿

相比 C 和 objective - C 中的 switch 语句,Swift 中的 switch 语句不会默认的掉落到每个 case 的下面进入另一个 case.相反,第一个匹配的 switch 语句当第一个匹配的 case 一完成, 就完成了它整个的执行。而不需要一个明确的 break 语句。这使得 switch 语句比在 C 语言中使用更安全、更简单,并避免错误地执行多个 case。

注

如果需要，你仍然可以在 case 完成其执行前中断已匹配的 switchcase

详情见打破 Switch 语句

每个 case 语句体必须至少包含一个可执行语句。如果编写以下代码，则本程序无效，因为第一个 case 为空：

```
let anotherCharacter： Character = "a"
switch anotherCharacter {
case "a"：
case "A"：
println("The letter A")
default：
println("Not the letter A")
}
// this will report a compile-time error
```

不像 C 语言中的 switch 语句，此 switch 语句不能匹配“a”和“A”。相反，在编译时会把 case“a”报错“a”：不包含任何可执行语句。这种方法避免了意外从一个 case 掉入另一个,并且使代码更安全,目的更清晰。

如果列表很长，一个 switch case 有多个匹配对象的，可以由逗号分隔,并可以编写多行：

```
switch some value to consider {
case value 1 ,
value 2 ：
statements
}
```

注

为一个特定的 switch case 选择掉落行为,使用 fallthrough 关键字,如落空中详解。

4.23 范围匹配

Switch 中 case 的值可检查它们内在的范围。本示例使用数值范围可以提供任意大小数字的自然语言计数：详见图 4-2

```
let count = 3_000_000_000_000
let countedThings = "stars in the Milky Way"
var naturalCount： String
switch count {
case 0：
naturalCount = "no"
case 1...3：
naturalCount = "a few"
case 4...9：
naturalCount = "several"
10...99：
naturalCount = "tens of"
100...999：
naturalCount = "hundreds of"
1000...999_999：
naturalCount = "thousands of"
```

```
default:
naturalCount = "millions and millions of"
("There are \(naturalCount) \(countedThings).")
prints "There are millions and millions of stars in the Milky Way."
```

4.24 元组

你可以使用元组测试同一个 switch 语句内的测试多个数值。可以通过不同的数值或一系列的数值测试元组中的每个元素。或者，使用下划线（）标识符匹配任何可能值。

下面的例子使用一个点坐标(x,y),表示为一个简单的元组型(Int,Int),并在示例后面的图中将其分类：详见图 4-3

```
let somePoint = (1, 1)
switch somePoint {
case (0, 0):
println("(0, 0) is at the origin")
case (_, 0):
println("(\(somePoint.0), 0) is on the x-axis")
case (0, _):
println("(0, \(somePoint.1)) is on the y-axis")
case (-2...2, -2...2):
println("(\(somePoint.0), \(somePoint.1)) is inside the box")
default:
println("(\(somePoint.0), \(somePoint.1)) is outside of the box")
prints "(1, 1) is inside the box"
```

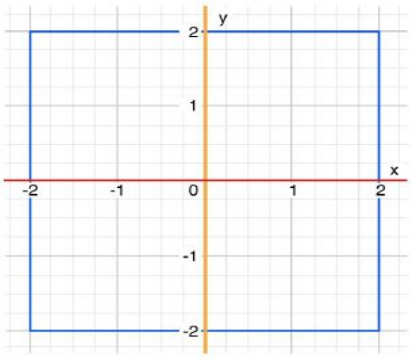


图 4-2 自然语言计数

switch 语句决定点是否在原点(0,0)上，在红色 x 轴上;在橙色 y 轴上，在蓝色 4×4 的矩形为中心的原点内,还是在矩形的外面。

与 C 语言不同的，Swift 可允许多个 switch 的 case 考虑为相同值或多值。事实上，在本示例中，点（0,0）能匹配例子中所有的四个 case.但是，如果可能存在多个匹配，通常使用第一个匹配成功的 case.点(0,0)将首先匹配 case(0,0),所以其他所有能匹配的 case 将会被忽略。

4.25 值绑定

一个 switch 的 case 可以将值或匹配值与临时常量或变量绑定,用于 case 的主体。这就是所谓的值绑定,因为这些值是在 case 主体中“绑定”到临时常量或变量的。

下面的例子有一个(x,y)点,表示为一个简单的元组型(Int,Int),并在下面的图中将其分类：

```
let anotherPoint = (2, 0)
switch anotherPoint {
case (let x, 0):
println("on the x-axis with an x value of \(x)")
case (0, let y):
println("on the y-axis with a y value of \(y)")
case let (x, y):
```

```
println("somewhere else at (\(x), \(y))")
}
prints "on the x-axis with an x value of 2"
```

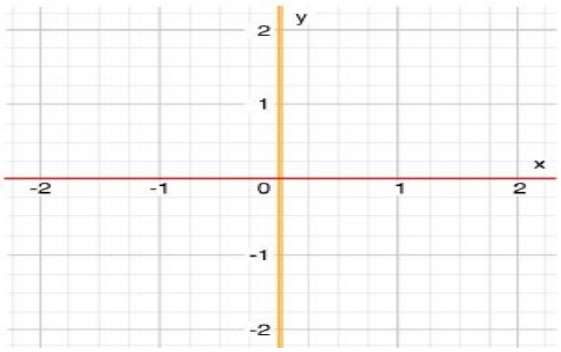


图 4-3 自然语言计数

switch 语句决定点是在红色的 x 轴上,橙色 y 轴上,或其他地方。

三个 switch 的 case 声明占位符常量 x 和 y,它暂时从 anotherPoint 占有一个或两个元组值。第一种 case 里,case(let x,0),匹配 y 值为 0 的任何点，并且将点的 x 值赋值到临时常数 x。同样的,第二种 case 里,case(0,let y),匹配 x 值为 0 的任何点并且将点的 y 值赋值到临时的常数 y。

一旦临时常量被声明，它们可以在 case 的代码块中使用。在这里，它们被用作采用 println 函数打印各值的简写形式。

请注，这个 switch 语句没有 default 的 case.最后一个 case, case let(x,y),声明两个占位符常量可以匹配任何值的一个元组。因此,它匹配所有可能的剩余值,不需要 default 语句 switch 就很详尽了。

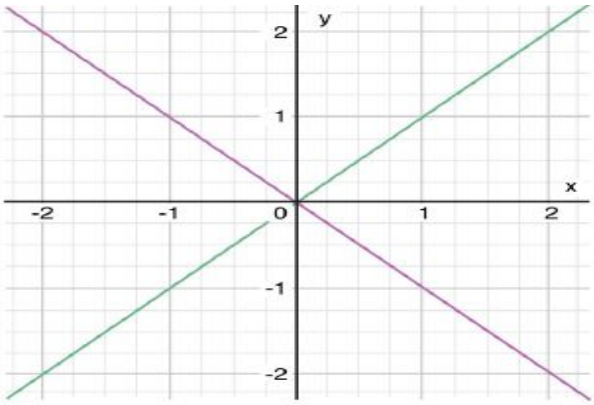
在上面的示例中, 声明 x 和 y 为常量用了 let 关键字,因为在 case 的主体中不需要修改他们的值。但是，它们可以使用 var 关键字，声明由变量来替代。如果用了变量,将创建一个临时变量，并使用适当的值将其初始化。该变量的任何改变只会在该 case 主体内产生影响。

4.26 Where

Switch 的 case 能使用 where 子句来检查其他条件。

下面的例子将点（x，y）在下图中分类：

```
let yetAnotherPoint = (1, -1)

switch yetAnotherPoint {

case let (x, y) where x == y:

println("(\(x), \(y)) is on the line x == y")

case let (x, y) where x == -y:

println("(\(x), \(y)) is on the line x == -y")

case let (x, y):

println("(\(x), \(y)) is just some arbitrary point")

}

prints "(1, -1) is on the line x == -y"
```



switch 语句决定点是在 x = = y 的绿色对角线上,在 x = = - y 的紫色对角线上,或都不是。

三个 switch 的 case 声明了占位符常量 x 和 y,它暂时占有两个元组值。这些常量作为一个 where 子句的一部分，用来创建一个动态筛选。只有在 where 子句的条件被判断为真值时，switch 的 case 则会匹配现有 point 的值。

因此,就像在前一个例子中，最后的 case 匹配所有可能的剩余值,并且不需要 default 语句 switch 就很详尽了。

4.27 控制转移语句

控制转移语句能改变已执行代码的顺序,通过从一个代码块转移到另一个来控制。Swift 有四个控制转移语句：

继续

control, break 和 fallthrough 语句在下文中详解。return 语句在函数中描述。

5 Continue

Continue 语句告诉一个循环停止正在做的事情,在下一个迭代循环开始时再一次开始。好像在说“我这次迭代做完了”，总之不会脱离循环体。

> 注
>
> 在一个 for-condition-increment 循环中,在调用 continue 语句后增量器仍然会计算。循环本身仍然照常运行;只有循环体中的代码被跳过。

下面的例子将所有元音和空格从一个小写字符串移除，来创建一个神秘的谜题短语：

```
let puzzleInput = "great minds think alike"
var puzzleOutput = ""
for character in puzzleInput {
switch character {
case "a", "e", "i", "o", "u", " ":
continue
default:
```

```
puzzleOutput += character
}
continue
break
fallthrough
return
(puzzleOutput)
prints "grtmndsthnklk"
```

上面的代码里，无论何时只要它匹配一个元音字母或空格,就会调用 continue 关键字，导致当前迭代循环立即结束，然后直接跳转到下一个迭代的开头。这种方式使 switch 块只匹配(忽视)元音字母和空格,而不是要求 switch 块把每一个要打印字符都匹配一次。

5.1 Break

break 语句能立即结束一个完整的控制流语句的执行。当你想提前终止 switch 或循环语句的执行时，在一个 switch 语句或循环语句中可以使用 break 语句。

5.2 循环语句中的 break 语句

当 break 语句使用在一个循环语句时,会立即终止循环语句的执行,并且将控制转移到循环语句的括号(})后第一行的代码里。当前迭代循环中，其他的代码不会被执行,并且不会开始下一次迭代循环。

5.3 语句中的 break 语句

当在 switch 语句内部使用 break 语句时,break 语句导致 switch 语句立即结束其执行,并且将控制转移到 switch 语句括号(})后第一行代码里。

这种形式可用来匹配、忽略 switch 语句中的一个或多个。因为 Swift 的 switch 语句是穷举的，并且不允许空 case,,有时为了使意图明确，有必要慎重匹配和忽略 case。可以通过编写 break 语句，作为你想要忽略的 case 的主体部分来实现这一点。 当那种 case 与 switch 语句匹配时,case 内部的 break 语句能直接终止 switch 语句的执行。

注

一个 switch 的 case 只包含一个被看做是编译错误的注释。注释不是语句，不会导致 switch 的 case 被忽略。 始终使用 break 语句忽略 switch 的 case。

下面的例子中 switch 有一个字符值，并确定它是否代表四种语言之一的一个数字符号。为了简洁起见，一个 switch 的 case 包含了多个值：

```
let numberSymbol： Character = "三" // Simplified Chinese for the number 3
var possibleIntegerValue： Int?
switch numberSymbol {
case "1", "١", "一", " ":
possibleIntegerValue = 1
case "2", "٢", "二", " ":
possibleIntegerValue = 2
case "3", "٣", "三", " ":
possibleIntegerValue = 3
"4", "٤", "四", " ":
possibleIntegerValue = 4
default:
break
integerValue = possibleIntegerValue {
println("The integer value of \(numberSymbol) is \(integerValue).")
{
println("An integer value could not be found for \(numberSymbol).")
```

```
prints "The integer value of 三 is 3."
```

这个示例通过检查来确定是否为拉丁,阿拉伯语,中文,或泰文符号来得到 1 至 4 的数字。如果找到匹配项,其中的一个 case 里会赋给可选型的 Int 变量一个适当的整数值。

switch 语句完成其执行后,示例使用可选型绑定来确定是否找已到一个值。变量 possibleIntegerValue 有一个隐型的初始值 nil,具有可选型的优点,所以只有在前四个 case 中，当 possibleIntegerValue 被 switch 语句赋予了实际的值时，可选绑定才会成功。

在上面的例子中列出所有可能的字符值不太实际,因此 default case 为不匹配的任何字符提供了一个总受器。这个 default case 不需要执行任何操作,所以它用单个 break 语句作为它的主体。一旦 default 语句被匹配,break 语句终结 switch 语句的执行,并且继续执行 if let 语句。

5.4 Fallthrough

Swift 中的 switch 语句不会掉到 case 的底部进入到下一个 case.相反,在完成第一个 case 匹配后，整个 switch 语句尽快完成其执行。相比之下,C 语言需要你在每个 switch 的 case 结尾处插入一个明确的 break 语句来防止 fallthrough。相比于 C 语言，避免默认 fallthrough 意味着 Swift 的 switch 语句更简洁且可预测,因而他们避免错误的执行多个 switch case.

如果你真的需要 c 语言式的的 fallthrough 特性,你可以使用 case-by-case 基础上选择此种特性。下面的示例使用 fallthrough 创建了一些描述数字的文本：

```
let integerToDescribe = 5

var description = "The number \(integerToDescribe) is"

switch integerToDescribe {

case 2, 3, 5, 7, 11, 13, 17, 19:

description += " a prime number, and also"

fallthrough

default:

description += " an integer."

}
```

(description)

prints "The number 5 is a prime number, and also an integer."

此示例声明了名为 descripton 的新的字符串变量，并赋予其一个初始值。然后，该函数考虑了使用 switch 语句匹配 integerToDescribe 的值。如果值在列表中是一个素数,该函数将文本附加到的结尾,要注数字都是素数。然后，同样使用 fallthrough 关键字让代码“掉到”default case 中。Default case 在最后增加了一些额外的文本,switch 语句结束。

如果的值不是列表中不是已知的素数,它根本就不会匹配 switch 的第一个 case.没有其他特殊的 case,所以相匹配。

Switch 语句执行完成后,使用 println 函数将数字描述打印出来。在这个例子中，数字 5 被正确地判定为一个素数。

注

Fallthrough 关键字不检查 case 里的条件，因为 switch case 会导致执行失败。Fallthrough 关键字简单的让代码直接执行到下一个 case(或 default)代码块中,和标准中 C 语言的 switch 语句特性一样。

5.5 标签语句

在 Swift 中，在其他循环和 switch 语句内部，您可以嵌套循环和 switch 语句，来创建复杂的控制流结构。然而,循环和 switch 语句都可以使用 break 语句过早地结束他们的执行。因此,它有时候对于明确 break 语句终止哪一个循环或 switch 语句是有用的。同样的,如果有多个嵌套循环,它对于明确 continue 语句影响哪一个循环是有用的。

为了实现这些目标,您可以用标记循环语句和 switch 语句,与 break 语句或 continue 语句一起使用这个标签来结束或继续标记语句的执行。

标记语句作为引导关键词在标签的同一行来作出指示,后跟一个冒号。这里有一个用了此种语法的 while 循环例子,尽管对于所有的循环和 switch 语句，规则是一样的：

label name ： while condition {

statements

}

下面的例子有一个带有标签 while 循环，使用了的 break 和 continue 语句。是你在上文中看到的游戏的改编版本。这一次，游戏包含一条附加规则：

如果一个特定的 dice roll 将带你超越方块，骰子必须投掷在方块 25。

游戏板和上文中是一样的：

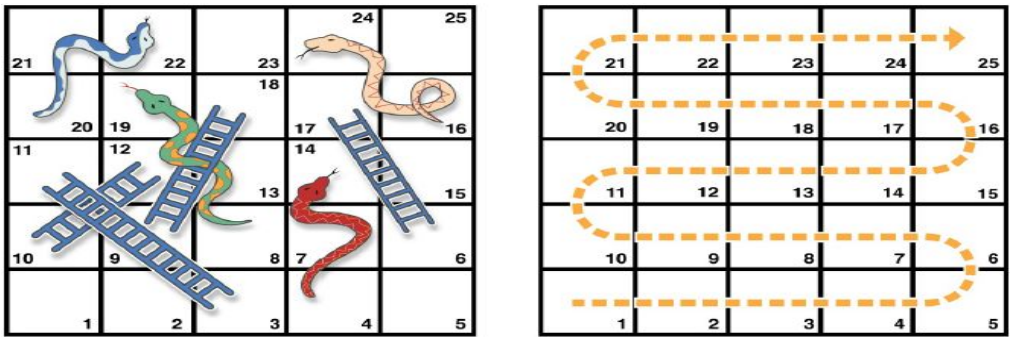


图 5-1 简单游戏的改编版本

FinalSquare, board, square, and diceRoll 值的初始化与前文一样：

```
let finalSquare = 25
var board = Int[](count:  finalSquare + 1, repeatedValue:  0)
board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
var square = 0
var diceRoll = 0
```

这个版本使用一个 while 循环语句和一个 switch 语句来实现游戏的逻辑性。While 循环有一个称为 gameLoop 的语句标签,表明它是游戏主要的游戏循环。

While 循环的条件是 while square! = finalSquare,意思是您必须准确地落到 25 号方块上：

```
gameLoop:  while square != finalSquare {
```

To win, you must land exactly on square 25.

```
if ++diceRoll == 7 { diceRoll = 1 }
switch square + diceRoll {
case finalSquare:
// diceRoll will move us to the final square, so the game is over
break gameLoop
case let newSquare where newSquare > finalSquare:
// diceRoll will move us beyond the final square, so roll again
continue gameLoop
default:
// this is a valid move, so find out its effect
square += diceRoll
square += board[square]
("Game over!")
```

在每一个循环语句的开始，摇动骰子。不立即移动玩家, switch 语句用于判断移动的结果,如果此举是被允许的，便移动：
如果玩家摇骰子于最后的方块上停下来, 游戏便结束。该打破的 gameLoop 语句传输控制到编码的第一排循环外, 表示游戏结束。

如果玩家摇骰子超过最后一个方块, 则此次移动无效，玩家需要再次摇骰子。继续 gameLoop 语句结束现行的 while 循环迭代并继续下一循环迭代。

在其他情况下，摇骰子是有效的移动。玩家通过 diceRoll 方块前移，游戏逻辑检查任一蛇形和梯子。此循环将结束，控制返回至 while 条件决定是否进行下一轮

注

如果上面的 break 语句中没有使用 gameLoop 标签,这将中断 switch 语句,而不是 while 语句。使用 gameLoop 标签可以明确让哪个控制流终止。

还请注,当调用 continue gameLoop 跳到下一个对象的迭代时,它并不一定需要使用 gameLoop 标签。在游戏中只有一个循环,所以 continue 语句不会影响其他循环体。然而,使用带有 continue 语句的 gameLoop 标签声明并没有损害。在 break 语句中,这样做符合标签的使用,并有助于让游戏逻辑能更加清晰的阅读和理解。

6 函数

函数都是独立的代码块,各自执行特定的任务。你赋予一个函数可以标识其功能的名称,并且需要时,这个名字用于“调用”这个函数来执行其任务。

Swift 的统一函数语法足够灵活,能表达没有参数名称的简单的 C 型函数的任何东西,本地和外部复杂 Objective-C-style 方法参数名称为每个参数。一旦函数完成其执行,参数可以提供默认值来简化函数调用,并且可以作为输入输出参数来传递,修改一个传递变量。

Swift 中的每个函数都有各自的类型,包括函数的参数类型和返回类型。此类型类似于 Swift 中的任何其他类型,您可以使用此类型,使它很容易将函数作为参数传递给其他函数,并从函数返回函数。函数也可以写在其他函数中,来封装一个嵌套函数范围内的有用的功能。

6.1 定义和调用函数

当你定义一个函数时,您可以选择性地定义一个或多个命名类型值,函数作为输入(称为参数),和/或一个类型的值,当它完成后函数将作为输出返回(称为其返回类型)。

每一个函数都有一个函数名,用于描述该函数所执行的任务。使用一个函数,你用它的名字“调用”函数并将其传递给匹配函数参数类型的输入值(称为参数)。所提供的函数的参数必须与函数的参数列表顺序相同。

以下示例中的函数称为 greetingForPerson,因为这就是它的任务——它需要输入一个人的名字,并返回那个人问候语。为了实现这一点,您定义一个输入参数——一个称为 personName 的字符串值——和字符串的返回类型,它将包含那个人的问候语:

```
func sayHello(personName： String) -> String {
let greeting = "Hello, " + personName + "!"
return greeting
}
```

所有这些信息都包括到函数的定义中,是 func 关键词的前置式。您用返回箭头- >(连字符后跟一个右尖括号)指示函数返回类型，紧随其后的是返回类型名称。

该定义描述了函数的作用、它期望接受的对象以及该函数被实施后返回结果。在其他代码中，该定义可以清楚明白的调用函数。

```
println(sayHello("Anna"))
// prints "Hello, Anna!"
println(sayHello("Brian"))
// prints "Hello, Brian!"
```

您可以通过将它放在括号内，如 sayHelo（“Anna”）的 String 参数值调用 sayHelo 函数。如上所说，因为该函数返回一个字符串值，sayHelo 可以包裹在 println 函数的调用中，去打印该字符串，并返回其值。

SayHelo 函数的首先定义一个名为 greeting 的新字符串常数，将其设置成一个简单的问候消息 personName。然后，使用 return 关键字，将这个问候语传回到函数以外。只要问候被调用，函数执行完毕并返回问候语的当前值。

你可以通过输入不同的值，多次调用 sayHelo 函数。上面的例子显示了输入值分别为“Anna”和“Brain”时的结果。该函数在不同情况下都会返回一个对应的问候语句。

为了简化这个函数的主体部分，将信息创建和 return 语句合并到一行中：

```
func sayHelloAgain(personName： String) -> String {
return "Hello again, " + personName + "!"
}
println(sayHelloAgain("Anna"))
```

```
// prints "Hello again, Anna!"
```

6.2 函数参数和返回值

在 Swift 中，函数参数和返回值是非常灵活的。您可以定义任何东西，从未具名参数的简单效用函数，到具有表达参数名称和不同的参数选项的复杂函数。

6.3 多个输入参数

函数可以有多个输入参数，这些可以在函数后的括号内表示，并用逗号分开。

此函数需要一个开始和一个半开区间终点指数，并计算出该范围包含了多少个元素：

```
func halfOpenRangeLength(start： Int, end： Int) -> Int {
return end - start
}
println(halfOpenRangeLength(1, 10))
// prints "9"
```

6.4 无参数函数

不要求函数对输入参数进行定义。这是一个没有输入参数的函数，当被调用时，它总是返回相同的字符串消息。

```
func sayHelloWorld() -> String {
return "hello, world"
}
println(sayHelloWorld())
// prints "hello, world"
```

虽然该函数不采取任何参数，但是该函数名依然需要放在括弧中。当函数被调用时，函数名后要有一对空括号。

6.5 无返回值的函数

不要求函数对返回类型进行定义。这里有一个版本的 sayHelo 函数，称为 waveGoodbye，它会打印自

己的字符串值，而不是返回它：

```
func sayGoodbye(personName： String) {
println("Goodbye, \(personName)!")
}
sayGoodbye("Dave")
// prints "Goodbye, Dave!"
```

因为它并不需要返回一个值，该函数的定义不包括返回箭头（ ->）或返回类型。

注

严格地说，SayGoodbye 函数仍然返回一个值，虽然没有定义返回值。无定义的返回类型的函数会返回一个特殊类型的值 Void。这仅仅是一个空的元组，实际上是一个具有零元素的元组，可以被写作（）。

当调用一个函数时，该函数的返回值可以忽略不计：

```
func printAndCount(stringToPrint: String) -> Int {
println(stringToPrint)
return countElements(stringToPrint)
}
func printWithoutCounting(stringToPrint: String) {
printAndCount(stringToPrint)
}
printAndCount("hello, world")
// prints "hello, world" and returns a value of 12
```

```
printWithoutCounting("hello, world")

prints "hello, world" but does not return a value
```

第一个函数 printAndCount 输出一个字符串，然后以 Int 类型返回其字符数..。第二个函数，printWithoutcounting，调用了第一个函数，但忽略它的返回值。当第二函数被调用时，内容是被第一个函数打印的，但是返回值却不被采用。

注

函数返回值可以被忽略，但是如果一个函数说它将返回一个值，它必须这么做。一个定义返回类型不允许控制下的底层函数没有返回值的函数，尝试这样做将导致编译时错误。

6.6 具有多个返回值的函数

您可以使用元组类型作为函数的返回类型，返回多个值作为一个复合返回值。

下面的例子定义了一个名为 count 函数，它计算元音，辅音，和某一字符串中的其他字符，基于美国英语中使用元音和辅音的标准设定。

```
func count(string: String) -> (vowels: Int, consonants: Int, others: Int) {

var vowels = 0, consonants = 0, others = 0

for character in string {

switch String(character).lowercaseString {

case "a", "e", "i", "o", "u":

++vowels

case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",

"n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":

++consonants

default:
```

```
++others
}
return (vowels, consonants, others)
```

你可以使用这个计数函数计算任意字符串的字符，并检索统计总数为三名为 int 值的元组：

```
let total = count("some arbitrary string!")
println("\(total.vowels) vowels and \(total.consonants) consonants")
// prints "6 vowels and 13 consonants"
```

注元组的成员不需要在元组从函数中返回时就被命名，因为他们的名字已经被指定为函数的返回类型的一部分。

6.7 函数参数名

上述所有的函数都定义了其参数的参数名：

```
func someFunction(parameterName: Int) {
// function body goes here, and can use parameterName
// to refer to the argument value for that parameter
}
```

然而，这些参数名称只能在函数本身中使用，而不能在调用函数时使用。这些类型的参数名称被称为当地的参数名称,因为它们只在函数内部使用。

6.8 外部参数名称

有时候，去调用一个函数时，去命名函数每个参数是有用的，来表示每个传递给函数的每个引用的目的

如果当你的函数被调用时，你希望你的函数的使用者提供参数名称，除了函数本身的本地参数外，请准确命名每个函数的外部参数，你在它所支持的本地参数名称之前写一个外部参数名称,之间用一个空格来分隔)：

```
func someFunction(externalParameterName localParameterName: Int) {
```

```
// function body goes here, and can use localParameterName
// to refer to the argument value for that parameter
}
```

注

如果您为参数提供一个外部参数名称，外部名字必须在调用函数时使用。

例如下面的函数;在两个函数间插入第三个字符串“joiner”，以此来连接两个字符串

```
func join(s1: String, s2: String, joiner: String) -> String {
return s1 + joiner + s2
}
```

当你调用这个函数，你传递给函数的这三个字符串的目的是不清楚的

```
join("hello", "world", ", ")
// returns "hello, world"
```

为了使这些字符串的目的更清晰，为每个连接函数参数提供外部参数名称。

```
func join(string s1: String, toString s2: String, withJoiner joiner: String)
-> String {
return s1 + joiner + s2
}
```

这个版本的函数,第一个参数有一个外部名称的字符串和 s1 的本地名称,第二个参数有一个外部名称 toString 和 s2 的本地名称,第三个参数有一个外部的名字 withJoiner 和当地 joiner 的名字。

现在，您可以明确的、毫不含糊的使用这些外部参数名称调用该函数

```
join(string: "hello", toString: "world", withJoiner: ", ")
```

```
// returns "hello, world"
```

使用外部参数名称使连接函数的第二个版本更具表达力,用户习惯使用类似句子的方式,同时也提供了一个可读的、明确的函数体。

注

对于第一次阅读函数代码而不清楚函数目的时，请考虑使用外部函数名称。当函数被调用时，如果每个函数的目的是清楚的，毫不含糊的，你就不需要指定外部函数名称。

6.9 外部参数名称

如果你想为某个函数指定外部参数名称，而参数本身的名字可以恰当的表达其本身，你就不需要重复为这个参数命名。相反，参数名写一次，并用一个 hash 符号（＃）给该函数名加上前缀。这就是 Swift 将一个名字作为内部参数名称和外部参数名称的方法。

这个例子中定义了一个名为 containsCharacter 函数，它通过将一个 hash 符号，在当地的参数名称前定义了函数的两个参数的外部名称：

```
func containsCharacter(#string: String, #characterToFind: Character) -> Bool {
for character in string {
if character == characterToFind {
return true
}
}
return false
}
```

这个函数选择的参数名称是一个清晰的、可读的函数体,同时也使该函数没有什么歧义

```
let containsAVee = containsCharacter(string: "aardvark", characterToFind: "v")
```

```
// containsAVee equals true, because "aardvark" contains a "v"
```

6.10 默认参数值

可以为任何参数定义一个默认值，作为函数定义的一部分。如果定义了一个默认值，那么在调用一个函数时，该参数可以省略。

注

将使用默认值的参数放在函数的参数列表的末尾。这确保了对该函数的所有调用使用相同的非默认参数顺序,并明确在每种情况下都调用相同的函数。

这里有一个从早期版本的连接函数,它为其 jointer 参数提供了一个默认值:

```
func join(string s1: String, toString s2: String,
withJoiner joiner: String = " ") -> String {
return s1 + joiner + s2
}
```

如果在连接函数被调用时。字符串会起到连接字符的功能，像以前一样，用于连接两个字符。

```
join(string: "hello", toString: "world", withJoiner: "-")
// returns "hello-world"
```

但是，如果当函数被调用时没有提供 jointer 上午值，单个空格（“”）将会作为一个默认值来代替：

```
join(string: "hello", toString: "world")
// returns "hello world"
```

6.11 含有默认值的参数的外部名称

大多数情况下，提供（如果需要）具有缺省值的任何参数一个外部名称是有用的当函数调用提供的值时，这将确保函数的目的是明确的，

为了使这个过程更容易，如果您不能提供外部名称，Swift 自动为你提供缺省参数下的外部名称，自动外部名称与本地名称相同，就像您再在您的本地代码名称前写了一个 hash 字符。

这是一个早期功能连接的版本，它不提供任何外部名称，但是仍要提供了 jointer 参数的默认值。

```
func join(s1: String, s2: String, joiner: String = " ") -> String {
return s1 + joiner + s2
}
```

在这种情况下，Swift 将自动提供一个连接默认值作为外部函数名当调用函数时，外部名字将会被应用，进而使函数可以清楚明确的表达。

```
join("hello", "world", joiner: "-")
// returns "hello-world"
```

注

当你定义一个函数时，您可以选择用下划线来替代明确的外部名字，然而，缺省值的外部名称在适当的地方总是首选。

6.12 可变参数

一个可变参数可以接受零值或指定类型的多个值。当函数被调用时，您可以指定一个可变参数用于传递不同数量的输入值。通过插入省略号来写可变参数的类型名称。

就函数本身，可以通过可变参数值作为适当类型的数组。例如，一个数字的名称和类型...可变的参数在函数体作为常量数组称为型。

下面的例子计算任何长度的列表的算数平均（也成为平均）值

```
func arithmeticMean(numbers: Double...) -> Double {
var total: Double = 0
for number in numbers {
```

```
total += number
}
return total / Double(numbers.count)
}
arithmeticMean(1, 2, 3, 4, 5)
// returns 3.0, which is the arithmetic mean of these five numbers
arithmeticMean(3, 8, 19)
returns 10.0, which is the arithmetic mean of these three numbers
```

注

一个函数可以有至多一个 varadic 参数，它必须出现在参数列表的最后，避免当调用该函数的参数时产生异议。

如果您的函数有一个或多个参数有默认值，也有一个可变参数，请将这多个参数值放在默认参数的后面。

6.13 常量和变量参数

功能参数默认情况下均是常数。编译阶段的错误将试图去改变函数本身的函数参数。这意味着你无法错误地改变参数的值。

然而，对于一个函数，有一个参数的可变副本是有用的你可以通过在函数内指定一个或者多个变量参数，进而避免定义新的函数变量在您的函数中，可变参数可作为变量而不是常量给您的函数提供新的修改副本。

用关键字 var 为参数名添加前缀，从而对变量参数进行定义：

```
func alignRight(var string: String, count: Int, pad: Character) -> String {
let amountToPad = count - countElements(string)
```

```
for _ in 1...amountToPad {
string = pad + string
}
return string
}
let originalString = "hello"
let paddedString = alignRight(originalString, 10, "-")
paddedString is equal to "-----hello"
originalString is still equal to "hello"
```

这个例子定义了一个新功能叫做 alignRight，它对准一个较长字符的左边缘输出字符。用指定的填充字符填充左侧的任何空格。在该示例中，字符串“hello”被转换为字符串“----- hello”。

该 alignRight 函数定义了该输入参数字符串为一个可变参数。这意味着字符串现在可以作为一个局部变量，可以在函数内进行操作，使传入的字符串值初始化。

该函数首先计算有多少个字符需要被添加到字符串的左边，在字符串中以右对齐的方式出现。此值被存储在在一个名为 amountToPad 的局部常量中。该函数将填充字符的 amountToPad 拷贝到现有的字符串的左边，并返回结果。它使用其字符串变量参数实现其所有字符串操作。

注

您对一个可变函数的更改都会在调用函数结束之前进行，在函数体外时无法遇见的。可变参数只能存在于该函数调用的周期中。

6.14 In-Out 参数

如上所述，可变参数只能在函数本身内进行更改。

如果你想要一个函数来修改参数的值，并且这些函数持续到函数调用结束后，请把参数定义为输入输出参数。

您在参数定义的开始部分放一个 inout 关键字，可以写一个输入输出参数在函数传递中，一个输入输出函数有其价值，并被函数修改，并传回了函数来替换原来的值

您只可以传递一个变量，作为一个 in-out 参数的自变量。你不能传递一个常数或常值作为参数，因为常数和常值不能被修改。当你把它作为一个参数传递给一个 inout 参数，请直接在变量名前放置一个连字符（&），以表明它可以通过该功能进行修改。

注

在输出参数不能有默认值，可变参数的参数不能被标记为 INOUT。如果你将一个参数标记为 INOUT，那么，它不能同时被标记为 var 或 let 形式。

这里的一个叫做 swapTwoInts 函数的例子，它有两个被称为 a 和 b 的输入输出的整数参数。

```
func swapTwoInts(inout a: Int, inout b: Int) {
let temporaryA = a
a = b
b = temporaryA
}
```

该 swapTwoInts 函数只是简单将 b 的值转化为 a，以及从 a 到 b，该功能通过一个 temporary A 的函数对 a 的临时存储，将 b 的值传递到 a，然后将 temporaryA 和 b 进行交换。

你可以使用 ant 类型的两个变量来调用 swapTwoInts 函数。需要注的是，当传递给 swapTwoInts 函数时，someInt 和 anotherInt 的名称前缀与符号

```
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
println("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
```

```
// prints "someInt is now 107, and anotherInt is now 3"
```

上面的例子表明，someInt 和 anotherInt 的原始值由 swapTwoInts 功能修改，尽管起初定义时不在此函数范围内。

注

In-out 参数不同于从一个函数中返回一个值。上述 swapTwoInts 例子没有定义返回类型或返回一个值，但它仍然会修改 someInt 和 anotherInt 的值。输出函数是一个函数作用其功能体的范围之外的可供选择的方法。

6.15 函数类型

每一个函数都有特定的功能，用来弥补参数类型和函数的返回类型。

例如：

```
func addTwoInts(a: Int, b: Int) -> Int {
return a + b
}
func multiplyTwoInts(a: Int, b: Int) -> Int {
return a * b
}
```

这个例子中定义了两个称为 addTwoInts 和 multiplyTwoInts 的简单的数学函数这些功能需要两个 int 值，并返回一个 int 值，它是执行适当的数学运算的结果

所有这些函数的类型均为(Int, Int) -> Int。这可以理解为：

“一个函数类型含有两个参数，这两个参数均为 Int 类型的，并返回一个 Int 类型的值。”

下面是另一个例子，一个关于无参数或无返回值的函数的例子：

```
func printHelloWorld() {
```

```
println("hello, world")
}
```

这个函数的类型为（） ->（），或者“一个无参数和无返回值 Void 的函数”不指定返回值的函数总是返回 void，这相当于 Swift 中的一个空的元组，显示为（）。

6.16 使用函数类型

在 Swift 中，你可以像使用任何其他类型一样使用函数类型。例如，你可以定义一个常量或变量为一个函数类型，并指定适当的函数给该变量：

```
var mathFunction: (Int, Int) -> Int = addTwoInts
```

这可以理解为：

“定义一个名为 mathFuncton 的变量，该变量的类型为'a 函数，它接受两个 int 值，并返回一个 int 值。”设置这个新的变量来引用所调用的函数 addTwoInts。

该 addTwoInts 函数与 mathFuncton 变量具有相同的类型，所以这个赋值是被 Swift 的类型检查允许的。

现在你可以调用指定的名为 mathFuncton 的函数：

```
println("Result: \(mathFunction(2, 3))")
// prints "Result: 5"
```

一个不同的但具有相同的匹配类型的函数可以被分配给相同的变量，以和非功能类型同样的方式：

```
mathFunction = multiplyTwoInts
println("Result: \(mathFunction(2, 3))")
// prints "Result: 6"
```

与任何其他类型一样，你可以留给 Swift 去推断函数类型，当你指定一个函数给一个常量或变量时：

```
let anotherMathFunction = addTwoInts
// anotherMathFunction is inferred to be of type (Int, Int) -> Int
```

6.17 可作为参数类型的函数

您可以使用一个函数类型，如(Int, Int) -> Int，作为另一个函数的参数类型。这使您可以留下函数实现

的某些方面给函数的调用者，当函数被调用的时候

下面举出了一个输出上面的数学函数的结果的例子：

```
func printMathResult(mathFunction: (Int, Int) -> Int, a: Int, b: Int) {

println("Result: \(mathFunction(a, b))")

}

printMathResult(addTwoInts, 3, 5)

// prints "Result: 8"
```

这个例子中定义了一个名为 printMathResult 的函数，该函数包含三个参数。第一个参数被称为 mathFuncton，是一个(Int, Int) -> Int 类型的参数。 您可以传递该类型的任何函数，作为此参数的一个自变量。第二个和第三个参数为 a 和 b，都属于 Int 类型的参数。这些参数被用作所提供的数学函数的两个输入值。

当 printMathResult 被调用时，addTwoInts 以及整数值 3 和 5 被传递。它调用和值 3 和 5 同时提供的函数，并且打印结果 8。

printMathResult 的作用是打印调用一个适当类型的数学函数的结果。那个函数的行为是什么其实并不要紧。该函数是正确类型就可以。这使 printMathResult 以类型安全的方式交出它的一些功能给函数的调用者。

6.18 可作为返回类型的函数

您可以使用一个函数类型作为另一个函数的返回类型。在返回的函数返回箭头（->）后，立即写一个完整的函数类型可以做到这一点。

下面的例子详述了两个简单的名为 stepForward 和 stepBackward 的函数。stepForward 函数返回比其输入值多一的值，而 stepBackward 函数返回比其输入值少一的值。这两个函数的类型均为(Int)-> Int 型：

```
func stepForward(input: Int) -> Int {

return input + 1

}

func stepBackward(input: Int) -> Int {

return input - 1
```

```
}
```

这里有一个叫 chooseStepFuncton 的函数，它的返回类型是“a function of type (Int) -> Int”。chooseStepFuncton 返回 stepForward 函数或 stepBackward 函数，基于一个叫做 backwards 的布尔参数：

```
func chooseStepFunction(backwards: Bool) -> (Int) -> Int {
return backwards ? stepBackward : stepForward
}
```

您现在可以使用 chooseStepFuncton 获得将迈向一个或另一个方向的函数：

```
var currentValue = 3
let moveNearerToZero = chooseStepFunction(currentValue > 0)
// moveNearerToZero now refers to the stepBackward() function
```

前面的例子解决了一个正的或负的步骤是否需要，以便逐步移动一个名为 currentValue 变量到接近零。currentValue 初始值为 3，这意味着当前值> 0，则返回 true，这表明 currentValue > 0 是 true，导致 chooseStepFuncton 返回 stepBackward 函数。对返回函数的引用被存储在一个名为 moveNearerTozero 的常数中。

由于 moveNearerTozero 指的是一个校正函数，那么可以用它来计数：

```
println("Counting to zero:")
// Counting to zero:
while currentValue != 0 {
println("\(currentValue)... ")
currentValue = moveNearerToZero(currentValue)
}
println("zero!")
// 3...
```

```
// 2...
1...
zero!
```

6.19 嵌套函数

所有你迄今为止在本章中遇到的函数都是全局函数的例子，这是在全球范围内定义的函数。你还可以在函数的语句体中定义其他函数，这种函数被称为嵌套函数。

嵌套函数默认隐藏，但仍然可以调用，并使用他们的封闭功能。一个封闭函数也可以返回其嵌套函数之一，以便在另一个范围内使用该嵌套函数。

你可以重写上面的 chooseStepFuncton 例子来练习使用并返回嵌套函数：

```
func chooseStepFunction(backwards: Bool) -> (Int) -> Int {
func stepForward(input: Int) -> Int { return input + 1 }
func stepBackward(input: Int) -> Int { return input - 1 }
return backwards ? stepBackward : stepForward
}
var currentValue = -4
let moveNearerToZero = chooseStepFunction(currentValue > 0)
// moveNearerToZero now refers to the nested stepForward() function
while currentValue != 0 {
println("\(currentValue)... ")
currentValue = moveNearerToZero(currentValue)
("zero!")
4...
3...
2...
1...
zero!
```

7 闭包

闭包是自包的函数组块，可以在您的代码中被传递和使用。Swift 的闭包类似于 C 和 Objective-C 语言的 blocks，和其他编程语言的 lambdas

闭包可以捕获和存储它们被定义的范围内的任何常量和变量的引用。它的特点就是用于关闭这些常数和变量，因此命名为“closures”。Swift 可帮你处理所有 capturing 的内存管理。

注

如果你不熟悉“capturing”的概念，不要担心。我们将会在下面捕获值的部分对它做详细的解释。

广泛及嵌套函数，实际上是闭包的特殊情况，如在函数部分的介绍，。Closures 采用下列三种形式之一：

全局函数是一种有一个函数名，但不会捕获任何值的 Closures。

嵌套函数是有一个名称、可以从他们的封闭函数捕获值的闭包。

闭合表达式是用 lightweight 语法写的、可以从他们的周围上下文捕获值的未命名闭包。

Swift 的闭合表达式有一个干净，清晰的风格，优化过程鼓励常见场景简单，整洁的语法。这些职能包括：

通过上下文推断参数和返回值的类型。

源自单表达式闭包的隐含返回值

速记参数名

尾随闭包句法

7.1 闭合表达式

嵌套函数，如在嵌套函数一章的介绍，是命名和定义的代码组块作为一个更大的函数的一部分的方便手段。但是，有时写短版本的没有完整名称的函数类似的结构是有益的。当你处理需要将其它函数作为它们的一个或多个引数的函数等功能为一体的论据或更多的功能，尤其如此。

闭合表达式是一种精简编写内联闭包的方式。闭合表达式提供了一些语法上的优化，可以在不损害清晰和意义的前提下以最简单的形式写闭包。下面的闭合表达式示例语法上的优化，通过优化一个 sort 函数的几次改进，每次的表达都以更简洁的方式表述相同的函数。

7.2 排序函数

Swift 的标准库提供了一个叫做 sort 的函数，这个函数根据您提供的分拣闭包对已知类型的值的组进行分类。一旦它完成了分拣过程中，排序函数返回与原来的相同类型和大小的新数组，其中的组分都已经在正确的排序顺序上。

下面的闭合表达式示例使用排序功能，将字符串值以字母倒序排序。这里是要进行排序的最初的数组：

```
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
```

