

# Introduction · GitBook

---

 [runoob.com/manual/gitbook/swift5/source/\\_book](http://runoob.com/manual/gitbook/swift5/source/_book)

2016.9.23: 已经更新到 Swift 3.0。

## 3.0 更新说明

---

Swift 3.0 是自 Swift 开源以来第一个大的版本更新。从语言角度不兼容之前的 Swift 2.2 和 Swift 2.3 版本。Swift 3.0 的更新说明，大家可以查看[官方 blog 的说明](#)，也可以关注[SwiftGG 最新的文章](#)。学习官方文档，是掌握语言特性点的最佳途径，感谢翻译的小伙伴们为 Swift 社区所做贡献！

## 3.0 译者记录

---

相关issue

- Functions: [cravygy](#)
- Control Flow: [Realank](#)
- Closures: [LanfordCai](#)
- Protocols: [chenmingbiao](#)
- The Basics:[chenmingbiao](#)
- Advanced Operators: [mmoaay](#)

Language Reference:

- Attributes: [WhoJave](#)
- Statements: [chenmingjia](#)
- Declarations: [chenmingjia](#)
- Expressions: [chenmingjia](#)
- Types: [lettleprince](#)
- Generic Parameters and Arguments: [chenmingjia](#)

感谢阅读！

# 关于 Swift · GitBook

---

 [runoob.com/manual/gitbook/swift5/source/\\_book/chapter1/01\\_about\\_swift.html](https://runoob.com/manual/gitbook/swift5/source/_book/chapter1/01_about_swift.html)

## 关于 Swift

---

Swift 是一种非常好的编写软件的方式，无论是手机，台式机，服务器，还是其他运行代码的设备。它是一种安全，快速和互动的编程语言，将现代编程语言的精华和苹果工程师文化的智慧，以及来自开源社区的多样化贡献结合了起来。编译器对性能进行了优化，编程语言对开发进行了优化，两者互不干扰，鱼与熊掌兼得。

Swift 对于初学者来说也很友好。它是第一个既满足工业标准又像脚本语言一样充满表现力和趣味的系统编程语言。它支持代码预览（playgrounds），这个革命性的特性可以允许程序员在不编译和运行应用程序的前提下运行 Swift 代码并实时查看结果。

Swift 通过采用现代编程模式来避免大量常见编程错误：

- 变量始终在使用前初始化。
- 检查数组索引起超出范围的错误。
- 检查整数是否溢出。
- 可选值确保明确处理 `nil` 值。
- 内存被自动管理。
- 错误处理允许从意外故障控制恢复。

Swift 代码被编译和优化，以充分利用现代硬件。语法和标准库是基于指导原则设计的，编写代码的明显方式也应该是最好的。安全性和速度的结合使得 Swift 成为从“Hello，world！”到整个操作系统的绝佳选择。

Swift 将强大的类型推理论和模式匹配与现代轻巧的语法相结合，使复杂的想法能够以清晰简洁的方式表达。因此，代码不仅更容易编写，而且易于阅读和维护。

Swift 已经进行了多年，并且随着新特性和功能的不断发展。我们对 Swift 的目标是雄心勃勃的。我们迫不及待想看到你用它创建出的东西。

# 版本兼容性 · GitBook

---



[runoob.com/manual/gitbook/swift5/source/\\_book/chapter1/02\\_version\\_compatibility.html](https://runoob.com/manual/gitbook/swift5/source/_book/chapter1/02_version_compatibility.html)

## 版本兼容性

---

本书描述的是在 Xcode 11 中的默认 Swift 版本 Swift 5.1。你可以使用 Xcode11 来构建 Swift 5.1、Swift 4.2 或 Swift 4 写的项目。

当您使用 Xcode 11 构建 Swift 4 和 Swift 4.2 代码时，除了下面的功能仅支持 Swift 5.1，其他大多数功能都依然可用。

- 返回值是不透明类型的函数依赖 Swift 5.1 运行时。
- **try?** 表达式不会为已返回可选类型的代码引入额外的可选类型层级。
- 大数字的整型字面量初始化代码的类型将会被正确推导，例如 **UInt64(0xffff\_ffff\_ffff\_ffff)** 将会被推导为整型类型而非溢出。

用 Swift 5.1 写的项目可以依赖用 Swift 4.2 或 Swift 4 写的项目，反之亦然。这意味着，如果你将一个大的项目分解成多个框架（framework），你可以每次一个框架地迁移 Swift 4 代码到 Swift 5.1。

# Swift 初见 · GitBook

---

 [runoob.com/manual/gitbook/swift5/source/\\_book/chapter1/03\\_a\\_swift\\_tour.html](http://runoob.com/manual/gitbook/swift5/source/_book/chapter1/03_a_swift_tour.html)

## Swift 初见

---

通常来说，编程语言教程中的第一个程序应该在屏幕上打印“Hello, world”。在 Swift 中，可以用一行代码实现：

```
print("Hello, world!")
```

如果你写过 C 或者 Objective-C 代码，那你应该很熟悉这种形式——在 Swift 中，这行代码就是一个完整的程序。你不需要为了输入输出或者字符串处理导入一个单独的库。全局作用域中的代码会被自动当做程序的入口点，所以你也不需要 `main()` 函数。你同样不需要在每个语句结尾写上分号。

这个教程会通过一系列编程例子来让你对 Swift 有初步了解，如果你有什么不理解的地方也不用担心——任何本章介绍的内容都会在后面的章节中详细讲解到。

### 注意

最好的体验是把这一章作为 Playground 文件在 Xcode 中打开。Playgrounds 允许你可以编辑代码并立刻看到输出结果。

[Download Playground](#)

## 简单值

---

使用 `let` 来声明常量，使用 `var` 来声明变量。一个常量的值，在编译的时候，并不需要有明确的值，但是你只能为它赋值一次。这说明你可以用一个常量来命名一个值，一次赋值就即可在多个地方使用。

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

常量或者变量的类型必须和你赋给它们的值一样。然而，你不用明确地声明类型。当你通过一个值来声明变量和常量时，编译器会自动推断其类型。在上面的例子中，编译器推断出 `myVariable` 是一个整数类型，因为它的初始值是整数。

如果初始值没有提供足够的信息（或者没有初始值），那你需要在变量后面声明类型，用冒号分割。

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

## 练习

创建一个常量，显式指定类型为 `Float` 并指定初始值为 4。

值永远不会被隐式转换为其他类型。如果你需要把一个值转换成其他类型，请显式转换。

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

## 练习

删除最后一行中的 `String`，错误提示是什么？

有一种更简单的把值转换成字符串的方法：把值写到括号中，并且在括号之前写一个反斜杠 (\)。例如：

```
let apples = 3  
let oranges = 5  
let appleSummary = "I have \(apples) apples."  
let fruitSummary = "I have \(apples + oranges) pieces of fruit."
```

## 练习

使用 `\()` 来把一个浮点计算转换成字符串，并加上某人的名字，和他打个招呼。

使用一对三个单引号 ( """ ) 来包含多行字符串内容，字符串中的内容（包括引号、空格、换行符等）都会保留下来。举个例子：

```
let quotation = """  
I said "I have \(apples) apples."  
And then I said "I have \(apples + oranges) pieces of fruit."  
"""
```

使用方括号 `[]` 来创建数组和字典，并使用下标或者键 (key) 来访问元素。最后一个元素后面允许有个逗号。

```
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数组或者空字典。

```
let emptyArray = [String]()  
let emptyDictionary = [String: Float]()
```

如果类型信息可以被推断出来，你可以用 `[]` 和 `[:]` 来创建空数组和空字典——就像你声明变量或者给函数传参数的时候一样。

```
shoppingList = []
occupations = [:]
```

## 控制流

---

使用 `if` 和 `switch` 来进行条件操作，使用 `for-in`、`while` 和 `repeat-while` 来进行循环。包裹条件和循环变量的括号可以省略，但是语句体的大括号是必须的。

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

在 `if` 语句中，条件必须是一个布尔表达式——这意味着像 `if score { ... }` 这样的代码将报错，而不会隐形地与 `o` 做对比。

你可以一起使用 `if` 和 `let` 一起来处理值缺失的情况。这些值可由可选值来代表。一个可选的值是一个具体的值或者是 `nil` 以表示值缺失。在类型后面加一个问号 (`?`) 来标记这个变量的值是可选的。

```
var optionalString: String? = "Hello"
print(optionalString == nil)

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

### 练习

把 `optionalName` 改成 `nil`，`greeting` 会是什么？添加一个 `else` 语句，当 `optionalName` 是 `nil` 时给 `greeting` 赋一个不同的值。

如果变量的可选值是 `nil`，条件会判断为 `false`，大括号中的代码会被跳过。如果不是 `nil`，会将值解包并赋给 `let` 后面的常量，这样代码块中就可以使用这个值了。另一种处理可选值的方法是通过使用 `??` 操作符来提供一个默认值。如果可选值缺失的话，可以使用默认值来代替。

```
let nickName: String? = nil
let fullName: String = "John Appleseed"
let informalGreeting = "Hi \(nickName ?? fullName)"
```

`switch` 支持任意类型的数据以及各种比较操作——不仅仅是整数以及测试相等。

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

### 练习

删除 `default` 语句，看看会有什么错误？

注意 `let` 在上述例子的等式中是如何使用的，它将匹配等式的值赋给常量 `x`。

运行 `switch` 中匹配到的 `case` 语句之后，程序会退出 `switch` 语句，并不会继续向下运行，所以不需要在每个子句结尾写 `break`。

你可以使用 `for-in` 来遍历字典，需要一对儿变量来表示每个键值对。字典是一个无序的集合，所以他们的键和值以任意顺序迭代结束。

```
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
    }
  }
}
print(largest)
```

### 练习

添加另一个变量来记录最大数字的种类 (`kind`)，同时仍然记录这个最大数字的值。

使用 `while` 来重复运行一段代码直到条件改变。循环条件也可以在结尾，保证能至少循环一次。

```
var n = 2
while n < 100 {
    n *= 2
}
print(n)
```

```
var m = 2
repeat {
    m *= 2
} while m < 100
print(m)
```

你可以在循环中使用 `..<<` 来表示下标范围。

```
var total = 0
for i in 0..<4 {
    total += i
}
print(total)
```

使用 `..<` 创建的范围不包含上界，如果想包含的话需要使用 `...<`。

## 函数和闭包

使用 `func` 来声明一个函数，使用名字和参数来调用函数。使用 `->` 来指定函数返回值的类型。

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

练习

删除 `day` 参数，添加一个参数来表示今天吃了什么午饭。

默认情况下，函数使用它们的参数名称作为它们参数的标签，在参数名称前可以自定义参数标签，或者使用 `_` 表示不使用参数标签。

```
func greet(_ person: String, on day: String) -> String {
    return "Hello \(person), today is \(day)."
}
greet("John", on: "Wednesday")
```

使用元组来生成复合值，比如让一个函数返回多个值。该元组的元素可以用名称或数字来获取。

```

func calculateStatistics(scores: [Int]) -> (min: Int, max: Int, sum: Int) {
    var min = scores[0]
    var max = scores[0]
    var sum = 0

    for score in scores {
        if score > max {
            max = score
        } else if score < min {
            min = score
        }
        sum += score
    }

    return (min, max, sum)
}

let statistics = calculateStatistics(scores:[5, 3, 100, 3, 9])
print(statistics.sum)
print(statistics.2)

```

函数可以嵌套。被嵌套的函数可以访问外侧函数的变量，你可以使用嵌套函数来重构一个太长或者太复杂的函数。

```

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