FAI 排序函数有两个自变量：

一个已知类型值得数组。

一个闭包取同类型的两个参数作为数组的内容, 一旦该值被存储，其将返回and returns a Bool 值来判断第一个值应于第二个值前还是后出现。如果第一个值应出现在第二个值之前，排序闭包则需返回true, 否则返回false。

这个例子是对字符串值的数组进行排序，所以分拣闭包必须是类型的函数(String, String) -> Bool。

提供分拣闭包的一种方法是编写一个正确类型的正常函数，并把它作为排序函数的第二个参数传递：

```
func backwards(s1: String, s2: String) -> Bool {
return s1 > s2
}
var reversed = sort(names, backwards)
// reversed is equal to ["Ewa", "Daniella", "Chris", "Barry", "Alex"]
```

如果第一个字符串（S1）大于第二个字符串（S2），向后函数将返回 true，表示在排序后的数组中 S1 应该出现 S2 之前，。在字符串中的字符，“较大”意思是 “在字母表中出现的较晚”。这意味着，字母“b”“比”字母“a”大,字符串"Tom"大于字符串"Tim"。如果将"Barry"放在"Alex"之前，那么这将提供一个字母反序排列，依此类推。

然而，这是以一个相当冗长的方式来写本质上的一个单一表达式函数(a > b)。在这个例子中，最好用闭合表达式语法写排序闭包。

7.3 闭合表达式句法

闭合表达式句法具有下列通式：

```
{ ( parameters ) -> return type in
statements
}
```

闭包表达式语法可以使用常量参数，变量参数和 INOUT 参数。不能提供默认值。只要你命名一个可变参数，并将其置于参数列表的最后，则可以使用可变参数。元组也可以被用作参数类型和返回类型。

下面的例子显示了早期 backwards 函数的闭包表达式版本：

```
reversed = sort(names, { (s1: String, s2: String) -> Bool in
return s1 > s2
})
```

请注，这个内嵌闭包参数和返回类型的声明与 backwards 函数的声明相同。 在这两种情况下，它被写作 (s1: String, s2: String) -> Bool。然而，对于内嵌闭包表达式，参数和返回类型都写在大括号内，不是在括号外面。

闭包体的开始通过关键字进行介绍。这个关键字表明闭包的参数和返回类型的定义已经完成，闭包体即将开始

因为闭包体很短，它甚至可以写入单行中：

```
reversed = sort(names, { (s1: String, s2: String) -> Bool in return s1 > s2 } )
```

这说明对排序函数的全部调用保持不变。一对圆括弧仍然可以包含函数的整组参数。然而，现在，这些参数中的其中一个是一个内联闭包。

7.4 通过上下文推断类型

因为排序闭包作为函数参数被传递，Swift 可以从排序函数第二个参数类型推断出它的参数和它返回值的类型。这个参数期望的函数类型为 (String, String)-> Boo。这意味着 和 类型不必写成闭包表达式定义的一部分。因为可推断所有类型，返回箭头和参数名称周围的括号也可以省略

```
reversed = sort(names, { s1, s2 in return s1 > s2 } )
```

把一个闭包作为内联闭包表达式传递给函数时，总可以推断参数类型和返回类型。因此，你很少需要在其最充分的形式下编写一个内联闭包。

然而，如果你想，你可以使类型明确。如果这样做可避免读者对你的代码产生歧义，则鼓励这样做。在排序函数中，闭包的目的与排序正在发生的事实无关。而且读者推断闭包很可能正在处理 值是安全的，因为它正在协助一个字符串数组的排序。

7.5 源自单表达式闭包的隐含返回值

单表达式闭包可通过省略其声明中的返回关键字隐式地返回单表达式的结果，和在之前示例版本中相同：

```
reversed = sort(names, { s1, s2 in s1 > s2 } )
```

在这里，排序函数的第二个参数的函数类型清楚地表明，必须经闭包返回。因为闭包体包含返回 的单表达式 ，没有歧义，所以返回关键字可以省略。

7.6 速记参数名称

Swift 自动提供内联闭包的速记参数名称，可用来引用闭包参数值，使用 $0、$1、$2 等等名称。

如果你在闭包表达式中使用这些速记参数名称，你可以从闭包的定义中省略其参数列表。速记参数名称的数量和类型会从预期的函数类型推断。关键字也可以省略，因为闭包表达完全由其主体组成：

```
reversed = sort(names, { $0 > $1 } )
```

在这里，$0 和 $1 分别是指该闭包的第一个和第二个实参。

7.7 运算符函数

实际上，有一个更简短的方式，可以表达上述闭包表达式。Swift 的 类型将字符串特有的大于运算符为一个函数，它有两类 参数，并返回一个类型为 BOO 的值。这与排序函数的第二个参数所需的函数类型完全匹配。

因此，你可以只传递大于运算符，Swift 将推断出你想要使用其字符串的特有实现：

```
reversed = sort(names, >)
```

欲了解更多关于运算符函数的信息，请参阅运算符函数部分的内容。

7.8 尾随闭包

如果你需要将一个闭包表达式传递给一个函数作为其最后的实参，且这个闭包表达式很长，则反而将其写成尾随闭包很有效。尾随闭包是写于其支持的函数调用括号外（和后）的闭包表达式：

```
func someFunctionThatTakesAClosure(closure: () -> ()) {
// function body goes here
}
// here's how you call this function without using a trailing closure:
someFunctionThatTakesAClosure({
// closure's body goes here
})
here's how you call this function with a trailing closure instead:
someFunctionThatTakesAClosure() {
trailing closure's body goes here
```

注

如果一个闭包表达式作为函数的唯一实参被提供，且你提供那个表达式作为一个尾随闭包，则当你调用该函数时你不需要在函数的名称后写一对圆括号。

上述闭包表达式语法部分的字符串分拣闭包可以作为尾随闭包写在排序函数的括号之外：

```
reversed = sort(names) { $0 > $1 }
```

当闭包足够长，以至于不可能在同一行中写，尾随闭包最有用。作为一个例子，Swift 的数组类型有一个映射方法，即将闭包表达式作为其唯一实参。闭包为数组中的每个项目调用一次，并且返回该项目的替代映射值（可能是某些其它类型的）映射的性质和返回值的类型是留于闭包指定。

应用提供的闭包到每个数组元素后，映射方法以与它们在原始数组中的相对应值的相同顺序返回一个

包含所有的新映射值的一个新数组。

以下是你如何用利用尾随闭包的映射方法将一个 Int 值数组转换为 值数组。数组 [16，58，510] 用于创建新的数组。 ["OneSix", "FveEght", "FiveOneZero"]:

```
let digitNames = [
0: "Zero", 1: "One", 2: "Two", 3: "Three", 4: "Four",
5: "Five", 6: "Six", 7: "Seven", 8: "Eight", 9: "Nine"
]
let numbers = [16, 58, 510]
```

上面的代码创建了整数及其英文版名称间的映射字典。它还定义了一个准备被转换成字符串的整数数组。

你现在可以使用数字数组创建一个 值数组，通过将一个闭包表达式作为尾随闭包传递给数组的映射方法。注，调用 numbers.map 不需要在映射后使用任何括号，因为映射的方法只有一个参数，并且该参数作为一个尾随闭包提供：

```
let strings = numbers.map {
(var number) -> String in
var output = ""
while number > 0 {
output = digitNames[number % 10]! + output
number /= 10
}
return output
}
strings is inferred to be of type String[]
value is ["OneSix", "FiveEight", "FiveOneZero"]
```

对于数组中的每一项，映射函数都会调用一次闭包表达式。无需指定闭包输入参数类型、数量，因为该类型可从要映射数组中的值进行推断。

在此例中，闭包数量参数被定义为一个可变参数，如常量和变量参数部分所描述，以便可在闭包体内修改该参数值，而不是声明一个新的本地变量和将传递的数字值分配给它。闭包表达式还指定 的返回类型，以表明将要存储在映射输出数组中的类型。

闭包表达式在每一次被调用时都会构建一个名为 output 的字符串。其通过使用剩余运算符 (number % 10) 计算数字的最后一位，并使用这个数字在 digitNames 词典中查找适当的字符串。

注

digitNames 字典下标调用后跟一个感叹号，因为字典下标返回一个可选值，表明如果该关键词不存在，词典查找可能失败。在上例中，保证 % 10 始终是有效的 digitNames 字典下标关键词，所以感叹号是用来强行解开存储在下标可选返回值中的值。

从 digitNames 词典中检索到的字符串被添加到输出前面，有效地建立了数字的相反字符串版本。（表达式 10 将 16 赋值为 6，将 58 赋值为 8，将 510 赋值为 0。）

然后数字变量除以 10，因为它是一个整数，在除法过程中被舍去余数，所以 16 变成 1，58 变为 5，510 变为 51。

重复该过程，直到 number10 等于 0 时输出字符串由闭包返回，并且由映射函数添加到输出数组。

在上例中使用尾随闭包语法，巧妙地在闭包支持的函数后融入了闭包功能，而不需要将整个闭包包含到映射函数的外括号内。

7.9 获取值

闭包可以在其定义的范围内捕捉常量和变量。闭包可以引用和修改这些值，即使定义的常量和变量已经不复存在了依然可以修改和引用。

在 Swift 中，闭包的最简单形式是一个嵌套函数，写入另一个函数内。嵌套函数可以捕获任何外部函数的参数，也可以捕获外部函数中定义的任何常量和变量。

下例为称之 makeincrementor 的函数，其中包含一个名为 incrementor 的嵌套函数。嵌套的增量函数从它周围的环境中捕获两个值 runningTota 和 amount。捕获这些值后，由 makeIncrementor 返回作为闭包的 incrementor，每次被调用时产生增量 runningTota。

```
func makeIncrementor(forIncrement amount: Int) -> () -> Int {
var runningTotal = 0
func incrementor() -> Int {
runningTotal += amount
return runningTotal
}
return incrementor
}
```

makeIncrementor 的返回类型为 () -> Int。这意味着，它返回的是一个函数，而不是一个简单的值。它所返回的函数没有参数，并在每次调用时返回一个 Int 值。如需了解函数如何返回其他函数的相关信息，请参见作为返回类型的函数类型。

该 makeIncrementor 函数定义一个名为 runningTotal 的整型变量，用于存储当前正在运行且将要被返回的增量总量。此变量的初始值为 0。

该 makeIncrementor 函数含有一个外部名称为 forIncrement 且本地名称为 amount 的 参数。传递到此参数的实参值指定每次返回的函数 incrementor 被调用时的 runningTotal 的增量。

makeIncrementor 定义了一个名为 incrementor 的嵌套函数，它执行实际的递增。这个函数只是将 amount 添加到 runningTotal，并将结果返回。

单独看时，嵌套函数 incrementor 可能看起来不同寻常：

```
func incrementor() -> Int {
runningTotal += amount
return runningTotal
}
```

函数没有任何参数，但在其函数体内却指向 runningTota 和 amount。它通过从其周围的函数捕获 runningTotal 和 amount 的现有值并在其函数体内使用它们来执行此操作。

因为它没有修改 amount，incrementor 实际上捕获和存储存储在 amount 中的值副本。这个值与新建的 incrementor 函数一起存储。

然而，由于 runningTotal 每次被调用时都会修改其变量，所以 incrementor 捕获 runningTota 变量的当前引用，而不是仅仅是初始值的拷贝。捕获引用确保当调用 makeIncrementor 结束时 runningTota 不会消

失，并确保 runningTota 在下一次调用 incrementor 函数时仍可用。

注

Swift 决定什么应该通过引用进行捕获，什么应该通过数值进行复制。您不需要注释 amount 或 runningTotal，说明它们可在嵌套函数 中使用。当 incrementor 函数不再需要时，Swift 也处理处置 runningTotal 时涉及的所有内存管理。

下例为正在运行的函数 makeIncrementor：

```
let incrementByTen = makeIncrementor(forIncrement: 10)
```

本例设置名为 incrementByTen 的一个常量指代 incrementor 函数，该函数在每次调用 runningTotal 变量时会增加 10 至其中。多次调用该函数表明该行为正在运行：

```
incrementByTen()
// returns a value of 10
incrementByTen()
// returns a value of 20
incrementByTen()
// returns a value of 30
```

如果您创建另一个 incrementor，它会让其自身的引用指代一个新的、独立的 runningTota 变量。在下例中，incrementBySeven 捕获一个新 runningTota 变量的引用，而且该变量与 incrementByTen 捕获的变量无关：

```
let incrementBySeven = makeIncrementor(forIncrement: 7)
incrementBySeven()
// returns a value of 7
incrementByTen()
// returns a value of 40
```

注

如果你将闭包指定给类实例属性，而且该闭包通过引用实例或其成员捕获该实例，你将在闭包和实例间创建强引用循环。Swift 采用捕获列表来打破这些强引用循环。欲了解更多相关信息，请参阅闭包强引用循环部分的内容。

7.10 引用类型闭包

在上例中，incrementBySeven 和 incrementByTen 是常量，但这些常量所指的闭包仍可增加它们已经捕获的 runningTotal 变量。这是因为函数和闭包都是引用类型。

当你指定一个函数或一个闭包常量或变量时，你实际上是在设置该常量或变量为函数或闭包引用。在上例中， 闭包的选择是 incrementByTen 指的是闭包常量，而不是闭包本身的内容。

这也意味着，如果你为两个不同的常量或变量分配一个闭包，这两个常量或变量将引用同一个闭包：

```
let alsoIncrementByTen = incrementByTen

alsoIncrementByTen()

// returns a value of 50
```

7.11 枚举

枚举定义了一组相关值的通用类型，并让你能够在你的代码中安全的操作这些值。

如果你熟悉 C 语言，你就会知道，C 语言中的枚举将指定一组整数值的相关名称。在 Swift 中枚举更为灵活，不必为枚举的每个成员提供一个值。如果一个值（被称为“原始”的值）被提供给每个枚举成员，则该值可以是一个字符串，一个字符，或者任何整数或浮点类型的值。

另外，枚举成员可指定要与每个不同成员值一起存储的任何类型相关值，就像其他语言中集合或变体那样。你可将一组通用的相关成员定义为枚举的一部分，每部分都有与其相关的不同的一组适当类型的值。

在 Swift 中，枚举凭本身的力量成为最重要的类型。它们采用了传统上只有类支持的许多特性，如计算性能，以提供有关枚举当前值的更多信息，并采用实例方法以提供与枚举所代表的值相关的功能。枚举也可以定义初始化以提供一个初始成员值；可扩展到超出其原来执行范围的功能超；并可遵照协议提供标准功能。

欲了解更多有关这些功能的信息，请参见属性、方法、初始化、扩展和协议。

7.12 枚举句法

你用 enum 关键字介绍枚举并将其整个定义放在一对大括号内：

```
enum SomeEnumeration {
// enumeration definition goes here
}
```

下面是有关指南针四个要点的一个例子：

```
enum CompassPoint {
case North
case South
case East
case West
}
```

在枚举中定义的值（如,，和 ）是枚举的成员值（或成员）。关键字表明将对新一行的成员值进行定义。

注

不像 C 和 Objective-C，Swift 枚举成员在创建时不分配默认整数值。在上面的 CompassPoints 例子中，North，South，East 和 West 不等于隐值 0，1，2 和 3。相反，不同的枚举成员凭本身的力量成为成熟的值，含明确定义的 CompassPoint 类型。

多个成员值可出现在一行上，用逗号隔开：

```
enum Planet {
case Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune
}
```

每个枚举定义中定义了一个全新的类型。像 Swift 中的其他类型，它们的名称（如 CompassPoint 和 Planet）应以大写字母开头。为列举类型提供单数而不是复数名称，因此它们就可以不言自明了：

```
var directionToHead = CompassPoint.West
```

当 directonToHead 与 CompassPoint 的一个可能值初始化时， directonToHead 的类型即可推断。一旦 directonToHead 被声明为一个 CompassPoint，您可以使用更短的点语法将其设置为不同的 CompassPoint 值：

```
directionToHead = .East
```

directionToHead 的类型是已知的，所以你可以在设定它的值时删除该类型。在处理显式类型的枚举值时，可以用来编制高度可读的代码。

7.13 采用 switch 语句匹配列举值

你可以采用 switch 语句匹配单个的枚举值：

```
directionToHead = .South
switch directionToHead {
case .North:
println("Lots of planets have a north")
case .South:
println("Watch out for penguins")
case .East:
println("Where the sun rises")
case .West:
println("Where the skies are blue")
prints "Watch out for penguins"
```

你可以将此代码读作：

“考虑 directionToHead 的值。在它等于.North 的情况下，输出 “Lots of planets have a north”。在它等于.South 的情况下，输出 “Watch out for penguins”

…等等。

如控制流所述， 当考虑一个枚举的成员时，switch 语句必须详尽无遗。如果省略了 West 的 case，这段代码无法编译，因为它没有考虑 CompassPoint 成员的完整列表。要求竭力确保枚举成员不会偶然遗漏。

当为每一个枚举成员规定一个 case 不适当时,你可以提供一个 default 的案例以覆盖任何未被被明确处理的成员。

```
let somePlanet = Planet.Earth

switch somePlanet {

case .Earth:

println("Mostly harmless")

default:

println("Not a safe place for humans")

}

// prints "Mostly harmless"
```

7.14 关联值

在上一节中的示例显示了枚举的成员是如何在自己的权利内定义（和类型）值的。你可以设置一个 Planet. Earth 常量或变量,随后检查这个值。但是，有时能够将其它类型的关联值 在这些成员值旁边存储是有用的。这使你能够将额外的自定义信息与成员值一起存储，并允许这些信息在你每次在你的代码中使用该成员时发生变化。

您可以定义 Swift 枚举以存储任何给定类型的关联值，如果需要，枚举的每个成员的值类型均可以改变。与此类似的枚举被称为可识别联合，标记联合，或在其他编程语言中称为变体。

例如，假设一个库存跟踪系统需要跟踪两种不同类型的条形码产品。有些产品上标有 UPC -A 格式的的一维条码，使用数字 0 到 9，每一个条码均有一个“数字系统”的位，后跟十个“标识符”的位。这些后跟一个“检查”位以验证已正确扫描的代码：详见图 7-1

图 7-1 条形码

其他产品也标注有 QR 代码格式的二维条码，此类产品可使用任何 ISO 8859-1 字符，并且可对长达 2953 个字符的字符串进行编码：详见图 7-2

图 7-2 二维条码

便于库存跟踪系统能够将 UPC-A 条形码以三个整数的元组的形式进行存储，以及将 QR 代码条形码以任何长度的字符串进行存储。

在 Swift 内，可定义任何类型的产品条形码的枚举可能是这样的：

```
enum Barcode {
case UPCA(Int, Int, Int)
case QRCode(String)
}
```

这可被解读为：

“定义一个枚举类型为 Barcode，其可取含有类型（Int, Int, Int）关联值的 UPCA 的值，或者可取含有类型关联值的 QRCode 的值。

String'.”

此定义并没有规定任何 Int 或 String 的实际值-它只定义了关联值的类型，即 Barcode 的常量和变量等同于 Barcode.UPCA 或 Barcode.QRGod6 时，可进行存储。

然后，可利用任何类型创建新条形码：

```
var productBarcode = Barcode.UPCA(8, 85909_51226, 3)
```

该例产生了一个称为 productBarcode 的新变量并赋予其含有一个关联元组值 (8, 8590951226, 3) 的 Barcode.UPCA 的值。所规定的“标识符”值在其整数值变量—85909—51226—内含有下划线，便于以条形码的形式阅读。

相同的产品可以被分配为一个不同类型的条形码：

```
productBarcode = .QRCode("ABCDEFGHIJKLMNOP")
```

此时，原 Barcode.upcA 及其整数值被新的 Barcode.QRCode 和其字符串值所替换。Barcode 的常量和变量可对一个 UPCA 或一个 QRCode（连同其关联值）进行存储，但是其只能在给定的时间内存储其中之一。

如上所述，可以通过执行 switch 语句检查不同的条码类型。但是，此时关联值可以作为 switch 语句的一部分提取。可提取每个关联值作为常量（以 let 为前缀）和变量（以 var 为前缀）以在 switch 事例块主体内部使用。

```
switch productBarcode {
case .UPCA(let numberSystem, let identifier, let check):
println("UPC-A with value of \(numberSystem), \(identifier), \(check).")
case .QRCode(let productCode):
println("QR code with value of \(productCode).")
}
// prints "QR code with value of ABCDEFGHIJKLMNOP."
```

若枚举成员的所有关联值都被提取为常量或若所有的都被提取为变量，你可将单一 var 或 let 注释置于成员名称之前，为了简洁：

```
switch productBarcode {
case let .UPCA(numberSystem, identifier, check):
println("UPC-A with value of \(numberSystem), \(identifier), \(check).")
case let .QRCode(productCode):
println("QR code with value of \(productCode).")
}
// prints "QR code with value of ABCDEFGHIJKLMNOP."
```

7.15 原始值

关联值中的条形码实例说明了枚举成员是如何声明它们能够存储不同类型的关联值。枚举成员作为关

联值的替代项，可以使用默认值（称为原始值）预先填充，此类值均为同一类型。

下述示例为将初始的 ASCII 值储存在命名的枚举成员旁：

```
enum ASCIIControlCharacter: Character {
case Tab = "\t"
case LineFeed = "\n"
case CarriageReturn = "\r"
}
```

此处，称为 ASCIIControlCharacter 的枚举的原始值被定义为类型 Character，并且被设置为某些更常见的 ASCII 控制字符。Character 以字符串和字符表述。

注，初始值与关联值不同。当你初次定义代码中的枚举时，如上所述的三个 ASCII 代码，原始值应设置为预先填充值。一个特定的枚举成员的初始值始终保持不变。当你根据其中一个枚举成员创建新的常量或变量时，此时可进行关联值设置，并且每次进行的此类设置可不同。

初始值可以是字符串、字符或任何整数或浮点型数值。在其枚举声明范围内，每个初始值必须是唯一的。当用整数定义原始值时，若没有为某些枚举成员指定值，此类原始值会自动增加。

下述为早期 Planet 枚举改进后的枚举，其原始的整数值表示每一个来自于 sun 的 planet 的顺序：

```
enum Planet: Int {
case Mercury = 1, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune
}
```

自动增量意味着 Planet.Venus 的初始值为 2，依此类推。

利用其 toRaw 类函数访问枚举成员的初始值：

```
let earthsOrder = Planet.Earth.toRaw()
// earthsOrder is 3
```

使用枚举的 fromRaw 类函数可以试图找到具有特定原始值的枚举成员。本示例将 Uranus 的初始值定为 7：

```
let possiblePlanet = Planet.fromRaw(7)
```

```
// possiblePlanet is of type Planet? and equals Planet.Uranus
```

然而，并不是所有可能的 Int 值都能找到一个匹配的 planet。正因为如此，该 fromRaw 类函数返回一个可选的枚举成员。在上述示例中，possiblePlanet 为 Planet 类型或“可选的 Planet”。

若你试图找到位置 9 上的 Planet 枚举类型，由 fromRaw 方法返回的可选的 Planet 值将为 nil：

```
let positionToFind = 9
if let somePlanet = Planet.fromRaw(positionToFind) {
switch somePlanet {
case .Earth:
println("Mostly harmless")
default:
println("Not a safe place for humans")
}
} else {
println("There isn't a planet at position \(positionToFind)")
prints "There isn't a planet at position 9"
```

此例使用可选的联编试图利用 9 的原始值进入 Planet。语句 if let somePlanet = Planet.fromRaw (9) 检索到可选的 Planet，如果能被检索到，它将设置 somePlanet 枚举类型为可选的 Planet 的内容。在这种情况下，利用 9 的位置不可能检索到一个 planet 枚举类型，因此将执行 else 分句。

7.16 类别和结构

类别和结构是通用的，灵活的结构有助于编制程序代码。你可以通过准确使用有关常量、变量和功能同样的句法，定义属性和类函数以为类别和结构增加函数性。

Swift 不同于其他编程语言，其不需要为自定义类别和结构创造独立的界面和实施文件。在 Swift 语言中，你可在单个文件里定义类别和结构，其外部界面自动供其他代码使用。

注

一个类的实例一般被称为一个对象。然而，Swift 语言的类别和结构在功能性方面比在其他语言中相近的多，而且本章节所述的功能性，可用于类别或结构类型的实例中。正因为如此，将会用到更普通的术语实例。

7.17 比较类别和结构

Swift 的类和结构有许多共同之处。均可以：

定义存储值的属性

定义规定功能的类函数

定义下标利用下标句法访问其数值

定义初始化块设置其初始状态

扩展其默认实现外的功能

依照协议规定某种标准功能

若需了解更多信息，请参见属性，类函数，下标，初始化，扩展和协议。

类具有结构不具有的额外功能：

继承功能使一个类继承另一个类的特征。

若需了解更多信息，请参见继承，类型转换，初始化和自动引用计数。

注

结构总是在被传递至代码时被处理，并且不使用引用计数。

7.18 定义语法

类和结构具有相似的定义语法。你利用 class 关键字引用类,struct 关键字引用结构。其整个定义必须置于一对大括号内：

```
class SomeClass {

// class definition goes here
```

```
}
struct SomeStructure {
// structure definition goes here
}
```

注

每当你定义一个新类或结构时，你有效地定义一个全新的 Swift 类型。赋予类型 UpperCamelCase 名称（例如此处为 SomeClass 和 SomeStructure）以匹配标准 Swift 类型（例如字符串、整数型和布尔型）的大写操作。相反地，始终赋予属性和类函数 lowerCamelCase 名称（例如 frameRate 和 incrementCount）将他们以类型名称区分开。

下述示例为结构定义和类定义：

```
struct Resolution {
var width = 0
var height = 0
}
class VideoMode {
var resolution = Resolution()
var interlaced = false
var frameRate = 0.0
var name: String?
```

上面的例子详述了一个新建结构称为 Resolution，用于说明基于像素的显示分辨率。这种结构包含两个存储属性，即 width 和 Height。存储属性为常量或者变量，可以作为类别或结构的一部分进行组合和存储。通过将其设置为初始整数值 0，可推出这两种属性为 Int 类型。

上面的例子也详述了一个新建类别称为 VideoMode，用于说明视频显示的特定视频模式。这个类具有四个变量存储属性。第一个，resolution，与一个新建 Resolution 结构实体进行初始化，可推出 Resolution 的属性类别。对于其它三个属性，新建 VideoMode 实体将与 false 的一个 interlaced 设置（指非交织视讯）进行初始化，0.0 的播放帧率，以及一个可选 String 值称为 name。name 的属性将自动设置默认值为 nil，或设置为“无 name 值”，因其为可选类型。

7.19 类和结构实体

Resolution 结构定义以及 VideoMode 类别定义只说明什么是 Resolution 或 VideoMode。其本身不能描述一个特定的分辨率或视频模式。要做到这一点，你需要创建一个结构或类实体。

创建实体的语法与结构和类非常类似：

```
let someResolution = Resolution()

let someVideoMode = VideoMode()
```

结构和类均使用初始化语法创建新实体。初始化语法最简单的形式是采用类或结构的类型名称加空括号的形式，如 Resolution（）或 VideoMode（）。这就为类或结构创建了一个新建实体，所有属性均被初始化至默认值。在初始化中更加详细地描述了类和结构初始化。

7.20 访问属性

您可以利用点语法访问实体属性。在点语法中，你可以在实体名称后接属性名称，用句号（.）隔开，中间不要空格：

```
println("The width of someResolution is \(someResolution.width)")

// prints "The width of someResolution is 0"
```

在这个例子中，someResoluton.width 指 someResolution 的 width 属性，并恢复其默认初始值至 0。

你可以继续探究其子属性，如 VideoMode 中 resolution 属性的 width 属性:

```
println("The width of someVideoMode is \(someVideoMode.resolution.width)")

// prints "The width of someVideoMode is 0"
```

你也可以利用点语法给变量属性赋一个新值：

```
someVideoMode.resolution.width = 1280
```

```
println("The width of someVideoMode is now \(someVideoMode.resolution.width)")
// prints "The width of someVideoMode is now 1280"
```

注

与 Objective-C 不同，使用 Swift 你可以直接设置结构属性的子属性。在上述最后一个例子中，someVideoMode 中 resolution 属性中的 width 属性被直接设置，你无需为所有 resolution 属性设置新值。

7.21 结构类型的成员逐一初始化程序

所有结构均含有一个自动生成的成员逐一初始化程序，你可以使用其初始化新建结构实体中的成员属性。新建实体中的属性初始值可以通过名称传递至成员逐一初始化程序。

```
let vga = Resolution(width: 640, height: 480)
```

与结构不同，类实体不能接收默认的成员逐一初始化程序。在初始化更加详细地描述了初始化程序。

7.22 结构和枚举均属于值类型

值类型被赋值为变量或常量或当其被传递到函数时所复制的类型。

在前述章节中，实际上你已经广泛应用了值类型。实际上，所有 Swift 中的所有基本类型—整数、浮点数、布尔值、串符、数组及词典---均为值类型，并作为后台应用结构。

在 Swift 内，所有结构和枚举均为值类型。这意味着你所创建的任何结构或枚举实体，以及其所有作为属性的值类型，在通过你的代码时常被复制。

考虑这个例子，即从先前例子中选用一个 Resolution 结构：

```
let hd = Resolution(width: 1920, height: 1080)
var cinema = hd
```

这个例子说明了一个称为 hd 的常量并将其设置到 Resolution 中，与全高清视频的宽度和高度（1920 像素宽 x1080 像素高）进行初始化。

然后，声明一个称为 cinema 的变量并设置其当前值为 hd。因为 Resolution 是一个结构，因此会拷贝现有实体，新建拷贝将赋值到 cinema 中。虽然现在 hd 和 cinema 都含有相同的宽度和高度，但是它们却是后

台中完全不同的两个实体。

然后，修改 cinema 的 width 属性，使宽度达到较宽 2K 标准，用于数字电影投影（2048 像素宽 x1080 像素高）：

```
cinema.width = 2048
```

检查 cinema 的 width 属性显示确实已经更改为 2048：

```
println("cinema is now \(cinema.width) pixels wide")
// prints "cinema is now 2048 pixels wide"
```

然而，原 hd 实体的 width 属性的旧值仍为 1920：

```
println("hd is still \(hd.width) pixels wide")
// prints "hd is still 1920 pixels wide"
```

当 cinema 配置为当前数值 hd 时，存储在 hd 中的数值将被拷贝至新建 cinema 实体中。最终结果是产生两种完全独立的实体。而其又恰好包含相同的数值。因为这两个实体完全独立，将 cinema 宽度设置为 2048 并不会影响 hd 中存储的宽度。

同样的行为适用于枚举：

```
enum CompassPoint {
case North, South, East, West
}
var currentDirection = CompassPoint.West
let rememberedDirection = currentDirection
currentDirection = .East
if rememberedDirection == .West {
println("The remembered direction is still .West")
}
```

```
prints "The remembered direction is still .West"
```

当 rememberedDirection 配置为 currentDirection 时，实际上是对该数值进行了一次拷贝。改变 currentDirection 的数值之后并不会影响存储在 rememberedDirection 的原始数值的拷贝。

7.23 类是引用类型

与数值类型不同，引用类型被赋值为变量或常量或当其被传递至函数时，并不会进行拷贝。相反使用现有的相同实体的引用，而不是副本。

下述示例中使用了以上定义的 VdeoMode 类：

```
let tenEighty = VideoMode()

tenEighty.resolution = hd

tenEighty.interlaced = true

tenEighty.name = "1080i"

tenEighty.frameRate = 25.0
```

这个例子说明了一个新建常量，即 tenEighty，并将其设置为指代 Video Mode 类别中的一个新建实体。视频模式由之前的 1080 配置为高清分辨率 1920 的副本。其被设置为隔行扫描并命名为“1080i”。最后，其帧速率被设置为 25.0 帧/秒。

然后，tenEighty 被赋值为一个新建常量，即 alsoTenEghty，此时 alsoTenEghty 的帧率会被改变：

```
let alsoTenEighty = tenEighty

alsoTenEighty.frameRate = 30.0
```

因为类是引用类型，tenEighty 和 alsoTenEighty 实际上均指代同一个 VideoMode 实体。实际上，他们只是相同单一实体的两个不同的名称。

通过检查 tenEighty 的 frameRate 属性，我们发现其可以从相关的 VideoMode 实体中正确报告出新帧率为 30.0。

```
println("The frameRate property of tenEighty is now \(tenEighty.frameRate)")

// prints "The frameRate property of tenEighty is now 30.0"
```

注，声明 tenEighty 和 TenEighty 为常量，而不是变量。但是，因为 tenEighty 和 alsoTenEighty 的常量

数值本身不会发生实际变化，你仍然可以改变 tenEighty.frameRate 和 alsoTenEighty.frameRate。tenEighty 和 alsoTenEighty 本身不会“存储”VideoMode 实体，但是两者均指代后台的 VideoMode 实体发生变化的是相关的 VideoMode 的 frameRate 属性，而非该 VideoMode 引用的常量值。

7.24 恒等运算符

因为类是引用类型，因此多个常量和变量可以指代后台某一类别的相同的单一实体。(结构和枚举则不同，因为其为数值类型且被赋值到一个常量或变量或传递给函数时，通常会被拷贝。)

这有时可以帮助识别两个常量或变量是否指代某一类的同一实体。为了启用该特性，Swift 规定了两个恒等运算符：

等于 (===)

不等于 (!==)

使用这些操作程序检查两个常量或变量是否指代相同的单一实体。

```
if tenEighty === alsoTenEighty {
println("tenEighty and alsoTenEighty refer to the same Resolution instance.")
}
// prints "tenEighty and alsoTenEighty refer to the same Resolution instance."
```

注“相同”（用三个等号或===表示）并不与“等于”（用两个等号或==表示）表示同一事物：

same class instance.

“等于” 指两个实例的值是“相等的” or “等价物”, 对于“等于”的某些适当意思, 由类型的计划着定义。

当你在定义你自己的自定义类和结构时，你应该负责决定两个实体“相同”的依据是什么。“等于”及“不等于”运算符的自身实现的定义过程，见等价运算符。

7.25 指示字

如果你有 C、C++或 Objective-C 方面的经验，你可能就知道这些语言使用指示字指代内存中的地址。一个 Swift 常量或者变量指代一些引用类型中的一个实体类似于 C 中的指示字，但这个指示字不会直接指示内存地址，你也无需用星号（*）来注明正在创建一个引用。相反，和任何常量或变量一样，在 Swift 内也定义了这些引用。

7.26 在类和结构之间选择

你可以使用类和结构定义自定义数据类型，并作为程序代码的组块使用。

然而，结构实体总是按值传递，类实体总是按引用传递。这意味着类和结构适合于不同类型的任务。当你考虑根据项目所需的数据结构和功能，决定每个数据结构是否应被定义为类或结构。

作为一般指引，当适用其中一个或多个条件时考虑创建一个结构：

该结构的最初目的是压缩几个比较简单的数据值。

当用户配置或传递该结构实例时，封装值会被拷贝而不是被引用，这是合理的期望。

由该结构存储的任何属性是其自身的数值类型，其也会被期望拷贝而不是引用。

该结构不需要从另外的现存类型继承属性或行为。

优秀入选的结构示例包括：

几何形状的尺寸,或许以宽属性和高属性进行封装，均以Double类型。

引用一个系列的排列的方法，或许以开始属性和长度属性进行封装，均以Int型。

A point in a 3D coordinate system三维坐标系里的一个点，或许以x, y, z属性进行封装，均以Double类型。

在所有其他情况下，定义一个类，并创建该类的实体，以便按引用进行管理和传递。在实践中，这意味着大多数自定义数据结构应为类，而不是结构。

8 集合类型的赋值和拷贝行为

Swift 的 Array 和 Dictonary 类型作为结构应用。然而，当被赋值到一个常量或变量时及被传递到一个函数或类函数时，数组与字典和其他结构具有略微不同的拷贝行为。

以下 Array 和 Dictionary 所描述的行为与基础中 NSArray 和 NSDictionary 的行为又有所不同，该行为按类而不是按结构实现。NSArray 和 NSDictionary 实体总是按引用而不是按拷贝赋值并传递给现有实体。

注

以下描述指数组、字典、字符串和其他数值的“拷贝”。涉及拷贝的，在代码中看到的行为将始终像拷贝一样。然而，Swift 只在绝对必要时在幕后进行实际拷贝。Swift 对所有值拷

贝进行管理，以确保获得最佳性能，你不应回避试图优先优化的赋值。

9 字典的赋值和拷贝行为

每当向常量或变量赋值一个 Dictionary 实体时，或向一个函数或类函数调用传递一个作为参数的字典实体时，字典就会在赋值或调用时进行拷贝。这将在结构和枚举是数值型中被描述。

如果 Dictionary 实体中存储的密钥和/或值属于值类型（结构或枚举），当赋值或调用时，对这些密钥和/或值进行拷贝。相反，如果密钥和/或值属于引用类型（类或函数），这些引用会进行拷贝，但不是他们所指的类实体或函数。这种拷贝字典密匙和值的行为与结构拷贝时结构的存储属性的拷贝行为相同。

下面的例子定义了一个称为 ages 的字典，其中存储了四个人的名字和年龄。然后，将该 ages 字典赋值到一个名称为 copiedAges 的新变量中，并在赋值时进行拷贝。赋值后，ages 和 copiedAges 是两个独立的字典。

```
var ages = ["Peter": 23, "Wei": 35, "Anish": 65, "Katya": 19]

var copiedAges = ages
```

此字典的密匙为 String 类型，其数值为 Int 型。在 Swift 中，这两种类型均属于值类型，所以在对字典进行拷贝的同时对该密匙和值进行拷贝。

可证明 ages 字典已通过更改其中一个字典中的年龄值并按照另一字典中相应的值进行了拷贝。如果将 copiedAges 字典中“Peter”的值设为 24，拷贝前，年龄字典仍返回旧值 23：

```
copiedAges["Peter"] = 24

println(ages["Peter"])

// prints "23"
```

10 数组的赋值和拷贝行为

Swift 的 Array 类型的赋值和拷贝行为比其 Dictionary 类型的赋值和拷贝行为更加复杂。当使用数组内容时，Array 规定了类一样的性能，并在拷贝必要时拷贝数组的内容。

当向常量或变量赋值一个 Array 实体时或向一个函数或类函数调用传递一个作为参数的 Array 实体时，在赋值或调用时该数组的内容不会进行拷贝。相反，两数组共用相同序列的元素值。当你通过数组对其中

一个元素值进行修改时，通过另一个阵列是可以观察到结果的。

对于数组，拷贝只发生在执行有可能修改数组长度的行为时。其中包括项目追加、插入或移除，或使用远程下标取代数组中一系列项目。只有当进行数组拷贝时，对数组内容拷贝的行为才与字典密匙和值的拷贝行为相同，如字典的配置与拷贝行为所述。

下面的示例向 a 变量赋值了一个 Int 值的新数组。还将该数组赋值到另外两个进一步变量 b 和 c 中：

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

var b = a

var c = a
```

你可以通过下标语法在任一 a, b, 或 c 检索数组内的第一个值：

```
println(a[0])

// 1

println(b[0])

// 1

println(c[0])

// 1
```

如果将数组中的一个项目采用下标语法设为新值，a、b 和 c 三个都将返回新值。注，采用下标语法设置新值时，未进行数组拷贝，因为采用下标语法设置单个值不可能会改变数组的长度：

```
a[0] = 42

println(a[0])

// 42

println(b[0])

// 42

println(c[0])

// 42
```

但是，如果你在 a 中添加一个新项，你实际上修改了数组的长度。这将提示 Swift 在追加新值时对数组进行新的拷贝。从此，a 为本数组的一个单独且独立的副本。

如果在拷贝后改变 a 中的一个值，a 将返回一个不同于 b 和 c 的值，拷贝前，b 和 c 两个值仍引用原有的数组内容：

```
a.append(4)

a[0] = 777

println(a[0])

// 777

println(b[0])

// 42

println(c[0])

// 42
```

10.1 确保数组的唯一性

确保在对数组内容进行任何行为前或将数组传递给函数或类函数前单独对数组进行唯一拷贝，这可能有用。通过调用数组类型变量上的 unshare 类函数确保数组引用的唯一性。（在常量数组内，不能调用 unshare 类函数。）

如果多个变量目前指的是同一数组，并调用这些变量之一上 unshare 类函数，对该数组进行拷贝，从而使变量对数组进行单独拷贝。然而，如果已经声明该变量仅能由数组引用，则不能进行拷贝操作。

在前述示例的最后，b 和 c 均引用了相同的数组。调用 unshare 类函数使 b 成为独特拷贝：

```
b.unshare()
```

如果在调用 unshare 类函数后改变 b 中的第一个值，所有三个数组现在会报告不同的值：

```
b[0] = -105

println(a[0])

// 777

println(b[0])
```

```
// -105
println(c[0])
// 42
```

10.2 检查是否两个数组共用相同的元素

通过与恒等运算符比较，检查两个数组或子数组是否共享相同的存储器和元素（===和！==)

下面的示例使用“相同”运算符（===）来检查 b 和 c 是否仍共享相同的数组元素：

```
if b === c {
println("b and c still share the same array elements.")
} else {
println("b and c now refer to two independent sets of array elements.")
}
// prints "b and c now refer to two independent sets of array elements."
```

或者，使用恒等运算符检查两个子数组是否共享相同的元素。下面的示例对 b 中两个相同的子数组进行了比较，并确认其指的是相同元素：

```
if b[0...1] === b[0...1] {
println("These two subarrays share the same elements.")
} else {
println("These two subarrays do not share the same elements.")
}
// prints "These two subarrays share the same elements."
```

10.3 数组的强制拷贝

通过调用数组的拷贝方法，强制拷贝数组的显式副本。该类函数对数组进行了浅拷贝，并返回一个包含拷贝项目的新数组。

下面的示例定义了一个名称为 names 的数组，其中存储了七个人的名字。将一个名称为 copedNames 的新变量设为调用 names 数组上 copy 类函数的结果：

```
var names = ["Mohsen", "Hilary", "Justyn", "Amy", "Rich", "Graham", "Vic"]
var copiedNames = names.copy()
```

可证明 names 数组已通过更改其中一个数组中的一个项并按照另一数组中相应的值进行了拷贝。如果将 copiedNames 数组中第一项设为“MO”，而不是“Mohsen”，names 数组在拷贝前仍返回旧值“Mohsen”：

```
copiedNames[0] = "Mo"
println(names[0])
// prints "Mohsen"
```

注

如果只需要确保对数组内容的引用是现存的唯一引用，那么调用 unshare 类函数，而不是 copy 类函数。unshare 类函数并不拷贝数组，除非有必要这么做。copy 类函数可一直拷贝数组，即使已取消了本数组的共享条件。

11 属性

属性关联值具有特定的类、结构或枚举。存储属性将常量和变量值存为实体的一部分，但是计算属性计算（而不是存储）值。由类、结构和枚举规定计算属性。仅由类和结构规定存储属性。

存储和计算属性通常与一个特定类型的实体相关联。然而，属性也可以与其本身的类型相关联。此类属性称为类型属性。

此外，可对属性观察器进行定义以便观察属性值的变化情况，这样可以使用自定义操作进行响应。可以为你自己定义的存储属性添加属性观察器，也可以为继承父类的子类属性添加。

11.1 存储属性

最简单的情形，作为特定类或结构实体的一部分，存储属性是常量或者变量。存储属性可分为变量存储属性（关键字 var 描述）和常量存储属性（关键字 let 描述）。

你可以为存储属性规定一个默认值作为其定义的一部分，如默认属性值中所述。你也可以设置并修改初始化过程中存储属性的初始值。这个准则对常量存储属性也同样适用，如初始化期间修改常量属性所述。

下面的例子定义了一个叫 FixedLengthRange 的结构，它描述了一系列整数，一旦创建其范围长度不可改变：

```
struct FixedLengthRange {
var firstValue: Int
let length: Int
}
var rangeOfThreeItems = FixedLengthRange(firstValue: 0, length: 3)
// the range represents integer values 0, 1, and 2
rangeOfThreeItems.firstValue = 6
// the range now represents integer values 6, 7, and 8
```

FixedLengthRange 的实体包含名为 firstValue 的变量存储属性和名为 length 的常量存储属性。在上例中，创建新范围后，初始化 length，但在此后无法改变，因其为常量属性。

11.2 常量结构实体的存储属性

如果你创建一个结构的实体，并将其赋给一个常量，则无法改变该实体属性，即使其声明为变量属性：

```
let rangeOfFourItems = FixedLengthRange(firstValue: 0, length: 4)
// this range represents integer values 0, 1, 2, and 3
rangeOfFourItems.firstValue = 6
// this will report an error, even thought firstValue is a variable property
```

因为 rangeOfFourItems 声明为一个常量（关键字 let），所以即便 firstValue 是变量属性，也无法改变其 firstValue 属性。

这种特性取决于值类型的结构。当一个值类型的实体被标记为常量时，则成为所有数值的共有属性。类则不同，其仅仅是引用类型。如果你将引用类型的实体赋值给常量，依然能够改变实体的变量属性。

11.3 延迟存储属性

延迟存储属性是第一次使用时才进行初值计算的属性。在声明之前，你可以通过编写@lazy 属性表示

延迟存储属性。

注

你必须始终声明延迟存储属性为变量属性（关键字 var），因其初始值直到实体初始化完成之后才被检索。常量属性在实体初始化完成之前必须始终有一个值，因此常量属性不能声明为延迟存储属性。

当属性初始值取决于外部因素时，延迟存储属性就派上用场了，其值在实体初始化完成之前不能够确定。

当属性初始值需要复杂或计算上高代价的设置（除非或直到需要时才执行）时，延迟属性也有用。

下例使用延迟存储属性来避免复杂类的不必要初始化操作。该例定义了 DataImporter 类和 DataManager 类，两者均未完全显示：

```
class DataImporter {
/*
DataImporter is a class to import data from an external file.
The class is assumed to take a non-trivial amount of time to initialize.
*/
var fileName = "data.txt"
// the DataImporter class would provide data importing functionality here
}
DataManager {
@lazy var importer = DataImporter()
var data = String[]()
the DataManager class would provide data management functionality here
```

```
manager = DataManager()

manager.data += "Some data"

manager.data += "Some more data"

the DataImporter instance for the importer property has not yet been created
```

DataManager 类有一个称为 data 的存储属性，使用新的具有 String 值得空数组对其进行初始化。虽然 DataManager 其它的功能并未显示，但 DataManager 类的目的是管理 String 数据组并为其规定访问接口。

DataManager 类的部分功能可以从文件中导入数据。

这个功能由 DataImporter 类规定，被假定为需一定时间来初始化。这可能是因为初始化 DataImporter 实体时，需要打开文件并将其内容读到存储器中。

因为 DataManager 实体管理其数据而无需导入文件中的数据是可能的，所以在 DataManager 本身被创建时，并不需要创建一个新的 DataImporter 实体。相反，如果且当首次使用 DataImporter 实体时，创建 DataImporter 实体更有意义。

因被标以 @lazy 属性，所以 importer 属性的 DataImporter 实体仅在其被第一次访问时才被创建，例如其 fileName 属性需要被询问时：

```
println(manager.importer.fileName)

// the DataImporter instance for the importer property has now been created

// prints "data.txt"
```

11.4 存储属性和实体变量

如果你使用过 Objective-C，你应该知道它规定两种方式来存储作为类实体一部分的值与引用。除了属性，你可以使用实体变量作为属性中所存储值的后备存储。

Swift 使用一个单一属性声明来统一这些概念。Swift 属性没有与之相符的实体变量，并且属性的后备存储也不能直接访问。这种方法避免了混淆如何在不同上下文中访问值，并将属性声明简化为一个单一的明确的声明。关于属性的所有信息－包含名称、类型和内存管理特征－定义为单一位置中类型定义的一部分。

11.5 计算属性

除了存储属性，类、结构和枚举能够定义实际上并不存储值的计算属性。相反，它们规定吸气剂和可选的调节器来间接地检索和设置其他属性和值。

```
struct Point {
var x = 0.0, y = 0.0
}
struct Size {
var width = 0.0, height = 0.0
}
struct Rect {
var origin = Point()
var size = Size()
var center: Point {
get {
let centerX = origin.x + (size.width / 2)
let centerY = origin.y + (size.height / 2)
return Point(x: centerX, y: centerY)
set(newCenter) {
origin.x = newCenter.x - (size.width / 2)
origin.y = newCenter.y - (size.height / 2)
square = Rect(origin: Point(x: 0.0, y: 0.0),
size: Size(width: 10.0, height: 10.0))
initialSquareCenter = square.center
```

```
square.center = Point(x: 15.0, y: 15.0)

("square.origin is now at (\(square.origin.x), \(square.origin.y))")

prints "square.origin is now at (10.0, 10.0)"
```

本示例定义了制作几何图形的三种结构：

Point encapsulates an (x, y) coordinate.

Size encapsulates a width and a height.

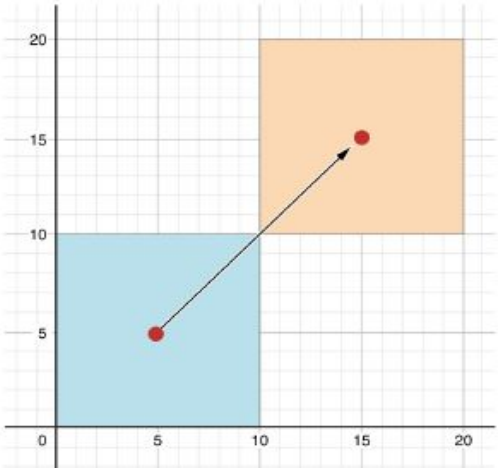
Rect defines a rectangle by an origin point and a size.

Rect 结构还规定了计算属性，即 center。Rect 的当前中心位置的坐标通常可通过其 origin 和 size 属性得来，所以并不需要将中心点存储为明确的 Point 值。相反，Rect 自定义称为 center 的计算属性的吸气剂和调节器，以使你能够操作长方形的 center，就像它是一个真正的存储属性一样。

前面的示例新建了一个 Rect 变量，称为 square。square 变量的原点初始化为 (0, 0)，高度和宽度初始化为 10。由以下图形中的蓝色正方形表示。

然后，通过点语法(square.center)访问 square 变量的 center 属性，这会调用 center 的吸气剂，以检索当前属性值。不同于直接返回一个存在的值，吸气剂要通过计算才能返回代表正方形中心点的新 Point。从上述示例可以看出，吸气剂可准确返回中心点（5,5）。

然后 center 属性被设置成新的值 (15, 15)，这样就把这个正方形向右向上移动到了图中橙色部分所表示的新的位置。通过调用 center 的 调节器来设置 center 属性，这将改变存储 origin 属性的 x 和 y 值，并将正方形移动到新的位置。



12 速计调节器声明

如果计算属性的调节器未定义要设置新值的名称，则会默认使用名称 newValue。下面是利用这种速记符号的另一种版本的 Rect 结构：

```
struct AlternativeRect {
var origin = Point()
var size = Size()
var center: Point {
get {
let centerX = origin.x + (size.width / 2)
let centerY = origin.y + (size.height / 2)
return Point(x: centerX, y: centerY)
}
set {
origin.x = newValue.x - (size.width / 2)
origin.y = newValue.y - (size.height / 2)
```

12.1 只读计算属性

仅包含吸气剂不包含调节器的计算属性被称为只读计算机属性。只读计算属性通常返回值，并可通过点语法访问，但无法设置为一个不同的值。

注

你必须使用 var 关键字将计算属性－包含只读计算属性－声明为变量属性，因其值并不固定。let 关键字仅供常量属性使用，以表明一旦将其设置为实体初始化一部分，它们的值就不可改变。

通过移除 get 关键字和它的大括号，可以简化只读计算属性的声明：

```
struct Cuboid {
var width = 0.0, height = 0.0, depth = 0.0
var volume: Double {
return width * height * depth
}
}
let fourByFiveByTwo = Cuboid(width: 4.0, height: 5.0, depth: 2.0)
println("the volume of fourByFiveByTwo is \(fourByFiveByTwo.volume)")
// prints "the volume of fourByFiveByTwo is 40.0"
```

该例定义了称为 Cuboid 的新结构，其代表包含 width, height 和 depth 属性的三维长方体结构。该结构还有一个只读计算属性 volume，它计算并返回该长方体的当前 volume。因为就应用于特定 volume 值的 width, height 和 depth 的值而言，volume 是否可设置这一点并不明确。然而，这可用于 Cubic 规定只读计算属性以使得外部用户能够访问到其当前计算的容积。

12.2 属性观察者

借助属性观察者观察属性值并对其变化做出应答。每次设定属性值时，需调用属性观察者，即使新值与当前属性值相同。

可将属性观察者加到你定义的存储属性中，懒惰存储属性除外。也可将属性观察者加到继承属性（无论是否是存储型的还是计算型的）中，通过在一个子类之内重写属性而获得该属性。在重写部分对通过重写属性而获得该属性进行了描述。

注

无需对未经过重写的计算型属性定义属性观察者，因为在属性赋值器内，可直接对其值变化做出观察和应答。

可选择这些属性观察者中的其中任何一个或二者皆可：

启用 willSet 观察者时，新的属性值将作为一个常量参数于该观察者中通过。可指定此参数的名称，作为 willSet 执行的一部分。启用观察者时，若未输入参数名称和圆括号，该参数则会以默认参数名称 newValue 来表示。

同样，若启用 didSet 观察者，包含原属性值的常量参数会于该观察者中通过。如需要，可对该参数进行命名或使用默认参数名称 oldValue。

注

当一个属性首次初始化时，无需调用 willSet 和 didSet 观察者。当在初始化语境外设置属性值时，可调用上述观察者。

下面给出了关于运行中的 willSet 和 didSet 的示例：在下列示例中，定义一个称为 StepCounter 的新类，通过 StepCounter 可以记录人们行走时所迈出的总步数。这一类可能要用到来自步数计或其他步进计数器的数据以便记录一个人在日常生活中的步行数。

```
class StepCounter {
var totalSteps: Int = 0 {
willSet(newTotalSteps) {
println("About to set totalSteps to \(newTotalSteps)")
}
didSet {
if totalSteps > oldValue {
println("Added \(totalSteps - oldValue) steps")
}
stepCounter = StepCounter()
stepCounter.totalSteps= 200
About to set totalSteps to 200
```

```
Added 200 steps
stepCounter.totalSteps = 360
About to set totalSteps to 360
Added 160 steps
stepCounter.totalSteps = 896
About to set totalSteps to 896
Added 536steps
```

该 StepCounter 类定义了一个 Int 类型的 totalSteps 属性。这是一个带有 willSet 和 didSet 观察者的存储属性。

当属性被赋予一个新值时，可调用 totalSteps 的 willSet 和 didSet 观察者。即使新值与当前值是相同的，也是如此。

例如，wilSet 观察者对即将产生的新值使用自定义参数名称 newTotalSteps。在该示例中，仅简单地输出了待设定值。

当 totalSteps 的值更新后，调用 didSet 观察者。它对 totalSteps 的新值与旧值做出比较。若总步数增加，则会输出一条信息来表明走了多少步。didSet 观察者并没有为旧值提供一个自定义参数，而是用默认名称 oldValue 来代替。

注

若在属性本身具备的 didSet 观察者内对属性赋予一个值，所赋予的新值则会取代刚才已设定的值。

全局变量和局部变量

上述计算型属性和观察型属性对全局变量和局部变量均有效。全局变量是指被定义在任何函数、类函数、闭包或上下语境之外的变量。局部变量是指被定义在函数、类函数或闭包上下文的变量。

在上述章节中提及的全局变量和局部变量均是存储变量。存储变量，像存储属性一样，储存特定类型值且允许对该值进行设置和检索。

但是，在全局或局部范围内，也可以定义计算型变量以及定义一个存储变量的观察者。计算型变量是对其进行计算而非存储，计算变量的写入方式与计算型属性的写入方式相同。

注

全局常量和变量总是以一种和懒惰存储属性相似的方式被延迟地计算。不同于懒惰存储属性，全局常量和变量不必标有@lazy 标识。

局部常量和变量从来没有徐缓地计算过。

类型属性

实例属性是指属于特定类型的实例的属性。每当创造该类型新实例时，该实例有自身已设定的属性值，并与其他的实例是不同的。也可以定义属于该类型自身的属性，而不是针对该类型的任一实例。创建的所有该类型的实例，这里都会有这些属性的副本。该类型的属性被称为类型属性。

类型属性可定义对某个特定类型所有实例通用的值，例如所有实例都能使用的常量属性（像在 C 中的静态常量）或存储该类型（像在 C 中的静态变量）所有实例全局值的变量属性。

可以为值类型（即，结构和枚举）定义存储类型和计算类型属性。对类而言，只能够定义计算类型属性。

值类型的存储类型属性可以是变量，也可以是常量。而计算类型属性通常声明成变量属性，类似于计算实例属性。

注

不同于存储实例属性，需要为存储类型属性设定一个默认值。这是因为类型本身没有能在初始化阶段的存储类型属性设定一个值的初始化器。

12.3 类型属性句法

在 C 和 Objective-C 中，定义静态常量、变量和全局静态变量一样。但是在 Swift 中，类型属性的定义要放在类型定义中进行，在类型定义的大括号中，显示地声明它在类型中的作用域。

你用 static 关键字对值类型的类型属性进行定义且用 class 关键字对类类型的类型属性进行定义。下面的示例展示了存储类型属性和计算类型属性的用法：

```
struct SomeStructure {

static var storedTypeProperty = "Some value."

static var computedTypeProperty: Int {

// return an Int value here
```

```
}
}
enum SomeEnumeration {
static var storedTypeProperty = "Some value."
static var computedTypeProperty: Int {
return an Int value here
SomeClass {
class var computedTypeProperty: Int {
return an Int value here
```

注

上述示例是针对只读计算类型属性而言的，不过你也可以像计算实例属性一样定义可读可写的计算类型属性。

12.4 查询和设置类型属性

和实例属性一样，类型属性通过点语法进行查询和设置。但是类型属性的查询与设置是针对类型而言的，并不是针对类型的实例。例如：

```
println(SomeClass.computedTypeProperty)
// prints "42"
println(SomeStructure.storedTypeProperty)
// prints "Some value."
SomeStructure.storedTypeProperty = "Another value."
println(SomeStructure.storedTypeProperty)
// prints "Another value."
```

下列示例在一个结构中使用两个存储类型属性来展示一组声音通道的音频等级表。每个通道使用 0 到

10 来表示声音的等级。

从下面的图表中可以看出，使用了两组音频通道来表示一个立体声音频等级表。当一个通道的音频等级为 0 时，该通道的所有灯都不会亮。 当音频等级为 10 时，该通道的所有灯都会亮。在该图中，左通道表示等级为 9，右通道表示等级为 7：详见图 12-1

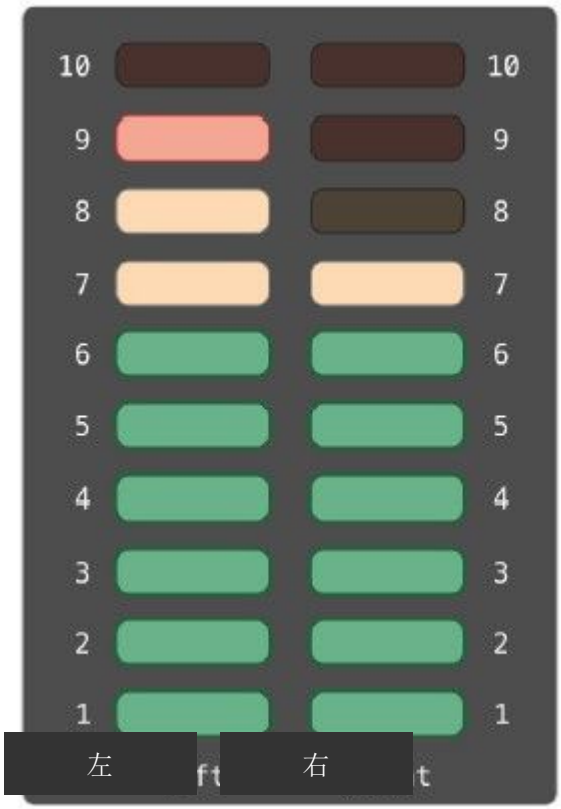


图 12-1 音频通道

上述的音频通道是由 AudioChannel 结构实例所表示。

结构：

```
struct AudioChannel {

static let thresholdLevel = 10

static var maxInputLevelForAllChannels = 0

var currentLevel: Int = 0 {

didSet {
```

```
if currentLevel > AudioChannel.thresholdLevel {
// cap the new audio level to the threshold level
currentLevel = AudioChannel.thresholdLevel
}
if currentLevel > AudioChannel.maxInputLevelForAllChannels {
// store this as the new overall maximum input level
AudioChannel.maxInputLevelForAllChannels = currentLevel
}
```

AudoChannel 结构定义两种存储类型属性去支持其函数性。第一个类型属性中，thresholdLevel 定义了音频所能达到的最高等级。对所有的 AudoChannel 实例而言，是个值为 10 的常量。当一个音频信号的值超过 10 时，这个信号将被限制在这个阈值（如下所述）。

第二个类型属性是一种被称为 maxInputLevelForAllChannels 的变化存储属性。该属性会记录任一 AudoChannel 实例接收的音频的最高等级。它被初始化为 0。

AudoChannel 结构也定义一种被称为 currentLevel 存储实例属性，该属性表示该通道的当前音频等级的范围在 0 至 10 之间。

currentLevel 属性使用 didSet 属性观察者来检查 currentLevel 值的改变。该观察者进行两项检查：

注

第一道检查中，didSet 为 currentLevel 设置了新值。这并不会造成观察者再次被调用。

可以使用 Audiochannel 结构创建两个新的 leftchannel 和 rightchannel 的音频通道来表现立体声音响系统的音频等级：

```
var leftChannel = AudioChannel()
var rightChannel = AudioChannel()
```

若设置左通道 currentLevel 至 7，maxInputLevelForAllChannels 类型属性则被更新为 7：

```
leftChannel.currentLevel = 7
println(leftChannel.currentLevel)
```

```
// prints "7"

println(AudioChannel.maxInputLevelForAllChannels)

// prints "7"
```

若设置右边通道的 currentLevel 至 11，右边通道的 currentLevel 属性被限制到最大值 10，且 maxInputLevelForAllChanneb 类型属性被更新为 10：

```
rightChannel.currentLevel = 11

println(rightChannel.currentLevel)

// prints "10"

println(AudioChannel.maxInputLevelForAllChannels)

// prints "10"
```

13 方法

方法是与特定类型相关联的函数。类，结构体以及枚举均可以定义实例方法，该方法为指定类型的实例封装了特定的任务与功能。类，结构体以及枚举也能定义类型方法，该方法与类型自身相关联。类型方法类似于在 Objective-C 中的类方法。

在 Swift 中,结构体和枚举能够定义方法；事实上这是 Swift 与 C/Objective-C 的主要区别之一。在 Objective-C 中,类是唯一能定义方法的类型。在 Swift 中，你可以选择是否定义一个类，结构体或枚举，且仍可以灵活地对你所创建的类型进行方法的定义。

13.1 实例方法

实例方法是某个特定类，结构体或枚举类型的实例的方法。他们通过提供访问的方式和修改实例属性，或提供与实例目的相关的功能性来支持这些实例的功能性。准确的来讲，实例方法的语法与函数完全一致，参考函数说明。

实例方法要写在它所属的类型的前后括号之间。实例方法能够访问他所属类型的所有的其他实例方法和属性。实例方法只能被它所属的类的特定实例调用。实例方法不能被孤立于现存的实例而被调用。

下面定义了一个简单的类 Counter 的示例（Counter 可以用来对一个动作发生的次数进行计数）：

```
class Counter {
```

```
var count = 0

func increment() {

count++

}

func incrementBy(amount: Int) {

count += amount

}

func reset() {

count = 0
```

类 Counter 可以定义三种实例方法：

类 Counter 还声明了一个变量属性、count 和 counter 值。

和调用属性一样，用点语法调用实例方法：

```
let counter = Counter()

// the initial counter value is 0

counter.increment()

// the counter's value is now 1

counter.incrementBy(5)

// the counter's value is now 6

counter.reset()

// the counter's value is now 0
```

13.2 方法的局部参数名称和外部参数名称

函数参数有一个局部名称(在函数体内部使用)和一个外部名称(在调用函数时使用),参考外部参数名

称。对方法参数也是一样的，因为方法仅仅是与某一类型相关的函数。但是，局部名称和外部名称的默认行为不同于函数和方法。

在 Swift 中的方法和在 Objective-C 中的方法极其相似，像在 Objective-C 一样，在 Swift 中方法的名称名称通常用一个介词指向方法的第一个参数，比如：with、for 以及 by 等等，前面的 Counter 类的例子中 incrementBy 方法就是这样的。当其被访问时，介词的使用使方法可被解读为一个句子介词的使用让方法在被调用时能像一个句子一样被解读。Swift 这种方法命名约定很容易落实,因为它是用不同的默认处理方法参数的方式，而不是用函数参数(来实现的)。

具体来说，Swift 默认仅给方法的第一个参数名称一个局部参数名称；但是默认同时给第二个和后续的参数名称局部参数名称和外部参数名称。这个约定与典型的命名和调用约定相匹配，这与你在写 Objective-C 的方法时很相似。这个约定还让 expressive method 调用不需要再检查/限定参数名。

看看下面这个 Counter 的替换版本（它定义了一个更复杂的 incrementBy 方法）：

```
class Counter {

var count: Int = 0

func incrementBy(amount: Int, numberOfTimes: Int) {

count += amount * numberOfTimes

}

}
```

amount 和 numberOfTimes。默认地，Swift 仅把 amount 当做一个局部名称，但是把 numberOfTimes 既看作局部名称又看做外部名称。调用方法如下：

```
let counter = Counter()

counter.incrementBy(5, numberOfTimes: 3)

// counter value is now 15
```

你不必对第一个参数值进行外部参数名称的定义，因为从函数名 incrementBy 已经能很清楚地看出它的目的/作用。但是，第二个参数，就要被一个外部参数名称所限定,以便在方法被调用时让他目的/作用明确。

这种默认的行为能够有效的检查方法，比如你在参数 numberOfTimes 前写了个井号（#）时：

```
func incrementBy(amount: Int, #numberOfTimes: Int) {
count += amount * numberOfTimes
}
```

这种默认行为使上面代码意味着：在 Swift 中定义方法使用了与 Objective-C 同样的语法风格，并且方法将以自然表达式的方式被调用。

13.3 修改该方法的外部参数名称

有时，对一个方法的第一个参数提供一个外部参数名是有用的，即使这不是默认行为。你可以自己添加一个明确的外部名称或你也可以用一个 hash 符号作为第一个参数的前缀，然后用这个局部名字作为外部名字。

相反，若你不想为方法的第二或后续参数提供一个外部名称，你可以通过使用下划线(_)作为该参数的显式外部名称来覆盖默认行为。

13.4self 属性

类型的每一实例都有一个被称为 self 的隐含属性，该属性完全等同于该实例本身。可以在一个实例的实例方法中使用这个隐含的 self 属性来引用当前实例。

在上面的例子中，increment 方法也可以被写成这样：

```
func increment() {
self.count++
}
```

实际上，你不必在你的代码里面经常写 self。不论何时，在一个方法中使用一个已知的属性或者方法名称，如果你没有明确的写 self，Swift 假定你是指当前实例的属性或者方法。这种假定在上面的 Counter 中已经示范了：Counter 中的三个实例方法中都使用的是 count(而不是 self.count)。

这条规则的主要例外发生在当实例方法的某个参数名称与实例的某个属性名称相同时。在这种情况下，参数名称享有优先权，并且在引用属性时必须使用一种更恰当(被限定更严格)的方式。你可以使用隐藏的 self 属性来区分参数名称和属性名称。

下面的例子演示了 self 消除方法参数 x 和实例属性 x 之间的歧义：

```
struct Point {
var x = 0.0, y = 0.0
func isToTheRightOfX(x: Double) -> Bool {
return self.x > x
}
}
let somePoint = Point(x: 4.0, y: 5.0)
if somePoint.isToTheRightOfX(1.0) {
println("This point is to the right of the line where x == 1.0")
prints "This point is to the right of the line where x == 1.0"
```

如果不使用 self 前缀，Swift 就认为两次使用的 x 都指的是名称为 x 的函数参数。

13.5在实例方法中修改值类型

结构体和枚举均属于值类型一般情况下，值类型的属性不能在其实例方法内被修改。

但是，如果在某个具体方法中，你需要对结构体或枚举的属性进行修改，你可以选择变异（mutating）这个方法。方法可以从内部变异它的属性；并且它做的任何改变在方法结束时都会回写到原始结构。方法会给它隐含的 self 属性赋值一个全新的实例,这个新实例在方法结束后将替换原来的实例。

对于变异方法，将关键字 mutating 放到方法的 func 关键字之前就可以了：

```
struct Point {
var x = 0.0, y = 0.0
mutating func moveByX(deltaX: Double, y deltaY: Double) {
x += deltaX
y += deltaY
}
```

```
}

var somePoint = Point(x: 1.0, y: 1.0)

somePoint.moveByX(2.0, y: 3.0)

("The point is now at (\(somePoint.x), \(somePoint.y))")

prints "The point is now at (3.0, 4.0)"
```

上文 Point 结构体定义一个变异（mutating）方法 moveByx，该方法按一定的数量移动 Point 结构体。moveByX 方法在被调用时修改了这个 point,而不是返回一个新的 point。方法定义是加上 mutating 关键字,因此，方法可以修改值类型的属性。

注：不能在结构体类型的常量上调用变异方法，因为其属性不能被改变，参考常量结构体实例的存储属性：

```
let fixedPoint = Point(x: 3.0, y: 3.0)

fixedPoint.moveByX(2.0, y: 3.0)

// this will report an error
```

13.6 在变异方法中给 self 赋值

变异方法可以赋予隐含属性 self 一个全新的实例。上述 Point 的示例也可以采用下面的方式来改写：

```
struct Point {

var x = 0.0, y = 0.0

mutating func moveByX(deltaX: Double, y deltaY: Double) {

self = Point(x: x + deltaX, y: y + deltaY)

}

}
```

新版的变异方法 moveByX 创建了一个新的分支结构(他的 x 和 y 的值都被设定为目标值了)。调用这个版本的方法和调用上个版本的最终结果是一样的。

枚举的变异方法可以让 self 从相同的枚举设置为不同的成员：

```
enum TriStateSwitch {
case Off, Low, High
mutating func next() {
switch self {
case Off:
self = Low
case Low:
self = High
case High:
self = Off
}
ovenLight = TriStateSwitch.Low
ovenLight.next()
ovenLight is now equal to .High
ovenLight.next()
ovenLight is now equal to .Off
```

上述示例中定义了一个三态开关的枚举。每次调用 next 方法时，开关在不同的电源状态(Off,Low,High)之前循环切换。

14 类型方法

如上所述，实例方法是被类型的某个实例调用的方法。你也可以定义调用类型本身的方法。这种方法就叫做类型方法。声明类的类型方法，在方法的 func 关键字之前加上关键字 class；声明结构体和枚举的类型方法，在方法的 func 关键字之前加上关键字 static。

注

在 Objective-C 中，你可以定义仅适用于 Objective-C 的类的 type-evel 方法。在 Swift，你可以对所有的类，结构体和枚举进行 type-evel 方法定义。每种类型方法仅适用于其所支持的类型。

与实例方法相同，也可利用点语法调用类型方法。但是，你需要在本类型上调用类型方法，而不是其类型实例上。下面是如何在 SomeClass 类上调用类型方法的示例：

```
class SomeClass {
class func someTypeMethod() {
// type method implementation goes here
}
}
SomeClass.someTypeMethod()
```

在一个类型方法的方法体内， self 指向该类型本身，而不是类型的某个实例。对结构体和枚举来讲，这意味着你可以用 self 来消除静态属性和静态方法参数之间的二意性(类似于我们在前面处理实例属性和实例方法参数时做的那样)。

更广泛地说，你在一个类型方法主体内使用的任何未经限定的方法和属性名称将指的是其他 type-level 方法和属性。一种类型方法能用其他方法名称调用另一种类型方法，而无需在方法名称前面加上类型名称的前缀。同样，结构体和枚举的类型方法也能够通过使用不带有类型名称前缀的静态属性名称来访问静态属性。

下述示例中定义名为 LevelTracker 的结构体，该结构体通过不同级别或阶段的游戏对玩家的进度进行监测。这是一个单人游戏，但也能在单个设备上储存多个玩家的信息。

所有游戏级别（除了级别 1）在游戏初始时都被锁定。每当玩家完成一个级别，在该设备上，该级别对所有的玩家解锁。LevelTracker 结构体用静态属性和方法来监测解锁的游戏级别。也可监测每个玩家的当前等级。

```
struct LevelTracker {
static var highestUnlockedLevel = 1
```

```
static func unlockLevel(level: Int) {
if level > highestUnlockedLevel { highestUnlockedLevel = level }
}
static func levelIsUnlocked(level: Int) -> Bool {
return level <= highestUnlockedLevel
}
var currentLevel = 1
mutating func advanceToLevel(level: Int) -> Bool {
if LevelTracker.levelIsUnlocked(level) {
currentLevel = level
return true
} else {
return false
}
```

LevelTracker 结构体监测任何玩家已解锁的最高级别。该值被存储在成为 hghestunockedLeve 的静态属性中。

LeveTracker 还定义了两个类型函数与 hghestUnockedLeve 配合工作。第一个为 unockLevel 类型函数，一旦新的级别被解锁，该函数会更新 hghestUnockedLevel 的值。第二个为 evelIsUnocked 便利型函数，若某个给定级别数已经被解锁，该函数则返回 true。（注：没用使用 LevelTracker.highestUnlockedLevel，这个类型方法还是能够访问静态属性 highestUnlockedLevel。）

除了其静态属性和类型方法之外，LeveTracker 还监测每个玩家的游戏进程。它使用 currentLevel 实例属性来监测玩家当前进行的级别。

为便于管理 currentLevel 属性，LeveTracker 定义了实例方法 advanceToLevel。在更新 currentLevel 之前，该方法检查所要求的新级别是否已经解锁。advanceToLevel 方法返回布尔值以指示是否确实能够设置 currentLevel 了。

```
class Player {
var tracker = LevelTracker()
let playerName: String
func completedLevel(level: Int) {
LevelTracker.unlockLevel(level + 1)
tracker.advanceToLevel(level + 1)
}
init(name: String) {
playerName = name
```

Player 类使用 LevelTracker 来监测该玩家的游戏进程。它也提供 competedLevel 方法，一旦玩家完成某个指定等级，调用该方法。该方法为所有玩家解锁下一个级别并将当前玩家进程更新为下一个级别。（忽略了 advanceToLevel 布尔返回值，因为之前调用上行时就知道了这个等级已经被解锁了。）

你可以为一个新玩家创建一个 Player 类的实例，然后看这个玩家完成等级一时发生了什么：

```
var player = Player(name: "Argyrios")
player.completedLevel(1)
println("highest unlocked level is now \(LevelTracker.highestUnlockedLevel)")
// prints "highest unlocked level is now 2"
```

如果你创建了第二个玩家，并尝试让他开始一个没有被任何玩家解锁的等级，关于设置玩家当前等级的尝试会失败：

```
player = Player(name: "Beto")
if player.tracker.advanceToLevel(6) {
println("player is now on level 6")
} else {
```

```
println("level 6 has not yet been unlocked")
}
// prints "level 6 has not yet been unlocked"
```

14.1 下标

类、结构体以及枚举均可以定义下标，该下标是访问集合、列表或序列成员元素的捷径。通过下标索引就可以方便地检索和设置相应的值，而不需要其他的额外操作。例如，你可以通过 someArray[index]来访问数组中的元素，或者通过 someDictionary[key]来对字典进行索引。

你可以为一个类型定义多个下标，以及适当的下标重载用来根据传递给下标的索引来设置相应的值。下标不仅可以定义为一维的，还可以根据需要定义为多维的输入参数。

14.2 小标语法

借助下标，你可以在实例名称之后通过在方括号内写入一个或多个数值的形式来查询某个类型的实例。其语法类似于实例方法语法和计算属性语法。与实例方法相类似，通过使用关键字 subscript 定义下标，并指定一个或多个输入参数以及一个返回类型值。与实体方法不同的是，下标仅为读写的或只读的。用与针对计算型属性相同的方式，该行为通过 getter 和 setter 语句联通：

```
subscript(index: Int) -> Int {
get {
// return an appropriate subscript value here
}
set(newValue) {
// perform a suitable setting action here
}
}
```

newValue 的类型与下标的返回值相同。和计算型属性一样，你可以选择不指定 setter（newValue）的参数。因为当未指定时，将默认参数 newValue 赋给 setter。

和计算型属性一样，针对只读下标，可以不需要关键字 get：

```
subscript(index: Int) -> Int {
// return an appropriate subscript value here
}
```

下面给出了一个只读下标执行的示例，定义了一个 TimesTable 结构来表示一个整数的 n-times-table：

```
struct TimesTable {
let multiplier: Int
subscript(index: Int) -> Int {
return multiplier * index
}
}
let threeTimesTable = TimesTable(multiplier: 3)
println("six times three is \(threeTimesTable[6])")
// prints "six times three is 18"
```

在该示例中，实例 TimesTabe 被创建为 three-times-table。这是通过结构体的初始化器为实例的 multiplier 参数传入的数值 3 设置的。

你可以通过调用其下标查询 threeTimesTabe 实例，如 hreeTimesTablep[6]调用中所示。这要求第 6 次进入 three-times-table，three-times-table 则会返回数值 18，或 3*6.

注

n-times-table 是根据特定的数学规则设置的。因此，不应该为 three Times Table[some Index] 元素设置一个新值，且 TimesTable 的下标被定义为只读下标。

14.3 下标的使用

“下标”的具体含义根据使用时的上下文来确定。下标主要用来作为集合、列表和序列的元素快捷方式。你可以自由的为你的类或结构定义你所需要的下标。

例如，Swift 中的 Dictonary 类型使用一个下标来设置和检索存储在 Dictonary 实例中的值。你可以在下标括号内，通过提供关键字 dictionary 关在字典内设置一个值，且对该下标赋予一个字典值：

```
var numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]

numberOfLegs["bird"] = 2
```

上面的例子中定义了一个变量 numberOfLegs，然后通过键值对初始化。经推断，numberOfLegs dictionary 的类型为 Dictonary <String, Int>。在字典创建后，该示例使用下标赋值方法添加了一个类型为字符串的键“bird”和 Int 值 2 到字典中。

更过关于 Dictonary 字典下标的信息，详见 Dictionary 的存储和修改。

注

Swift 中的 Dictonary 类型实现的键值对下标是可选类型。对于上述 numberOfLegs 字典而言，关键值下标采取和返回 Int?型或“可选 Int”的一个值。字典类型的这种使用可选类型下标的方式说明不是所有的键都有对应的值。同样也可以通过给键赋值 nil 来删除这个键。

13.4 下标选项

下标可以接收任意数量的参数，参数的类型也可以各异。下标还可以返回任何类型的值。下标可以使用变量参数和可变参数，但是不能使用输入、输出参数或者提供默认参数的值。

类或结构体可以根据其需要实现各种下标方式，可以在需要时使用合适的下标通过中括号中的参数返回需要的值。。这种多下标的定义被称作下标重载。

当然，最常见的下标用法是单个参数，也可以定义多个参数的下标。下列示例定义了一个 Matrix 结构体，它含有二维的 Double 值。Matrix 结构的下标采用两个整型参数：

```
struct Matrix {

let rows: Int, columns: Int

var grid: Double[]
```

```
init(rows: Int, columns: Int) {
self.rows = rows
self.columns = columns
grid = Array(count: rows * columns, repeatedValue: 0.0)
}
func indexIsValidForRow(row: Int, column: Int) -> Bool {
return row >= 0 && row < rows && column >= 0 && column < columns
subscript(row: Int, column: Int) -> Double {
get {
assert(indexIsValidForRow(row, column: column), "Index out of range")
return grid[(row * columns) + column]
}
set {
assert(indexIsValidForRow(row, column: column), "Index out of range")
grid[(row * columns) + column] = newValue
}
```

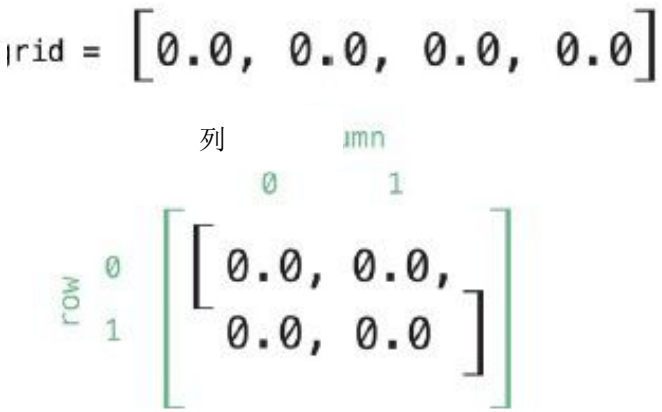
Matrix 提供一个初始化器，使用两个参数 rows 和 columns，然后建立了一个数组来存储类型为 Double 的 rows*columns 值。矩阵的每个位置的初始值设定为 0.0。通过传递数组长度和初始值 0.0 给数组初始化器完成上述操作。数组的初始化方法参照数组的创建和初始化。

你可以传递两个参数 row 和 column 来完成 Matrix 的初始化：

```
var matrix = Matrix(rows: 2, columns: 2)
```

上述初始化操作创建了一个两行两列的 Matrix 实例。该 Matrix 实例的 grid 数组看起来是平坦的，但是实际上是矩阵从左上到右下的一维存储形式：

网格



矩阵中的值可以通过使用包含 row 和 column 以及逗号的下标来设置：

```
matrix[0, 1] = 1.5
matrix[1, 0] = 3.2
```

这两种语句调用下标 setter 对矩阵右上角（横着是 0，竖着是 1）和左下角（横着是 1，竖着是 0）的两个元素分别赋值 1.5 和 3.2：

$$\begin{bmatrix} 0.0 & 1.5 \\ 3.2 & 0.0 \end{bmatrix}$$

矩阵下标的 getter 和 setter 方法都包括了一个断言语句来检查下标 row 和 column 是否有效。对于该断言，通过 Matrix 中的 indexIsValid 方法来判断 row 或 column 是否在矩阵的范围内。

```
func indexIsValidForRow(row: Int, column: Int) -> Bool {
return row >= 0 && row < rows && column >= 0 && column < columns
}
```

当访问矩阵边界外的下标时，触发断言：

```
let someValue = matrix[2, 2]
// this triggers an assert, because [2, 2] is outside of the matrix bounds
```

14.5 继承

一个类可继承另一个类的方法、属性以及其他特点。

当一个类继承其它类，继承类叫子类，被继承类叫超类（或父类）。在 Swift 中，继承是区分类与其它类型的一个基本特征。

在 Swift 中的类可以调用和访问超类的方法、属性和下标，并且可以重写这些方法、属性和下标来优化或修改它们的行为。Swift 通过检查重写定义在超类中是否有匹配的定义，以此确保你的重写行为是正确的。

可以为类中继承来的属性添加属性观察者，而当属性值改变时，类就会被通知到。可以为任何属性添加属性观察者，无论它原本被定义为存储型属性还是计算型属性。

14.5定义一个基类

不继承其它类的任意类被称为基类。

注

Swift 中的类并不是从一个通用的基类继承而来。若未能为你定义的类指定一个超类，这个类则自动成为基类。

下述示例定义了名为 vehicle 的基类。这个基类声明了两个对所有车辆均通用的属性（numberOfWheels 和 maxPassengers）。这些属性在 description 方法中使用，这个方法返回一个 String 类型的，对车辆特征的描述。

```
class Vehicle {
var numberOfWheels: Int
var maxPassengers: Int
func description() -> String {
return "\(numberOfWheels) wheels; up to \(maxPassengers) passengers"
}
```

```
init() {

numberOfWheels = 0

maxPassengers = 1
```

在 Vehicle 类中还定义了一个初始化器设置其属性。初始化器会在初始化中详细介绍，这里我们做一下简单介绍，以便于讲解子类中继承来的属性可以被如何修改。

初始化器用于创建某个类型的一个新实例。尽管初始化器不是方法，但是他们被以相同的语法写入实例方法中。初始化器的工作是准备新实例以供使用，并确保实例中的所有属性都拥有有效的初始化值。

初始化器的最简单形式就像一个没有参数的实例方法，使用关键字 init 写入：

```
init() {

// perform some initialization here

}
```

如果要创建一个 Vehicle 类的新实例，使用初始化器语法调用上面的初始化器，即在 TypeName 后写入一个空括号：

```
let someVehicle = Vehicle()
```

这个 Vehicle 类的初始化器为任意的一辆车设置一些初始化属性值（numberOfWheels = 0 和 maxPassengers =1）。

Vehicle 类定义了车辆的共同特性，但这个类本身并没太大用处。为了让其更实用，你需要进一步细化它来描述更具体的车辆。

14.6子类化

Subclassing 指的是在一个已有类的基础上创建一个新的类。子类继承现有类的所有属性，你可以细化这些属性。你还可以在子类中添加新的属性。

为了指明某个类的超类，将超类名写在子类名的后面，用冒号分：

```
class SomeClass: SomeSuperclass {

// class definition goes here
```

```
}
```

下一个示例中定义了一个更具体的车辆类 Bicycle。这个新类是在 Vehicle 类的基础上创建起来。因此你需要将 Vehicle 类放在 Bicycle 类后面，用冒号分隔。

我们可将其读作：

“定义一个名为 Bicycle 的新类，此新类可继承 Vehicle 的全部属性”：

```
class Bicycle: Vehicle {
init() {
super.init()
numberOfWheels = 2
}
}
```

Bicycle 是 Vehicle 的一个子类，而 Vehicle 是 Bicycle 的超类。新的 Bicycle 自动获得 Vehicle 的所有属性，例如其 maxPassengers 和 numberOfWheefe 属性。可以在子类中定制此类属性，或添加新的属性以便更好地描述 Bicycle 类。

Bicycle 类定义了一个初始化器来设置它定制的特性。 Bicycle 的初始化器调用了其超类 Vehicle 的初始化器 superinit，以此确保在 Bicycle 类试图修改那些继承来的属性前，Vehicle 类已将其初始化。

> 注
>
> 不同于 Objective-C，在 Swift 中，默认情况下不能继承初始化器。更多相关信息，参见初始化器的继承和重写。

Vehicle 类中 maxPassengers 的默认值对自行车而言是正确的，因此在 Bicycle 的初始化器中并没有对其进行更改。而 numberOfWheels 原值是不正确的，因此在初始化器中将它更改为新值 2。

Bicycle 不仅可以继承 Vehicle 的属性，也能继承其类下的方法。当创建一个 Bicycle 时，你就可以调用它继承来的 description 方法，并且可以看到，它输出的属性值已经发生了变化：