```

函数是第一等类型，这意味着函数可以作为另一个函数的返回值。

```

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

```

函数也可以当做参数传入另一个函数。

```
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  
}  
var numbers = [20, 19, 7, 12]  
hasAnyMatches(list: numbers, condition: lessThanTen)
```

函数实际上是一种特殊的闭包：它是一段能之后被调取的代码。闭包中的代码能访问闭包作用域中的变量和函数，即使闭包是在一个不同的作用域被执行的 - 你已经在嵌套函数的例子中看过了。你可以使用 `{}` 来创建一个匿名闭包。使用 `in` 将参数和返回值类型的声明与闭包函数体进行分离。

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

### 练习

重写闭包，对所有奇数返回 0。

有很多种创建更简洁的闭包的方法。如果一个闭包的类型已知，比如作为一个代理的回调，你可以忽略参数，返回值，甚至两个都忽略。单个语句闭包会把它语句的值当做结果返回。

```
let mappedNumbers = numbers.map({ number in 3 * number })  
print(mappedNumbers)
```

你可以通过参数位置而不是参数名字来引用参数——这个方法在非常短的闭包中非常有用。当一个闭包作为最后一个参数传给一个函数的时候，它可以直接跟在括号后面。当一个闭包是传给函数的唯一参数，你可以完全忽略括号。

```
let sortedNumbers = numbers.sorted { $0 > $1 }  
print(sortedNumbers)
```

## 对象和类

使用 `class` 和类名来创建一个类。类中属性的声明和常量、变量声明一样，唯一的区别就是它们的上下文是类。同样，方法和函数声明也一样。

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

## 练习

使用 `let` 添加一个常量属性，再添加一个接收一个参数的方法。

要创建一个类的实例，在类名后面加上括号。使用点语法来访问实例的属性和方法。

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

这个版本的 `Shape` 类缺少了一些重要的东西：一个构造函数来初始化类实例。使用 `init` 来创建一个构造器。

```
class NamedShape {  
    var numberOfSides: Int = 0  
    var name: String  
  
    init(name: String) {  
        self.name = name  
    }  
  
    func simpleDescription() -> String {  
        return "A shape with \(numberOfSides) sides."  
    }  
}
```

注意 `self` 被用来区别实例变量 `name` 和构造器的参数 `name`。当你创建实例的时候，像传入函数参数一样给类传入构造器的参数。每个属性都需要赋值——无论是通过声明（就像 `numberOfSides`）还是通过构造器（就像 `name`）。

如果你需要在对象释放之前进行一些清理工作，使用 `deinit` 创建一个析构函数。

子类的定义方法是在它们的类名后面加上父类的名字，用冒号分割。创建类的时候并不需要一个标准的根类，所以你可以根据需要添加或者忽略父类。

子类如果要重写父类的方法的话，需要用 `override` 标记——如果没有添加 `override` 就重写父类方法的话编译器会报错。编译器同样会检测 `override` 标记的方法是否确实在父类中。

```

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)."
    }
}

let test = Square(sideLength: 5.2, name: "my test square")
test.area()
test.simpleDescription()

```

### 练习

创建 `NamedShape` 的另一个子类 `Circle`，构造器接收两个参数，一个是半径一个名称，在子类 `Circle` 中实现 `area()` 和 `simpleDescription()` 方法。

除了储存简单的属性之外，属性可以有 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 triangle with sides of length \(sideLength)."
    }
}

var triangle = EquilateralTriangle(sideLength: 3.1, name: "a triangle")
print(triangle.perimeter)
triangle.perimeter = 9.9
print(triangle.sideLength)

```

在 `perimeter` 的 setter 中，新值的名字是 `newValue`。你可以在 `set` 之后显式的设置一个名字。

注意 `EquilateralTriangle` 类的构造器执行了三步：

1. 设置子类声明的属性值
2. 调用父类的构造器
3. 改变父类定义的属性值。其他的工作比如调用方法、getters 和 setters 也可以在这个阶段完成。

如果你不需要计算属性，但是仍然需要在设置一个新值之前或者之后运行代码，使用 `willSet` 和 `didSet`。写入的代码会在属性值发生改变时调用，但不包含构造器中发生值改变的情况。比如，下面的类确保三角形的边长总是和正方形的边长相同。

```
class TriangleAndSquare {  
    var triangle: EquilateralTriangle {  
        willSet {  
            square.sideLength = newValue.sideLength  
        }  
    }  
    var square: Square {  
        willSet {  
            triangle.sideLength = newValue.sideLength  
        }  
    }  
    init(size: Double, name: String) {  
        square = Square(sideLength: size, name: name)  
        triangle = EquilateralTriangle(sideLength: size, name: name)  
    }  
}  
  
var triangleAndSquare = TriangleAndSquare(size: 10, name: "another test shape")  
print(triangleAndSquare.square.sideLength)  
print(triangleAndSquare.triangle.sideLength)  
triangleAndSquare.square = Square(sideLength: 50, name: "larger square")  
print(triangleAndSquare.triangle.sideLength)
```

处理变量的可选值时，你可以在操作（比如方法、属性和子脚本）之前加 `?`。如果 `?` 之前的值是 `nil`，`?` 后面的东西都会被忽略，并且整个表达式返回 `nil`。否则，`?` 之后的东西都会被运行。在这两种情况下，整个表达式的值也是一个可选值。

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

## 枚举和结构体

使用 `enum` 来创建一个枚举。就像类和其他所有命名类型一样，枚举可以包含方法。