```
let bicycle = Bicycle()
println("Bicycle: \(bicycle.description())")
```

```
// Bicycle: 2 wheels; up to 1 passengers
```

子类还可以继续被其它类继承：

```
class Tandem: Bicycle {
init() {
super.init()
maxPassengers = 2
}
}
```

上面的例子创建了 Bicycle 的一个子类：双人自行车（tandem）。Tandem 从 Bicycle 继承了两个属性，而这两个属性是 Bicycle 从 Vehicle 继承而来的。Tandem 并不修改轮子的数量，毕竟它仍是一辆自行车。但它需要修改 maxPassengers 的值，因为双人自行车可以坐两个人。

> 注
>
> 子类只允许修改从超类继承来的变量属性。而无法修改继承的常量属性。

创建一个 Tandem 实例，打印它的描述，即可看到它的属性已被更新：

```
let tandem = Tandem()
println("Tandem: \(tandem.description())")
// Tandem: 2 wheels; up to 2 passengers
```

注：descripton 方法也可以由 Tandem 继承。类的实例方法可以由全部子类继承。

15 重写

子类可以为继承来的实例方法、类方法、实例属性或其另外从一个超类处继承的下标自定义实现。我们把这种行为称为重写。

如果要重写某个特性，你需要在重写定义的前面加上关键字 override。这么做，你就表明了你是想提

供一个重写版本，而非错误地提供了一个相同的定义。意外地进行重写可能引起意想不到的行为，且任何缺少关键字 override 的重写都会在编译时被诊断为错误。

关键字 override 会提醒 Swift 编译器去检查该类的超类（或其中一个父类）是否有匹配重写版本的声明。此检查可确保你的重写定义是正确的。

15.1访问超类的方法、属性和下标

当你为子类提供重写的方法、属性或下标时，有时在你的重写版本中使用已经存在的超类实现会大有裨益。比如，你可以优化已有实现的行为，或在一个继承来的变量中存储一个修改过的值。

在合适的地方，你可以通过使用 super 前缀来访问超类版本的方法、属性或下标：

重写方法 someMethod 可以在重写方法实现内通过调用 super.someMethod()来调用 someMethod 的超类版本。

重写属性 someProperty 可在 getter 或 setter 实现内将 someProperty 超类版本存储为 super.someProperty。

An overridden subscript for someIndex 的重写下标可在重写下标内将同一下标的超类版本存储为 super[someIndex]

15.2重写方法

在子类中，你可以重写继承来的实例方法或类方法，提供一个定制或替代的方法实现。

下列示例中定义了被称为 Car 的 Vehicle 的一个新子类，该子类重写其从 Vehicle 继承的 description 方法：

```
class Car: Vehicle {
var speed: Double = 0.0
init() {
super.init()
maxPassengers = 5
numberOfWheels = 4
}
override func description() -> String {
return super.description() + "; "
```

```
+ "traveling at \(speed) mph"

Car
```

Car 声明了一个名为 speed 的新存储 Double 属性。此属性默认值 0.0，表示“时速是 0 英里”。Car 有自己的初始化器，该初始化器把乘客最大数量设置到 5，车轮默认数量为 4.

按来自从 Vehicle 的 descripton 方法，通过用相同的描述提供一个方法，car 重写其继承的 descripton 方法。重写方法定义以关键字 override 作为前缀。

Car 中的 description 方法并非完全自定义，而是通过 super.description 使用了超类 Vehicle 中的 description 方法。然后再追加一些额外的信息，比如汽车的当前速度。

如果你创建一个 Car 的新实例，并且打印 descripton 方法输出，你可以看到描述信息已经被更改：

```
let car = Car()

println("Car: \(car.description())")

// Car: 4 wheels; up to 5 passengers; traveling at 0.0 mph
```

15.3 重写属性

你可以重写继承来的实例或类属性，提供自己定制的 getter 和 setter，或添加属性观察者使重写的属性观察属性值的更改时间。

15.4 重写属性 Getters 和 Setters

你可以提供定制的 getter（或 setter，如有）来重写继承来的任一属性，不论其为存储型属性或是计算型属性。子类并不知道继承来的属性是存储型的还是计算型的，它只知道继承来的属性会有一个名称和类型。。你在重写一个属性时，必需将它的名称和类型都写出来。这样才能使编译器去检查你重写的属性是与超类中同名同类型的属性相匹配。

你可以将一个继承来的只读属性重写为一个读写属性，只需要你在重写版本的属性里提供 getter 和 setter 即可。但是，你不能将一个继承来的读写属性重写为一个只读属性。

> 注
>
> 如果在重写属性中提供了 setter，则还需提供 getter。如果你不想在重写版本中的 getter 里修改继承来的属性值，你可以直接返回 super.someProperty 来返回继承来的值。正如下列的

SpeedLimitedCar 的例子所示。

以下示例中定义一个被称为 SpeedLimitedCar 新类，它是 Car 的一个子类。类 SpeedLimitedCar 表示安装了限速装置的车，它的最高速度只能达到 40mph。你可以通过重写继承来的 speed 属性来实现这个速度限制：

```
class SpeedLimitedCar: Car {
override var speed: Double {
get {
return super.speed
}
set {
super.speed = min(newValue, 40.0)
}
}
```

当你设置 SpeedumitedCar 实例的 speed 属性时，属性 setter 实现检查新值且将其限制到 40mph。它会将超类的 speed 设置为 newValue 和 40.0 中的较小者。通过 min 函数决定两个值中的较小者，它是 Swift 标准库中的一个全局函数。min 函数接收两个或更多的数，返回其中的最小值。

若你尝试设置一个 SpeedLimitedCar 的 speed 属性超过 40mph，然后打印其 descripton 方法输出，你会发现该速度被限制在：40mph

```
let limitedCar = SpeedLimitedCar()
limitedCar.speed = 60.0
println("SpeedLimitedCar: \(limitedCar.description())")
// SpeedLimitedCar: 4 wheels; up to 5 passengers; traveling at 40.0 mph
```

15.5 重写属性观察者

你可以在属性重写中为一个继承来的属性添加属性观察者。当继承属性值改变时，你就会被通知到，

无论那个属性原本是如何实现的。更多关于属性观察者的信息，参阅属性观察者。

注

禁止为继承来的常量存储型属性或继承来的只读计算型属性添加属性观察者。这些属性值是不可以被设置的，因此，不应为其重写提供 willSet 或 didSet。

此外还要注，你不可以同时提供重写的 setter 和重写的属性观察者。若想观察属性值的变化，并且你已经为那个属性提供了定制的 setter，你在 setter 中则可以观察到任何值变化了。

以下示例中定义一个被称为 AutomaticCar 的新类，它是 Car 的一个子类。AutomaticCar 类表明汽车带有自动变速器，该变速器能根据当前速度自动选择一个合适的档位。AutomaticCar 还提供了定制的 descripton 方法，可以输出当前档位。

```
class AutomaticCar: Car {
var gear = 1
override var speed: Double {
didSet {
gear = Int(speed / 10.0) + 1
}
}
override func description() -> String {
return super.description() + " in gear \(gear)"
```

当设置一个 AutomaticCar 实例的 speed 属性时，这个属性的 didSet 观察者为新的速度自动设置档位属性至一个合适的档位。具体来说，属性观察者将新的速度值除以 10，然后向下取得最接近的整数值，最后加 1 来得到档位 gear 的值。例如，速度为 10.0 时，档位为 1；速度为 35.0 时，档位为 4：

```
let automatic = AutomaticCar()
automatic.speed = 35.0
println("AutomaticCar: \(automatic.description())")
```

```
// AutomaticCar: 4 wheels; up to 5 passengers; traveling at 35.0 mph in gear 4
```

15.6 防止重写

你可以通过把方法、属性或下标作为 final 来防止其被重写。只需要在声明关键字前加上@final 特性即可。（例如：@final var，@final func，@final class func 以及@final subscript）。

如果你重写了 final 方法、属性或下标，在编译时会报错。在扩展中，你添加到类里的方法、属性或下标也可以在扩展的定义里标记为 final。

你可以通过在关键字 class 前添加@final 特性（@final class）来将整个类标记为 final 的。这样的类是不可被继承的，否则会报编译错误。

16 初始化

初始化是一个编译供使用的一个类、结构体或枚举实例的一个过程。这个过程包含了为实例中的每个属性设置初始值和为其执行必要的准备和初始化任务。

你通过定义初始化器来实施初始化，该初始化器和特别方法相似，该方法可以被调用用来创建某个具体类型的一个新实例。与 Objective-C 初始化不同，Swift 初始化无需返回值。它们的主要任务是保证新实例在第一次使用前完成正确的初始化。

类实例也可以通过 deinitializer 在类实例释放之前执行特定的清除工作。更过关于 deinitializers 的信息，请参阅 Deinitialization。

16.1存储属性的初始赋值

类和结构体在实例创建时，必须为所有存储型属性设置合适的初始值。存储型属性的值不能处于一个未知的状态。

你可以在一个初始化器中为存储型属性赋初值，也可以在定义属性时为其设置默认值。以下章节将详细介绍这两种方法。

> 注
>
> 当你为存储型属性设置默认值或者在初始化器中为其赋值时，它们的值是被直接设置的，而无需调用任何属性观察者。

16.2 初始化器

创建一个新的特定类型的实例时，需要调用初始化器。初始化器的最简形式类似于一个不带任何参数的实例方法，以关键字 init 命名。

下面的例子定义了一个新的结构体 Fahrenheit 来存储在华氏标度中显示的温度。Fahrenheit 结构体有一个存储属性 temperature，属于 Double 类型：

```
struct Fahrenheit {
var temperature: Double
init() {
temperature = 32.0
}
}
var f = Fahrenheit()
println("The default temperature is \(f.temperature)° Fahrenheit")
// prints "The default temperature is 32.0° Fahrenheit"
```

该结构体定义一个不带参数的初始化器 init，并将存储型属性 temperature 的值初始化为 32.0（当用华氏标度表示时的水的冰点）。

16.3 默认属性值

你可以从一个初始化器内设置一个存储属性的初始值，如上所示。同样，你也可以在属性声明时为其设置默认值。当属性被定义时，你通过赋予一个初始值给该属性来指定默认属性值。

> 注
>
> 若一个属性一直采用同一个初始值，则在一个初始化器内提供一个默认值而非设定一个值。最终结果是相同的，只不过默认值跟属性初始化过程结合的更紧密。通过采用默认值使初始化器更简洁、更清晰，且能通过默认值自动推导出属性的类型。同时，借助默认值也能充分利用默认初始化器以及初始化器继承（后续章节将讲到）等特性。

你可以使用更简单的方式在定义结构体 Fahrenheit 时为属性 temperature 设置默认值：

```
struct Fahrenheit {
var temperature = 32.0
}
```

16.4 定制初始化

你可以通过输入参数和可选属性类型来定制初始化过程，也可以在初始化过程中修改常量属性。这些都将在后面章节中提到。

16.5 初始化参数

你可以在定义初始化器时提供初始化参数，为其提供定制初始化所需值的类型和名称。初始化参数有与函数和方法参数相同的功能和语法。

以下示例中定义了一个结构体 Celsius，该结构体存储用摄氏标度表示的温度。结构体 Celsiu 定义了两个不同的定制初始化器：init(fromFahrenheit:)和 init(fromKelvin:)，二者分别通过接受不同刻度表示的温度值来创建新的实例：

```
struct Celsius {
var temperatureInCelsius: Double = 0.0
init(fromFahrenheit fahrenheit: Double) {
temperatureInCelsius = (fahrenheit - 32.0) / 1.8
}
init(fromKelvin kelvin: Double) {
temperatureInCelsius = kelvin - 273.15
}
}
boilingPointOfWater = Celsius(fromFahrenheit: 212.0)
boilingPointOfWater.temperatureInCelsius is 100.0
```

```
freezingPointOfWater = Celsius(fromKelvin: 273.15)
freezingPointOfWater.temperatureInCelsius is 0.0
```

第一个初始化器拥有一个初始化参数，其外部名称为 fromFahrenheit，内部名称为 fahrenheit。第二个初始化器也拥有一个初始化参数，其外部名称为 fromKelvin，内部名称为 kelvin。这两个初始化器都将唯一的参数值转换成摄氏温度值，并保存在属性 temperatureInCelsius 中。

16.6 内部参数和外部参数名称

跟函数和方法参数相同，初始化参数也存在一个在初始化器内部使用的参数名称和一个在调用初始化器时使用的外部参数名称。

然而，初始化器并不像函数和方法那样在括号前有一个可辨别的名称。因此，在调用初始化器时，主要通过初始化器中的参数名称和类型来对其进行确认。正因为参数如此重要，如果你在定义初始化器时未能提供参数的外部名称，Swift 会为每个初始化器的参数自动生成一个跟内部名称相同的外部名称，就相当于在每个初始化参数之前加了一个 hash 字符。

注

若你不想为在初始化器内的参数提供一个外部名称，则可以为该参数提供一个下划线（_）作为一个明确的外部名称来重写上述默认行为。

下列示例中定义了一个结构体 Color，该结构体包含三个常量属性 red、green 和 blue。这些属性存储 0.0 到 1.0 的值来表明颜色中红、绿和蓝成分的含量。

Color 提供了一个初始化器，其中包含三个 Double 类型的经命名的参数：

```
struct Color {
let red = 0.0, green = 0.0, blue = 0.0
init(red: Double, green: Double, blue: Double) {
self.red = red
self.green = green
self.blue = blue
}
```

```
}
```

当创建一个新的 Color 实例时，你都需要通过三种颜色的外部参数名称来传值，并调用初始化器。

```
let magenta = Color(red: 1.0, green: 0.0, blue: 1.0)
```

注，如果不通过外部参数名称传值，你是没法调用这个初始化器的。只要初始化器定义了某个外部参数名称，你就必须使用它，并忽略它将导致编译错误：

```
let veryGreen = Color(0.0, 1.0, 0.0)
// this reports a compile-time error - external names are required
```

16.7 可选属性类型

如果你定制的类型包含一个逻辑上允许取值为空的存储型属性--不管是因为它无法在初始化时赋值，还是因为它可以在之后某个时间点可以赋值为空--你都需要将它定义为可选类型 optional type。

可选类型的属性将自动初始化为 nil，表示这个属性是故意在初始化时设置为“no value yet”的。

下列示例中定义了类 SurveyQueston，它包含一个可选字符串属性 response：

```
class SurveyQuestion {
var text: String
var response: String?
init(text: String) {
self.text = text
}
func ask() {
println(text)
}
cheeseQuestion = SurveyQuestion(text: "Do you like cheese?")
cheeseQuestion.ask()
```

```
prints "Do you like cheese?"

cheeseQuestion.response = "Yes, I do like cheese."
```

调查问题在问题提出之后，我们才能得到回答。因此，我们将属性回答 response 声明为 String?类型，或者说是“可选字符串类型”。当 SurveyQuestion 实例化时，它将自动赋值为 nil，表明“no string yet”。

16.8 初始化过程中常量属性的修改

只要在初始化过程结束前常量的值能确定，你可以在初始化过程中的任意时间点修改常量属性的值。

> 注
>
> 对于类实例，一个常量属性仅仅在引入该属性的类的初始化过程中才能被修改。该属性不能被子类修改。

你可以修改上面的 SurveyQuestion 示例，用常量属性替代变量属性 text，指明问题内容 text 在其创建之后不会再被修改。尽管 text 属性现在是常量，我们仍然可以在其类的初始化器中修改它的值：

```
class SurveyQuestion {

let text: String

var response: String?

init(text: String) {

self.text = text

}

func ask() {

println(text)

}

beetsQuestion = SurveyQuestion(text: "How about beets?")

beetsQuestion.ask()

prints "How about beets?"
```

```
beetsQuestion.response = "I also like beets. (But not with cheese.)"
```

16.9 默认初始化器

Swift 将为所有属性已提供默认值的且自身没有定义任何初始化器的结构体或基类，提供一个默认的初始化器。这个默认初始化器将简单的创建一个所有属性值都设置为默认值的实例。

以下示例中创建了一个类 ShoppingListItem，它封装了购物清单中的某一项的属性：名称、数量和购买状态：

```
class ShoppingListItem {
var name: String?
var quantity = 1
var purchased = false
}
var item = ShoppingListItem()
```

因为 ShoppingystItem 类所有属性有默认值，且因为该类是基类且没有超类，它将自动获得一个可以为所有属性设置默认值的默认初始化器（尽管代码中没有显式为 name 属性设置默认值，但由于 name 是可选字符串类型，它将默认设置为 nil）。上述示例中使用默认初始化器创造了一个 ShoppingListItem 类的实例（使用 ShoppingListItemo 形式的初始化器语法），并将其赋值给变量 item。

16.10 结构体类型的逐一成员初始化器

除了上述提及的默认初始化器之外，若结构体类型对其存储属性提供了默认值且自身没有提供定制的初始化器，它们能自动获得一个逐一成员初始化器。

逐一成员初始化器是用来初始化结构体新实例里成员属性的快捷方法。我们在调用逐一成员初始化器时，通过与成员属性名称相同的参数名称进行传值来完成对成员属性的初始赋值。

以下示例中定义了一个结构体 Size，它包含两个属性 width 和 height。Swift 可以根据这两个属性的初始赋值 0.0 自动推导出它们的类型 Double。

由于这两个存储型属性都有默认值，结构体 Size 自动获得了一个逐一成员初始化器 init(width:height:)。你可以调用其对新建的 Size 实体进行初始化：

```
struct Size {
var width = 0.0, height = 0.0
```

```
}

let twoByTwo = Size(width: 2.0, height: 2.0)
```

16.11 值类型的初始化器代理

初始化器可以调用其他初始化器来对一个实例进行部分初始化。这个过程，被称为初始化器代理，它能够减少多个初始化器间的代码重复。

初始化器代理的实现规则和形式在值类型和类类型中有所不同。值类型（结构体和枚举类型）不支持继承，因此，初始化器代理的过程相对简单，因为它们只能代理任务给本身提供的其它初始化器。类则不同，它可以继承自其它类。请参阅继承。这意味着类有责任保证其所有继承的存储型属性在初始化时也能正确的初始化。这些责任将在后续章节类的继承和初始化过程中介绍。

对于值类型，你可以使用 self.init 在自定义的初始化器中引用其它的属于相同值类型的初始化器。并且你只能在 initializer 调用 sef.init。

注：若你对值类型定义一个定制初始化器，你将无法访问该类型的默认初始化器（如果是一个结构体，则无法访问逐一成员初始化器）。这个限制可以防止你在为值类型定义了一个更复杂的、完成了重要准备初始化器之后，别人仍错误的使用了那个自动生成的初始化器。

注

假如你想通过默认初始化器、逐一成员初始化器以及你自己定制的初始化器为值类型创建实例，你应将自己定制的初始化器写到扩展（extension）中，而不是跟值类型定义混在一起。更多相关信息，请查看扩展章节。

以下示例中定义一个结构体 Rect，用来展现几何矩形。这个例子需要两个辅助结构体 Size 和 Point，它们各自为其属性提供了初始值 0.0：

```
struct Size {

var width = 0.0, height = 0.0

}

struct Point {

var x = 0.0, y = 0.0

}
```

你可以通过以下三种方式为结构体 Rect 创建实例--使用默认的 0 值来初始化属性 origin 和 size；使用特定的 origin 和 size 实例来初始化，或使用特定的 center 和 size 来初始化。在下列结构体 Rect 定义中，我们为这三种方式提供了三个定制的初始化器：

```
struct Rect {
var origin = Point()
var size = Size()
init() {}
init(origin: Point, size: Size) {
self.origin = origin
self.size = size
}
init(center: Point, size: Size) {
let originX = center.x - (size.width / 2)
let originY = center.y - (size.height / 2)
self.init(origin: Point(x: originX, y: originY), size: size)
```

第一个 Rect 初始化器 init ()，在功能上跟未定制初始化器时自动获得的默认初始化器是一样的。这个初始化器是一个空函数，使用一对大括号{}来描述，它没有执行任何定制的初始化过程。 调用该初始化器将返回一个 Rect 实例，该实例的 origin 和 size 属性都使用定义时的默认值 Point(x: 0.0, y: 0.0)和 Size(width: 0.0, height: 0.0)：

```
let basicRect = Rect()
// basicRect's origin is (0.0, 0.0) and its size is (0.0, 0.0)
```

第二个 Rect 初始化器 init(origin:size:)，在功能上跟结构体在未定制初始化器时获得的逐一成员初始化器是一样的。该初始化器只是简单的将 origin 和 size 的参数值赋给对应的存储型属性：

```
let originRect = Rect(origin: Point(x: 2.0, y: 2.0),
```

```
size: Size(width: 5.0, height: 5.0))
// originRect's origin is (2.0, 2.0) and its size is (5.0, 5.0)
```

第三个 Rect 初始化器 init(center:size:)稍微复杂一点。它先通过 center 和 size 的值计算出 origin 的坐标。然后再调用（或代理给）init(origin:size:) 初始化器来将新的 origin 和 size 值赋值到对应的属性中：

```
let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
size: Size(width: 3.0, height: 3.0))
// centerRect's origin is (2.5, 2.5) and its size is (3.0, 3.0)
```

初始化器 init(center:size:)可以自己将 origin 和 size 的新值赋值到对应的属性中。然而尽量利用现有的初始化器和它所提供的功能来实现 init(center:size:)的功能，是更方便、更清晰和更直观的方法。

注

如果你想用另外一种不需要自己定义 init()和 init(origin:size:)的方式来实现这个例子，请参阅扩展。

16.12 类的继承和初始化

所有类存储属性-包括从其超类继承的类——在初始化过程中必须被赋予一个初始值。Swift 提供了两种类型的类初始化器来确保所有类实例中存储型属性都能获得初始值。被称为指定初始化器和便利初始化器。

16.13 指定初始化器和便利初始化器

指定初始化器是类中最主要的初始化器。一个指定的初始化器将初始化类中提供的所有属性，并根据超类链往上调用超类的初始化器来实现超类的初始化。

类通常很少指定初始化器，通常情况下，一个类只有一个初始化器。通过所发生的初始化和初始化持续至超类链，被指定的初始化器是“funnel”点。

每一个类都必须至少有一个指定初始化器。在某些情况下，许多类通过继承了超类中的指定初始化器而满足了这个条件。具体内容请参阅后续章节自动初始化器的继承。

便利初始化器是类中比较次要的、辅助型的初始化器。你可以定义便利初始化器来调用同一个类中的指定初始化器，并为其参数提供默认值。你也可以定义便利初始化器来创建一个特殊用途或特定输入的实例。

你应当只在必要的时候为类提供便利初始化器。比方说某种情况下通过使用便利初始化器来快捷调用某个指定初始化器，能够节省更多开发时间并让类的初始化过程更清晰、明确。

16.14 初始化器链

为简化指定初始化器和便利初始化器之间的调用关系，Swift 采用以下三条规则来限制初始化器之间的代理调用：

指定初始化器必须从其直接超类中调用指定初始化器。

便利初始化器必须调用同一类中定义的其它初始化器。一个更方便记忆的方法是：

这些规则说明如下图所示：详见图 16-1

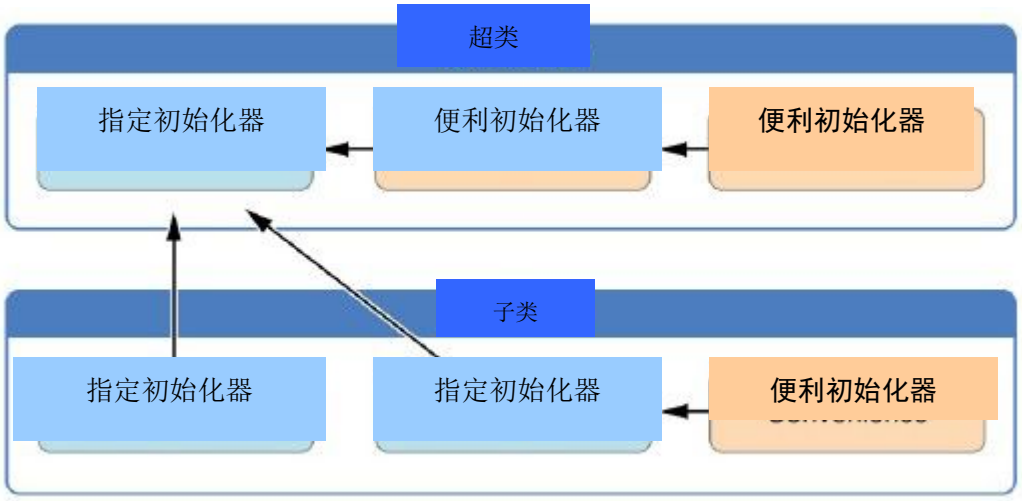


图 166-1 初始化器链

如图所示，该超类包含一个指定初始化器和两个便利初始化器。其中一个便利初始化器调用了另外一个便利初始化器，而后者又调用了唯一的指定初始化器。满足上述规则 2 和 3。这个超类没有对应的超类，因此规则 1 不适用。

此图中的子类包含两个指定初始化器和一个便利初始化器。便利初始化器必须调用两个指定初始化器中的任一个，这是因为该便利初始化器只能从同一个类中调用另一个指定初始化器。满足上述规则 2 和 3。这两个指定初始化器均须从超类中调用唯一的指定初始化器，以满足上述规则 1 的要求。

注

这些规则不会影响你的类用户创建每个类实例的方式。上图中的任何初始化器均可用于为其所属类创建完全初始化实例。这些规则只在实现类的定义时有影响。

下图显示了四个类的更复杂的类层级结构。它演示了指定初始化器是如果在类层级中充当“funnel”

的作用，在类的初始化器链上简化了类之间的内部关系：详见图 16-2

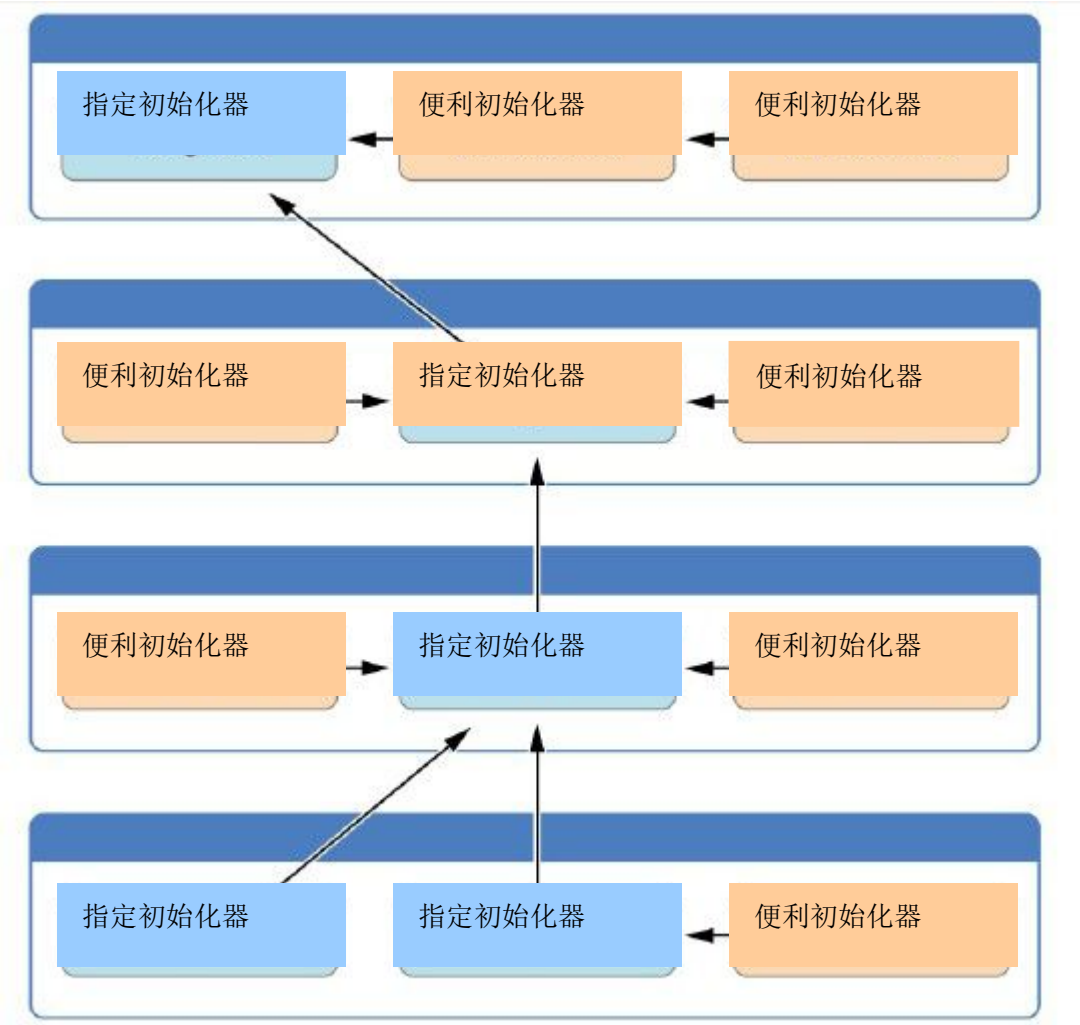


图 166- 2 类层级结构

16.15 两段式初始化过程

Swift 中类的初始化过程分两个阶段。在第一阶段，引入所存储各属性的类为每个属性分配一个初始值。一旦确定各存储属性的初始状态后即开始第二阶段，并且各类均有机会在新实例准备运行前对其所存储属性进行进一步定制。

两段式初始化过程的使用让初始化过程更安全，同时在整个类层级结构中给予了每个类完全的灵活性。两段式构造过程可以防止属性值在初始化之前被访问；也可以防止属性被另外一个初始化器意外地赋予不同的值。

注

Swift 的两段式初始化过程与 Objective-C 中的初始化过程类似，主要的区别在于：在第一阶段中，Objective-C 将各属性赋值零或 null 值（0 或 nil）。Swift 的初始化流程则更加灵活，它可允许用户设置定制的初始值，并自如应对某些属性不能以 0 或 nil 作为合法默认值的情况。

Swift 编译器将执行四种有效的安全检查，以确保两段式构造过程能顺利完成：

指定初始化器必须确保它所在类引入的所有属性都必须先初始化完成，之后才能将其它构造任务向上代理给超类中的初始化器。

如上所述，某对象的内存仅在已知其所有存储属性的初始状态后才可被视为完全初始化。为满足此规则，指定初始化器必须确保它所在类引入的属性在它往上代理之前先完成初始化。

指定初始化器必须先向上代理调用超类初始化器，然后再为继承的属性设置新值。否则，指定初始化器赋予的新值将被超类中的初始化器所重写。

在为任一属性（包括同一类定义的属性）复制前，便利初始化器必须先代理调用同一类中的其它初始化器。否则，便利初始化器赋予的新值将被同一类中其它指定初始化器所重写。

在第一阶段初始化完成前，初始化器无法调用任何实例方法、读取任何实例的属性值或将 self 作为一个值引用。

类未完全实例化，直到第一阶段结束。一旦在第一阶段结束后已知类实例有效，则仅可对属性进行访问，且仅可对方法进行调用。

基于以上四次安全检查，下文介绍如何完成两段式初始化：

16.15.1 阶段 1

一个配置的或方便的初始化函数被称为一类。

对该类的新实例的记忆进行配置，该记忆还未被初始化。

为该类配置的初始化函数证实了由该类引进的所有存储属性均有，这些存储属性的记忆现在被初始化。.

直到抵达该链的顶端，该配置的初始化函数不干涉超类初始化函数执行相同的 This 继续类的继承链。

一旦达到链的顶端, 该链的最后类已确保其所有的存储属性均有值, 该实例的记忆被认为已经全部初始化, 阶段 1 完成了。

16.15.2 阶段 2

从链的顶端往下工作，该链内的每个配置的初始化函数均有进一步自定义该实例的选项，初始化函数现在可以访问 self 并修改其属性，调用其实例类函数等等。

最后，任何该链的方便初始化函数均有自定义选项。

下文介绍阶段 1 如何针对假设子类和超类搜索初始化调用：详见图 16-3

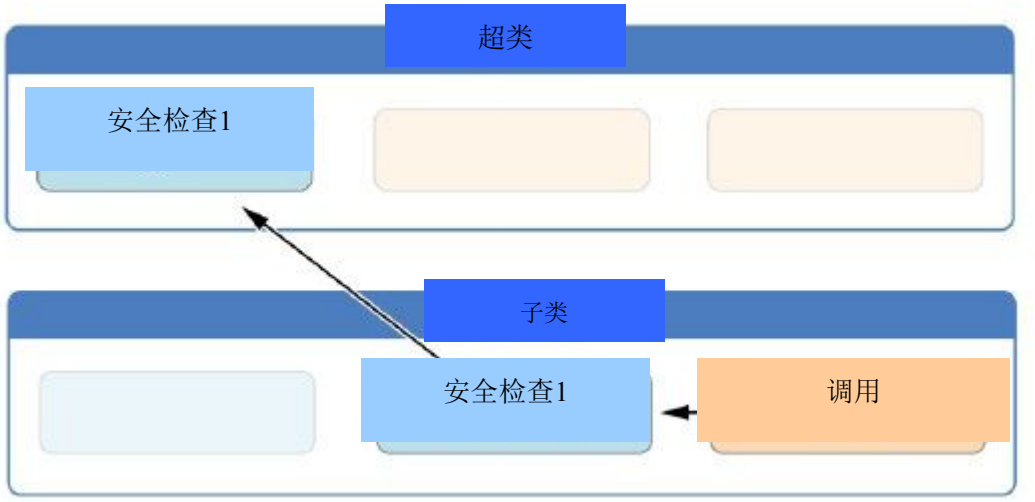


图 166-1 阶段 1 初始化过程

本例中，对子类的便利初始化器进行调用后即开始初始化过程。便利初始化器尚不能修改任何属性。它把初始化任务代理给同一类中的指定初始化器。

根据安全检查 1，指定初始化器将确保子类的所有属性均有值；然后，该初始化器调用其超类的指定初始化器，以继续在初始化器链上进行初始化。

超类的指定初始化器确保所有超类属性均有值。由于没有更多的超类需要初始化，也就无需继续向上做初始化代理。

当超类的所有属性均有初始值时，实例的内存被认为是完全初始化，而阶段 1 也已完成。

下述为在阶段 2 查找相同的初始化调用的过程：详见图 16-4

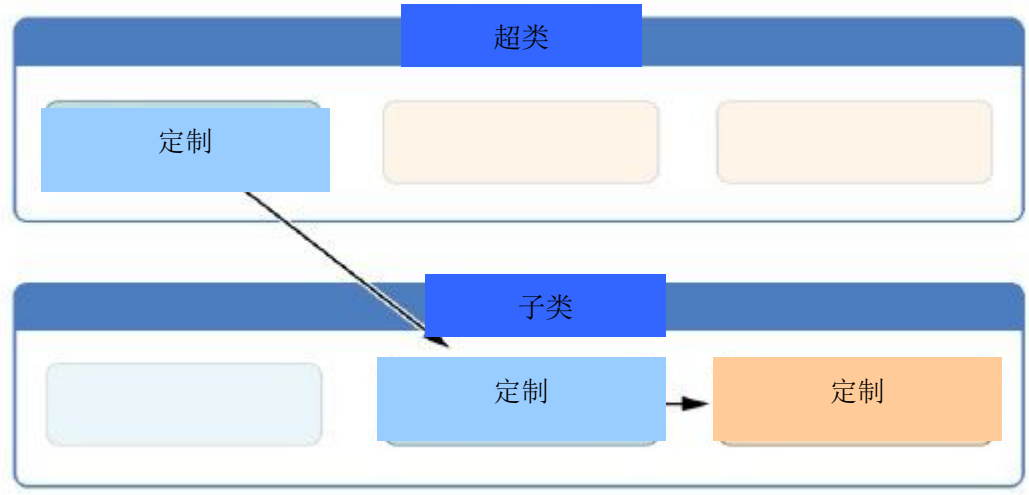


图 166-4 阶段 2 初始化过程

超类中的指定初始化器现在有机会进一步来定制实例（尽管它没有这种必要）。

一旦超类中的指定初始化器完成调用，子类的构指定构造器可以执行更多的定制操作（同样，它也没有这种必要）。

最终，一旦子类的指定初始化器完成调用，最开始被调用的便利初始化器可以执行更多的定制操作。

16.16 始化器的继承和重写

与 Objective-C 中的子类不同，Swift 中的子类不以默认的方式继承其超类的初始化器。Swift 的这种机制可以防止一个超类的简单初始化器被一个更专业的子类继承，并被错误的用来创建子类的实例。

假如你希望定制的子类中能实现一个或多个跟超类相同的初始化器--也许是为了完成一些定制的初始化过程--你可以在你定制的子类中提供和重写与超类相同的初始化器。

如果你重写的初始化器是一个指定初始化器，你可以在子类里重写它的实现，并在自定义版本的初始化器中调用超类版本的初始化器。

如果你重写的初始化器是一个便利初始化器，你的重写过程必须通过调用同一类中提供的其它指定初始化器来实现。这一规则的详细内容请参阅初始化器链。

注

与方法、属性和下标不同的是，重写初始化器时无需使用关键字 override。

16.17 自动初始化器的继承

如上所述，子类不通过默认的方式继承其超类初始化器。但是，如果满足特定条件，超类初始化器将被自动继承。实际上，这意味着在多数一般情况下，用户无需重写超类初始化器，并可在安全情况下通过

最少的操作步骤继承超类初始化器。

假设要为子类中引入的任意新属性提供默认值，请遵守以下两个规则：

如果子类不对任何指定初始化器进行定义，则它将自动继承其所有超类的指定初始化器。

如果子类可实现其所有超类的指定初始化器-无论是按照规则 1 继承超类初始化器还是通过定制（作为其定义的一部分）实现-则该子类将自动继承所有超类便利初始化器。

即使你在子类中添加了更多的便利初始化器，这些规则也同样适用。

注

子类可以通过部分满足规则 2 的方式，使用子类便利初始化器来实现超类的指定初始化器。

16.18 指定初始化器和便利初始化器的语法

类的指定初始化器的写法跟值类型简单初始化器一样：

```
init( parameters ) {
statements
}
```

便利初始化器也采用相同样式的写法，但需要在关键字 init 之前写入关键字 convenience，并使用空格将其分开：

```
convenience init( parameters ) {
statements
}
```

16.19 指定初始化器和便利初始化器的实践

下列示例将在实战中展示指定初始化器、便利初始化器和自动初始化器的继承。它定义了包含三个类 Food、RecipeIngredient 以及 ShoppingListItem 的类层次结构，并将演示它们的初始化器是如何相互作用的。

层次中的基类为 Food，它是一个简单的用来封装食物名字的类。Food 类引入了一个名为 name 的 String 属性，并提供两个初始化器来创建 Food 实例：

```
class Food {
```

```
var name: String

init(name: String) {

self.name = name

}

convenience init() {

self.init(name: "[Unnamed]")

}

}
```

下图为 Food 类的初始化器链：详见图 16-5

图 16-5 Food 类初始化器链

```
var quantity: Int

init(name: String, quantity: Int) {

self.quantity = quantity

super.init(name: name)

}

convenience init(name: String) {

self.init(name: name, quantity: 1)

}
```

下图为 RecipeIngredient 类的初始化器链：详见图 16-6

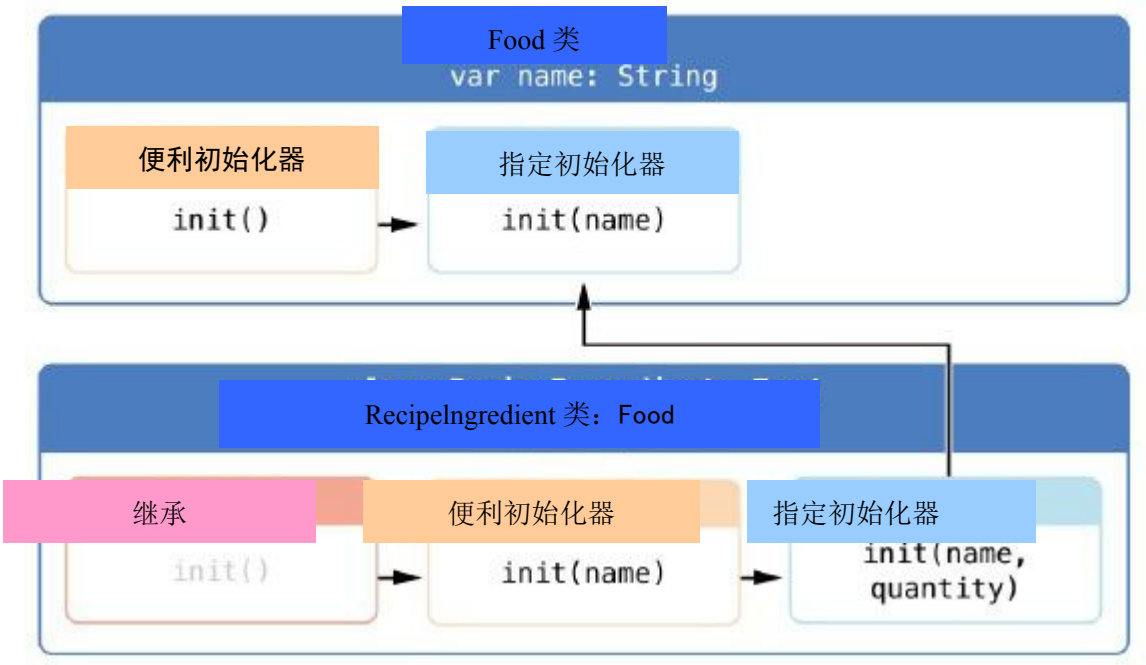


图 16-6

RecipeIngredient 类的初始化器链

RecipeIngredent 类拥有一个指定初始化器 int（name：String, quantity：Int），它可用于产生新 RecipeIngredient 实例的所有属性值。这个初始化器首先将传入的 quantity 参数赋值给属性 quantity，该属性也是唯一在 RecipeIngredient 中新引入的属性。。然后，初始化器将任务向上代理给超类 Food 的 init(name: String)。这个过程满足两段式初始化过程中的安全检查 1。

RecipeIngredent 还定义了一个便利初始化器 init（name：String），可以仅通过 name 新建一个 RecipeIngredent 实例。这个便利初始化器假设任意 RecipeIngredient 实例的 quantity 为 1，因此，不需要显示指明数量即可创建出实例。这个便利初始化器的定义可以让创建实例更加方便和快捷，并且避免了使用重复的代码来创建多个 quantity 为 1 的 RecipeIngredient 实例。这个便利初始化器只是简单的将任务代理给了同一类里提供的指定初始化器。

注：RecipeIngredent 提供的 init（name:String）便利初始化器与 Food 中的 init（name:String）指定初始化器具有相同参数。即使 RecipeIngredent 初始化器是便利初始化器，RecipeIngredient 依然提供了对所有超类指定初始化器的实现。因此，RecipeIngredent 也自动继承其所有超类的便利初始化器。

本例中，RecipeIngredent 的超类为 Food，其中含有一个名为 init()的便利初始化器。因此，RecipeIngredent 可继承该初始化器。该继承的 init()函数版本跟 Food 提供的版本是一样的，除了它是将任务代理给 RecipeIngredient 版本的 init(name: String)而不是 Food 提供的版本。

所有的这三种初始化器都可以用来创建新的 RecipeIngredient 实例：

```
let oneMysteryItem = RecipeIngredient()

let oneBacon = RecipeIngredient(name: "Bacon")

let sixEggs = RecipeIngredient(name: "Eggs", quantity: 6)
```

类层级中第三个也是最后一个类是 RecipeIngredient 的子类-ShoppingListItem。ShoppingListItem 构建了购物单中出现的某一种调味料。

购物单中的每一项总是从“unpurchased”状态开始的。为体现这一事实，ShoppingListItem 引入了名为 purchased 的布尔属性，它的默认值是 false。ShoppingListItem 还添加了计算型属性 description，该属性提供 ShoppingListItem 实例的一些文字描述：

```
class ShoppingListItem: RecipeIngredient {

var purchased = false

var description: String {

var output = "\(quantity) x \(name.lowercaseString)"

output += purchased ? " ✔" : " ✘"

return output

}

}
```

注

ShoppingListItem 没有定义初始化器来为 purchased 提供初始化值，这是因为任何添加到购物单（在此构建）的项的初始状态总是 unpurchased。

由于它为所有由其引入的属性提供默认值并且不自动对任何初始化器进行定义，因此，ShoppingListItem 将自动从其超类继承所有的指定和便利初始化器。

下图为所有三个类的完整初始化器链：详见图 16-7

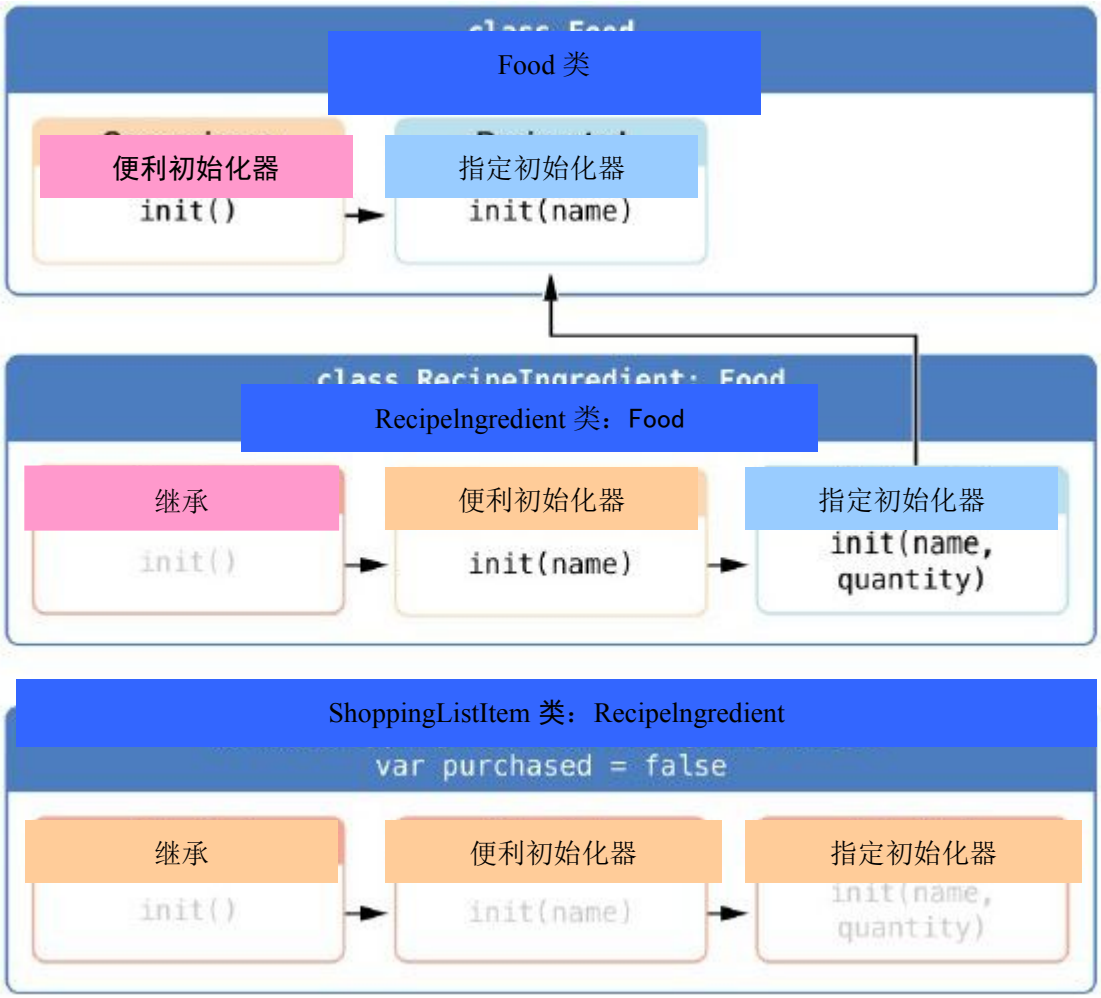


图 16-7 完整初始化器链

用户可使用所继承的所有这三个初始化函数来创建新的 ShoppingListItem 实例：

```
var breakfastList = [
ShoppingListItem(),
ShoppingListItem(name: "Bacon"),
ShoppingListItem(name: "Eggs", quantity: 6),
]
breakfastList[0].name = "Orange juice"
breakfastList[0].purchased = true
for item in breakfastList {
println(item.description)
```

x orange juice ✔

x bacon ✘

x eggs ✘

这时，在一个含有三个 ShoppingListItem 实例的数字标识符中创建了一个新的名为 breakfastList 的数组。经推断，该数组类型为 ShoppingListItem【】。创建数组后，数组前部的 ShoppingListItem 名称由“【Unnamed】”更改为“Orange juice”，即标记为已购买。通过对数组中的各项说明进行打印表明，其默认状态已设置为预期状态。

16.20 以闭包或函数设置默认属性值

如果所存储属性的默认值需要进行自定义或设置，用户可使用闭包或全局函数来为该属性提供自定义默认值。当对属性所属类型的新实例进行初始化时，闭包或函数被调用，并且其返回值被赋值为属性默认值。

以上类别的闭包或函数通常创建一个与属性类型相同的临时值，并调整该值使其可表示所需的初始状态；然后，返回将要作为属性默认值使用的临时值。

本文针对如何将闭包用于提供默认属性值进行了概述：

```
class SomeClass {
let someProperty: SomeType = {
// create a default value for someProperty inside this closure
// someValue must be of the same type as SomeType
return someValue
}()
}
```

注，闭包的终止花括号后为一对空白圆括号。其命令 Swift 立即执行闭包。如果漏掉了圆括号，则会将闭包本身分配给属性，而不是闭包的返回值。

注

如果采用闭包来对属性进行初始化，切记当执行闭包时，其余实例并未被初始化。这意味着用户无法从自身闭包内部访问其他任何属性值，即使这些属性有默认值。同时，用户也无法使用隐式 self 属性或调用任何该实例的类函数。

以下示例对名为 Checkerboard 的结构进行了定义，该结构对跳棋（也叫国际跳棋）所用的游戏板进行了模拟：详见图 16-8

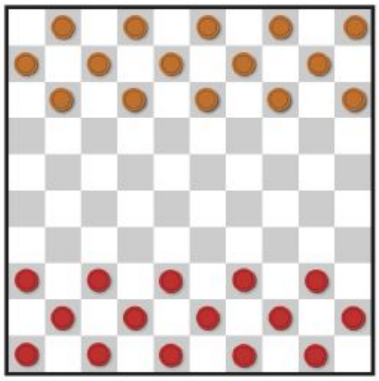


图 16- 8 Checkerboard 的结构

跳棋所用的游戏板为 10×10 的黑白方块相间的正方形游戏板。Checkerboard 结构具有名为 boardColors 的单一属性用以创建该游戏板，该属性是含有 100 个 Bool 值的数组。数组中的 true 值代表一个黑色方块，false 值代表一个白色方块。阵列中的第一项代表游戏板上的左上角方格，最后一项代表游戏板上的右下角方格。

利用一个闭包初始化 boardColors 数组以设置其颜色值：

```
struct Checkerboard {
let boardColors: Bool[] = {
var temporaryBoard = Bool[]()
var isBlack = false
for i in 1...10 {
for j in 1...10 {
temporaryBoard.append(isBlack)
isBlack = !isBlack
}
isBlack = !isBlack
}
```

```
return temporaryBoard
}()
func squareIsBlackAtRow(row: Int, column: Int) -> Bool {
return boardColors[(row * 10) + column
```

当创建好一个新的 Checkerboard 实例后，执行闭包，并且计算并返回 boardColors 的默认值。以上示例中的闭包为名为 temporaryBoard 的临时阵列中游戏板上的每个方格计算并设置适当的颜色，并在完成设置后，将此临时数组作为闭包的返回值返回。所返回的数组值被存储在 boardColors 中，并且可通过 squareIsBlackAtRow 效用函数对其进行查询：

```
let board = Checkerboard()
println(board.squareIsBlackAtRow(0, column: 1))
// prints "true"
println(board.squareIsBlackAtRow(9, column: 9))
// prints "false"
```

16.21 反初始化

在解除类实例配置之前，应立即调用反初始化。用户通过使用 deinit 关键字编写反初始化函数，方法类似于采用 init 关键字编写初始化函数。反初始化只适用于类类型。

16.22 反初始化的操作原理

当不再需要用户实例时，Swift 自动对其解除配置以释放资源。Swift 通过自动引用计数（ARC）管理实例内存-见自动引用计数下的介绍。通常，当解除用户实例配置时，用户无需进行手动清理。但用户使用自身资源工作时，可能需要另外亲自清理。例如，如果用户创建一个自定义类并用其来打开文件和向其中写入数据，则可能需要在解除类实例配置前关闭该文件。

每个类的类定义最多可有一个反初始化。反初始化时不用采用任何参数，且写入时不需要圆括号：

```
deinit {
// perform the deinitialization
}
```

反初始化函数在解除实例配置前才自动被调用。用户不得自行调用反初始化函数。超类反初始化函数

由其子类继承，并在子类反初始化函数实现过程结束时自动被调用。超类反初始化函数经常被调用，即使超类不提供其自身的反初始化函数。

由于实例分配仅在实例的反初始化函数被调用后才被解除，因此在反初始化函数在被调至某实例后可访问该实例的所有属性，还可以根据这些属性修改其特性（例如查找需关闭的文件的名称）。

16.23 运转中的反初始化函数

下面是关于运转中的反初始化函数的例子：本示例定义了一个简单游戏的两种新类型，即 Bank 和 Player。Bank 结构管理虚拟货币，该虚拟币流通数额不得超过 10,000 个虚拟币。该游戏中只可能有一个 Bank,所以，该 Bank 作为带有静态属性和类函数的结构实现，并用于存储和管理其当前状态：

```
struct Bank {
static var coinsInBank = 10_000
static func vendCoins(var numberOfCoinsToVend: Int) -> Int {
numberOfCoinsToVend = min(numberOfCoinsToVend, coinsInBank)
coinsInBank -= numberOfCoinsToVend
return numberOfCoinsToVend
}
static func receiveCoins(coins: Int) {
coinsInBank += coins
```

Bank 通过其 coinsInBank 属性持续追踪其所持有虚拟币的当前数量。同时，还提供两个类函数 -vendCoins 和 receveCoins-用于处理虚拟币分配和收集。

在分发之前，vendCoins 检查 bank 是否有足够的虚拟币。如果虚拟币不足，Bank 会返还少于所需数额的虚拟币（如果银行中没有虚拟币则返还至零）。vendCoins 声明 numberOfCoinsToVend 为可变参数，因此，该数量可在类函数体内进行修改（无需定义新变量）。它返还一个整数值以表示所提供虚拟币的实际数量。

receiveCoins 类函数仅向银行的虚拟币库返还其所收到的虚拟币。

Player 类描述游戏中的一个玩家。在任何时候，每个玩家的钱包内均存有一定数量的虚拟币。由玩家的 coinsInPurse 属性表示：

```
class Player {
var coinsInPurse: Int
init(coins: Int) {
```

```
coinsInPurse = Bank.vendCoins(coins)
}
func winCoins(coins: Int) {
coinsInPurse += Bank.vendCoins(coins)
}
deinit {
Bank.receiveCoins(coinsInPurse)
```

在初始化过程中，各 Player 实例均在允许银行放出一定数量的虚拟币起开始初始化，但如果虚拟币不足，Player 实例可能会收到少于此数量的虚拟币。

Player 类对 winCoins 类函数进行了定义，该函数从银行取得一定数量的虚拟币并将其放入玩家钱包。Player 类还实现了一个反初始化函数，该函数在解除 Player 实例配置前才被调用。此处，反初始化函数仅可以将玩家的所有虚拟币返回到银行内：

```
var playerOne: Player? = Player(coins: 100)
println("A new player has joined the game with \(playerOne!.coinsInPurse) coins")
// prints "A new player has joined the game with 100 coins"
println("There are now \(Bank.coinsInBank) coins left in the bank")
// prints "There are now 9900 coins left in the bank"
```

已创建好新的 Player 实例，需要 100 个虚拟币（如有）。该 Player 实例存储于一个名为 playerOne 的可选 Player 变量内。由于玩家可以在任何时候退出游戏，所以这里使用一个可选变量。可选项可以使你跟踪查看在当前的游戏中是否有一个玩家。

由于 playerOne 为可选项，所以当访问其 coinsInPurse 属性以打印其虚拟币的默认数量时以及当调用其 winCoins 类函数时，带有叹号（!）才为合格：

```
playerOne!.winCoins(2_000)
println("PlayerOne won 2000 coins & now has \(playerOne!.coinsInPurse) coins")
// prints "PlayerOne won 2000 coins & now has 2100 coins"
println("The bank now only has \(Bank.coinsInBank) coins left")
// prints "The bank now only has 7900 coins left"
```

在这里，玩家已赢得了 2000 个虚拟币。现在，玩家的钱包内有 2100 个虚拟币，银行内仅余下 7900 个虚拟币。

```
playerOne = nil
println("PlayerOne has left the game")
// prints "PlayerOne has left the game"
println("The bank now has \(Bank.coinsInBank) coins")
// prints "The bank now has 10000 coins"
```

现在，玩家已经退出了游戏。这通过将可选 playerOne 变量设为 nil（意为“无 Player 实例”）来显示。出现这种情况时，playerOne 变量对 Player 实例的引用被破坏。不存在针对 Player 实例引用的其他属性或变量，因此，对其解除配置以释放其内存。在此之前，其反初始化函数被自动调用，并且其虚拟币也被返还给银行。

16.24 自动引用计数

Swift 采用自动引用计数（ARC）来追踪和管理用户对应用程序内存的使用。在大多数情况下，这说明内存管理“仅工作”于 Swift 中，并且用户无需考虑自身内存管理。当不再需要这些类实例时，ARC 自动释放被类实例所占用的内存。

但是，ARC 偶尔会需要更多有关用户代码部分之间关系的信息，以便帮助用户管理内存。本章介绍了这些情况并说明了如何启用 ARC 来管理用户所有应用程序的内存。

注

引用计数仅适用于类实例。结构和枚举是值类型而不是引用类型，它们不通过引用进行存储或传递。

16.25 ARC 运行原理

每次为类创建新实例时，ARC 均会分配出一个内存组块来存储有关此实例的信息。该内存保存有关实例类型的信息以及与该实例相关的任何所存储属性的值。

此外，当不再需要某实例时，ARC 会释放出该实例占用的内存以便作其他用途。这样，保证了在不再需要某些实例的情况下，这些类实例不会占用内存空间。

但是，如果 ARC 对处于使用状态的实例解除配置；则将无法再次访问该实例的属性，也无法调用该实例类函数。事实上，如果用户试图访问本实例，其应用程序很可能会崩溃。

为确保实例不在被需要时消失，ARC 对当前各类实例引用的属性、常量和变量进行追踪。若至少存在一个针对该实例的主动引用，则 ARC 不会对此实例解除配置。

要使这成为可能，当用户为某属性、常量或变量分配一个类实例时，该属性、常量或变量则成为针对该实例的强引用。由于该引用对该实例具有有力控制，因此被成为“强”引用,并且只要此强引用一直存在，其将不会被允许解除配置。

16.26 运转中的 ARC

本示例为自动引用计数的工作原理。此示例以一个名为 Person 的简单类开始，该类定义了所存储的名为 name 的常量属性：

```
class Person {
let name: String
init(name: String) {
self.name = name
println("\(name) is being initialized")
}
deinit {
println("\(name) is being deinitialized")
}
```

Person 类具有一个初始化函数，该函数设置实例的 name 属性并打印出一条消息以显示正在进行初始化。Person 类还具有一个反初始化函数，该函数在该类的一个实例被解除配置后打印出一条消息。

下面的代码段定义了 Person？类型的 3 个变量它用于在后续代码片段中针对新 Person 实例设置多次引用。由于这些变量为可选类型（Person?不是 Person），所以，他们将自动与 nil 值进行初始化，并且它们目前不引用 Person 实例。

```
var reference1: Person?
var reference2: Person?
var reference3: Person?
```

用户现在可创建一个新的 Person 实例并将其分配给这三个变量中的一个：

```
reference1 = Person(name: "John Appleseed")
```

```
// prints "John Appleseed is being initialized"
```

注：在用户调用 Person 类的初始化函数时，将打印出“John Appleseed is being initialized”这条消息。这证实了初始化已经发生。

由于已将新的 Person 实例分配给 reference 1 的变量，所以，自 reference1 到新增 Person 实例之间存在强引用。由于至少存在一个强引用，ARC 确保将此 Person 存储在内存中并不对其解除配置。

如果用户将同一 Person 实例分配给两个或以上的变量，则将因此建立出针对该实例的两个更强的引用：

```
reference2 = reference1

reference3 = reference1
```

这个单一的 Person 实例现有三个强引用。

如果用户通过将 nil 分配给该变量中的两个而破坏这些强引用中的两个（包括原始引用），则系统将会保留单一较强的引用，且 Person 实例不会被解除配置：

```
reference1 = nil

reference2 = nil
```

在第三即最终强引用被破坏前，ARC 不会针对 Person 实例解除配置，这时，可以确定的是用户将无法再次使用该 Person 实例：

```
reference3 = nil
// prints "John Appleseed is being deinitialized"
```

16.27 类实例之间的强引用周期

在上述示例中，ARC 可追踪用户为新增 Person 实例所建立的引用的数量，并可在不再需要该实例时解除配置。

但是，可以编写代码，该代码中某类的实例永远不会出现不具有强引用的情况。如果两个类实例间具有强引用则可能出现这种情况，以使各实例均确保对方的存在。这就是所谓的强引用周期。

可通过将类之间的部分关系定义为弱引用或无主引用而非强引用来解析强引用周期。在解析类实例件的强引用周期下对此过程进行了介绍。但是，在学习如何解析强引用周期前，先了解该周期产生的原因会很有帮助。

此示例为如何意外创建一个强引用周期。此示例定义了名为 Person 和 Apartment 的两个类，这两个类

对公寓楼及其住户进行了模拟：

```
class Person {
let name: String
init(name: String) { self.name = name }
var apartment: Apartment?
deinit { println("\(name) is being deinitialized") }
}
class Apartment {
let number: Int
(number: Int) { self.number = number }
var tenant: Person?
deinit { println("Apartment #\(number) is being deinitialized") }
```

各 Person 实例均有一个类型为 String 的 name 属性和一个初始值为 nil 的可选 apartment 属性。由于不一定每个人都有住所，所以该 apartment 属性为可选项。

同样，各 Apartment 实例均有一个类型为 Int 的 number 属性和一个初始值为 nil 的可选 tenant 属性。由于公寓并不总是有租户，所以可以租赁房子为可选项。

这两个类均定义了一个反初始化函数，该函数将打印出“正在初始化该类实例”这条消息。这使得用户可以查看 Person 和 Apartment 实例是否均按预计被解除配置。

此后续编码片段定义了类型为可选型的两个变量：john 和 number73，这两个变量将被设置为以下具体的 Apartment 和 Person 实例。由于这两个变量为可选项，所以其初始值均为 nil：

```
var john: Person?
var number73: Apartment?
```

这时，用户可创建具体的Person实例和Apartment实例并将这些新增实例分配给变量john和number73：

```
john = Person(name: "John Appleseed")
number73 = Apartment(number: 73)
```

以下介绍强引用如何创建和分配这两个实例。这时，变量 john 具有针对新增 Person 实例的强引用，变

量 number73 具有针对新增 Apartment 实例的强引用：详见图 16-8

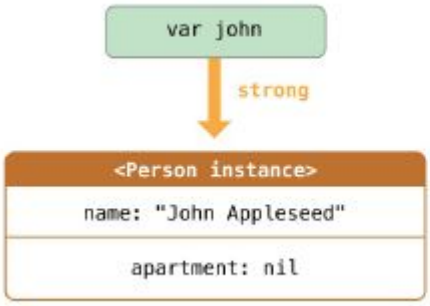


图 16-8 Var John

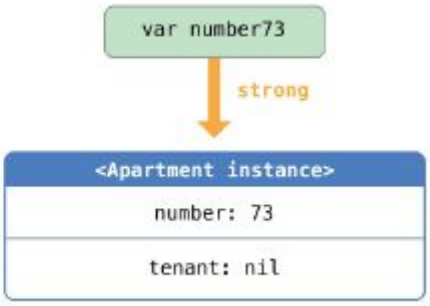


图 16-8 Var Number 73

这时，用户可将这两个实例连接起来，以使住户有住所、住所内有租户。注：叹号（!）用于展开并访问可选变量 john 和 number73 中存储的实例，以便为这些实例设置属性：

john!.apartment = number73

number73!.tenant = john

以下介绍强引用如何帮助用户将两个实例连接在一起：详见图 16-9

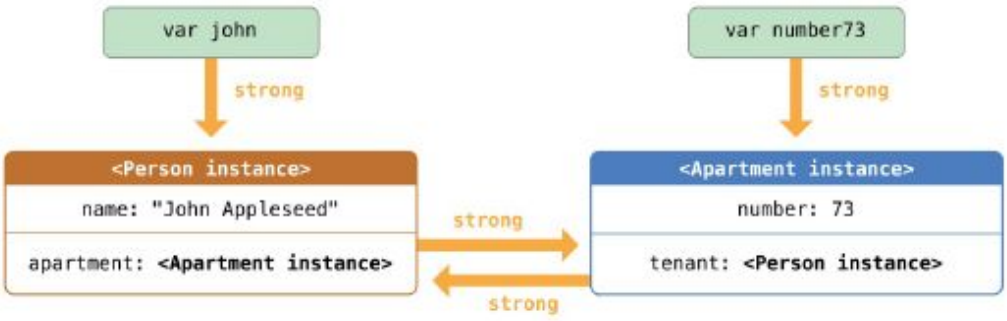


图 16-9

可惜，对这两个实例进行连接会在其间创建出一个强引用周期。这时，Person实例具有针对Apartment实例的强引用，Apartment实例也具有针对Person实例的强引用。因此，当破坏变量john和number73保存的强

引用时，引用计数不降为零，且ARC不针对该实例解除配置：

john = nil

number73 = nil

注，当用户设置这两个变量值为 nil，不得调用任一反参数化函数。强引用周期防止 Person 和 Apartment 实例被解除配置，并防止用户应用程序中出现内存泄露。

下文介绍强引用如何帮助用户将变量 john 和 number73 设为 nil：详见图 16-10

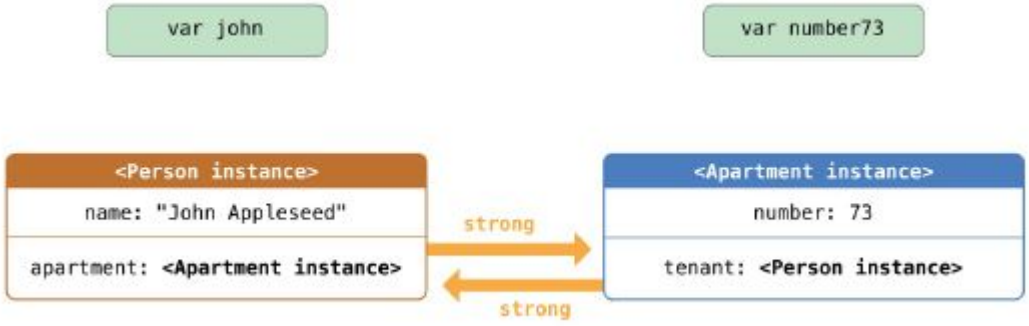


图 16-10

Person 实例和 Apartment 实例间仍保留强引用且该引用无法被破坏。

16.28 解析类实例之间的强引用周期

如果用户采用类型属性，Swift 可提供两种解析强引用周期的方法：弱引用和无主引用。

弱引用及无主引用使得引用周期中的一个实例可在不强控制另一实例的情况下对其进行引用。这些实例可互相引用，无需创建强引用周期。

当弱引用在其生命期的某一时刻成为 nil 时是有效的，则使用弱引用。相反地，如果用户已知在初始化过程中设置引用后该引用将不会为 nil，这时应采用无主引用。

16.29 弱引用

住所宜在其生命期某时刻能够处于“无租户”状态；因此，在这种情况下，弱引用适用于破坏引用周期。

注

必须声明弱引用为变量，以表明可在运行过程中对其值做出更改。不能声明弱引用为常量。

由于允许弱引用具有“无值”，因此用户须声明各弱引用均有可选类型。可选类型为表示 Swift 中可能存在“无值”的优选方法。

由于弱引用不对其引用的实例进行有力控制，因此在该弱引用仍对该实例进行引用期间，该实例有可能被解除配置。因此，当弱引用所引用的实例被解除配置后，ARC 自动将弱引用设为 nil。像检查任何其他可选值一样，用户可检查弱引用中是否存在任何值；并且，无法以针对不复存在的无效实例的引用结束。

以下示例与上述 Person 和 Apartment 示例相同，但有一点重要的不同之处。大约在这时，Apartment 类型的 tenant 属性被声明为弱引用：

```
class Person {
let name: String
init(name: String) { self.name = name }
var apartment: Apartment?
deinit { println("\(name) is being deinitialized") }
}
class Apartment {
let number: Int
(number: Int) { self.number = number }
weak var tenant: Person?
deinit { println("Apartment #\(number) is being deinitialized") }
```

引自这两个变量（john 和 number73）的强引用及两个实例间的链接按照前述方法建立：

```
var john: Person?
var number73: Apartment?
john = Person(name: "John Appleseed")
number73 = Apartment(number: 73)
john!.apartment = number73
number73!.tenant = john
```

以下将介绍引用如何帮助用户将这两个实例连接在一起：详见图 16-11

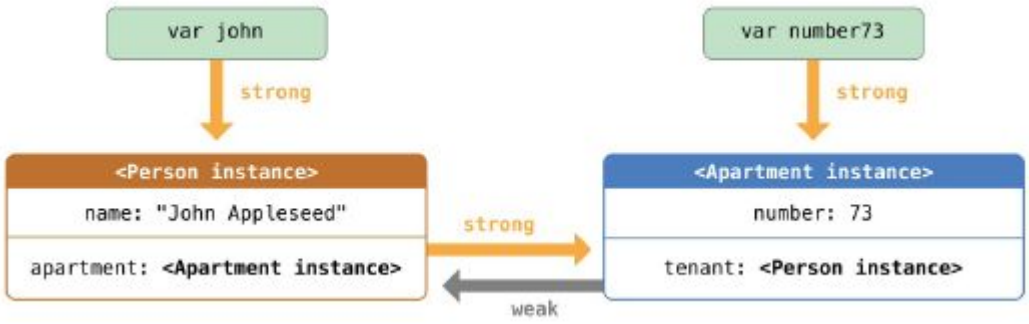


图 16-11

这时，Person 实例仍具有针对 Apartment 实例的强引用，但 Apartment 实例具有针对 Person 实例的弱引用。这说明，当变量 john 的强引用被破坏后，不存在针对 Person 实例的其他强引用：详见图 16-12

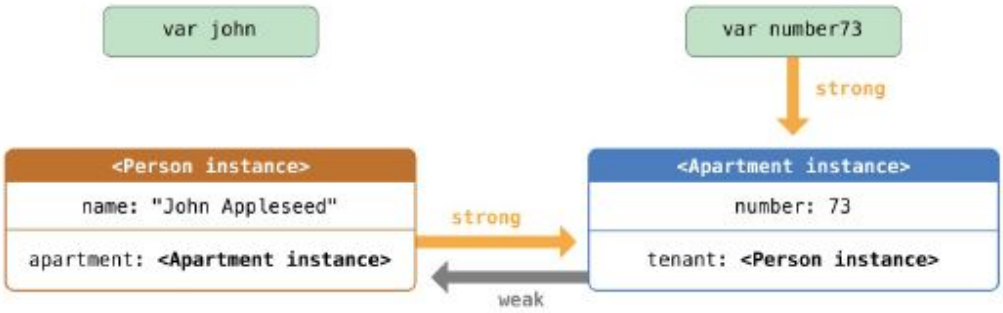


图 16-12

由于不存在针对 Person 实例的其他强引用，因此，该实例被解除配置：

```
john = nil

// prints "John Appleseed is being deinitialized"
```

唯一存留的针对 Apartment 实例的强引用引自变量 number73。如果该强引用被破坏，则不存在针对 Apartment 实例的其他强引用：详见图 16-13

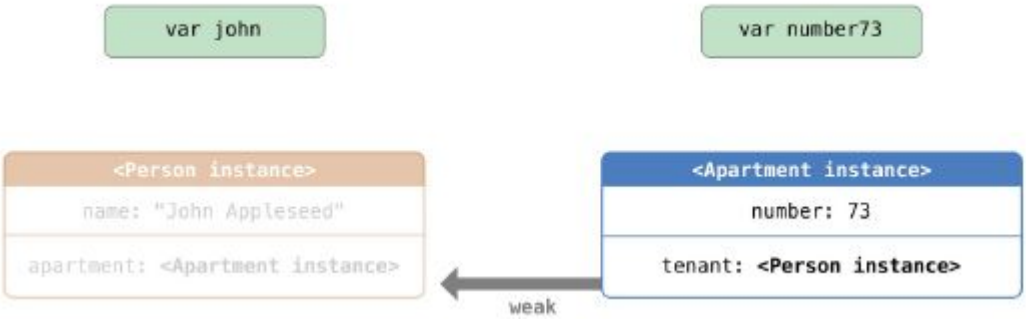


图 16-13

由于不存在针对 Apartment 实例的其他强引用，因此，该实例也将被解除配置：

```
number73 = nil

// prints "Apartment #73 is being deinitialized"
```

以上最后两个代码片段表明，在将变量 john 和 number73 设为 nil 后，Person 实例 和 Apartment 实例的反初始化函数打印出其“完成反初始化”消息。这证明，引用周期已被打破。

16.30 无主引用

正如弱引用一样，无主引用不对其引用的实例产生有力控制。但与弱引用不同的是，无主引用被假定为始终有值。正因为如此，无主引用总是被定义为非可选类型。用户通过将 unowned 关键字置于属性或变量说明前即可表示无主引用。

由于无主引用为非可选项，因此用户无需在每次使用该引用时均将其展开。总是可以直接访问无主引用。但是，由于无法将非可选型变量设为 nil，所以，ARC 无法在其引用的实例被解除配置后将引用设为 nil。

注

如果在其引用的实例被解除配置后尝试访问无主引用，则会触发运行时错误。仅在用户确定引用始终引用一个实例的情况下才可使用无属引用。

Swift 还提醒用户注：如果用户在其引用的实例被解除配置后访问无主引用，则用户的应用程序一定会崩溃。在这种情况下，你绝对不会邂逅意外行为。即使用户一定会防止应用程序崩溃，应用程序还是确实会崩溃。

以下示例定义了两个类：Customer 和 CreditCard，这两个类对银行客户和可能适用于该客户的信用卡进行了模拟。这两个类分别存储其它类的实例作为属性。这种关系有可能创建一个强引用周期。

Customer 与 CreditCard 间的关系和上文弱引用示例中显示的 Apartment 与 Person 间的关系略有不同。在这个数据模型中，客户可能有或没有信用卡，但一个信用卡始终关联至一个客户。为表示这一情况，Customer 类具有可选 card 属性，但 CreditCard 类具有非可选 customer 属性。

而且，仅可通过将一个 number 值和 customer 实例传递至自定义 CreditCard 初始化函数来新增 CreditCard 示例。这样，可以确保 CreditCard 示例在创建后，始终配有与其相关的 customer 实例。

由于信用卡始终配有客户，因此，用户可将其 customer 属性定义为无属引用，以避免强引用周期：

```
class Customer {
let name: String
var card: CreditCard?
init(name: String) {
self.name = name
}
deinit { println("\(name) is being deinitialized") }
}
CreditCard {
number: Int
unowned let customer: Customer
(number: Int, customer: Customer) {
self.number = number
self.customer = customer
```

```
deinit { println("Card #\(number) is being deinitialized") }
```

以下代码片段定义了名为 john 的可选 Customer 变量，该变量将用于存储针对具体客户的引用。作为可选项，该变量的初始值应为空：

```
var john: Customer?
```

这时，用户可创建一个 Customer 实例，并将其作为该客户的 card 属性对新增 creditcard 实例进行初始化和分配：

```
john = Customer(name: "John Appleseed")
john!.card = CreditCard(number: 1234_5678_9012_3456, customer: john!)
```

以下介绍引用如何帮助用户将两个实例连接在起义：详见图 16-14

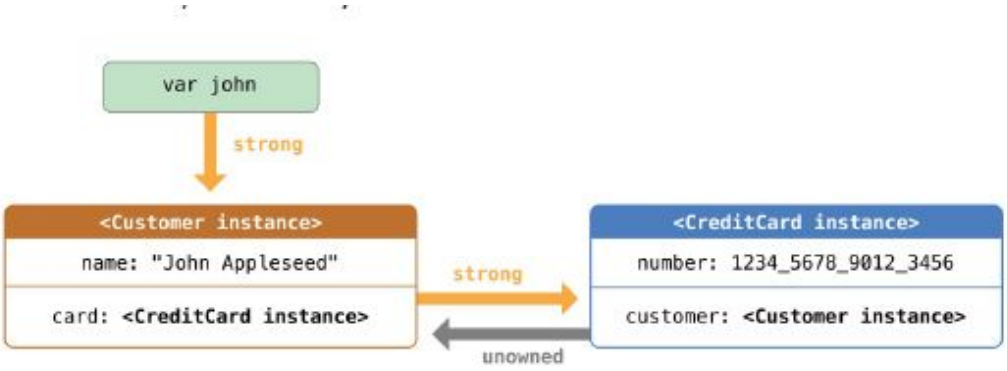


图 16-14

这时，Customer 实例具有针对 CreditCard 实例的强引用，CreditCard 实例也具有针对 Customer 实例的无主引用。

由于无主 Customer 引用，当用户破坏了变量 john 的强引用后，不存在针对 Customer 实例的其他强引用：详见图 16-15

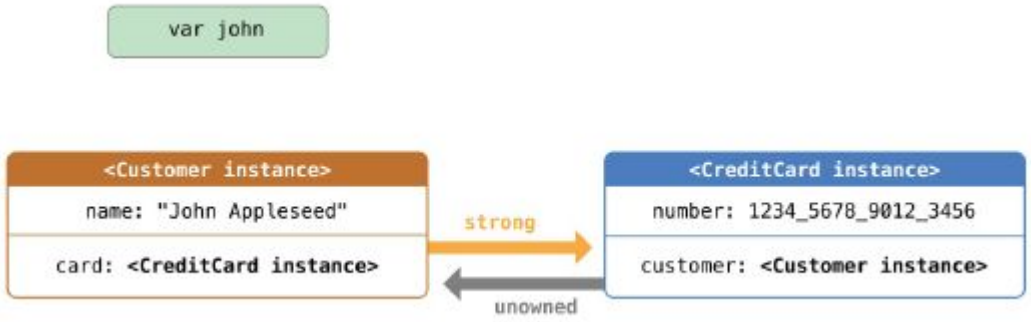


图 16-15

由于不存在针对 Customer 实例的其他强引用，因此，该实例被解除配置。此后，不存在针对 CreditCard 实例的其他强引用，并且实例也被解除配置：

```
john = nil
// prints "John Appleseed is being deinitialized"
// prints "Card #1234567890123456 is being deinitialized"
```

以上最后的代码片段表示，在将变量 john 设为 nil 后，Customer 实例和 CreditCard 实例的反初始化函数均分别打印出“完成反初始化”消息。

16.31 无主引用和隐式拆包的可选属性

以上弱引用和无主引用示例包括两种或两种以上一般情况，在这些情况下需要破坏强引用周期。

Person 和 Apartment 示例说明了一种情况，在这种情况下两个属性（均允许为 nil）均有可能造成强引用周期。这种情况下，最好通过弱引用解决。

Customer 和 CreditCard 示例说明了一种情况，在这种情况下允许一种属性 nil 和不允许另一属性为 nil 有可能造成强引用周期。这种情况下，最好通过无主引用解决。

但是，还有第三种情况，在这种情况下这两个属性均有值，并且在反初始化完成后这两个属性均不得为 nil。在这种情况下，将一个类的一个无主属性与另一个类的隐式展开的可选属性结合会有所帮助。

这样，使得这两个属性在初始化完成后可直接被访问（无需可选性展开），同时，还防止引用周期。本节说明如何建立这样的关系。

以下示例定义了两个类：Country 和 City，这两个类均分别将另一个类实例作为属性存储。在这个数据模型中，各城市必须始终具有首都城市，各城市必须始终属于一个国家。为表示 这一情况，Country

类具有 capitalCity 属性，City 类具有 country 属性：

```
class Country {
let name: String
let capitalCity: City!
init(name: String, capitalName: String) {
self.name = name
self.capitalCity = City(name: capitalName, country: self)
}
}
City {
name: String
unowned let country: Country
(name: String, country: Country) {
self.name = name
self.country = country
```

为在这两个类之间设置相关性，City 的初始化函数获取一个 Country 实例，并将该实例存储在其 country 属性中。

在 Country 初始化范围内调用 City 初始化函数。但是，在新增 Country 实例被完全初始化前（如两阶段初始化下介绍），Country 的初始化函数无法将 self 传递给 City 的初始化函数。

为满足这一要求，用户应声明 Country 的 capitalCity 属性为隐式展开的可选属性，由此类型注释后的叹号表示（City!）。这意味着 capitalCity 属性与其他可选项一样具有一个默认值 nil，但可以在不展开其值的情况下被访问-如隐式展开的可选项下所介绍。

由于 capitalCity 具有一个默认的 nil 值，因此，当 Country 实例在其初始化函数内设置 name 属性后，新增 Country 实例则被认为已完全被初始化。这意味着，设置好 name 属性后，Country 初始化函数可开始引用并传递隐式 self 属性。因此，当 Country 初始化函数设置其自身 capitalCity 属性时，Country 初始化函数可将 self 作为 City 初始化函数的其中一个参数进行传递。

这些均意味着用户可在单一指令中创建 Country 和 City 实例，无需创建强引用周期；并且 capitalCity 属性可直接被访问，无需使用叹号来展开其可选值：

```
var country = Country(name: "Canada", capitalName: "Ottawa")
println("\(country.name)'s capital city is called \(country.capitalCity.name)")
// prints "Canada's capital city is called Ottawa"
```

在以上示例中，使用隐式展开的可选项意味着已满足所有对两阶段类初始化函数的要求。完成初始化后，capitalCity 属性可像非可选值一样被使用和访问；同时，避免了强引用周期。

16.32 闭包的强引用周期

通过上文可了解当两个类属性均针对对方保存了强引用时如何创建强引用周期。同时，还可以了解到如何使用弱引用和无主引用来破坏这些强引用周期。

如果用户将一个闭包分配给了一个类实例属性，则也可能出现循环强引用，并且该闭包体会捕获该实例。由于闭包体访问了实例属性，例如 self.someProperty，或闭包调用了实例类函数，例如 self.someMethod()，则可能出现此类捕获。无论在哪种情况下，这些访问均会导致闭包对 self 进行“捕获”，并因此创建出强引用周期。

由于闭包与类一样为引用型，则可能出现强引用周期。当将一个闭包分配给一个属性时，也同时将一个引用分配给了该属性。实际上，这跟上文出现的问题一样-两个强引用均各自保证对方的存在。但是，在这种情况下，类实例和闭包而不是两个类实例各自保证对方的存在。

Swift 提供一个可以解决此问题的优质解决方案，被称为闭包捕获列表。但是，在学习如何利用闭包捕获列表来破坏强引用周期之前，应了解该周期是如何产生的。

以下示例说明了在使用引用 self 的闭包时，如何创建一个强引用周期。本示例定义了名为 HTMLElement 的 HTML 元素类，该类为 HTML 文件中的单个元素提供了一个简单模型：

```
class HTMLElement {
let name: String
let text: String?
@lazy var asHTML: () -> String = {
if let text = self.text {
return "<\(self.name)>\(text)</\(self.name)>"
} else {
return "<\(self.name) />"
}
```

```
(name: String, text: String? = nil) {
self.name = name
self.text = text
deinit {
println("\(name) is being deinitialized")
```

HTMLElement 类定义了一个 name 属性，该属性说明了元素的名称，例如段落元素中的“p”或换行符元素中的“br”。HTMLElement 还定义了一个可选 text 属性，该属性可设为一个字符串用来表示预计在该 HTML 元素中进行转义的文本。

除以上两个简单属性外，HTMLElement 类还定义了一个名为 asHTML 的懒惰属性。该属性引用一个闭包，该闭包将 name 和 text 结合在一个 HTML 字符串片段中。asHTML 属性为()->String 类型，或者是“一个不含有任何参数的函数，并会返回一个 String 值”。

默认将一个闭包分配给 asHTML 属性，该闭包会返回一个含 HTML 标签的字符串表达式。这个标签包含可选 text 值（如有），或者不包含文本内容（如果不存在 text）。对于段落元素，闭包将根据 text 属性是否等于“some text”或 nil 而返回“ some text ”或“ ”。

该 asHTML 属性的命名和应用类似于实例类函数。但是，由于 asHTML 是一个闭包属性而不是实例类函数，因此，如果用户希望针对某特定 HTML 元素改变 HTML 转义，则可用自定义闭包代替 asHTML 属性的默认值。

注

asHTML 属性被声明为懒惰属性，因为只有当实际需要将该元素作为某 HTML 输出目标的字符串值进行转义时才需要此属性。asHTML 属性为懒惰属性，这意味着用户可在默认闭包内引用 self，这是因为在初始化完成且已知 self 存在前懒惰属性不会被访问。

HTMLElement 类提供了一个单一的初始化函数，该函数采用 name 变元和（如需要）text 变元来对新元素进行初始化。该类还定义了一个反初始化函数，该函数打印出消息来说明 HTMLElement 实例何时被解除配置。

以下介绍了用户怎样使用 HTMLElement 类新建并打印一个实例：

```
var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
println(paragraph!.asHTML())
```

```
// prints "<p>hello, world</p>"
```

注

以上的 paragraph 变量被定义为一个可选 HTMLElement，这样即可将其设置为 nil 以显示存在强引用周期。

可惜的是，上述 HTMLElement 类在 HTMLElement 实例和作为其默认 asHTML 值的闭包之间创建了一个强引用周期。以下为测量周期的方法：详见图 16-16

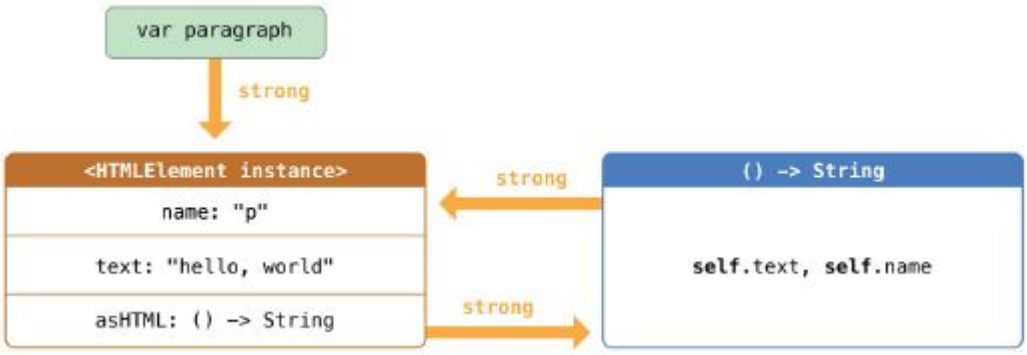


图 16-16 测量周期

该实例的 asHTML 属性保存了针对其闭包的强引用。但是，由于闭包在其体内引用了 self（作为一种引用 self.name 和 self.text 的方法），则该闭包捕获了 self；这说明该闭包反过来保存了针对 HTMLElement 实例的强引用。在两者之间创建强引用周期。（有关闭包内捕获数值的更多信息，参阅捕获值。）

注

即使该闭包多次引用了 self，它也仅可捕获一个针对 HTMLElement 实例的强引用。

由于存在强引用周期，所以，如果将 paragraph 变量设为 nil 或破坏其针对 HTMLElement 实例的强引用，HTMLElement 实例和闭包均会被解除配置：