```
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.rawValue)
        }
    }
}
let ace = Rank.ace
let aceRawValue = ace.rawValue
```

### 练习

写一个函数，通过比较它们的原始值来比较两个 `Rank` 值。

默认情况下，Swift 按照从 0 开始每次加 1 的方式为原始值进行赋值，不过你可以通过显式赋值进行改变。在上面的例子中，`Ace` 被显式赋值为 1，并且剩下的原始值会按照顺序赋值。你也可以使用字符串或者浮点数作为枚举的原始值。使用 `rawValue` 属性来访问一个枚举成员的原始值。

使用 `init?(rawValue:)` 初始化构造器来创建一个带有原始值的枚举成员。如果存在与原始值相应的枚举成员就返回该枚举成员，否则就返回 `nil`。

```
if let convertedRank = Rank(rawValue: 3) {
    let threeDescription = convertedRank.simpleDescription()
}
```

枚举的关联值是实际值，并不是原始值的另一种表达方法。实际上，如果没有比较有意义的原始值，你就不需要提供原始值。

```

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"
        }
    }
}
let hearts = Suit.hearts
let heartsDescription = hearts.simpleDescription()

```

### 练习

给 `Suit` 添加一个 `color()` 方法，对 `spades` 和 `clubs` 返回 “black”，对 `hearts` 和 `diamonds` 返回 “red”。

注意在上面的例子中用了两种方式引用 `hearts` 枚举成员：给 `hearts` 常量赋值时，枚举成员 `Suit.hearts` 需要用全名来引用，因为常量没有显式指定类型。在 `switch` 里，枚举成员使用缩写 `.hearts` 来引用，因为 `self` 的值已经是一个 `suit` 类型，在已知变量类型的情况下可以使用缩写。

如果枚举成员的实例有原始值，那么这些值是在声明的时候就已经决定了，这意味着不同枚举实例的枚举成员总会有一个相同的原始值。当然我们也可以为枚举成员设定关联值，关联值是在创建实例时决定的。这意味着不同的枚举成员的关联值都可以不同。你可以把关联值想象成枚举成员的寄存属性。例如，考虑从服务器获取日出和日落的时间的情况。服务器会返回正常结果或者错误信息。

```

enum ServerResponse {
    case result(String, String)
    case failure(String)
}

let success = ServerResponse.result("6:00 am", "8:09 pm")
let failure = ServerResponse.failure("Out of cheese.")

switch success {
    case let .result(sunrise, sunset):
        print("Sunrise is at \(sunrise) and sunset is at \(sunset)")
    case let .failure(message):
        print("Failure... \(message)")
}

```

### 练习

给 `ServerResponse` 和 `switch` 添加第三种情况。

注意日升和日落时间是如何从 `ServerResponse` 中提取到并与 `switch` 的 `case` 相匹配的。

使用 `struct` 来创建一个结构体。结构体和类有很多相同的地方，包括方法和构造器。它们之间最大的一个区别就是结构体是传值，类是传引用。

```
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()
```

### 练习

给 `Card` 添加一个方法，创建一副完整的扑克牌并把每张牌的 `rank` 和 `suit` 对应起来。

## 协议和扩展

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遵循协议。

```
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()  
let aDescription = a.simpleDescription
```

```
struct SimpleStructure: ExampleProtocol {  
    var simpleDescription: String = "A simple structure"  
    mutating func adjust() {  
        simpleDescription += " (adjusted)"  
    }  
}  
var b = SimpleStructure()  
b.adjust()  
let bDescription = b.simpleDescription
```

## 练习

写一个实现这个协议的枚举。

注意声明 `SimpleStructure` 时候 `mutating` 关键字用来标记一个会修改结构体的方法。`SimpleClass` 的声明不需要标记任何方法，因为类中的方法通常可以修改类属性（类的性质）。

使用 `extension` 来为现有的类型添加功能，比如新的方法和计算属性。你可以使用扩展让某个在别处声明的类型来遵守某个协议，这同样适用于从外部库或者框架引入的类型。

```
extension Int: ExampleProtocol {  
    var simpleDescription: String {  
        return "The number \(self)"  
    }  
    mutating func adjust() {  
        self += 42  
    }  
}  
print(7.simpleDescription)
```

## 练习

给 `Double` 类型写一个扩展，添加 `absoluteValue` 属性。

你可以像使用其他命名类型一样使用协议名——例如，创建一个有不同类型但是都实现一个协议的对象集合。当你处理类型是协议的值时，协议外定义的方法不可用。

```
let protocolValue: ExampleProtocol = a  
print(protocolValue.simpleDescription)  
// print(protocolValue.anotherProperty) // 去掉注释可以看到错误
```

即使 `protocolValue` 变量运行时的类型是 `simpleClass`，编译器还是会把它的类型当做 `ExampleProtocol`。这表示你不能调用在协议之外的方法或者属性。

## 错误处理

使用采用 `Error` 协议的类型来表示错误。

```
enum PrinterError: Error {  
    case outOfPaper  
    case noToner  
    case onFire  
}
```

使用 `throw` 来抛出一个错误和使用 `throws` 来表示一个可以抛出错误的函数。如果在函数中抛出一个错误，这个函数会立刻返回并且调用该函数的代码会进行错误处理。

```
func send(job: Int, toPrinter printerName: String) throws -> String {
    if printerName == "Never Has Toner" {
        throw PrinterError.noToner
    }
    return "Job sent"
}
```

有多种方式可以用来进行错误处理。一种方式是使用 `do-catch`。在 `do` 代码块中，使用 `try` 来标记可以抛出错误的代码。在 `catch` 代码块中，除非你另外命名，否则错误会自动命名为 `error`。

```
do {
    let printerResponse = try send(job: 1040, toPrinter: "Bi Sheng")
    print(printerResponse)
} catch {
    print(error)
}
```

### 练习

将 printer name 改为 `"Never Has Toner"` 使 `send(job:toPrinter:)` 函数抛出错误。

可以使用多个 `catch` 块来处理特定的错误。参照 switch 中的 `case` 风格来写 `catch`。