```
paragraph = nil
```

注：未打印出 HTMLElement 反初始化函数中的消息，这说明未解除 HTMLElement 实例分配。

16.33 解析闭包的强引用周期

通过定义一个捕获列表并将其作为闭包定义的一部分，用户可以解析闭包和类实例间的强引用周期。针对在闭包体内捕获一种或多种引用类型，捕获列表定义了一项规则。关于两个类实例间的强引用周期，用户可声明各捕获到的引用为弱引用或无主引用，而不是强引用。根据用户代码不同部分间的关系来适当地选择弱引用或无主引用。

注

在引用闭包内的 self 时，Swift 要求用户编写 self.someProperty 或 self.someMethod（而不仅仅是 someProperty 或 someMethod）。有助于你记录偶然捕获的数据信息。

16.34 定义捕获列表

捕获列表中的各项是对 weak 关键字或 unowned 关键字与针对类实例（如 self 或 someInstance）的引用的配对。应在成对的方括号内编写这些配对并用逗号分隔。

将捕获列表置于闭包参数列表前并返回类型（如有）：

```
@lazy var someClosure: (Int, String) -> String = {
[unowned self] (index: Int, stringToProcess: String) -> String in
// closure body goes here
}
```

如果由于参数列表或返回类型可从语境推论得出而导致闭包未对其做出规定，则应将捕获列表置于闭包的最前方，其后为 in 关键字：

```
@lazy var someClosure: () -> String = {
[unowned self] in
// closure body goes here
}
```

16.35 弱和无主引用

当闭包和其捕获的实例始终互相引用且始终同时被解除配置时，将闭包中的一个捕获定义为无主引用。

相反地，当被捕获的引用可能在未来某时刻变为 nil 时，则将捕获定义为弱引用。弱引用一般为可选类型，

并且当实例解除其引用配置时，可以自动称为 ni。这使得用户能够检查其是否存在于闭包体内。

注

如果被捕获的引用不可能变为 nil，则该引用被捕获时应始终作为无主引用而非弱引用。

无主引用是一种适用于解析上文 HTMLElement 示例中强引用周期的捕获方法。以下介绍用户如何编写 HTMLElement 类，以避免周期：

```
class HTMLElement {
let name: String
let text: String?
@lazy var asHTML: () -> String = {
[unowned self] in
if let text = self.text {
return "<\(self.name)>\(text)</\(self.name)>"
} else {
return "<\(self.name) />"
}
(name: String, text: String? = nil) {
self.name = name
self.text = text
deinit {
println("\(name) is being deinitialized")
```

除在 asHTML 闭包中增加了捕获列表外，这里 LElement 的实现与上文实现相同。在这种情况下，捕获列表为[unowned self]，这意味着“捕获自身时将其作为无主引用而非强引用”。

用户可以像之前一样创建并打印一个 HTMLElement 实例：

```
var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
println(paragraph!.asHTML())
// prints "<p>hello, world</p>"
```

当捕获列表就位后，引用进行测量的方法：详见图 16-17

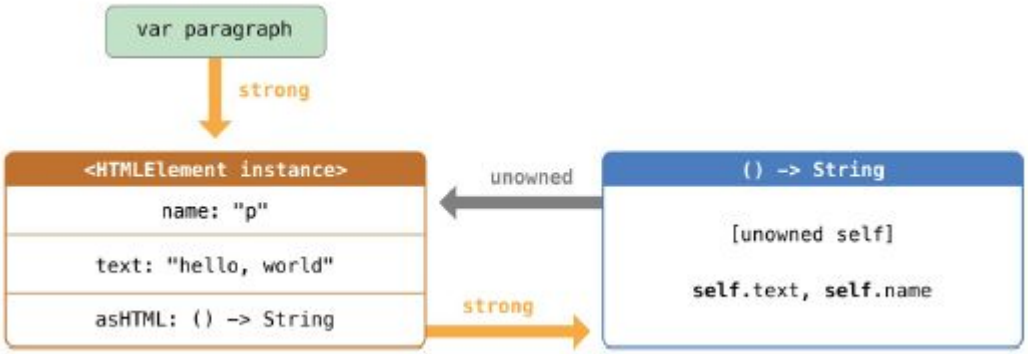


图 16-17 捕获列表就位后，引用进行测量的方法

这时，由闭包执行的 self 捕获为无主引用，并且不对所捕获的 HTMLElement 进行有力控制。如果将引自 paragraph 变量的强引用设为 nil，则 HTMLElement 实例将被解除配置，这一点可通过打印以下示例中的反初始化消息看出：

```
paragraph = nil
// prints "p is being deinitialized"
```

16.36 可选链接

可选链接是查询和调用属性、类函数以及目前可能为 nil 的可选项上的下标的过程。如果可选项含有一个值，则属性、类函数或下标调用成功；如果可选项为 nil，则属性、类函数或下标调用返回 nil。可将多次查询链接起来，如果链接中的任何环节为 nil，则整个链接失败。

注

Swift 中的可选链接类似于在 Objective-C 中传送 nil，但该链接适用于任何类型并且允许其检查成功或失败。

16.37 可选链接替代强制展开

通过将问号（？）置于可选值（当可选项为非 nil 时在该值上调用属性、类函数或下标）后即可指定可选链接。这与将叹号（!）置于可选值后以强制展开其值类似。主要不同之处在于，当可选项为 nil 时可选链接失败，但同时，如果可选项为 nil，则强制展开将触发运行时错误。

为反映出可选链接可在 nil 值上被调用这一事实，可选链接调用结果始终为可选值，即使所查询的属

性、类函数或下标返回一个非可选值。用户可使用此可选返回值来检查可选链接调用是成功（所返回的可选项含有一个值）或是由于链接中存在 nil 值而失败（所返回的可选值为 nil）。

具体来说，可选链接调用的结果与预期返回值类型相同，但包含在可选项中。当通过可选链接访问时，通常情况下返回 Int 的属性将返回 Int?。

以下几个代码片段说明了可选链接与强制展开有何不同以及可选链接如何为用户检查成功与否创造条件。

首先，名为 Person 和 Residence 两个类被定义为：

```
class Person {
var residence: Residence?
}
class Residence {
var numberOfRooms = 1
}
```

Residence 实例具有一个单一的名为 numberOfRooms 的 Int 属性，默认值为 1，Person 实例具有一个类型为 Residence?的可选 residence 属性。

如果用户创建一个新的 Person 实例，则凭借其可选性，默认为将其 residence 属性初始化为 nil。在下面的代码中，john 有一个值为零的 residence 属性：

```
let john = Person()
```

如果用户试图通过将叹号置于 residence 后以强制展开其值的方法来访问该住户 residence 的 numberOfRooms 属性，则会触发运行时错误,因为不存在 residence 值来展开：

```
let roomCount = john.residence!.numberOfRooms
// this triggers a runtime error
```

当 john.resdence 具有一个非 nil 值并且会将 roomCount 设为包含适当的房间数量的 Int 值时，则以上代码成功。但当 residence 为 nil 时，该代码总是会触发运行时错误-见上文介绍。

可选链接提供了另一种方式访问 numberOfRooms 值。若使用可选链接，需使用问号代替感叹号：

```
if let roomCount = john.residence?.numberOfRooms {
println("John's residence has \(roomCount) room(s).")
```

```
} else {
println("Unable to retrieve the number of rooms.")
}
// prints "Unable to retrieve the number of rooms."
```

这种情况使得 Swift 在可选 residence 属性上进行“链接”并检索出 numberOfRooms 的值（如果 residence 存在）。

由于访问 numberOfRooms 可能会失败，因此，可选链接会试图返回一个 Int?类型的值或“可选 Int”。当 residence 为 nil 时，如以上示例所示，此可选 Int 也将为 nil，以反映出无法访问 numberOfRooms 这一事实。

注，即使 numberOfRooms 值为非可选 Int 类型，这也是真的。通过可选链接进行查询这一事实说明：对 numberOfRooms 的调用始终返回一个 Int?而不是 Int。

用户可将 Residence 实例分配给 john.residence，以使其不再具有 nil 值：

```
john.residence = Residence()
```

john.residence 现在包含一个实际 Residence 实例，而不是 nil。如果用户试图以与上文相同的可选链接访问 numberOfRooms，则此时其将返回一个 Int?，其中包含 numberOfRooms 默认值 1：

```
if let roomCount = john.residence?.numberOfRooms {
println("John's residence has \(roomCount) room(s).")
} else {
println("Unable to retrieve the number of rooms.")
}
// prints "John's residence has 1 room(s)."
```

16.38 定义可选链接的模型类

用户可在对深度等级大于一级的属性、类函数和下标进行调用时使用可选链接。这样，用户可对互联型复杂模型中的子属性进行深度查询，并可检查是否有可能访问属性、类函数及这些子属性上的下标。

以下代码片段定义了用于下文几个示例（包括多级可选链接示例）的四个模型类。这些类通过添加具有相关属性、类函数和下标的 Room 和 Address 类对上述 Person 和 Residence 模型进行了扩展。

按照之前的方式，定义 Person 类：

```
class Person {
var residence: Residence?
```

与之前相比，Residence 类更加复杂。本次，Residence 类定义了一个名为 rooms 的变量属性，该属性与 Room[]类型的空数组进行初始化：

```
class Residence {
var rooms = Room[]()
var numberOfRooms: Int {
return rooms.count
}
subscript(i: Int) -> Room {
return rooms[i]
}
func printNumberOfRooms() {
println("The number of rooms is \(numberOfRooms)")
var address: Address?
```

由于此 Residence 版本存储了一个 Room 实例的数组，因此，其 numberOfRooms 属性作为计算属性实现，而不是存储属性。计算的 numberOfRooms 属性仅从 rooms 数组返回 count 属性的值。

作为一个访问其 rooms 阵列的快捷途径，此版本的 Residence 提供了只读下标，该下标以以下声明为开始：传递至下标的索引为有效索引。如果索引有效，则下标将置于请求的 rooms 阵列索引中的 room 返回。

此版本的 Residence 还提供了一个名为 printNumberOfRooms 的方法，该方法仅打印出 residence 中 rooms 的数量。

最后，Residence 定义了一个名为 address 的可选属性，其类型为 Address?.此属性的 Address 类型定义如下。

用于 rooms 阵列的 Room 类是一个简单类，其中有一个名为 name 的属性及一个用于将该属性设为适当的 room 名称的初始化函数：

```
class Room {
```

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

在这个模型中，最终类被称为 Address.这个类有三个可选择的 String？类型的属性前两个属性 buildingName 和 buildingNumber 是两种用于确认 address 中具体 building 的可选方法。第三个属性 street 被用来命名这个 address 的 street：

```
class Address {
var buildingName: String?
var buildingNumber: String?
var street: String?
func buildingIdentifier() -> String? {
if buildingName {
return buildingName
} else if buildingNumber {
return buildingNumber
} else {
return nil
}
```

Address 类还提供了一个名为 buildingIdentifer 的方法，该方法具有一个 String?返回类型。该方法检查 buildingName 和 buildingNumber 属性，并返回 buildingName（如有值）或 buildingNumber（如有值）或 nil（如果这两个属性均没有值）。

16.39 通过可选链接调用属性

如强制展开的可替代性可选链接介绍，用户可使用可选链接访问可选值上的属性，并检查该属性访问是否成功。但是，你不能通过可选链接设置属性值。

使用上文定义的类来创建一个新 Person 实例，并以上文所述方式尝试访问其 numberOfRooms 属性：

```
let john = Person()

if let roomCount = john.residence?.numberOfRooms {

println("John's residence has \(roomCount) room(s).")

} else {

println("Unable to retrieve the number of rooms.")

}

// prints "Unable to retrieve the number of rooms."
```

由于 john.resaence 为空，此可选链接调用以上述同样的方式失败，且未出现错误。

16.40 通过可选链接调用方法

用户可使用可选链接来调用可选值上的方法，并检查该方法调用是否成功。虽然该方法不能定义一个返回值，你仍可以这样做。

调用 Residence 类上的 printNumberOfRooms 方法打印出 numberOfRooms 的当前值。以下为此方法的展示：

```
func printNumberOfRooms() {

println("The number of rooms is \(numberOfRooms)")

}
```

此方法不能指定返回类型。但是，不带有返回类型的函数和方法具有一个隐式返回类型 Void-见无返回值的函数下的介绍。

如果在一个带有可选链接的可选值上调用此方法，则该方法的返回类型将为 Void？而不是 Void，这是因为当通过可选链接进行调用时返回值始终为可选类型。这使得用户可使用 if 语句来检查是否有可能调用 printNumberOfRooms 方法，即使该方法本身并未定义一个返回值。如果通过可选链接成功调用该方法，则从 printNumberOfRooms 隐式返回的值将等于 Void，否则该返回值为空。

```
if john.residence?.printNumberOfRooms() {

println("It was possible to print the number of rooms.")
```

```
} else {
println("It was not possible to print the number of rooms.")
}
// prints "It was not possible to print the number of rooms."
```

16.41 通过可选链接调用下标

用户可使用可选链接尝试检索可选值上的下标值，并检查该下标调用是否成功。但是，你不能通过可选链接设置下标值。

注

当通过可选链接访问可选值上的下标时，需要将问号置于下标的大括号前面而不是后面。可选链接问号始终紧跟在表达式可选部分之后。

以下示例尝试通过使用在 Residence 类中定义的下标来检索 john.residence 属性 rooms 阵列中第一个 room 名称。因为 john.residence 的当前值为空，所以下标调用失败：

```
if let firstRoomName = john.residence?[0].name {
println("The first room name is \(firstRoomName).")
} else {
println("Unable to retrieve the first room name.")
}
// prints "Unable to retrieve the first room name."
```

在这个下标调用中，可选链接问号紧随 john.residence 之后、下标括号之前，这是因为 john.residence 是可选链接试图在其上建立的可选值。

如果建立一个实际 Residence 实例并将其赋值给 john.residence，并且 rooms 阵列中有一个或多个 Room 实例，则可使用 Residence 下标通过可选链接访问 rooms 阵列中的实际项：

```
let johnsHouse = Residence()
```

```
johnsHouse.rooms += Room(name: "Living Room")

johnsHouse.rooms += Room(name: "Kitchen")

john.residence = johnsHouse

if let firstRoomName = john.residence?[0].name {

println("The first room name is \(firstRoomName).")

} else {

println("Unable to retrieve the first room name.")

prints "The first room name is Living Room."
```

16.42 链接多层次链接

用户可将多个等级的可选链接联系起来，对处于模块中更深层次的属性、方法和下标进行深度查询。但是，多个等级的可选链接并不在所返回的值中添加更多等级的可选性。

以另一种角度说：如果试图检索的类型不是可选性的，那么它会由于可选链接变为可选性的。如果试图检索的类型已经是可选性的，那么它会由于可选链接变得更加有可选性。因此：如果试图通过可选链接来检索一个 Int 值，不论用了多少层的链接，Int?始终是可返回的。

同样的，如果试图通过可选链接检索一个 Int?值，不论用了多少层的链接，Innt?始终是可返回的。

以下示例试图访问 john-residence 属性-address 属性-street 属性。本文采用两个级别的可选链接，以在 residence 和 address 属性间进行链接，这两种属性均为可选型：

```
if let johnsStreet = john.residence?.address?.street {

println("John's street name is \(johnsStreet).")

} else {

println("Unable to retrieve the address.")

}

// prints "Unable to retrieve the address."
```

目前 john.resaence 值含有一个有效的 Residence 实例。然而，john.residence.address 值仍为空。正因为如此，john.resdence？.address?.street 调用失败。

注：在以上示例中，用户试图检索 street 属性的值。此属性类型为 String？因此，即使应用了两个级别

的可选链接，除属性的底层可选类型外，john.resdence?.address?.street 的返回值也还是 String?。

如果将实际 Address 实例设为适用于 john.street.address 的值并为 address 的 street 属性设置一个实际值，则可通过多级可选链接访问属性值。

```
let johnsAddress = Address()

johnsAddress.buildingName = "The Larches"

johnsAddress.street = "Laurel Street"

john.residence!.address = johnsAddress

if let johnsStreet = john.residence?.address?.street {

println("John's street name is \(johnsStreet).")

} else {

println("Unable to retrieve the address.")

prints "John's street name is Laurel Street."
```

在将 address 实例赋值给 john.residence.address 的过程中，注叹号的使用。john.residence 属性具有一个可选类型，因此，在访问 residence 的 address 属性前，需要利用叹号展开其实际值。

16.43 链接方法和可选的返回值

上一示例说明了如何通过可选链接检索可选型属性的值。还可以使用可选链接来调用一个可返回可选类型值的方法，如果需要，还可在该方法的返回值上进行链接。

以下示例通过可选链接调用了 Address 类的 buildingIdentifer 方法。此方法可以返回一个 String？类型的值如上所述，在进行可选链接后，此方法调用的最终返回类型值仍为 String?：

```
if let buildingIdentifier = john.residence?.address?.buildingIdentifier() {

println("John's building identifier is \(buildingIdentifier).")

}

// prints "John's building identifier is The Larches."
```

如果要在此方法的返回值上进行进一步的可选链接，应将可选链接问号置于该方法的括号后：

```
if let upper = john.residence?.address?.buildingIdentifier()?.uppercaseString {
println("John's uppercase building identifier is \(upper).")
}
// prints "John's uppercase building identifier is THE LARCHES."
```

注

在以上示例中，用户将可选链接问号置于括号后，这是因为所链接的可选值为 buildingIdentifer 方法的返回值而不是该方法本身。

16.44 类型转换

类型转换是一种检查实例类型和/或处理该实例的方法，类似于其自身类层次中某处的一个不同的父类或子类。

通过 is 和 as 操作符实现 Swift 内的类型转换。这两个操作符提供了一个简单易懂的方法来检查值的类型或将值转换为不同的类型。

同时，还可以使用类型转换来检查某一类型是否符合协议-见检查协议一致性下的介绍。

16.45 定义类型转换的类层次结构

可以使用具有类层次和子类层次的类型转换来检查具体类实例的类型并在同一层次上将该实例转换为另一个类。以下三个代码片段定义了类层次和阵列，其中含有这些类的实例，以便在类型转换示例中使用。

第一个代码段定义了一个新的基类 MediaItem。这个类针对数字媒体库中出现的任何种类的项提供了基本功能。具体地说，它声明了一个 String 型 name 属性和一个 init name 初始化函数。（假定包括全部电影和音乐在内的所有媒体项目均有名称。）

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

以下代码断定义了 MediaItem 的两个子类。第一个子类-Movie-将其他有关电影或影片的信息进行了概述。它在基本 MediaItem 类顶部添加了一个 director 属性，并配有相应的初始化函数。第二个子类-Song-在基本类顶部添加了一个 artist 属性和初始化函数：

```
class Movie: MediaItem {
var director: String
init(name: String, director: String) {
self.director = director
super.init(name: name)
}
}
class Song: MediaItem {
var artist: String
(name: String, artist: String) {
self.artist = artist
super.init(name: name)
```

最后的片段创建了一个常量阵列 library，其中含有两个 Movie 实例和三个 Song 实例。利用阵列常值的内容将 library 阵列初始化，以此来推断出该阵列的类型。Swift 类型检查器可做出以下推论：Move 和 Song 具有 MedaItem 的通用超类；因此，推断出适用于 library 阵列的 MediaItem [ ]类型：

```
let library = [
Movie(name: "Casablanca", director: "Michael Curtiz"),
Song(name: "Blue Suede Shoes", artist: "Elvis Presley"),
Movie(name: "Citizen Kane", director: "Orson Welles"),
Song(name: "The One And Only", artist: "Chesney Hawkes"),
Song(name: "Never Gonna Give You Up", artist: "Rick Astley")
]
```

```
// the type of "library" is inferred to be MediaItem[]
```

存储于中的项仍然为后台的 Move 和 Song 实例。但是，如果对此阵列的内容进行迭代，则所收回的项目为 MediaItem 类型，而非 Move 或 Song 类型。为了使用其本机类型进行操作，需要检查其类型或将其向下转换为另一不同类型-见下文介绍。

16.46 检查类型

使用类型检查操作符检查某实例是否为特定子类类型。如果实例为特定子类类型，则类型检查操作符返回真；否则，返回假。

以下示例定义了两个变量—movecount 和 songCount，它们计算 library 阵列中 Movie 和 Song 实例的数量。

```
var movieCount = 0
var songCount = 0
for item in library {
if item is Movie {
++movieCount
} else if item is Song {
++songCount
}
("Media library contains \(movieCount) movies and \(songCount) songs")
prints "Media library contains 2 movies and 3 songs"
```

本示例演示了迭代阵列内所有项。每次传递中，for-in 循环将 item 常量设为阵列中的下一个 MedaItem。

如果当前 MediaItem 为 Movie 实例，则 Movie 项返回真；否则，返回假。同样，项 s Song 可检查此项是否为 Song 实例。在 for-in 循环的末尾，movecount 和 songCount 的值包含所找出的各类 MedaItem 实例总数。

16.47 向下转换

特定类类型的常量或变量可能实际引用后台的子类实例。当确定出现这种情况时，可尝试采用类型转

换操作符（as）向下转换为子类类型。

因为向下转换失败，类型转换运算符分为两种不同的形式。可选形式-as?-返回一个类型（需要向下转换为此类型）的可选值。

强制形式-as-试图进行向下转换，并强制将结果作为一个单一的复合动作展开。

当不确定向下转换是否会成功时，应使用类型转换操作符（as?）的可选形式。使用这个形式的操作符将始终返回可选值，并且如果不能实现向下转换，则该值将为 ni。可以使你检查是否已成功完成了向下转换。

当确定向下转换总是会成功时，应使用类型转换操作符（as）的强制形式。如果试图向下转换为一个错误的类类型，则使用这个形式的操作符将触发运行时错误。

以下示例对 library 中的各 MediaItem 进行迭代，并为每项打印出适当的描述。为此，它需要作为 true Movie 或 Song 而不仅是 MediaItem 来访问各项。这十分必要，这样，它即可访问用于描述的 Movie 或 Song 中的 director 或 artist 属性。

在本示例中，阵列中的每一项均可以为 Movie 或 Song。由于事先并不知道各项使用的实际类，因此，宜始终使用类型转换操作符（as?）的可选形式，每次通过循环对向下转换进行检查：

```
for item in library {

if let movie = item as? Movie {

println("Movie: '\(movie.name)', dir. \(movie.director)")

} else if let song = item as? Song {

println("Song: '\(song.name)', by \(song.artist)")

}

}

// Movie: 'Casablanca', dir. Michael Curtiz

Song: 'Blue Suede Shoes', by Elvis Presley

Movie: 'Citizen Kane', dir. Orson Welles

Song: 'The One And Only', by Chesney Hawkes

Song: 'Never Gonna Give You Up', by Rick Astley
```

本示例首先尝试把当前项向下转换为 Movie。由于 item 为 MediaItem 实例，因此，它有可能是 Movie；同样地，也有可能是 Song，或只是基本 MedaItem。由于这种不确定性，当试图向下转换为一个子类类型时，类型转换操作符的 as?形式返回一个可选值。Movie 项的结果为 Movie?或“可选 Movie”类型。

当应用 library 阵列中的两个 Song 实例时，向下转换为 Movie 失败。为解决这个问题，以上示例使用可选绑定来检查可选 Movie 是否确实包含一个值（即确认向下转换是否成功）。可选绑定编写为“if let move = item as?”可以被理解为：

“尝试访问 Movie 项。如果成功，将新增临时常量 movie 设为所返回的可选 Movie 中存储的值。

如果向下转化成功，则将 movie 属性用于打印该 Movie 实例的描述，包括 director 名称。当在 library 中发现 Song 时，用同样的原则检查 Song 实例，并打印出相应的描述（包括 artist 名称）。

注

实际上，类型转环并未修改实例或改变实例值。底层实例仍相同；它只是作为一个实例被处理和访问，该实例的类型为其转换的目标类型。

16.48 Any 和 AnyObject 之间的类型转换

Swift 为采用非特定类型提供了两种特殊的类型别名：

AnyObject 能够代表任何类型的一个实例

除了函数类型，Any 能够代表任何类型的一个实例。

注

仅当明确需要 Any 和 AnyObject 提供的特性和能力时，才使用 Any 和 AnyObject。应尽量明确你希望在代码中采用的类型。

16.49 AnyObject

采用 Cocoa APIs 时，经常会收到 AnyObject []类型的阵列或“任何对象类型值的阵列”。这是因为 Objective-C 中没有明确地类型数组。但是，通常只根据所掌握的该阵列有关 API 的信息，即可信任该阵列中包含的对象类型。

在这些情况下，可使用强制类型转换操作符（as）将阵列中的各项向下转换，使之变为比 AnyObject 更具体的类类型，无需可选展开。

以下示例定义了 AnyObject []类型的阵列，并将此阵列与三个 Movie 类实例一起进行了概述：

```
let someObjects: AnyObject[] = [
```

```
Movie(name: "2001: A Space Odyssey", director: "Stanley Kubrick"),
Movie(name: "Moon", director: "Duncan Jones"),
Movie(name: "Alien", director: "Ridley Scott")
]
```

由于已知此阵列仅包含 Movie 实例，因此，可使用强制类型转换操作符（as）直接向下转换并展开为非可选 Movie：

```
for object in someObjects {
let movie = object as Movie
println("Movie: '\(movie.name)', dir. \(movie.director)")
}
// Movie: '2001: A Space Odyssey', dir. Stanley Kubrick
// Movie: 'Moon', dir. Duncan Jones
// Movie: 'Alien', dir. Ridley Scott
```

对于此循环的较短形式，将 someObjects 阵列向下转换为 Movie []类型，而不是将所有项均向下转换：

```
for movie in someObjects as Movie[] {
println("Movie: '\(movie.name)', dir. \(movie.director)")
}
// Movie: '2001: A Space Odyssey', dir. Stanley Kubrick
// Movie: 'Moon', dir. Duncan Jones
 // Movie: 'Alien', dir. Ridley Scott
```

在以下示例中，将 Any 与不同类型（包括非类类型）的组合一同使用。本示例创建一个阵列 things，可以存储类型为 Any 的数值：

```
var things = Any[]()
```

```
things.append(0)

things.append(0.0)

things.append(42)

things.append(3.14159)

things.append("hello")

things.append((3.0, 5.0))

things.append(Movie(name: "Ghostbusters", director: "Ivan Reitman"))
```

Things 阵列包含两个 Int 值、两个 Double 值、一个 String 值、一个类型元组（Doube、Doube）和伊万·瑞特曼导演的电影《捉鬼敢死队》。

可以使用语句中的 is 和 as 操作符来找出常量和变量的具体类型，此类型仅已知为 Any 或 AnyObject 的类型。以下示例对 things 阵列中的项进行了迭代，并使用语句查询了各项的类型。语句的几个将其匹配值绑定成一个具有指定类型的常量，以使其值可被打印。

```
for thing in things {

switch thing {

case 0 as Int:

println("zero as an Int")

case 0 as Double:

println("zero as a Double")

case let someInt as Int:

println("an integer value of \(someInt)")

case let someDouble as Double where someDouble > 0:

println("a positive double value of \(someDouble)")

case is Double:

println("some other double value that I don't want to print")
```

```
case let someString as String:
println("a string value of \"\(someString)\"")
case let (x, y) as (Double, Double):
println("an (x, y) point at \(x), \(y)")
case let movie as Movie:
println("a movie called '\(movie.name)', dir. \(movie.director)")
fault:
println("something else")
```

```
zero as an Int
zero as a Double
an integer value of 42
positive double value of 3.14159
string value of "hello"
an (x, y) point at 3.0, 5.0
movie called 'Ghostbusters', dir. Ivan Reitman
```

注

语句的使用强制类型转换操作符（as,as?）来检查并转换为某特定类型。为安全起见，通常在 switch case 语句上下文中进行检查。

16.50 嵌套类型

枚举类型通常被用于实现某特定类或结构的功能。同样地，仅以使用为目的，可以在类型更为复杂的语境中方便地定义通用类和结构体。为了实现这种功能， Swift 允许你定义嵌套类型，可以在枚举类型、

类和结构体中定义嵌套支持的类型。

要在一个类型中嵌套另一个类型，应在被嵌套类型的外括号内编写需嵌套类型的定义。按照需要，类型可以多层次嵌套。

16.51 嵌套类型实例

以下示例定义了名为 BlackjackCard 的结构体，该结构体模拟了二十一点游戏）中使用的扑克牌点数。BlackJack 结构体包含两个嵌套式枚举类型 Suit 和 Rank。

在 Blackjack 游戏中，Ace 卡的点数可以为 1 或 11。这一特征由结构体 Values 表示，该结构体被嵌套在 Rank 枚举中：

```
struct BlackjackCard {
// nested Suit enumeration
enum Suit: Character {
case Spades = "♠", Hearts = "♡", Diamonds = "♢", Clubs = "♣"
}
// nested Rank enumeration
enum Rank: Int {
case Two = 2, Three, Four, Five, Six, Seven, Eight, Nine, Ten
case Jack, Queen, King, Ace
struct Values {
let first: Int, second: Int?
}
var values: Values {
switch self {
case .Ace:
return Values(first: 1, second: 11)
```

```
case .Jack, .Queen, .King:
return Values(first: 10, second: nil)
default:
return Values(first: self.toRaw(), second: nil)
}
}
BlackjackCard properties and methods
rank: Rank, suit: Suit
var description: String {
var output = "suit is \(suit.toRaw()),"
output += " value is \(rank.values.first)"
if let second = rank.values.second {
output += " or \(second)"
}
return output
```

Suit 枚举描述四种常用的扑克牌花色，分别用一个原始 Character 类型的值代表花色符号。Rank 枚举描述十三张可能的扑克牌排列方式，分别用一个原始 Int 类型的值表示牌的面值。（此原 Int 类型的值不适用于 Jack， Queen， King 和 Ace 的牌。）

如上所述，Rank 枚举定义了其本身更深层次的嵌套结构体 values。这个结构体概述了一项事实：大多数牌均有一个值，但 Ace 有两个值。该值结构体定义了两个属性，表述为：

Rank 还定义了一个计算属性，用于返回 Values 结构体的实例。这个计算属性会考虑牌的面值，并根据牌的面值，采用适当的值将新增 Values 实例初始化，。对于 Jack、 Queen、 King 和 Ace，它采用了特定值。对于数字面值的牌采用 rank 属性的原 Int 值。

该 BlackjackCard 结构体本身有两个属性—rank 和 suit.它还定义了一个计算属性 description，该属性采用 rank 和 suit 中存储的值来构建对这张牌名称和值的描述。并且使用可选绑定来检查是否有第二个值可以

显示，如果有，则针对第二个值插入其他描述细节。

由于 BlackjackCard 是一个不具有自定义初始化函数的结构体，正如 BlackjackCard 结构体有默认的成员初始化函数。你可以使用默认的 initializer 去初始化新的常量 the AceOfSpades：

```
let theAceOfSpades = BlackjackCard(rank: .Ace, suit: .Spades)

println("theAceOfSpades: \(theAceOfSpades.description)")

// prints "theAceOfSpades: suit is ♠, value is 1 or 11"
```

即使 Rank 和 Suit 被嵌套在 BlackjackCard 中，其类型也可根据上下文推断；因此，此实例的初始化可以通过枚举成员的成员名称（.Ace 和.Spades）来单独引用。在以上示例中，description 属性准确地报告出 AceOfSpades 有 1 或 11 两个值。

16.52 引用嵌套类型

要使用其定义上下文外的嵌套类型，用被嵌入类型的名称为前缀：

```
let heartsSymbol = BlackjackCard.Suit.Hearts.toRaw()

// heartsSymbol is "♡"
```

针对以上示例，这样做使 Suit，Rank 和 Values 的名称尽可能的简短，因为这些名称会自然的由被定义的上下文来限定。

16.53 扩展

扩展将新增功能添加到现有类、结构体或枚举类型。包括对类型（无法获取其初始源代码时）进行扩展的能力（被称为逆向建模）。扩展类似于 Objective-C 中的分类（不同之处在于，Swift 没有扩展名称）。

Swift 中的扩展可以：

添加计算属性和计算静态属性

定义实例方法和类型方法

提供新的初始化函数

定义下标

定义和使用新的嵌套类型

使现有的类型符合协议

注

如果将扩展定义为向现有类型中添加新功能，则这些新功能将在该类型的所有现有实例上可用；即使这些实例是在定义扩展前创建的，也是如此。

16.54 扩展语法

使用关键字声明扩展：

```
extension SomeType {
// new functionality to add to SomeType goes here
}
```

扩展功能可以扩展现有的类型，使其采用一个或多个协议。如果是这种情况，则协议名称以适用于类或结构同样的方法被准确写入：

```
extension SomeType: SomeProtocol, AnotherProtocol {
Add computed properties and computed static properties
Define instance methods and type methods
Provide new initializers
Define subscripts
Define and use new nested types
Make an existing type conform to a protocol
// implementation of protocol requirements goes here
}
```

以此种方式添加协议的一致性-见扩展添加协议一致性下的介绍。

16.55 计算属性

扩展可向现有类型中添加计算型实例属性和计算型类型属性。本示例向 Swift 的内置 Double 类型添加了五个计算型实例属性，以便为距离单位的使用提供基本支持：

```
extension Double {
var km: Double { return self * 1_000.0 }
var m: Double { return self }
var cm: Double { return self / 100.0 }
```

```
var mm: Double { return self / 1_000.0 }
var ft: Double { return self / 3.28084 }
}
let oneInch = 25.4.mm
println("One inch is \(oneInch) meters")
prints "One inch is 0.0254 meters"
threeFeet = 3.ft
("Three feet is \(threeFeet) meters")
prints " Three feet is 0.914399970739201 meters"
```

这些计算型属性表示：Double 值应被视为某长度单位。即使被作为计算属性实现，这些属性的名称仍可被添加到采用点语法的浮点文字值上，作为一种使用该文字值来执行距离转换的方法。

在这个例子中，人们认为 1.0 的 Double 值是代表“一米”。因此 m 计算属性返回 self-表达式 1.m 被视为用来计算 Double 值 1.0。

其他单位需要一些转换，被表示为以米为单位测得的值。1

一千米等于 1,000 米，因此，km 计算型属性用 1-000.00 乘以该值，以转换为以米为单位表示的数字。类似地，一米为 3.28024 英尺，因此，ft 计算属性用 3.28024 除以基础 Double 值，以将该值由英尺转换为米制。

这些属性为只读的计算属性，因此，为使其表达简洁，不使用 get 关键字。它们的返回值为 Double 类型，并且，在接受 Double 的情况下，可在数学计算中使用。

```
let aMarathon = 42.km + 195.m
println("A marathon is \(aMarathon) meters long")
// prints "A marathon is 42195.0 meters long"
```

注

扩展可添加新的计算属性，但无法添加存储属性，也无法将属性观察器添加到现有属性。

16.56 初始化函数

扩展功能可以为现有类型添加新的初始化函数。这使得用户可对其他类型进行扩展，以接受自定义属性并将其作为初始化函数参数，或提供未包括在该类型初始缺省参数的其他初始化选项。

扩展可向类中添加新的便利初始化函数，但无法向其中添加新的指定初始化函数或反初始化函数。指

定初始化函数和反初始化函数必须始终由原始类实现提供。

注

如果使用扩展向数值类型添加初始化函数，该值类型向存储的属性提供默认值，且不对任何自定义初始化函数进行定义，则可从扩展的初始化函数中调用适用于该值类型的默认初始化函数和按成员初始化函数。

如果已编写初始化函数并将其作为该值类型初始实现的一部分，则情况将有所不同-见适用于值类型的初始化函数委托下的介绍。

下面例子定义一个自定义 Rect 结构体来表示几何矩形。这个例子还定义了 2 个支撑结构体 Size 和 Point，它们 2 个都为其属性提供 0.0 的默认值：

```
struct Size {
var width = 0.0, height = 0.0
}
struct Point {
var x = 0.0, y = 0.0
}
struct Rect {
var origin = Point()
var size = Size()
```

由于 Rect 结构体为它的所有属性提供默认值，因此，它会自动收到默认初始化函数和按成员初始化函数-见默认初始化函数下的介绍。这些初始化程序可以用来创建新的实例：

```
let defaultRect = Rect()
let memberwiseRect = Rect(origin: Point(x: 2.0, y: 2.0),
size: Size(width: 5.0, height: 5.0))
```

用户可以扩展 Rect 结构体，以提供另外一个具有特定中心点和尺寸的初始化函数：

```
extension Rect {
init(center: Point, size: Size) {
let originX = center.x - (size.width / 2)
```

```
let originY = center.y - (size.height / 2)
self.init(origin: Point(x: originX, y: originY), size: size)
}
}
```

该初始化函数以提供的一个 center 点和一个 size 数值为依据，通过计算相应的原点而被启动。然后，该初始化函数调用该结构体的自动按成员初始化函数 init（origin:size:），该函数将新原点和尺寸值存储在相应属性中：

```
let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
size: Size(width: 3.0, height: 3.0))
// centerRect's origin is (2.5, 2.5) and its size is (3.0, 3.0)
```

注

如果利用扩展提供一个新的初始化函数，则在完成初始化函数后，还应确保各实例均完全初始化。

17 方法

扩展可向现有类型中添加新的实例方法和类型方法。下面的示例为 Int 类型添加了的新的实例方法 repetitons：

```
extension Int {
func repetitions(task: () -> ()) {
for i in 0..self {
task()
}
}
}
```

Repetition 方法具有一个类型为 () -> ()的单变元，该变元表示一个不含有参数且不返回值的函数。

定义此扩展后，可以调用任何整数上的 repetitions 方法以执行任务（可执行许多次）：

```
3.repetitions({
```

```
println("Hello!")
})
// Hello!
// Hello!
// Hello!
```

使用尾随闭包语法，使调用更加简洁：

```
3.repetitions {
println("Goodbye!")
}
// Goodbye!
// Goodbye!
// Goodbye!
```

17.1 变异实例方法

向实例方法添加扩展后，该方法还可以对其实例本身进行修改（或变异）。修改 self 或其属性的结构和枚举方法必须将实例方法标记为变异方法，类似于初始实现中的变异方法。

以下示例向 Swift 的 Int 类型中添加了一个新的变异方法 square，该方法对初始值进行平方运算：

```
extension Int {
mutating func square() {
self = self * self
}
}
var someInt = 3
someInt.square()
// someInt is now 9
```

17.2 下标

扩展功能可以为现有类型添加新的下标。本示例为 Swift 的内置 Int 类型添加了一个整数下标。下标[n]返回十进制数字符 n 并从该数字右侧将 n 置于其中：

等等：

```
extension Int {
subscript(digitIndex: Int) -> Int {
var decimalBase = 1
for _ in 1...digitIndex {
decimalBase *= 10
}
return (self / decimalBase) % 10
}
}
746381295[0]
returns 5
746381295[1]
returns 9
746381295[2]
returns 2
746381295[8]
returns 7
```

对于所请求的索引，如果 Int 值没有足够的位数，则下标实现将返回 0——这相当于在该数字的左边以零填补：

```
746381295[9]
// returns 0, as if you had requested:
0746381295[9]
```

17.3 嵌套类型

扩展可向现有类、结构体和枚举中添加新的嵌套类型：

```
extension Character {
enum Kind {
```

```
case Vowel, Consonant, Other
}
var kind: Kind {
switch String(self).lowercaseString {
case "a", "e", "i", "o", "u":
return .Vowel
case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":
return .Consonant
default:
return . Other
}
```

本示例为字符添加了一个新的嵌套枚举。这个名为 Kind 的枚举表示某特殊字符所代表的字母的种类。具体来说，它表示该字符在标准拉丁字母中是元音还是辅音（不考虑发音或地域差异），或者该字符是否是另外一种字符。

此示例还向 Character 中添加了新的名为 kind 的计算型实例属性，该属性为该字符返回适当的 Kind 枚举成员。

现在嵌套枚举可以与值一起使用：

```
func printLetterKinds(word: String) {
println("'\(word)' is made up of the following kinds of letters:")
for character in word {
switch character.kind {
case .Vowel:
print("vowel ")
case .Consonant:
print("consonant ")
case .Other:
```

```
print("other ")
}
print("\n")
printLetterKinds("Hello")
Hello' is made up of the following kinds of letters:
consonant vowel consonant consonant vowel
```

PrintLetterKinds 函数具有一个输入 String 值并对其字符进行迭代。针对每个字符，该函数均考虑适用于各字符的 kind 计算属性，并打印出该 kind 的相应描述。然后，printLetterKinds 函数可以被调用，以便以整个单词的形式打印出各种字母，如这里所示的“Helo”。

注

character.kind 是已知 CharacterKind 的类型之一。因此，在语句中，所有 Character.Kind 的成员值可以以简写形式编写，例如用 Vowel 代替 Character。Kind.Vowel.

18 协议

协议定义了方法、属性的大纲以及其他适用于特殊任务或功能块的要求。协议实际上不实现这些要求-只对实现的表现形式做出描述。然后，协议可被类、结构体或枚举采用以实现这些要求。满足协议要求的任何类型都被认为符合该协议。

协议可以要求符合规定的类型具有特定的实例属性、实例方法、类型方法、运算符以及下标。

18.1协议语法

采用与类、结构体和枚举非常相似的方式来定义协议：

```
protocol SomeProtocol {
// protocol definition goes here
}
```

自定义类型声明通过将协议名称放在类型名称后面，并用冒号隔开，采用一种特定的协议作为其定义的一部分。可以列出多种协议，各协议之间用逗号隔开：

```
struct SomeStructure: FirstProtocol, AnotherProtocol {
// structure definition goes here
```

}

如果类还有一个超类，在采用任何协议前列出超类的名称，后接 a

```
class SomeClass: SomeSuperclass, FirstProtocol, AnotherProtocol {
// class definition goes here
}
```

18.2 性能要求

协议可以要求任何符合要求的类型提供实例属性或有特定名称和类型的类型属性。协议并不指定属性是否应该是存储属性还是计算属性---其仅指定所需的属性名称与类型。协议还指定每一属性是否必须为可得到或是可得到且可设定。

如果协议要求某一属性可得到且可设定，则该属性要求不能通过常量存储属性或只读计算属性实现。如果协议只要求某一属性可得到，则该要求可通过任何类型的属性满足，如果它对你的代码有效，那是因为它还是可设定的属性，所以是有效的。

属性要求总是声明为变量属性，并带有 var 关键词前缀。可得到且可设定属性通过在其类型声明后写上{ get set }来表示，而可得到属性通过写上{ get }来表示。

```
protocol SomeProtocol {
var mustBeSettable: Int { get set }
var doesNotNeedToBeSettable: Int { get }
}
```

当你在某一协议中定义类型属性要求时，应始终为其加类关键字前缀。当通过结构体或枚举执行时，即使类型属性要求带有静态关键字前缀，也是真的：