```
do {
    let printerResponse = try send(job: 1440, toPrinter: "Gutenberg")
    print(printerResponse)
} catch PrinterError.onFire {
    print("I'll just put this over here, with the rest of the fire.")
} catch let printerError as PrinterError {
    print("Printer error: \(printerError).")
} catch {
    print(error)
}
```

### 练习

在 `do` 代码块中添加抛出错误的代码。你需要抛出哪种错误来使第一个 `catch` 块进行接收？怎么使第二个和第三个 `catch` 进行接收呢？

另一种处理错误的方式使用 `try?` 将结果转换为可选的。如果函数抛出错误，该错误会被抛弃并且结果为 `nil`。否则，结果会是一个包含函数返回值的可选值。

```
let printerSuccess = try? send(job: 1884, toPrinter: "Mergenthaler")
let printerFailure = try? send(job: 1885, toPrinter: "Never Has Toner")
```

使用 `defer` 代码块来表示在函数返回前，函数中最后执行的代码。无论函数是否会抛出错误，这段代码都将执行。使用 `defer`，可以把函数调用之初就要执行的代码和函数调用结束时的扫尾代码写在一起，虽然这两者的执行时机截然不同。

```

var fridgelsOpen = false
let fridgeContent = ["milk", "eggs", "leftovers"]

func fridgeContains(_ food: String) -> Bool {
    fridgelsOpen = true
    defer {
        fridgelsOpen = false
    }

    let result = fridgeContent.contains(food)
    return result
}
fridgeContains("banana")
print(fridgelsOpen)

```

## 泛型

---

在尖括号里写一个名字来创建一个泛型函数或者类型。

```

func makeArray<Item>(repeating item: Item, numberOfTimes: Int) -> [Item] {
    var result = [Item]()
    for _ in 0..<numberOfTimes {
        result.append(item)
    }
    return result
}
makeArray(repeating: "knock", numberOfTimes: 4)

```

你也可以创建泛型函数、方法、类、枚举和结构体。

```

// 重新实现 Swift 标准库中的可选类型
enum OptionalValue<Wrapped> {
    case none
    case some(Wrapped)
}
var possibleInteger: OptionalValue<Int> = .none
possibleInteger = .some(100)

```

在类型名后面使用 `where` 来指定对类型的一系列需求，比如，限定类型实现某一个协议，限定两个类型是相同的，或者限定某个类必须有一个特定的父类。

```

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

```

## 练习

修改 `anyCommonElements(_:_)` 函数来创建一个函数，返回一个数组，内容是两个序列的共有元素。

`<T: Equatable>` 和 `<T> ... where T: Equatable>` 的写法是等价的。

# Swift 版本历史记录 · GitBook

---

 [runoob.com/manual/gitbook/swift5/source/\\_book/chapter1/04\\_revision\\_history.html](https://runoob.com/manual/gitbook/swift5/source/_book/chapter1/04_revision_history.html)

## Swift 文档修订历史

---

### 2019-03-25

---

- 更新至 Swift 5。
- 新增 拓展字符串分隔符 章节。更新 字符串字面量 章节，拓展有关字符串分隔符的内容。
- 新增 动态调用 章节，其中包含使用 `dynamicCallable` 属性动态调用实例作为函数的内容。
- 新增 unknown 和 未来枚举匹配 章节，其中包含了使用 `unknown` 来处理未来枚举可能发生改变的情形。
- 在 Key-Path 表达式 章节新增了有关标示 key path (`\.self`) 的内容。
- 在 可选编译块 章节新增了有关小于比较符 `<` 的内容。

### 2018-09-17

---

- 更新至 Swift 4.2。
- 在 遍历枚举情形 章节新增了有关访问所有枚举情形的内容。
- 在 编译诊断 章节新增了有关 `#error` 和 `#warning` 的内容。
- 在 属性声明 章节中新增了有关 `inlinable` 和 `usableFromInline` 属性的内容。
- 在 属性声明 章节中新增了有关 `requires_stored_property_inits` 和 `warn_unqualified_access` 属性的内容。
- 在 可选编译块 章节新增了有关如何根据 Swift 编译器版本对代码进行对应编译处理的内容。
- 在 字面量语法 章节新增了有关 `#dsohandle` 的内容。

### 2018-03-29

---

- 更新至 Swift 4.1。
- 在 等价运算符 章节新增了有关等价运算符的合成实现的内容。
- 在 声明 篇章中 申明拓展 章节和 协议 篇章中 有条件地遵循协议 章节新增了有关协议有条件遵循的内容。
- 在 关联类型约束中使用协议 章节中新增了有关递归协议约束的内容。
- 在 条件编译块 章节中新增了有关 `canImport()` 和 `targetEnvironment()` 平台条件的内容。

### 2017-12-04

---

- 更新至 Swift 4.0.3。
- 更新 Key-Path 表达式 章节，现在 key path 支持下标子路径。

### 2017-09-19

---

- 
- 更新至 Swift 4.0。
  - 在 内存安全 章节新增了有关内存互斥访问的内容。
  - 新增 带有泛型 Where 子句联类型 章节，现在可以使用泛型 `where` 子句约束关联类型。
  - 在 字符串和字符 篇章中 字面量 章节以及 词法结构 篇章的 字符串字面量 章节中新增了有关多行字符串字面量的内容。
  - 更新 声明属性 中 `objc` 属性的讨论，现在该属性会在更少的位置被推断出来。
  - 新增 范型下标 章节，现在下标也支持范型特性了。
  - 更新 协议 篇章中 协议组合 章节和 类型 篇章中 协议组合类型 章节的讨论，现在协议组合类型支持进行父类约束了。
  - 更新 拓展声明 中有关协议扩展的讨论，现在它们不支持 `final` 特性了。
  - 在 断言和前置条件 章节中新增了部分前置条件和致命错误的内容。

## 2017-03-27

---

- 更新至 Swift 3.1。
- 新增 范型 Where 子句扩展 章节，包含需要的扩展内容。
- 在 For-In 循环 章节中新增了区间迭代的例子。
- 在 到可失败构造器 章节中新增了可失败数值转换的例子。
- 在 声明特性 章节中新增了有关使用 Swift 语言版本的 `available` 特性的内容。
- 更新 函数类型 章节中的讨论，注意在写函数类型时不允许使用参数标签。
- 更新 条件编译块 章节中的 Swift 语言版本号的讨论，现在可以使用可选的补丁版本号。
- 更新 函数类型 章节的讨论，现在 Swift 区分了采用多参数的函数和采用元组类型的单个参数的函数。
- 在 表达式 篇章中删除了动态表达式的章节，现在 `type(of:)` 是 Swift 标准库函数。

## 2016-10-27

---

- 更新至 Swift 3.0.1。
- 更新 自动引用计数 章节中有关 `weak` 和 `unowned` 引用的讨论。
- 在 声明标识符 章节中新增了有关新的标识符 `unowned`，`unowend(safe)` 和 `unowned(unsafe)` 的内容。
- 在 Any 和 AnyObject 的类型转换 章节中新增了一条提示，有关使用类型 `Any` 作为可选值。
- 更新 表达式 章节，把括号表达式和元组表达式的描述分开。

## 2016-09-13

---

- 更新至 Swift 3.0。
- 更新 函数 篇章和 函数声明 章节中有关函数的讨论，所有函数参数默认都有函数标签。
- 更新 高级操作符 篇章中有关操作符的讨论，现在你可以作为类型函数来实现，替代之前的全局函数实现方式。

- 在 访问控制 章节中新增有关对新的访问级别描述符 `open` 和 `fileprivate` 的内容。
- 更新 函数声明 章节中有关 `inout` 的讨论，注意它现在出现在参数类型的前面，而不是在参数名称的前面。
- 更新 逃逸闭包 和 自动闭包 章节还有 属性 篇章中有关 `@noescape` 和 `@autoclosure` 的讨论，现在他们是类型属性，而不是定义属性。
- 在 高级操作符 篇章中 自定义中缀操作符的优先级 章节和 定义 篇章中 优先级组声明 章节中新增了有关操作符优先级组的内容。
- 更新一些讨论，使用 `macOS` 替换掉 `OS X`，`Error` 替换掉 `ErrorProtocol`。更新一些协议名称，比如使用 `ExpressibleByStringLiteral` 替换掉 `StringLiteralConvertible`。
- 更新 泛型 篇章中 泛型 Where 语句 章节和 泛型形参和实参 篇章的讨论，现在泛型的 `where` 语句写在一个声明的最后。
- 更新 逃逸闭包 章节中的讨论，现在闭包默认为非逃逸的。
- 更新 基础部分 篇章中 可选绑定 章节和 语句 篇章中 While 语句 章节中的讨论，现在 `if`, `while` 和 `guard` 语句使用逗号分隔条件列表，不需要使用 `where` 语句。