```
protocol AnotherProtocol {
class var someTypeProperty: Int { get set }
}
```

下面是一个具有单一实例属性要求的协议的例子：

```
protocol FullyNamed {
var fullName: String { get }
}
```

该 FullyNamed 协议定义了具有完全限定名称的任何类的事物。其并不指定该事物是什么类型---它只指

定该事物必须能为自己提供全称。它通过声明来指定该要求，即任何 FullyNamed 类型都必须有名为 fullName 的可得到实例属性，且应为 String 类型。

下面是一个采用并符合 FullyNamed 协议的简单结构体的例子：

```
struct Person: FullyNamed {
var fullName: String
}
let john = Person(fullName: "John Appleseed")
// john.fullName is "John Appleseed"
```

这个例子定义了一个名为 Person 的结构体，该结构代表有特定名称的个体。它声明其采用 FullyNamed 协议作为其定义的第一行。

Person 的每个实例都只有一个名为 fullName 的储存属性，且应为 String 类型。这与 FullyNamed 协议的单一要求相匹配，也意味着 Person 符合协议。（只要有一个协议的要求未能满足，Swift 就会报告编译时的一个错误。）

还有更复杂的类，它也采用并符合 FullyNamed 协议：

```
class Starship: FullyNamed {
var prefix: String?
var name: String
init(name: String, prefix: String? = nil) {
self.name = name
self.prefix = prefix
}
var fullName: String {
return (prefix ? prefix! + " " : "") + name
ncc1701 = Starship(name: "Enterprise", prefix: "USS")
ncc1701.fullName is "USS Enterprise"
```

对某一星舰来说，该类将 fullName 属性要求作为计算只读属性来执行。星舰类实例中都存储了一个强制性名称和一个可选前缀。如果存在前缀值，fullName 属性则使用前缀值，并将其插至名称的开始部分，以便为 starship 创建一个全称。

18.3 方法要求

协议可以要求符合要求的类型实现特定的实例方法和类型方法。这些方法以与正常实例以及类型方法完全相同的方式写入协议定义的一部分，但是没有大括号或方法体。允许存在 variadic 参数，但受同样的关于正常方法规则的制约。

注

协议可以将相同的语法用作正常方法，但是不允许其为方法参数指定默认值。

与类型属性要求一样，当在协议中定义类型方法要求时，应始终为其加上类关键字前缀。当通过结构体或枚举执行时，即使类型方法要求带有静态关键字前缀，也是一样的：

```
protocol SomeProtocol {
class func someTypeMethod()
}
```

下面的例子定义了一个协议与单个实例方法要求：

```
protocol RandomNumberGenerator {
func random() -> Double
}
```

RandomNumberGenerator 协议，要求任何符合要求的类型都应有名为 random 的实例方法，而在调用时其会返回 Double 值。（尽管其并没有被指定为协议的一部分，但我们假定该值是 0.0 到 1.0（包括 1.0）之间的某个数。）

RandomNumberGenerator 协议对每一随机数是如何生成的不作任何假设 — 其仅仅要求生成器为生成新的随机数提供标准的方式。

以下是一个关于采用并符合 RandomNumberGenerator 协议的类的实施的例子。这个类的实施使用了一

个名为线性同余发生器的虚拟随机数生成算法：

```
class LinearCongruentialGenerator: RandomNumberGenerator {
var lastRandom = 42.0
let m = 139968.0
let a = 3877.0
let c = 29573.0
func random() -> Double {
lastRandom = ((lastRandom * a + c) % m)
return lastRandom / m
}
generator = LinearCongruentialGenerator()
("Here's a random number: \(generator.random())")
prints "Here's a random number: 0.37464991998171"
("And another one: \(generator.random())")
prints "And another one: 0.729023776863283"
```

18.4 变异方法的要求

有时候某一方式有必要修改（或变异）它所属的实例。对于数值类型的实例方法来说（也就是，结构体与枚举），你可以将变异的关键字放在方法的 func 关键字之前，以表明允许该方法修改其所属的实例和/或该实例的任何属性。这个过程在“内部实例方法部分的修改值类型”中有所描述。

如果你要定义一个旨在改变采用该协议的任何类型的实例协议实例方法要求时，将改变的关键词标记为协议定义的一部分即可。这使结构体及枚举能够采用协议并且满足该方法的要求。

注

如果你将某个协议实例方法要求标记为变异，在为某一类编写该实施方法时，你就没必要编写改变关键字了。该变异关键字只用于结构体和列举。

下面的例子定义了 Togglable 协议，它定义了名为 toggle 的单个实例方法要求。正如其名，toggle 方法旨在切换或转化任何符合要求的类型的状态，通常是通过修改该类型的某一个属性的方法。

toggle 方法被使用变异关键字标记为 Togglable 协议定义的一部分，表明在调用该方法时，它应该改变符合要求的实例的状态：

```
protocol Togglable {
mutating func toggle()
}
```

如果你为某个结构或枚举执行 Togglable 协议，该结构体或枚举可通过实施同样标记为变异的 toggle 方法而符合该协议。

下面的例子定义了一个名为 OnOffSwitch 的枚举。该枚举在两种状态之间切换，由枚举实例的 On 及 Off 表示。枚举实施 toggle 被标记为变异，以满足 Togglable 协议的要求：

```
enum OnOffSwitch: Togglable {
case Off, On
mutating func toggle() {
switch self {
case Off:
self = On
case On:
self = Off
}
lightSwitch = OnOffSwitch.Off
```

```
lightSwitch.toggle()

lightSwitch is now equal to .On
```

18.5 作为类型的协议

事实上，协议本身并不会执行任何功能。 尽管如此，要在你的代码中使用，你创建的任何协议都会成为成熟的类型。

因为它是一种类型，你可以在很多允许使用其他类型的地方使用某个协议，包括：

作为函数、方法或初始化函数中的参数类型或返回类型

作为常量、变量或属性的类型

作为阵列、字典或其他容器项的类型

注

因为协议是以大写字母作为名称开头的类型（例如：FullyNamed 和 RandomNumberGenerator），用以匹配 Swift 中其他类型（例如，Int，String 以及 Double）的名称。

下面是一个协议用作类型的例子：

```
class Dice {

let sides: Int

let generator: RandomNumberGenerator

init(sides: Int, generator: RandomNumberGenerator) {

self.sides = sides

self.generator = generator

}

func roll() -> Int {

return Int(generator.random() * Double(sides)) + 1
```

这个例子定义了名为 Dice 的新类，它代表在棋盘游戏中使用的一个多面骰子。Dice 实例有调用面数的整数属性，用以表示有多少面数，还有能调用生成器的属性，其可以提供随机数字生成器以创建骰子滚动值。

生成器属性的类型为 RandomNumberGenerator.因此，你可以将其设置为选用 RandomNumberGenerator

协议的任何类型的实例。除必须采用 RandomNumberGenerator 协议外，你赋给该属性的实例不需要别的东西。

骰子也有一个初始化函数，用以设置它的初始状态。该初始化函数有一个能调用生成器的参数，且其也有 RandomNumberGenerator 类型的性质。当初始化新的 Dice 实例时，你可以将任何符合条件的类型的值输入到此参数中。

Dice 提供了一种实例方法，即滚动，其能返回介于 1 以及骰子上点数之间的整数值该方法调用生成器的随机方法以创建介于 0.0 与 1.0 之间的新的随机数，并且使用该随机数创建正确范围内的骰子滚动值。我们都知道生成器应采用 RandomNumberGenerator，保证其有随机方法可调用。

以下解释怎样才能使用 Dice 类创建六面骰子，该骰子具有 LinearCongruentialGenerator 实例作为其随机数发生器：

```
var d6 = Dice(sides: 6, generator: LinearCongruentialGenerator())
for _ in 1...5 {
println("Random dice roll is \(d6.roll())")
}
// Random dice roll is 3
// Random dice roll is 5
// Random dice roll is 4
// Random dice roll is 5
// Random dice roll is 4
```

19 委派

委托是一种能使某一类或结构将其某些责任切换（或委托）给另一种类型的实例的设计模式。该设计模式通过定义一个能封装委托责任的协议实现，这样就能确保符合条件的类型能提供被委托的功能。委托可被用来应对特定的动作，或者是检索来自外部源的数据而无需了解该来源的基本类型。

下面的例子定义了以掷骰子为基础的棋牌游戏所使用的两个协议：

```
protocol DiceGame {
```

```
var dice: Dice { get }
func play()
}
protocol DiceGameDelegate {
func gameDidStart(game: DiceGame)
func game(game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int)
func gameDidEnd(game: DiceGame)
}
```

DiceGame 协议是一个能被任何涉及骰子的游戏采用的协议。DiceGameDelegate 协议可被追踪 DiceGame 进程的任何类型采用。

以下是最初引入到控制流程中的游戏的一个版本。该版本适合于使用 Dice 实例作为其骰子滚动值；适合于采用 DiceGame 协议；并且适合于告知 DiceGameDeegate 其进程：

```
class SnakesAndLadders: DiceGame {
let finalSquare = 25
let dice = Dice(sides: 6, generator: LinearCongruentialGenerator())
var square = 0
var board: Int[]
init() {
board = Int[](count: finalSquare + 1, repeatedValue: 0)
board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
var delegate: DiceGameDelegate?
func play() {
square = 0
delegate?.gameDidStart(self)
gameLoop: while square != finalSquare {
```

```
let diceRoll = dice.roll()
delegate?.game(self, didStartNewTurnWithDiceRoll: diceRoll)
switch square + diceRoll {
case finalSquare:
break gameLoop
case let newSquare where newSquare > finalSquare:
continue gameLoop
default:
square += diceRoll
square += board[square]
}
}
delegate?.gameDidEnd(self)
```

关于游戏的描述，请参阅控制流程章节的跳出循环部分。

这个版本的游戏被包装成名为 SnakesAndLadders 的类，且其采用 DiceGame 协议。它提供了可返回的骰子属性以及游戏方式以符合该协议。（骰子属性被声明为常数属性，因为在初始化后其并不需要改变，并且协议仅要求其可得到。）

棋盘设置在类的 int()初始化器中进行。

所有游戏逻辑被加入到协议的游戏方法中，其使用协议所需要的骰子属性提供其骰子滚动值。

请注委托属性被定义为可选的 DiceGameDeegate，因为要想玩游戏并不一定需要委托。因为它是可选的类型，所以委托属性自动设置为空的初始值。此后，游戏实例化器可以选择将属性设置成合适的委托。

DiceGameDelegate 提供了三种方法，用于跟踪一个游戏的进程。这三种方法已被纳入上述游戏方法的游戏逻辑，并且在开始新游戏、开始新一轮或游戏结束时被调用。

因为委托属性是可选的 DiceGameDeegate，所以每次在调用委托方法时，游戏方法会使用可选链接。如果委托属性是空的，这些委托调用会优雅地失败且没有错误。如果 deegate 属性不为零时，

委托方法作为一个参数被调用，并被传给 SnakesAndLadders 实例。

下一个例子展示了名为 DiceGameTracker 的类，其采用了 DiceGameDelegate 协议：

```
class DiceGameTracker: DiceGameDelegate {
```

```
var numberOfTurns = 0
func gameDidStart(game: DiceGame) {
numberOfTurns = 0
if game is SnakesAndLadders {
println("Started a new game of Snakes and Ladders")
}
println("The game is using a \(game.dice.sides)-sided dice")
}
func game(game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int) {
++numberOfTurns
println("Rolled a \(diceRoll)")
func gameDidEnd(game: DiceGame) {
println("The game lasted for \(numberOfTurns) turns")
```

DiceGameTracker 将执行 DiceGameDelegate 所需的所有三种方法。并使用这些方法来记录游戏所进行的轮数。游戏开始时，其将 numberOfTurns 属性重置为 0；每当新的一轮开始时，增加其值；并且在游戏结束时，输出游戏的总轮数。

如上所示，gameDdStart 的实现使用参数输出某些关于游戏即将开始的介绍信息。该游戏具有 DiceGame 类型而不是 SnakesAndLadders 类型，因此 gameDdStart 只能访问和使用作为 DiceGame 协议的一部分而实现的方法和属性。然而，该方法仍然能够使用类型转换查询基础实例的类型。

在这个例子中，其检查了游戏是否是 SnakesAndLadders 的一个实际实例；如果是的话，会输出相应的信息。

gameDdStart 还访问了已通过游戏参数的骰子属性。因为我们知道游戏符合 DiceGame 协议，则保证其会有投资属性，因此，无论玩的是哪一种游戏，gameDidStart 方法能够访问并输出骰子的面数属性。

下面内容具体介绍了 DiceGameTracker 的运行情况：

```
let tracker = DiceGameTracker()
let game = SnakesAndLadders()
game.delegate = tracker
game.play()
```

```
// Started a new game of Snakes and Ladders
// The game is using a 6-sided dice
// Rolled a 3
// Rolled a 5
// Rolled a 4
Rolled a 5
The game lasted for 4 turns
```

19.1 使用扩展功能添加协议一致性

即使并没有访问现有类型的源代码的权限，你也可以将其扩展至采用并符合新的协议。扩展可以为现有类型添加新属性、方法以及下标，因此其可以添加协议可能需要的任何要求。欲了解更多有关扩展功能的信息，请参见“扩展功能”部分的内容。

> 注
>
> 当在一个扩展里的实例类型中添加该符合性时，类型的现有实例会自动采用并符合协议。

例如，该协议，即 TextRepresentabe，能够被任何用文字表示的类型实现。这可能是对其自身的描述，或者是其当前状态的文本版本：

```
protocol TextRepresentable {
func asText() -> String
}
```

早期的 Dice 类可被扩展为采用且符合 TextRepresentable：

```
extension Dice: TextRepresentable {
func asText() -> String {
return "A \(sides)-sided dice"
}
}
```

该扩展会明确采用新的协议，犹如 Dice 已在其原始实现中提供了该协议。协议名称在类型名称之后出现，用冒号隔开，并且协议所有要求的实现在扩展的大括号中提供。

此时，任何 Dice 都可以被视为 TextRepresentable：

```
let d12 = Dice(sides: 12, generator: LinearCongruentialGenerator())
println(d12.asText())
// prints "A 12-sided dice"
```

同样地，SnakesAndLadders 游戏类可扩展到采用并符合 TextRepresentable 协议：

```
extension SnakesAndLadders: TextRepresentable {
func asText() -> String {
return "A game of Snakes and Ladders with \(finalSquare) squares"
}
}
println(game.asText())
// prints "A game of Snakes and Ladders with 25 squares"
```

19.2 使用扩展功能采纳声明协议

如果一个类型符合某一协议的所有要求，但是还没有声明其已采纳该协议，那么可以通过空扩展使其采纳该协议：

```
struct Hamster {
var name: String
func asText() -> String {
return "A hamster named \(name)"
}
}
extension Hamster: TextRepresentable {}
```

在任何情况下，如果所需的类型为 TextRepresentabe，都可以使用 Hamster 的实例：

```
let simonTheHamster = Hamster(name: "Simon")
let somethingTextRepresentable: TextRepresentable = simonTheHamster
println(somethingTextRepresentable.asText())
// prints "A hamster named Simon"
```

注

类型不会仅仅通过满足其要求来自动采用一个协议。他们必须始终明确声明采纳该协议。

19.3协议类型集合

如“协议就是类型”章节中所述，协议可被用作存储在某个集合（例如数组或词典）中的类型。此示例创建了 TextRepresentabe 事物的一个数组：

```
let things: TextRepresentable[] = [game, d12, simonTheHamster]
```

现在可以迭代数组中的项，并输出每个项中的文本表达式：

```
for thing in things {
println(thing.asText())
}
// A game of Snakes and Ladders with 25 squares
// A 12-sided dice
// A hamster named Simon
```

请注，thing 常量的类型为 TextRepresentable。尽管实际实例是这些类型中的一个，但是它不是 Dice，或者 DiceGame,或者 Hamster。然而，因为它是 TextRepresentable 类型，并且我们知道属于 TextRepresentable 的任何东西都有 asText 方法，所以在循环中对 thing.asText 每次调用都是安全的。

19.4协议继承

一个协议可以继承一个或多个其他协议，并且能够在它继承的要求之上添加进一步的要求。协议继承的语法与类继承的语法相似，但可选择列出多个继承的协议，用逗号隔开：

```
protocol InheritingProtocol: SomeProtocol, AnotherProtocol {
// protocol definition goes here
```

```
}
```

以下是某个协议从上级继承 TextRepresentable 协议的例子：

```
protocol PrettyTextRepresentable: TextRepresentable {
func asPrettyText() -> String
}
```

这个例子定义了一个新协议，PrettyTextRepresentabe，其继承自 TextRepresentable。采用 PrettyTextRepresentabfe 的任何东西都必须满足 TextRepresentable 所实施的所有要求，加上 PrettyTextRepresentabe 所实施的附加要求。在这个例子中，PrettyTextRepresentable 添加一个单一要求以提供一个调用 asPrettyText 返回 String 的实例方法。

SnakeAndLadders 类可被扩展为采用且符合 PrettyTextRepresentable：

```
extension SnakesAndLadders: PrettyTextRepresentable {
func asPrettyText() -> String {
var output = asText() + ":\n"
for index in 1...finalSquare {
switch board[index] {
case let ladder where ladder > 0:
output += "▲ "
case let snake where snake < 0:
output += "▼ "
default:
output += "○ "
}
}
Return output
```

该扩展声明其采用了 PrettyTextRepresentabe 协议，并且为 SnakesAndLadders 类型提供了 asPrettyText 方法的实现。属于 PrettyTextRepresentabe 的任何东西也属于 TextRepresentable，因此，asPrettyText 实现开始于从 TextRepresentabe 协议中调用 asText 方法以开始输出字符串。它会添加一个冒号和一个换行符，并

将其作为优美文本表达式的开始。然后，其迭代棋盘方框中的所有数组，并且为每一方框加上一个表情符：

If the square's value is greater than 0, it is the base of a ladder, and is represented by ▲.

If the square's value is less than 0, it is the head of a snake, and is represented by ▼.

Otherwise, the square's value is 0, and it is a "free" square, represented by ○.

如果方格值大于 0，那它就是梯基，由▲表示。

如果方格值小于 0，那它就是蛇头，由▼表示。

否则，方格值为 0，那它就是游离方格，由○表示。

现在方法实现可用于输出任何 SnakesAndLadders 实例的优美的文本描述：

```
println(game.asPrettyText())
// A game of Snakes and Ladders with 25 squares:
// ○ ○ ▲ ○ ○ ▲ ○ ○ ▲ ▲ ○ ○ ○ ▼ ○ ○ ○ ○ ▼ ○ ○ ▼ ○ ▼ ○
```

19.5 协议组合

要求一个类型同时符合多种协议是很有用的。通过协议组合，你可以将多个协议组合成单一要求。

协议组合的形式有 protocok<SomeProtocol,AnotherProtocol>。可以在一对尖括号（<>）中列出你所需要的尽可能多的协议，用逗号隔开。

以下是将名为 Named 和 Aged 的两个协议组合成函数参数的单一协议组合要求的例子：

```
protocol Named {
var name: String { get }
}
protocol Aged {
var age: Int { get }
}
struct Person: Named, Aged {
var name: String
var age: Int
wishHappyBirthday(celebrator: protocol<Named, Aged>) {
println("Happy birthday \(celebrator.name) - you're \(celebrator.age)!")
birthdayPerson = Person(name: "Malcolm", age: 21)
```

```
wishHappyBirthday(birthdayPerson)

prints "Happy birthday Malcolm - you're 21!"
```

这个例子使用名为 Name 的可返回 String 属性的单一要求定义了名为 Named 的协议。这个例子还使用名为 age 的可返回 Int 属性的单一要求定义了名为 Aged 的协议。这两种协议是通过结构体 calledPerson 被采用的。

这个例子还使用名为 celebrator 的单个参数定义了名为 wishHappyBirthday 的函数。该参数的类型是 protocol<Named，Aged>，意即“符合 Named 以及 Aged 协议的所有类型。”只要符合这两个要求协议，传给函数的具体类型是什么并不重要。

随后，该例子创建了一个名为 birthdayPerson 的新的 Person 实例并将这个新实例传递给 wishHappyBirthday 函数。因为 Person 符合这两个协议，所以这是有效的调用，并且 wishHappyBirthday 函数可以输出其生日祝福。

注

协议组合未定义一个新的、永久性的协议类型。相反，它们定义一个临时的本地协议，该协议含有组合中所有协议的要求。

19.6 检查协议一致性

你可以使用在“类型转换”中所述的 is 和 as 运算符检查协议的符合性，并且转换某个特定的协议。根据与检查并转换类型完全一样的语法检查并转换某个协议：

如果实例符合协议，那么运算符返回 true；否则，返回 false。

向下转换运算符的 as?版本会返回协议类型的可选值，如果实例不符合协议那么这个值为 nil。

向下转换运算符的 as 版本会强制向下转换协议类型，如果转换不成功，就会发送运行错误。

这个例子使用 area 的可返回 Double 属性的单一属性要求定义了一个 HasArea 的协议。

```
@objc protocol HasArea {
var area: Double { get }
}
```

注

只有在协议被标记为@objc 属性的情况下，才能检查协议的符合性，可参见上述 HasArea 协议。该属性表明该协议可能会接触到 Objective-C 代码,这在《Using Swift with Cocoa and Objective-C》中有详细描述。尽管你并没有同时操作 Objective-C，但是如果检查协议的符合性，还是需要将协议标记为@objc 属性。

还要注@objc 协议只能被类采用，而不能被结构体或枚举采用。如果你将你的协议标记为@objc 以检查其一致性，那么你只能够将此协议应用于类的类型。

这里有两个类：Circle 和 Country，并且都符合 HasArea 协议：

```
class Circle: HasArea {
let pi = 3.1415927
var radius: Double
var area: Double { return pi * radius * radius }
init(radius: Double) { self.radius = radius }
}
class Country: HasArea {
var area: Double
init(area: Double) { self.area = area }
```

根据存储的半径属性，Circle 类将 area 属性要求作为计算属性执行。Country 类直接执行 area 的要求，作为一种存储属性。这两个类都正确地符合 HasArea 协议。

此处有一个名为 Anima 但不符合 HasArea 协议的类：

```
class Animal {
var legs: Int
init(legs: Int) { self.legs = legs }
}
```

Circle、Country 和 Animal 类之间没有共享的基类。但是，它们都是类，所以这三种类型的实例都可用来初始化存储 AnyObject 类型值的数组：

```
let objects: AnyObject[] = [
```

```
Circle(radius: 2.0),
Country(area: 243_610),
Animal(legs: 4)
]
```

对象数组被数组常值中包含半径为两个单位的 Circle 实例初始化；Country 实例被英国的表面面积（单位：千米），以及有四条腿的 Animal 实例初始化。

现在，可以迭代对象数组，并且可以检查数组中的每个对象以判断其是否符合 HasArea 协议：

```
for object in objects {
if let objectWithArea = object as? HasArea {
println("Area is \(objectWithArea.area)")
} else {
println("Something that doesn't have an area")
}
}
// Area is 12.5663708
// Area is 243610.0
Something that doesn't have an area
```

在数组中的对象符合 HasArea 的情况下，as?运算符返回的可选值被绑定到调用 objectWithArea 常数的可选值解包。我们知道 objectWithArea 常数是 HasArea 类型，因此其 area 属性可以类型安全的方式来访问并输出。

需要注的是，相关的对象没有因转换过程而发生改变。他们仍然是一个 Circle、Country 和 Animal。但是，在它们被存储在 objectWithArea 常数的情况下，我们只知道它们是 HasArea 类型，所以只能访问它们的 area 属性。

19.7 任择议定书的要求

你可以为协议定义任选要求，符合该协议的类型执行并不需要执行这些类型。任选要求以@optona 关键字做前缀并作为其定义的一部分。

任择议定书的要求可被可选链接调用，用来解释该要求不能被某个符合该协议的类型实现的可能性。欲了解有关可选链接的信息，请参阅“可选链接”部分。

当被调用时，你可以通过在要求的名称后编写一个问号来检查该可选要求的实现，例如，someOptonaiMethod?(someArgument)。

当访问或调用可选属性要求以及返回某个数值的可选方法要求时，它们将始终返回适当类型的一个可选值，以表示可选要求可能不会实现。

注

只有在你的协议标有@objc 属性时，任择议定书要求才会被指定。即使你没有同时操作 Objective-C，如果你想要指定可选要求你还是需要为协议标记@objc 属性。

还要注@objc 协议只能被类采用，而不能被结构体或枚举采用。如果将协议标记为@objc 以检查指定可选要求，那么你只能将此协议应用于类别类型。

下面的例子定义了一个调用 Counter 的整数计算类，它使用外部数据源提供其增加数量。该数据源通过 counterDataSource 协议定义，该协议有两个可选要求：

```
@objc protocol CounterDataSource {
@optional func incrementForCount(count: Int) -> Int
@optional var fixedIncrement: Int { get }
}
```

counterDataSource 协议定义了一个 incrementForCount 的可选方法要求以及一个 fixedIncrement 的可选属性要求。这些要求为数据源定义了不同的方式，以便为 Counter 实例提供适当的增加数量。

注

严格来说，你可以在不执行任一协议要求时，编写一个符合 CounterDataSource 的自定义类。毕竟，它们都是可选的。尽管在技术上来讲是允许的，但这不会有助于形成一个很好的数据源。

Counter 类，定义如下，具有 CounterDataSource?类型可选的 dataSource 属性：

```
@objc class Counter {
var count = 0
var dataSource: CounterDataSource?
```

```
func increment() {
if let amount = dataSource?.incrementForCount?(count) {
count += amount
} else if let amount = dataSource?.fixedIncrement? {
count += amount
}
```

Counter 类将其当前值存储在一个名为 count 的变量属性中。Counter 类还定义了一个 increment 的方法，每次调用方法时它都会增加 count 属性。

增量方法首先试图通过在其数据源中寻找 incrementForCount 方法的实现来检索一个增量。increment 方法是用可选链接试图调用 incrementForCount，并且将当前的 count 值作为方法的参数传输。

注此处有两个级别可选链接在起作用。 首先，dataSource 有可能为 nil，因此 dataSource 在其名称之后有一个问号，用来说明只有在 dataSource 不是空的时候 incrementForCount 才能够被调用。其次，即使 dataSource 确实存在，也不能保证它会执行 incrementForCount，因为它是一个可选要求。

这就是为什么 incrementForCount 在编写时还需要在其名称后面加一个问号。

因为这两个原因的某一个都可能会导致 incrementForCount 调用失败，所以调用返回一个可选的 Int 值。即使在 CounterDataSource 的定义中 incrementForCount 被定义为返回一个非可选 Int 值，这也是真的。

调用 incrementForCount 后，使用可选约束，其返回的可选 Int 会解包成一个数量的常量。如果可选 Int 确实包含一个数值—也就是说，如果指令与方法同时存在，并且该方法可返回一个数值—解包 amount 会添加到储存的 count 属性上，并且增量完成。

如果不可能从 incrementForCount 方法中检索一个数值—或者是因为 dataSource 是空的或者是因为数据源没有实现 incrementForCount---那么 increment 方法则会试图从数据源的 fixedIncrement 属性中检索一个数值。fixedIncrement 属性也是一个可选要求，因此其名称也使用可选链接编写并且以问号结束，以表明不能访问该属性值。像以前一样，返回的数值是可选 Int 值，即使在 CounterDataSource 协议的定义中 fixedIncrement 被定义为非可选 Int 属性。

以下是一个简单的执行 CounterDataSource 的例子，其中在每次查询时数据源都返回常量值 3。它通过执行可选 fixedIncrement 实现着一点。

```
class ThreeSource: CounterDataSource {
let fixedIncrement = 3
}
```

在新的 Counter 实例中你可以使用 ThreeSource 实例作为数据源：

```
var counter = Counter()
counter.dataSource = ThreeSource()
for _ in 1...4 {
counter.increment()
println(counter.count)
}
// 3
// 6
// 9
```

以上代码创建了一个新的 Counter 实例；将其数据源设置为新的 ThreeSource 实例；并且调用四次计数器的 increment 方法。同预计的一样，在每次调用 increment 时，计数器的 count 属性会增加 3。

以下是一个更复杂的数据源 TowardszeroSource，它能使一个 Counter 实例从其当前数值向上或向下计数到零。

TowardszeroSource 类在 counterDataSource 协议中使用可选 incrementForCount 方法，并且使用 count 参数值算出应从哪个方向开始计数。如果 count 已经是零，该方法则返回 0 以表明无需再进行进一步的计数。

你可以使用现有 Counter 实例中的一个 TowardszeroSource 实例从 4 计数到零。当计数器达到零时，不会发生更多的计数：

```
class TowardsZeroSource: CounterDataSource {
func incrementForCount(count: Int) -> Int {
if count == 0 {
return 0
} else if count < 0 {
return 1
} else {
return -1
```

```
}
```

20 泛型

根据你所定义的要求，通用代码能使你编写灵活、可重复使用并且对任何类型都有效的函数及类型。你可以编写避免重复的代码，并且以明确且抽象的方式表达其意图。

泛型是 Swift 最强大的功能之一，并且许多 Swift 标准程序库是使用泛型代码编写的。事实上，尽管你并没有意识到，但在该《语言指导》中你已经使用了泛型代码。例如，Swift 的 Array 和 Dictonary 类型都是泛型集合。你可以创建保存 Int 数值的数组，或者是保存 String 数值的数组，甚至是包含 Swift 能够创建的任何其他类型的数组。同样地，你可以创建一个字典来储存任何特定类型的数值，并且对该类型并没有任何限制。

20.1泛型所解决的问题

以下是一个标准的、非泛型函数 swapTwoInts，其可以交换两个 Int 数值：

```
func swapTwoInts(inout a: Int, inout b: Int) {
let temporaryA = a
a = b
b = temporaryA
}
```

该函数使用可写入读出参数交换两个数值 a 和 b，如“可写入读出参数章节”所述。swapTwoInts 函数把 b 的初始值交换给 a，把 a 的初始值交换给 b。你可以调用这个函数来交换两个 Int 变量的数值：

```
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
println("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
// prints "someInt is now 107, and anotherInt is now 3"
```

该 swapTwoInts 函数是有用的，但它只能用于 Int 值。如果你想要交换两个 String 数值，或两个 Double 数值，你需要编写更多函数，例如如下所述的 swapTwoStrings 以及 swapTwoDoubles 函数：

```
func swapTwoStrings(inout a: String, inout b: String) {
let temporaryA = a
a = b
b = temporaryA
}
func swapTwoDoubles(inout a: Double, inout b: Double) {
let temporaryA = a
a = b
= temporaryA
```

你可能已经发现 swapTwoInts、swapT woStrings 以及 swapTwoDoubles 函数的正文完全相同。唯一的不同是它们所接受的数值的类型（Int、String 和 Double）。

编写能够交换任何类型的两个数值的单一函数可能会更有用并且更灵活。这是通过泛型代码可以解决的一类问题。（这些函数的泛型版本，定义如下。）

注

在这三个函数中，重要的是 a 与 b 应被定义为相同的类型。如果 a 和 b 的类型不同，那么就不可能交换二者的值。Swift 是一种类型安全的语言，并且不允许（例如）String 类型的变量和 Double 类型的变量相互交换其值。如试图交换则会出现编译时的报告错误。

20.2泛型函数

泛型函数可以适用于任何类型。这是来自上级的 swapTwoInts 函数的泛型版本，为 swapTwoValues：

```
func swapTwoValues<T>(inout a: T, inout b: T) {
let temporaryA = a
a = b
b = temporaryA
}
```

swapTwovalues 函数的正文与 swapTwoInts 函数的正文完全相同。然而，swapTwovalues 的第一行与 swapTwoInts 略有不同。下面对两者的第一行做了比较：

```
func swapTwoInts(inout a: Int, inout b: Int)

func swapTwoValues<T>(inout a: T, inout b: T)
```

函数的泛型版本使用一个占位符类型名称（在这种情况下，称为 T）而不是实际的类型名称（例如，Int, String 或 Double）。占位符类型名称并不声明 T 必须是什么，但是无论 T 代表着什么，它确实声明 a 和 b 必须是相同类型的 T。代替 T 而使用的实际类型会在每次调用 swapTwovalues 函数时确定。

其他差异是泛型函数的名称（swapTwovalues）后面跟着一个在尖括号（<T>）里的占位符类型名称（T）。这个括号在 Swift 表示这个 T 是 swapTwovalues 函数定义中的一个占位符类型名称。因为 T 是一个占位符，Swift 不会查找一个名为 T 的实际类型。

除 swapTwovalues 函数可以传递任何类型的两个数值（两个数值必须同一类型）之外，现在其可以以与 swapTwoInts 相同的方式被调用。每次调用 swapTwovalues 时，则可以根据传递给函数的数值类型推断出用于 T 的类型。

在下面的两个例子中，T 分别被推断为 Int 和 String：

```
var someInt = 3

var anotherInt = 107

swapTwoValues(&someInt, &anotherInt)

// someInt is now 107, and anotherInt is now 3

var someString = "hello"

var anotherString = "world"

swapTwoValues(&someString, &anotherString)

// someString is now "world", and anotherString is now "hello"
```

注

定义上述 swapTwoValues 函数的灵感来自于 swap 泛型函数，它是 Swift 标准程序库的一部分，并且可以在应用程序使用中自动获得。如果在代码中需要使用 swapTwoValues 函数，你可以使用 Swift 的现有 swap 函数而无需自己操作。

20.3 类型参数

在上述 swapTwoValues 例子中，占位符类型 T 是类型参数的一个例子。类型参数指定并命名一个占位

符类型，并紧跟在函数名后在一对尖括号（<T>）中被编写。

一旦被指定，类型参数可被用于定义函数的参数（例如 swapTwoValues 函数中的参数 a 和 b）类型；或用作函数的返回类型；或用作函数正文中的类型注解。在不同情况下，当函数被调用时，以类型参数代表的占位符类型被替换为实际类型。（在上述 swapTwoValues 例子中，在函数第一次被调用时，T 被替换为 Int，而在第二次调用时被替换为 String。）

可以通过在尖括号中编写多个类型参数，并用逗号隔开以提供多个类型参数。

20.4 命名类型参数

在简单的情况下，如果通用函数或泛型类型需要引用占位符类型（例如上述 swapTwoValues 通用函数类型，或者储存单一类型的通用数集，例如 Array），则一般用单字符为类型参数命名 T。但是，你可以使用任何有效的标识符作为类型参数名。

如果用多个参数定义更复杂的通用函数或泛型类型，则可以帮助提供更多的描述型类型参数名。例如，Swift 的 Dictonary 类型有两个类型参数---一个用作其密钥，另一个用作其数值。如果你自己编写 Dictonary，你可以将这两个类型参数命名为 KeyType 和 ValueType 以提醒你在通用代码中使用它们的目的。

注

始终将类型参数命名为 UpperCamelCase（例如 T 和 KeyType）以表明它们是某一类型的占位符而非其数值。

20.5 泛型类型

除通用函数之外，Swift 能让你定义你自己的泛型类型。有可与任何类型协作的自定义类、结构和枚举，写作方式与 Array 和 Dictonary 类似。

本部分说明如何编写名为 Stack 的泛型集合类型。堆栈是数值的有序集合，与数组类似，但是它有比 Swift 的 Array 类型更受限制的操作集合。数组允许在数组的任何位置插入及删除新项目。但是，堆栈只允许将新条目添加到集合的结尾处（被称为将新值推向堆栈）。同样地，堆栈只允许从集合的结尾处删除条目（被称为从堆栈中弹出一个值）。

注

堆栈的概念被 UINavigatonController 类用来在其导航层次结构中模拟视图控制器。你调用一个 UINavigatonController 类 pushviewController:animated:将视图控制器添加（或推送）到导航堆栈中的方法，以及 popviewControllerAnimated：从导航堆栈中删除（或弹出）一个

视图控制器的方法。当你需要一个严格地“后进先出”法管理集合时，堆栈则是非常有用的集合模式。

下图显示了一个堆栈的推送/弹出动作：详见图 20-1

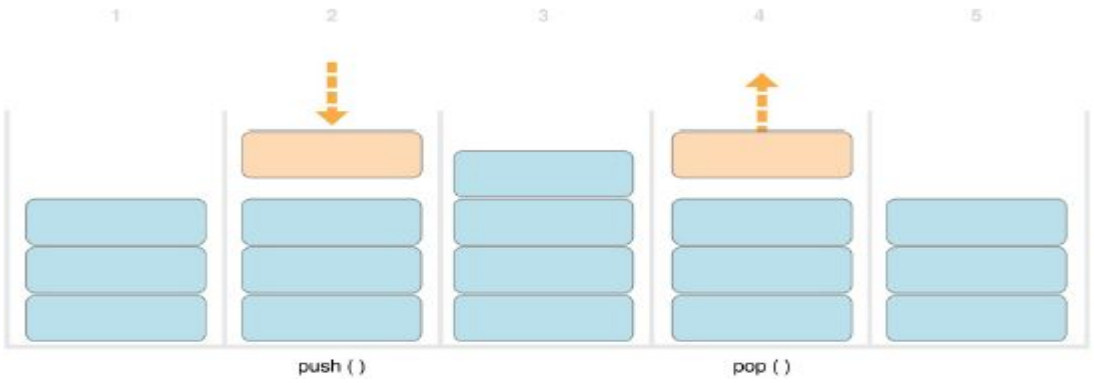


图 20- 1 堆栈的推送/弹出动作

1. 目前在堆栈上有三个值。
2. 第四个值被“推”到堆栈的顶部。
3. 现在，该堆栈拥有四个值，最新的一个值位于最顶部。
4. 堆栈顶部的条目被移除，或“弹出”。
5. 弹出一个值后，该堆栈再次拥有三个值。

以下是如何编写非泛型版本堆栈的例子，在这种情况下堆栈是 Int 值：

```
struct IntStack {
var items = Int[]()
mutating func push(item: Int) {
items.append(item)
}
mutating func pop() -> Int {
```

```
return items.removeLast()
}
}
```

这种结构使用名为 items 的 Array 属性来存储堆栈中的值。堆栈提供了两种方法：推送和弹出，即推送数值或将数值从堆栈中截取出来。这些方法被标记为改变，因为它们需要修改（或改变）结构的条目数组。

然而，上述 IntStack 类型只能用于 Int 值。定义一个能管理任何数值类型的堆栈的 Stack 类将会非常有用。

下面是一个相同代码的泛型版本：

```
struct Stack<T> {
var items = T[]()
mutating func push(item: T) {
items.append(item)
}
mutating func pop() -> T {
return items.removeLast()
}
}
```

注堆栈的泛型版本跟其非泛型版本基本上是一样的，不同之处为其有名为 T 的占位符类型参数而非名为 Int 的实际类型。该类型参数紧跟在结构名称后在一对尖括号（<T>）中被编写。

T 定义了一个名为“some type T”的占位符，稍后将会出现。在结构的定义中，这种未来类型可被称为“T”。在这种情况下，T 被用作三个地方的占位符：

创建名为items的一个属性，与类性值为T 的空数组进行初始化。

指定push类函数有一个名为item 的单一参数，其必须为T 型。

To specify 指定由 pop 类函数返回的值将具有 T 型值。

你可以用与 Array 和 Dictonary 类似的方式创建 Stack 实例，即当使用初始化器语句创建新的实例时，在类型名称后面的一对尖括号中编写用于该特定堆栈的实际类型：

```
var stackOfStrings = Stack<String>()

stackOfStrings.push("uno")

stackOfStrings.push("dos")

stackOfStrings.push("tres")

stackOfStrings.push("cuatro")

// the stack now contains 4 strings
```

下面是将堆栈的四个值推到堆栈顶部后，stackOfStrings 将变成什么样子： 详见图 20-2

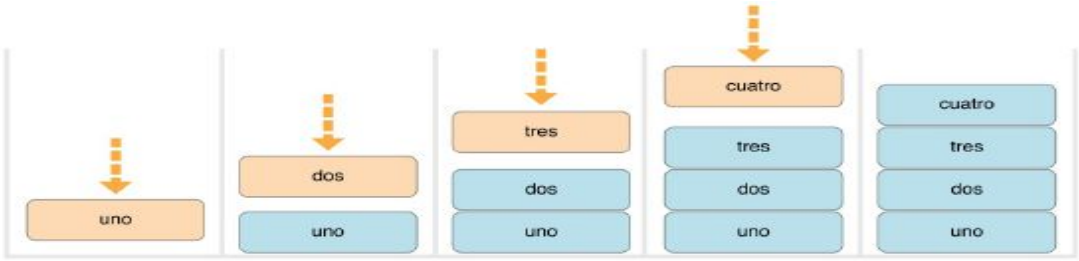


图 20-2 堆栈的四个值推到堆栈顶部后

从堆栈中弹出一个值，返回并移除顶部的值“cuatro”：

```
let fromTheTop = stackOfStrings.pop()

// fromTheTop is equal to "cuatro", and the stack now contains 3 strings
```

下图是堆栈顶部的值弹出后，该堆栈将变成什么样子：详见图 20-3

图 20-3 堆栈顶部的值弹出后

因其为泛型类型，Stack 可用于创建 Swift 中任何有效类型的堆栈，方式与 Array 和 Dictonary 类似。

20.6 类型约束

该 swapTwoValues 函数及堆叠类型适用于任何的类型。但是，其有时在可与泛型函数和泛型类型同时使用的类型上施加一定的类型约束非常有用。类型约束指定类型参数必须继承特定的类，或者符合特定的协议或协议组合。

例如，Swift 的 Dictonary 类型对能够用作词典密钥的类型施加了限制。正如在字典中所描述的那样，字典密钥的类型必须为 hashable。也就是说，它必须提供一种方式，这种方式使自己是唯一可代表的。Dictonary 要求其密钥能够散列（hashable）以便能检查其是否早已含有特定密钥的值。如果没有该要求，Dictonary 则不能判断其是否应该为某一特定密钥插入或替换数值，也不能判断它是否能为已经存在于词典中的给定密钥找到一个值。

该要求由类型约束强制在 Dictonary 的密钥类型中执行，并指定该密钥类型必须符合 Hashable 协议（在 Swift 标准程序库中定义的特殊协议）。Swift 的所有基本类型（例如 String、Int、Double 和 BOO）都可被默认值散列。

当创建自定义泛型类型时，你可以定义自己的类型约束，并且这些约束提供很多泛型编程的能力。抽象概念，如 Hashable 特征类型，是参照其概念上的特点而非其显式类型。

20.7 类型约束语句

你可以在类型参数名后加上一个类或者协议约束来编写类型约束，将其用冒号隔开，并将其作为类型参数列表的一部分。通用函数中类型约束的基础语句如下所示（该语句与泛型类型的语句相同）：

```
func someFunction<T: SomeClass, U: SomeProtocol>(someT: T, someU: U) {
// function body goes here
}
```

上述的假设函数有两个类型参数。第一个类型参数 T，有要求 T 是 SomeClass 的子类的类型约束。第二个类型参数 U，有要求 U 符合 SomeProtocol 协议的类型约束。

20.8 运行中的类型约束

以下是名为 findStr_dex 的非泛型函数，它能给定一个供查找的 String 值以及可供查找 String 值的 String 值数组。findStringIndex 函数返回一个可选 Int 值，如果能被找到，该值将会是数组中第一个匹配字符串的指针；否则为空：

```
func findStringIndex(array: String[], valueToFind: String) -> Int? {
```

```
for (index, value) in enumerate(array) {
if value == valueToFind {
return index
}
}
return nil
}
```

findStringIndex 函数可被用来在字符串数组中找出字符串值：

```
let strings = ["cat", "dog", "llama", "parakeet", "terrapin"]
if let foundIndex = findStringIndex(strings, "llama") {
println("The index of llama is \(foundIndex)")
}
// prints "The index of llama is 2"
```

但是，在数组中查找某一索引值的原则并不是只对字符串有用。使用某种类型 T 的值更换所有提到的字符串，你就可以编写与名为 findIndex 的泛型函数相同的泛函性。

以下是你可能会想到的 findStringIndex 的泛型版本（名为 findIndex）的编写过程。注，该函数的返回类型仍然是 Int?，因为该函数从数组中返回一个可选指数，而非可选值。注，though—this 函数不能编译，原因将在例子后面进行阐释：

```
func findIndex<T>(array: T[], valueToFind: T) -> Int? {
for (index, value) in enumerate(array) {
if value == valueToFind {
return index
}
}
```

```
return nil
}
```

此函数并没有按照上述方式进行编译。问题在于等式检查，“if value == valueToFind”.并非 Swift 中的每一个类型都可以与等于运算符（==）进行比较。例如，如果你创建你自己的类或结构来表示复杂的数据模型，那么该类或结构中的“等于”的意义就不是 Swift 能为你猜到的了。因此，不能保证该代码会对所有可能的类型 T 有效，并且，当你试图编译该代码时，会报告一个适当的错误。

然而，一切都没有丢失。Swift 标准程序库定义了名为 Equatabe 的协议，该协议要求任何符合要求的类型执行等于运算符（==）与不等于运算符（!=）以比较该类型的任意两个值。所有 Swift 的标准类型自动支持 Equatabe 协议。

Equatabe 的任何类型都可以安全地在 findIndex 函数中使用，因为能保证其支持等于运算符。要表达这一事实，在定义函数时你应该编写一个 Equatabe 的类型约束作为类型参数定义的一部分：

```
func findIndex<T: Equatable>(array: T[], valueToFind: T) -> Int? {
for (index, value) in enumerate(array) {
if value == valueToFind {
return index
}
}
return nil
}
```

FindIndex 的单一类型参数被表示为 T：Equatabe，它表示“任何符合 Equatabe 协议的类型 T。”现在 findIndex 函数可成功编译并可用于任何 Equatable 的类型，例如 Double 或 String：

```
let doubleIndex = findIndex([3.14159, 0.1, 0.25], 9.3)
// doubleIndex is an optional Int with no value, because 9.3 is not in the array
let stringIndex = findIndex(["Mike", "Malcolm", "Andrea"], "Andrea")
// stringIndex is an optional Int containing a value of 2
```

20.9 关联类型

在定义协议时，有时声明一个或多个相关类型为协议定义的一部分会非常有用。相关类型可赋予作为协议一部分的类型一个占位符名称（或别名）。在该协议被采用之前，不会指定用于该关联类型的实际类型。用 typealias 关键字指定关联类型。

20.10 运行中的关联类型

以下是名为 Container 的协议的例子，该协议声明了一个名为 Itemtype 的相关类型：ItemType：

```
protocol Container {
typealias ItemType
mutating func append(item: ItemType)
var count: Int { get }
subscript(i: Int) -> ItemType { get }
}
```

Container 协议定义了三个任何容器都须提供的功能：

必须能够利用附加函数为容器添加新项目。

必须能够通过带有 Int 返回值的计数属性访问这些容器类的项目的计数。

必须能够利用带有 Int 指数值得下标检索容器内的每一个项目。

该协议不指定该容器中的条目该如何被存储或它们应为何种类型。该协议仅指定要想成为 Container，任何类型都必须提供的三部分的函数性。只要类型符合这三个要求，其都能够提供附加函数性。

符合 Container 协议的任何类型必须能指定其所存储的数值类型。具体来说，它必须确保只有正确类型的条目才能被添加到容器中，并且它必须能通过下标明确所返回的条目类型。

要定义这些要求，Container 协议需要一种能查阅容器将容纳的元素类型的方法，而无需知道某个特定容器的类型是什么。Container 协议需要指明，任何传给添加方法的值必须有与容器的元素类型一样的类型，并且由容器下标返回的值的类型应该与容器的元素类型一样。

要做到这一点，Container 协议需声明名为 ItemType 的相关类型改写成 typealas ItemType。协议没有定义 ItemType 的别名---该信息将由任何符合要求的类型提供。但是，ItemType 别名提供了一种方法，来查阅容器中条目类型，以及使用添加方法和下标来定义使用的类型，以确保执行所有 Container 的预期行为。

以下是非泛型 IntStack 类型的一个早期版本，符合 Container 协议：

```
struct IntStack: Container {
// original IntStack implementation
var items = Int[]()
mutating func push(item: Int) {
items.append(item)
}
mutating func pop() -> Int {
return items.removeLast()
}
conformance to the Container protocol
typealias ItemType = Int
mutating func append(item: Int) {
self.push(item)
var count: Int {
return items.count
subscript(i: Int) -> Int {
return items[i]
```

IntStack 类型执行 Container 协议的三个要求，并且在不同情况下，其隐藏 IntStack 类型的现有函数性以满足这些要求。

此外，IntStack 指定要执行该 Container 应使用的合适的 ItemType 是 Int 类型。由于 Container 协议的执行，typealias ItemType = Int 的定义将 ItemType 抽象类型转化为 Int 具体类型。

由于 Swift 的类型推导，实际上你并不需要将 Int 具体的 ItemType 声明为 IntStack 定义的一部分。因为 IntStack 符合 Container 协议的所有要求，所以 Swift 仅通过查看添加方法的条目参数类型以及下标的返回类型就可以推断出应使用的适当的 ItemType。

事实上，如果你在上述代码中删除 typealias ItemType = Int，一切仍将正常运行，因为它清楚应用于 ItemType 的类型。

你还可以使泛型 Stack 类型符合 Container 协议：

```
struct Stack<T>: Container {
// original Stack<T> implementation
var items = T[]()
mutating func push(item: T) {
items.append(item)
}
mutating func pop() -> T {
return items.removeLast()
}
conformance to the Container protocol
mutating func append(item: T) {
self.push(item)
var count: Int {
return items.count
subscript(i: Int) -> T {
return items[i]
```

这次，占位符类型参数 T 被用作添加方法的条目参数类型以及下标的返回类型。因此，Swift 可以推断 T 为

20.11 扩展现有类型来指定关联类型

使用扩展添加协议符合性如使用扩展添加协议符合性中所述，你可以扩展现有类型来给某个协议添加符合性。包括具有关联类型的协议。

Swift 的 Array 类型已使用 Int 索引提供了一种添加方法、计数属性和下标，以便检索其元素。这三个功能符合 Container 协议的要求。这意味着仅仅通过声明 Array 采用该协议，你可以扩展 Array 使其符合 Container 协议。使用扩展声明协议采用如使用扩展声明协议采用所述，你可以通过空扩展做到这一点：

```
extension Array: Container {}
```

数组的现有添加方法和下标使 Swift 能推断出用于 ItemType 的合适类型，这与推断上述 Stack 类型一致。定义该扩展后，你可以使用任何一个 Array 作为 Container。

21 Where 语句

如类型约束所述，类型约束能使你在与泛型函数或类型相关的类型中定义要求。

定义关联类型的要求也是有用的。你可以通过将 where 语句定义为类型参数列表的一部分来实现这一点。Where 语句能使你要求某个相关类型符合一个特定的协议和/或要求某些类型参数与相关类型相同。你可以通过以下方式编写 where 语句：紧跟在类型参数列表后写下 where 关键字，然后写下一个或多个相关类型的限制，和/或写下类型与相关类型之间的一个或多个相等关系。

下面的例子定义了名为 allItemsMatch 的泛型函数，该函数能检查两个 Container 示例是否含有同样顺序排列的相同条目。如果所有条目都匹配，则该函数返回一个为“真”的布尔值；否则，则返回为“假”的布尔值。

被检查的两个容器不一定是相同类型的容器（当然也可以是相同类型），但是它们必须要含有相同类型的条目。该要求通过类型约束的组合及 where 语句来表达：

```
func allItemsMatch<

C1: Container, C2: Container

where C1.ItemType == C2.ItemType, C1.ItemType: Equatable>

(someContainer: C1, anotherContainer: C2) -> Bool {
```

```
// check that both containers contain the same number of items
if someContainer.count != anotherContainer.count {
return false
}
// check each pair of items to see if they are equivalent
for i in 0..someContainer.count {
if someContainer[i] != anotherContainer[i] {
return false
}
}
// all items match, so return true
return true
```

此函数需要两个参数：someContainer 和 anotherContaine。someContainer 参数是 C1 类型，而 anotherContainer 参数是 C2 类型。C1 和 C2 是这两个容器类型在函数调用时需要确定的占位符类型参数。

函数的类型参数列表为这两个类型参数指定了以下要求：

C1 必须服从 Container 协议 (被写作 C1: Container).

C2必须服从Container协议 (被写作C2: Container).

C1 的ItemType must be 必须与C2 的ItemType 相同(被写作 C1.ItemType ==

C2.ItemType).

C1 的 ItemType 必须服从 Equatable 协议 (被写作 C1.ItemType:

第三和第四个要求被定义为 where 语句的一部分，并在 where 关键词之后被编写为该函数类型参数列表的一部分。

这些要求意味着：

someContainer 是 类型 C1的容器

anotherContainer 是类型 C2的容器

someContainer 和 anotherContainer contain 包含相同类型的项目。

The items in someContainer can be checked with the not equal operator (!=) to see if

第三和第四个要求结合起来意味着 anotherContainer 中的条目也可以使用丨=运算符进行检查，因为它们与 someContainer 中的条目的类型完全一样。

尽管这两个容器是不一样的容器类型，但是这些要求使 allItemsMatch 函数能比较这两个容器。

AllItemsMatch 函数从检查这两个容器是否包含相同数量的条目开始。如果它们包含不同数量的条目，则没有办法使它们匹配，并且函数返回逻辑值 False。

经过此次检查，函数通过 for-in 循环以及半闭区间运算符遍历完 someContainer 中的所有条目。对于每一项，该函数都会核实 someContainer 中的项是否不等于 anotherContainer 中对应的项。如果两个项不相等，则这两个 containercontainer 容器不匹配，该函数会返回逻辑值 False。

如果“loop 循环”结束时没有找到任何不匹配项，则两个 containercontainer 容器匹配，该函数返回逻辑值 True。

下面是在运行时，allItemsMatch 函数是什么样子的：

```
var stackOfStrings = Stack<String>()

stackOfStrings.push("uno")

stackOfStrings.push("dos")

stackOfStrings.push("tres")

var arrayOfStrings = ["uno", "dos", "tres"]
```

Equatable).

someContainer is a container of type C1.

anotherContainer is a container of type C2.

someContainer and anotherContainer contain the same type of items.

The items in someContainer can be checked with the not equal operator (!=) to see if

they are different from each other.

```
if allItemsMatch(stackOfStrings, arrayOfStrings) {

println("All items match.")

{

println("Not all items match.")

prints "All items match."
```

以上示例创建了一个用来存储 String 字符串的值的 Stack 示例，同时，将三个字符串推送到堆栈（stack）中。该示例还创建了一个数组 Array 示例，用三个相同字符串作为堆栈的数组实量对该数组进行初始化。尽管堆栈和数组类型不同，但它们都符合 Container 协议，而且都包含相同类型的值。因此，你可以调用含有这两个 container 容器的 allItemsMatch 函数作为其参数。在以上示例中，allItemsMatch 函数正确报告：两个 container 容器中所有的项都匹配。

22 高级运算

基本运算符除了基本运算符中所描述的运算符外，Swift 还提供了几种高级运算符，以执行更复杂的值处理流程。其中包括你将在 C 及 Objective-C 语言中所熟悉的所有按位和移位运算符。

不像 C 语言算术运算符，Swift 算术运算符默认不会溢出。溢出行为会被捕获，并发出错误报告。要想选择溢出行为，请使用 Swift 中默认会溢出的第二套算术运算符，如溢出加法运算符（&+）。所有这些溢出运算符都以一个符号(8)开始。

当你定义自己的结构、类及枚举类型时，你自己执行这些自定义类型的标准 Swift 运算符可能会有用。Swift 可以很容易地自定义执行这些运算符，并针对你所创建的每个类型准确地确定其应有的行为。

不仅限于预定义运算符。Swift 让你利用自定义优先级和结合值随意定义自己的自定义中缀、前缀、后缀和赋值运算符。如同使用任何预定义运算符一样，在代码中使用这些运算符，你甚至可以扩展现有类型，以支持你所界定的自定义运算符。

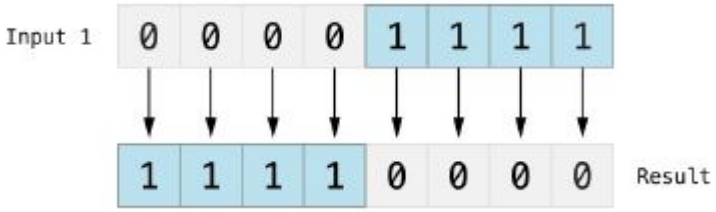
22.1 按位运算符

按位运算符使你能够处理数据结构中的单个原始数据位。通常将它们用于底层编程，如图形编程和设备驱动程序的创建。按位运算符还可帮助你处理外部来源原始数据，如帮助你编码和解码自定义协议通信数据。

Swift 支持所有 C 语言中发现的按位运算符，如下所述。

22.2 按位“非”运算符

按位“非”运算符（～）将一个数值的所有位进行倒置：详见图 22-1



按位“非”运算符是一个前缀运算符，直接显示在它作用的值之前，没有任何空格。

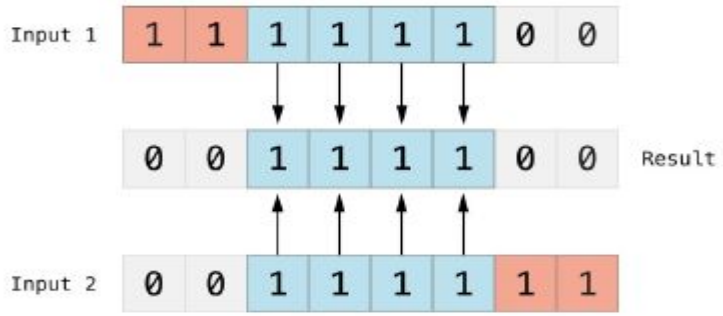
```
let initialBits: UInt8 = 0b00001111

let invertedBits = ~initialBits // equals 11110000
```

UInt8 整数有 8 位，可存储 0 到 255 之间的任意值。该示例对一个带有二进制值 00001111 的 UInt8 整数进行了初始化，该二进制值前四位设置为 0，后四设置为 1，其等于十进制值 15。

然后使用按位“非”运算符创建一个新的常数，称为 invertedBits，其等于 initalBits，但所有位元都倒置了。所有的零都变为一，所有的一都变为零。 invertedBits 的值是 11110000，等于无符号十进制值：240.

22.3 按位“和”运算符

按位“和”运算符（＆）结合两个数的位元。只有当两个输入数字中位元等于 1 时，它才会返回一个将位元设置为 1 的新数：



在以下示例中，firstSxBits 和 lastSixBits 均有四个中间位元为 1。按位“和”运算符将它们进行组合，生成数值 00111100，其等于无符号十进制值 60：

```
let firstSixBits: UInt8 = 0b11111100

let lastSixBits: UInt8 = 0b00111111
```

```
let middleFourBits = firstSixBits & lastSixBits // equals 00111100
```

22.4 按位“或”运算符

按位“或”运算符（|）将比较两个数的位元。如果任何一个输入数字中位元等于 1，它会返回一个将位元设置为 1 的新数：详见图 22-1

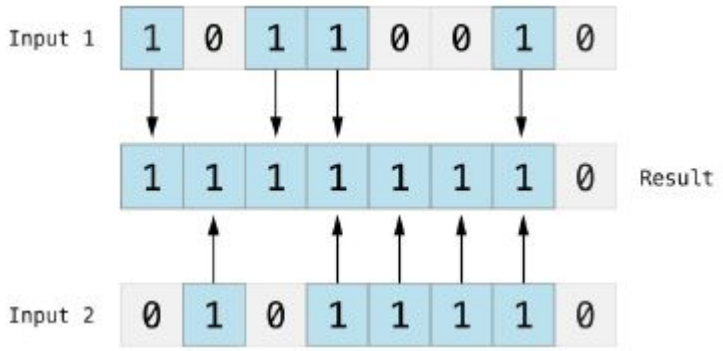


图 22-1 按位“或”运算符

在以下示例中，someBits 和 moreBts 值中不同位元被设置为 1。按位“或”运算符将它们进行组合，生成数值 11111110，其等于无符号十进制值 254：

```
let someBits: UInt8 = 0b10110010

let moreBits: UInt8 = 0b01011110

let combinedbits = someBits | moreBits // equals 11111110
```

22.5 按位“异或”运算符

按位“异或”运算符或“互斥或运算符”（^），比较两个数的位元。当输入数字中的位元不同并被设置成 1，以及当输入数字中的位元相同并被设置为 0 时，运算符返回一个新数： 详见图 22-2

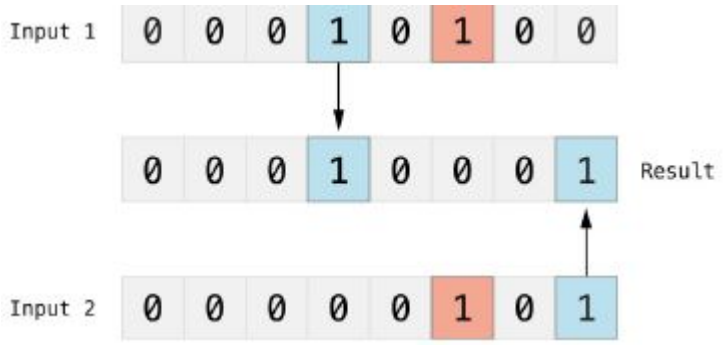


图 22-2 按位“异或”运算符

在以下示例中，firstBits 和 otherBits 的每一个值都有一个位元被设置为 1，而另一值中相同的位元不设置为 1。按位“异或”运算符将在其输出值中把两个数值的位元设置为 1。firstBits 和 otherBits 中所有其他位元都匹配，且输出值设置为 0：

```
let firstBits: UInt8 = 0b00010100

let otherBits: UInt8 = 0b00000101

let outputBits = firstBits ^ otherBits // equals 00010001
```

22.6 按位左移位、右移位运算符

据以下定义的规则，按位左移位运算符（<<）和按位右移位运算符（>>）将一个数的所有位向左或向右移动一定位置。

按位左移位和按位右移位运算都具有整数乘以或除以二者的系数的作用。整数的位向左移动一位会使其值加倍，而向右移动一位会使其值减半。

22.7 无符号整数的移位行为

无符号整数的的移位行为过程如下：

1. 将现有的位元向左或向右移位到所要求的位置。

2. 丢弃任何移位超过整数存储边界的位元。

3. 待原始位元向左或向右移动后，将“零”插入所留空间内。

这种方法被称为逻辑移位。

下图显示了 11111111 << 1（即 11111111 左移 1 位）和 11111111 >> 1（即 11111111 右移 1 位）的结果。蓝色数字移位，灰色数字删除，插入橙色数字“零”：详见图 22-3

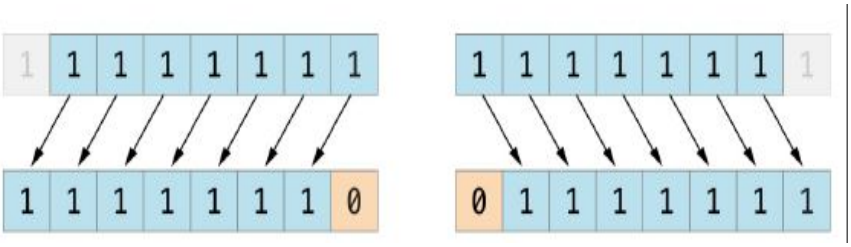


图 22-3 逻辑移位

下面展示的是 Swift 代码中的位元如何移位：

```
let shiftBits: UInt8 = 4 // 00000100 in binary

shiftBits << 1 // 00001000

shiftBits << 2 // 00010000

shiftBits << 5 // 10000000

shiftBits << 6 // 00000000

shiftBits >> 2 // 00000001
```

你可以使用位元移位来编码及解码其他数据类型范围内的值：

```
let pink: UInt32 = 0xCC6699

let redComponent = (pink & 0xFF0000) >> 16 // redComponent is 0xCC, or 204

let greenComponent = (pink & 0x00FF00) >> 8 // greenComponent is 0x66, or 102

let blueComponent = pink & 0x0000FF // blueComponent is 0x99, or 153
```

该示例使用一个 UInt32 常数，标为“粉色”，用来存储层叠样式表，颜色值位粉色。在 Swift 十六进制数表示法中将 CSS 颜色值＃CC6699 写为 0xCC6699。然后利用按位“与”运算符（＆）和按位右移位运算符（>>）将这种颜色分解成红色（CC）、绿色（66）和蓝色（99）三种分量。

通过执行数字 0xcc6699 和 0xFF0000 之间的按位“与”运算得到红色分量。0xFF0000 中的“零”有效“掩模”了 0xcc6699 的第二和第三字节，从而忽略 6699，将 0xcc0000 留作结果。

然后，该数字会向右移动 16 个位数（>> 16）。十六进制数中每对字符都有 8 位，所以右移 16 位会将 0xcc0000 转换为 0x0000cc。这与具有一个十进制的值为 204 的 0xCC 是一样的。

同样，通过执行数字 0xcc6699 和 0x00FF00 之间的按位“与”运算得到绿色分量，输出值为 0x006600。然后将该输出值向右移动 8 位，得出数值 0x66，即为十进制值 102。

最后，通过执行数字 0xcc6699 和 0x0000ff 之间的按位“与”运算得到蓝色分量，输出值为 0x000099。没有必要将它向右移位，因为 0x000099 已经是 0x99，即为十进制值 153。

22.8 将行为移位为符号整数

由于有符号整数用二进制表示，所以有符号整数的移位行为比无符号整数的移位行为要复杂。（为了简便，以下以 8 位有符号整数为例，但这些法则同样适用于任何位数的有符号整数。）

有符号整数利用其第一位元（称为符号位）来表示正负数。符号位 0 表示正，符号位 1 表示负。

其余位（被称为值位）用来存储实际值。正数的保存方式与无符号整数的保存方式完全相同，从 0 开始向上计数。以下为 Int8 类数中各位元找出数值 4 的方法：详见图 22-4

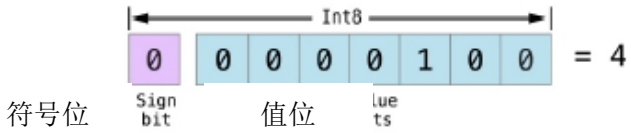


图 22-4 正数的保存方式

符号位为 0（表示“正数”），7 值位正好是以二进制符号表示的数值 4。

然而，负数的存储方式不同。从 2 的 n 次幂减去其绝对值，保存结果，其中，n 是值位数。一个 8 位的数值有七个值位，所以这指的是 2 到 7 或 128 的功率。

下面所示的是一个 Int8 内部的元是找出数值-4 的方法：详见图 22-5

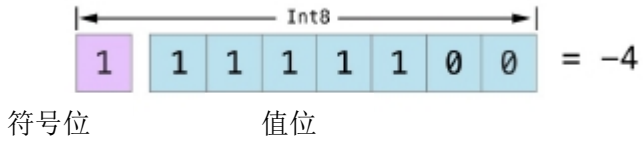


图 22-5 逻辑移位

此时，符号位为 1（表示“负数”），7 值位表示二进制值 124（即 128-4）：详见图 22-6

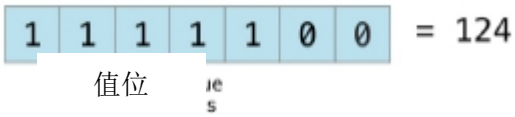


图 22-6 逻辑移位

负数的编码表示方式被称为是“2”的补码。这看似不是表示负数的常规方式，但其有几个优点。

首先，仅仅通过添加一个标准的 8 位（包括符号位）二进制数，并且在完成之后删去不适于 8 位二进制数的任何数字，你就可以从-1 添加至-4：详见图 22-7

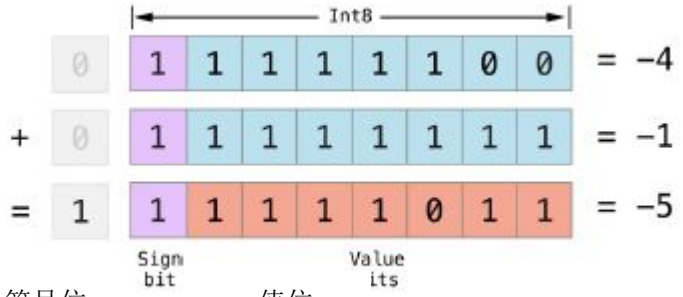


图 22-7 负数的编码

其次，使用二进制补码表示法也可以使你像移动正数那样向左及向右移动负数的位，并且仍然得到这样的结果：即每次左移位会使其值加倍，每次右移位会使其值减半。为此，当有符号整数右移位时，需使用一个附加法则：详见图 22-8

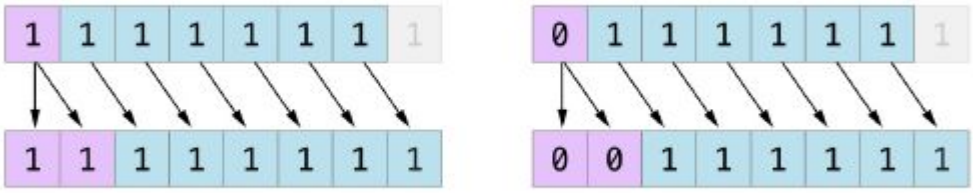


图 22-8

此操作可确保有符号整数右移位后具有相同的符号，称为算术移位。

由于正负数保存方案特殊，二者任一者右移位后都会更接近于零。在此移位过程中保持符号位相同

22.9 溢出运算符

如果您尝试在一个整数常量或其值不固定的变量中插入一个数字时，Swift 会默认报告“出错”而不是允许创建一个无效值。当您处理的数字过大或过小时，这种行为可以提供更多的安全性。

例如，Int16 整数类可容纳-32768～32767 之间的任何的有符号整数。尝试为此范围外的数字设定一个 Int16 常量或变量时会导致“出错”：

```
var potentialOverflow = Int16.max

// potentialOverflow equals 32767, which is the largest value an Int16 can hold

potentialOverflow += 1
```

```
// this causes an error
```

提供错误处理功能，当数值过大或过小时，可以使您更灵活地编码边界值条件。

不过，当您特别希望使用溢出条件截断有效位数字时，您可以选择性地认可该行为而不是触发一个“错误”。Swift 提供了五种可以选择性地认可整数运算中溢出行为的算术溢出运算符。这些运算符都开始于符号（&）：

上溢加法 (&+)

上溢减法 (&-)

上溢乘法 (&*)

上溢除法 (&/)

上溢余数(&%)

22.10 值溢出

以下示例揭示了运用溢出加法运算符（&+）允许无符号值溢出时的结果：

```
var willOverflow = UInt8.max
// willOverflow equals 255, which is the largest value a UInt8 can hold
willOverflow = willOverflow &+ 1
// willOverflow is now equal to 0
```

利用一个 uInt8 类可以容纳的最大值（255 或二进制 11111111）对变量 willOverfow 进行初始化。然后，通过使用溢出加法运算符（&+），该值将以 1 为单位递增。这会使得其二进制表示值超出 UInt8 可容纳的量值，使其溢出其边界，如下图所示。溢出加法运算后仍在 Uint8 范围内的值是 00000000 或零：详见图 22-9

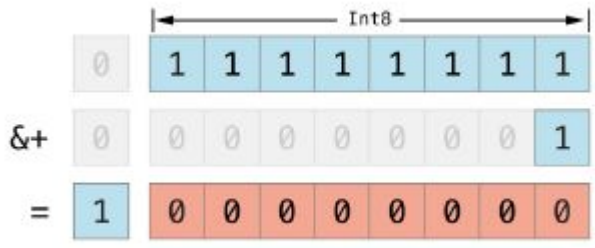


图 22-9

22.11 值下溢

数字也可能会变得太而无法适应其类型的最大边界。举一例来说明：

UInt8 类型可容纳的最小值是 0（即，8 位二进制数 00000000）。如果运用溢出减法运算符从 00000000 中减去 1，该数会溢出返回至 11111111 或十进制值 255：详见图 22-10

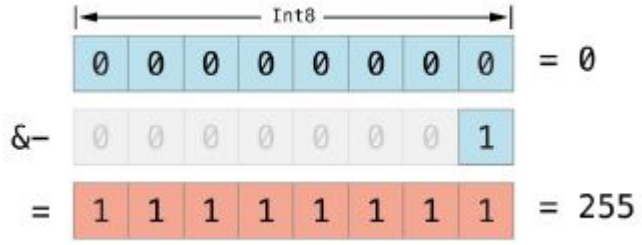


图 22-10

这里介绍了怎样访问Swift代码：

```
var willUnderflow = UInt8.min
// willUnderflow equals 0, which is the smallest value a UInt8 can hold
willUnderflow = willUnderflow &- 1
// willUnderflow is now equal to 255
```

有符号整数会发生类似的下溢。所有有符号整数的减法运算都按直接按照二进制减法执行，正如《按位左、右移位运算符》中所描述的将符号位作为被减数字的一部分。一个 INT8 可以保留的最小数值是-128，用二进制表示为 10000000。运用溢出运算符从该二进制数中减去 1，得出二进制值 01111111，切换符号位，得出正数 127，即 Int8 类可容纳的正数最大值：详见图 22-11

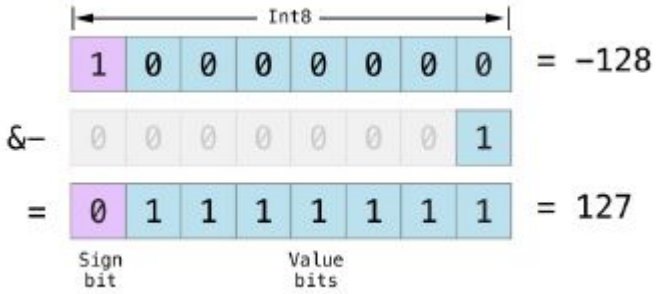


图 22-11《按位左、右移位运算符》

下面所示与 Swift 代码中的过程是一样的：

```
var signedUnderflow = Int8.min

// signedUnderflow equals -128, which is the smallest value an Int8 can hold

signedUnderflow = signedUnderflow &- 1

// signedUnderflow is now equal to 127
```

上述溢出及下溢行为的最终结果归纳如下，对有符号和无符号整数而言，溢出行为总是从最大有效整数值折回至最小有效整数值，而下溢行为总是从最小有效整数值折回至最大有效整数值。

22.12 被零除

某一数字除以零（i / 0）或尝试计算某一数除以零的余数（ i %0）都会导致“出错”：

```
let x = 1

let y = x / 0
```

然而，如果你除以零，这些操作符（&/和&%）的溢出版本会返回零值：

```
let x = 1

let y = x &/ 0

// y is equal to 0
```

优先级和结合性

运算符优先级使得一些运算符的优先级高于其他运算符；我们会首先计算这些优先级高的运算符。

运算符结合性界定了将优先级相同的运算符组合（或关联）在一起的方法 即，从左侧开始组合或从右侧开始组合。即指“它们关联它们左侧的表达式，”或“它们关联它们右侧的表达式。”

处理复合表达式的计算顺序时，重要的是要考虑每个运算符的优先级及结合性。举一例来说：为什么下面的表达式的值是 4？

```
2 + 3 * 4 % 5

// this equals 4
```

如果严格按照从左至右的顺序，你可能会将该表达式解读如下：

加 3 等于 5;

5 乘以 4 等于 20;

20 余除 5 等于 0

然而，实际答案是 4 而不是 0。首先要评估高优先级运算符然后才是低优先级运算符。如同在 C 语言中，在 Swift 中，乘法运算符（*）及求余运算符（%）的运算优先级都有要高于加法运算符（+）的运算优先级。结果是，它们都在增加值被考虑之前进行了评估。

然而，乘法和余数有相同的优先级。要制定出所使用的精确的评估顺序，您还需要考虑它们的结合性。乘法和余数都与其左边的表达式有联系。 即指，围绕表达式的这些部分从其左侧开始加上隐式括号：

2 + ((3 * 4) % 5)

(3 * 4) is 12, so this is equivalent to:

2 + (12 % 5)

(12 % 5) is 2, so this is equivalent to:

2 + 2

通过这种计算方法得出的最终答案是 4。

欲获得 Swift 运算符优先级和结合性法则完整列表，请参阅表达式。

注

Swift 运算符的优先级和结合性法则比 C 语言和 Objective-C 语言中的法则更简单，更易于预测。然而，这也意味着它们与 C 语言不同。将现有代码植入 Swift 时，务必确保运算符仍按您预定的方式发挥交互作用。

21.1 运算符函数

类和结构可以自行实现现有的运算符。这被称为现有运算符过载。

以下示例显示了自定义结构算术加法运算符（+）生效的方法。算术加法运算符是一个可在两个目标上进行运算的二元运算符，并且因为它出现在两个目标中间，故而，也被称为中缀，。

该示例首先定义了一个二维位置矢量（x，y）的 vector2d 结构，然后定义了添加到 vector2d 结构实例的运算符函数：

```
struct Vector2D {var x = 0.0, y = 0.0}

@infix func + (left: Vector2D, right: Vector2D) -> Vector2D {

return Vector2D(x: left.x + right.x, y: left.y + right.y)}
```

将运算符函数定义为一个名为+的全局函数，其接受两个 vector2d 类输入参数并返回一个 vector2d 类输出值。描述运算符函数时，您可以通过在函数关键字前写上@中缀属性使中缀运算符生效。

在实现过程中，将输入参数命名为“左”和“右”，用以表示+运算符左侧和右侧的 vector2d 实例。该函数返回一个新的 vector2d 实例，其中，利用相加的两个 vector2d 实例的 x 和 y 属性总和对这个新的 vector2d 实例的 x 和 y 属性进行了初始化。

将该函数定义为全局函数而非 vector2d 结构的方法，因此，可将其用作现有 vector2d 实例之间的中缀运算符：

```
let vector = Vector2D(x: 3.0, y: 1.0)

let anotherVector = Vector2D(x: 2.0, y: 4.0)

let combinedVector = vector + anotherVector

// combinedVector is a Vector2D instance with values of (5.0, 5.0)
```

该示例使矢量（3.0,1.0）和（2.0,4.0）相加，得出矢量（5.0,5.0）。详见图 22-12

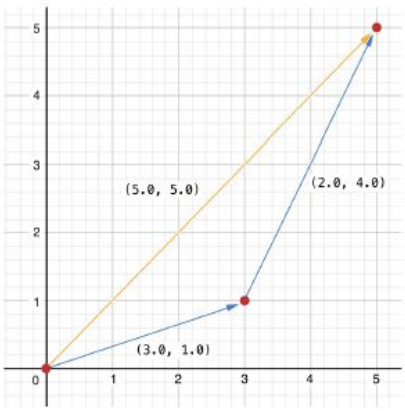


图 22-12 矢量图

22.13 前缀运算符和后缀运算符

如上所示示例演示了二元中缀运算符的自定义生效过程。类和结构也可以展示标准一元运算符的生效过程。一元运算符针对一个目标进行操作。如果在它们的目标之前（如-a），它们即为前缀；如果在它们的

目标之后（如 i++），则它们为后缀运算符。

描述运算符函数时，您可以通过在函数关键字前写上@前缀或@后缀属性的方法，使前缀或后缀一元运算符生效：

```
@prefix func - (vector: Vector2D) -> Vector2D {
return Vector2D(x: -vector.x, y: -vector.y)
}
```

上面的例子中对 Vector2D 实例执行了一元减运算符（-a）。一元减运算符是前缀运算符，所以该函数必须具有@前缀属性。

对于简单的数值，一元减运算符可将正数转换为等效负数，反之亦然。相应的 vector2d 实例实现过程在 x 和 y 属性上均进行此运算：

```
let positive = Vector2D(x: 3.0, y: 4.0)
let negative = -positive
// 负值为一个值为（-3.0，-4.0）的 vector2D 实例中
let alsoPositive = -negative
// 同样，正值为一个值为（3.0,4.0）的 vector2D 实例
```

22.14 复合赋值运算符

复合赋值运算符将赋值（=）与另一个运算结合在一起。例如，加法赋值运算符（+=）将加法及赋值合并到一次运算中。实现复合赋值运算的运算符函数必须具备@ assgnment 属性。您还必须将一个复合赋值运算符的左侧输入参数标记为 inout，因为该参数值将直接在运算符函数中予以修改。

以下示例给出了 vector2d 实例加法赋值运算符函数的生效过程：

```
@assignment func += (inout left: Vector2D, right: Vector2D) {left = left + right}
```

因为先前已定义了加法运算符，所以您无需在此再次实现加法运算过程。相反，加法赋值运算符函数运用了现有加法运算功能函数，并用它将左值设置为左值加右值：

```
var original = Vector2D(x: 1.0, y: 2.0)
let vectorToAdd = Vector2D(x: 3.0, y: 4.0)
original += vectorToAdd
```

```
// original now has values of (4.0, 6.0)
```

在 vector2d 实例的前缀增量运算符（++a）的生效过程中，您可以将@ assgnment 属性与@ prefx 或@ postfx 属性合并：

```
@prefix @assignment func ++ (inout vector: Vector2D) -> Vector2D {
vector += Vector2D(x: 1.0, y: 1.0)
return vector
}
```

上述前缀增量运算符函数运用了先前定义的加法赋值运算符。将 x、y 值为 1.0 的 vector2d 与据此所称的 vector2d 相加，并返回结果：

```
var toIncrement = Vector2D(x: 3.0, y: 4.0)
let afterIncrement = ++toIncrement
// toIncrement now has values of (4.0, 5.0)
// afterIncrement also has values of (4.0, 5.0)
```

注

不可能使默认的赋值运算符（=）过载。只有复合赋值运算符可以过载。同样，三元条件运算符（a ？ b:c）也不能过载。

22.15 等价运算符

自定义类和结构不接收等价运算符，即不接收“等于”运算符（==），与“不等于”运算符（！=）的默认生效过程。Swift 无法猜出针对您自己的自定义类型中，什么具备“等于”资格，因为“等于”的含义取决于这些类型在您的代码中所发挥的作用。

欲使用等价运算符核对自己的自定义类型的等价性，请按照与其他中缀运算符相同的方式给出运算符的生效过程：

```
@infix func == (left: Vector2D, right: Vector2D) -> Bool {
return (left.x == right.x) && (left.y == right.y)
}
```

```
@infix func != (left: Vector2D, right: Vector2D) -> Bool {
return !(left == right)
}
```

以上示例给出了“等于”运算符(==)的生效过程，用于核对两个 vector2d 实例的值是否等价。在 vector2d 情景下，可以将“等于”理解为“两个实例具有相同的 x 值和 y 值”，这也是 运算符实现过程中所运用的逻辑。该示例还给出了“不等于”运算符（！=）的生效过程，即仅返回与“等于”运算符结果相反的结果。

现在，您可以使用这些运算符来核对两个 vector2d 实例是否相等：

```
let twoThree = Vector2D(x: 2.0, y: 3.0)
let anotherTwoThree = Vector2D(x: 2.0, y: 3.0)
if twoThree == anotherTwoThree {
println("These two vectors are equivalent.")
}
// prints "These two vectors are equivalent."
```

22.16 自定义运算符

除了 Swift 提供的标准运算符外，您还可以描述并展示自己的自定义运算符的生效过程。自定义运算符只能用字符/ =来定义

- + * % < > ! & | ^ . ~.

可在全局范围内使用 operator 关键字用来描述新的运算符，并可将其描述为前缀、中缀或后缀运算符：

```
operator prefix +++ {}
```

上面的例子定义了一个新的名为+ + +的前缀运算符。该运算符在 Swift 中不具有实际意义，所以，下面给出了 vector2d 实例在具体处理情境中的自定义含义过程。为了说明该示例，将+++视为一个新的“前缀倍增加法器”运算符。利用先前定义的加法赋值运算符使矢量与自身相加，使 vector2d 实例的 x 和 y 值翻倍：

```
@prefix @assignment func +++ (inout vector: Vector2D) -> Vector2D {
```

```
vector += vector
return vector
}
```

+++实现过程与 vector2d 的++的实现过程非常相似，不同的是该运算符函数使矢量与自身相加，而不是与 vector2d（1.0，1.0）相加：

```
var toBeDoubled = Vector2D(x: 1.0, y: 4.0)
let afterDoubling = +++toBeDoubled
// toBeDoubled now has values of (2.0, 8.0)
// afterDoubling also has values of (2.0, 8.0)
```

22.17 自定义中缀运算符的优先级和结合性

自定义中缀运算符也可以指定一种优先级和结合性。参见《优先级和结合性》，查看这两个特点是如何影响一个中缀运算符与其他中缀运算符的交互作用的。

结合性的可能的值为 left、 right 和 none。如果左结合运算符写入时紧临相同优先级的其他左结合运算符，则其关联其左侧。同样地，如果右结合运算符写入时紧临相同优先级的其他右结合运算符，则其关联其右侧。无法紧临优先级相同的其他运算符写入非关联运算符。

如果未指定，则结合性的值默认为 none。如果未指定，则优先级的值默认为 100。

以下示例定义了一个新的自定义中缀运算符，即+-，其具有左结合性，且优先级为 140：

```
operator infix +- { associativity left precedence 140 }
func +- (left: Vector2D, right: Vector2D) -> Vector2D {
return Vector2D(x: left.x + right.x, y: left.y - right.y)
}
let firstVector = Vector2D(x: 1.0, y: 2.0)
let secondVector = Vector2D(x: 3.0, y: 4.0)
let plusMinusVector = firstVector +- secondVector
// plusMinusVector is a Vector2D instance with values of (4.0, -2.0)
```

该运算符使两个矢量的 x 值相加，并由第一个矢量的 y 值减去第二个矢量的 y 值。因为其本质上是一

个“加法”运算符，所以其被赋予与默认加法中缀运算符（如+和-）相同的（左）结合性和优先级值（140）。

欲获得 Swift 运算符优先级和结合性法则完整列表，请参阅《表达式》。

第 3 章　语言参考

1 关于语言参考

本书这一部分描述了 Swift 编程语言的形式文法。此处描述的文法旨在帮助您更详细地了解该语言，而不是让您直接编译一个分析程序或编译程序。

Swift 语言相对较小，因为实际上在 Swift 标准库中定义了普遍存在于 Swift 代码中的许多常见类型、函数和运算符。虽然这些类型、函数和运算符并不是 Swift 语言本身的一部分，但本书在这一部分却将它们广泛用在了探讨示例和代码示例中。

1.1 如何阅读语法

用来描述 Swift 编程语言形式文法的符号应符合以下几项规则：

箭头（→）用于标记文法产生式，可以理解为“可包含”。

句法类型用斜体文字表示，分列在文法生成法则的两侧。

文字和标点符号用等宽粗体文本表示，仅列于文法生成法则的右手侧。

替代语法产生式用竖线（|）隔开。 当替代性生成式太长不便阅读时，将它们分成多个新型文法生成法则。

在少数情况下，会使用常规字体文字来描述文法生成法则右手侧内容。

可选句法类型和文字通过一个尾部标 opt.进行标记。例如，一个 getter-setter 模块的语法定义方式如下：

GRAMMAR OF A GETTER- SETTER BLOCK

getter-setter-block → { getter-clause setter-clause opt } { setter-clause getterclause}