- 在 控制流 篇章中 Switch 章节和 语句 篇章中 Switch 语句 章节中新增了 `switch cases` 可以使用多模式的内容。
- 更新 函数类型 章节有关现在函数参数标签不包含在函数类型中的讨论。
- 更新 协议 篇章中 协议组合 章节和 类型 篇章中 协议组合类型 章节中有关使用新的 `Protocol1 & Protocol2` 语法的内容。
- 更新 动态类型表达式 章节中使用新的 `type(of:)` 表达式的讨论。
- 更新 行控制表达式 章节中使用 `#sourceLocation(file:line:)` 表达式的讨论。
- 更新 永不返回函数 章节中使用新的 `Never` 类型的讨论。
- 在 字面量表达式 章节中新增了有关 `playground` 字面量的内容。
- 更新 In-Out 参数 章节，标明只有非逃逸闭包能捕获 `in-out` 参数。
- 更新 默认参数值 章节，现在默认参数不能在调用时候重新排序。
- 更新 属性 篇章中有关属性参数使用分号的说明。
- 在 重新抛出函数和方法 章节中新增了有关在 `catch` 代码块中抛出错误的重新抛出函数的内容。
- 在 Selector 表达式 章节中新增了中有关访问 Objective-C 中 `Selector` 的 `getter` 和 `setter` 的内容。
- 在 类型别名声明 章节中中新增了有关泛型类型别名和在协议内使用类型别名的内容。
- 更新 函数类型 章节中有关函数类型的讨论，标明函数类型作为参数类型必须使用括号包裹。
- 更新 属性 篇章，标明 `@IBAction` , `@IBOutlet` 和 `@NSManaged` 隐式含有 `@objc` 属性。
- 在 声明属性 章节中新增了有关 `@GKInspectable` 的内容。
- 更新 可选协议要求 章节中有关只能在与 `Objective-C` 交互的代码中才能使用可选协议要求的内容。
- 删除 函数声明 章节中有关显式使用 `let` 关键字作为函数参数的内容。
- 删除 语句 章节中有关 `Boolean` 协议的内容，现在这个协议已经被 Swift 标准库删除。

- 更正 声明属性 章节中有关 `@NSApplicationMain` 协议的内容。

## 2016-03-21

---

- 更新至 Swift 2.2。
- 在 编译配置语句 章节新增了中有关如何根据 Swift 版本进行条件编译。
- 在 显示成员表达式 章节中新增了有关如何区分只有参数名不同的方法和构造器的内容。
- 在 选择器表达式 章节中新增了针对 Objective-C 选择器的 `#selector` 语法。
- 更新 关联类型 和 协议关联类型声明 章节中有关使用 `associatedtype` 关键词修饰关联类型的讨论。
- 更新 可失败构造器 章节中有关当构造器在实例完全初始化之前返回 `nil` 的相关内容。
- 在 比较运算符 章节中新增了比较元组的内容。
- 在 关键字和标点符号 章节中新增了使用关键字作为外部参数名的内容。
- 更新 声明特性 章节中有关 `@objc` 特性的讨论，并指出枚举和枚举用例。
- 更新 操作符 章节中对于自定义运算符的包含了 `.` 的讨论。
- 在 重新抛出错误的函数和方法 章节中新增了一条提示，重新抛出错误函数不能直接抛出错误。
- 在 属性观察器 章节中新增了一条提示，当作为 in-out 参数传递属性时，属性观察器的调用行为。
- 在 Swift 初见 篇章中新增了错误处理的章节。
- 更新 弱引用 章节中的图片用以更清楚的展示重新分配过程。
- 删除 C 语言风格的 `for` 循环，`++` 前缀和后缀运算符，以及 `--` 前缀和后缀运算符。
- 删除对变量函数参数和柯里化函数的特殊语法的讨论。

## 2015-10-20

---

- 更新至 Swift 2.1。
- 更新 字符串插值 和 字符串字面量 章节，现在字符串插值可包含字符串字面量。
- 在 逃逸闭包 章节中新增了有关 `@noescape` 属性的相关内容。
- 更新 声明特性 和 编译配置语句 章节中与 tvOS 相关的内容。
- 在 In-Out 参数 章节中新增了与 in-out 参数行为相关的内容。
- 在 捕获列表 章节新增了有关指定闭包捕获列表被捕获时捕获值的相关内容。
- 更新 可选链式调用访问属性 章节，阐明了如何通过可选链式调用进行赋值。
- 改进 自动闭包 章节中对自闭包的讨论。
- 在 Swift 初见 篇章中新增了一个使用 `??` 操作符的例子。

## 2015-09-16

---

- 更新至 Swift 2.0。
- 在 错误处理 篇章中新增了有关错误处理的相关内容，包括 Do 语句、Throw 语句、Defer 语句 以及 try 运算符。
- 更新 错误表示和抛出 章节，现在所有类型都可以遵循 `ErrorType` 协议了。
- 在 将错误装换成可选值 篇章增加了 `try?` 关键字相关内容。

- 在 枚举 篇章的 递归枚举 章节以及 声明 篇章的 任意类型用例 的 枚举 章节中新增了递归枚举相关内容。
- 在 控制流 篇章的 API 可用性检查 章节和 语句 篇章的 可用性条件 章节中新增了有关 API 可用性检查相关的内容。
- 在 控制流 篇章的 尽早退出 章节和 语句 篇章的 Guard 语句 章节新增了与 guard 语句相关的内容。
- 在 协议 篇章中 协议扩展 章节中新增了有关协议扩展的内容。
- 在 访问控制 篇章的 单元测试 target 的访问级别 章节中新增了有关单元测试访问控制相关的内容。
- 在 模式 篇章的 可选模式 章节中新增了可选模式相关内容。
- 更新 Repeat-While 章节中有关 repeat-while 循环相关的内容。
- 更新 字符串和字符 章节，现在 String 类型在 Swift 标准库中不再遵循 CollectionType 协议。
- 在 常量与变量打印 章节中新增了新 Swift 标准库中有关 print(\_:separator:terminator) 相关内容。
- 在 枚举 篇章的 原始值的隐式赋值 章节和 声明 篇章的 包含原始值类型的枚举 章节中新增了有关包含 String 原始值的枚举用例的行为相关内容。
- 在 自动闭包 章节中新增了有关 @autoclosure 特性的相关内容，包括它的 @autoclosure(escaping) 形式。
- 更新 声明特性 章节中有关 @available 和 warn\_unused\_result 特性的相关内容。
- 更新 类型特性 章节中有关 @convention 特性的相关内容。
- 在 可选绑定 章节中新增了有关使用 where 子句进行多可选绑定的相关内容。
- 在 字符串字面量 章节中新增了有关在编译时使用 + 运算符拼接字符串字面量的相关内容。
- 在 元类型 章节中新增了有关元类型值的比较和使用它们通过构造器表达式构造实例相关内容。
- 在 断言调试 章节中新增了一条提示，有关用户定义断言何时会失效。
- 更新 声明特性 章节中对 @NSManaged 特性的讨论，现在这个特性可以被应用到一个确定实例方法。
- 更新 可变参数 章节，现在可变参数可以声明在函数参数列表的任意位置中。
- 在 重写可失败构造器 章节中新增了有关非可失败构造器相当于一个可失败构造器通过父类构造器的结果进行强制拆包的相关内容。
- 在 任意类型用例 的 枚举 章节中新增了有关枚举用例作为函数的内容。
- 在 构造器表达式 章节中新增了有关显式引用一个构造器相关内容。
- 在 编译控制语句 章节中新增了有关编译内容以及行控制语句相关内容。
- 在 元类型 章节新增了一条提示，有关如何从元类型值中构造类实例相关内容。
- 在 弱引用 章节新增了一条提示，有关弱引用作为缓存所存在的不足。
- 更新 类型特性 章节，提到了存储型特性其实是懒加载。
- 更新 捕获类型 章节，阐明了变量和常量在闭包中如何被捕获。
- 更新 声明特性 章节，用以描述何时在类中使用 @objc 关键字。
- 在 错误处理 章节中新增了一条提示，有关执行 throw 语句的性能。在 Do 语句 章节的 do 语句部分也新增了类似内容。
- 更新 类型特性 章节中有关类、结构体和枚举的存储型和计算型特性相关的内容。

- 更新 [Break 语句](#) 章节中有关带标签的 break 语句相关内容。
- 在 [属性观察器](#) 章节更新了一处提示，用来明确 `willSet` 和  `didSet` 观察器的行为。
- 在 [访问级别](#) 章节新增了有关 `private` 作用域的相关内容说明。
- 在 [弱引用](#) 章节新增了有关弱应用在垃圾回收系统和 ARC 之间的区别的说明。
- 更新 [字符串字面量中特殊字符](#) 章节，对 Unicode 标量更精确定义。

## 2015-4-8

---

- 更新至 Swift 1.2。
- Swift 现在自身提供了一个 `Set` 集合类型，更多内容，请看 [Sets](#)。
- `@autoclosure` 现在是一个参数声明的属性，而不是参数类型的属性。这里还有一个新的参数声明属性 `@noescape`。更多内容，请看 [属性声明](#)。
- 对于类型属性和方法现在可以使用 `static` 关键字作为声明描述符，更多内容，请看 [类型变量属性](#)。
- Swift 现在包含一个 `as?` 和 `as!` 的向下可失败类型转换运算符。更多内容，请看 [协议遵循性检查](#)。
- 新增 [字符串索引](#) 的新指导章节。
- 在 [溢出运算符](#) 一节中删除了溢出除运算符（`&/`）和求余溢出运算符（`&%`）。
- 更新常量和常量属性在声明和构造时的规则，更多内容，请看 [常量声明](#)。
- 更新 [字符串字面量中 Unicode 标量集的定义](#)，请看 [字符串字面量中的特殊字符](#)。
- 更新 [区间运算符](#) 章节，注意当半开区间运算符含有相同的起止索引时，其区间为空。
- 更新 [闭包引用类型](#) 章节，对于变量的捕获规则进行了阐明。
- 更新 [值溢出](#) 章节，对有符号整数和无符号整数的溢出行为进行了阐明。
- 更新 [协议声明](#) 章节，对协议声明时的作用域和成员等内容进行了阐明。
- 更新 [捕获列表](#) 章节，对于闭包捕获列表中的弱引用和无主引用的使用语法进行了阐明。
- 更新 [运算符](#) 章节，明确指明一些例子来说明自定义运算符所支持的特性，如数学运算符，各种符号，Unicode 符号块等。
- 在函数作用域中的常量声明时可以不被初始化，它必须在第一次使用前被赋值。更多的内容，请看 [常量声明](#)。
- 在构造器中，常量属性有且仅能被赋值一次。更多内容，请看 [在构造过程中给常量属性赋值](#)。
- 多个可选绑定现在可以在 `if` 语句后面以逗号分隔的赋值列表的方式出现，更多内容，请看 [可选绑定](#)。
- 一个 [可选链表达式](#) 必须出现在后缀表达式中。
- 协议类型转换不再局限于 `@objc` 修饰的协议了。
- 在运行时可能会失败的类型转换可以使用 `as?` 和 `as!` 运算符，而确保不会失败的类型转换现在使用 `as` 运算符。更多内容，请看 [类型转换运算符](#)。

## 2014-10-16

---

- 更新至 Swift 1.1。
- 新增 [失败构造器](#) 的完整指引。

- 在协议中新增了 失败构造器要求 的描述。
- 常量和变量的 `Any` 类型现可以包含函数实例。更新了有关 `Any` 相关的示例来展示如何在 `switch` 语句中如何检查并转换到一个函数类型。
- 带有原始值的枚举类型增加了一个 `rawValue` 属性替代 `toRaw()` 方法，同时使用了一个以 `rawValue` 为参数的失败构造器来替代 `fromRaw()` 方法。更多的内容，请看 原始值 和 带原始值的枚举类型。
- 新增 Failable Initializer 的参考章节，它可以触发初始化失败。
- 自定义运算符现在可以包含 `?` 字符，更新了 运算符 涉及改进后的规则的部分，并且在 自定义运算符 章节中删除了重复的运算符有效字符集合。

## 2014-08-18

---

- 描述 Swift 1.0 的新文档。Swift 是苹果公司发布的全新编程语言，用于 iOS 和 OS X 应用开发。
- 在协议中新增了 对构造器的规定 章节。
- 新增 类专属协议 章节。
- 断言 现在可以使用字符串内插语法，并删除了文档中有冲突的注释。
- 更新 连接字符串和字符 章节来说明字符串和字符不能再用 `+` 号运算符或者复合加法运算符 `+=` 相互连接，这两种运算符现在只能用于字符串之间相连。请使用 `String` 类型的 `append` 方法在一个字符串的尾部增加单个字符。
- 在 属性申明 章节增加了有关 `availability` 特性的一些内容。
- 可选类型 若有值时，不再隐式的转换为 `true`，同样，若无值时，也不再隐式的转换为 `false`，这是为了避免在判别 optional `Bool` 的值时产生困惑。替代的方案是，用 `==` 或 `!=` 运算符显式地去判断 Optional 是否是 `nil`，以确认其是否包含值。
- Swift 新增了一个 Nil 合并运算符 (`a ?? b`)，该表达式中，如果 Optional `a` 的值存在，则取得它并返回，若 Optional `a` 为 `nil`，则返回默认值 `b`
- 更新和扩展 字符串的比较，用以反映和展示'字符串和字符的比较'，以及'前缀 (prefix) /后缀 (postfix) 比较'都开始基于扩展字符集 (extended grapheme clusters) 规范的等价比较。
- 现在，你可以通过下标赋值或者 可选调用链 中的可变方法和操作符来给属性设值。相应地更新了有关 通过可选链接访问属性 的内容，并扩展了 通过可选链接调用方法 时检查方法调用成功的示例，以显示如何检查属性设置是否成功。
- 在可选链中新增了 访问可选类型的下标脚注 章节。
- 更新 访问和修改数组 章节以标示，从该版本起，不能再通过 `+=` 运算符给一个数组新增一个新的项。对应的替代方案是，使 `append` 方法，或者通过 `+=` 运算符来新增一个只有一个项的数组。
- 新增一条提示，在 范围运算符 中，比如，`a..b` 和 `a..<b`，起始值 `a` 不能大于结束值 `b`。
- 重写继承 篇章：删除了本章中有关构造器重写的介绍性报道；转而将更多的注意力放到新增的部分——子类的新功能，以及如何通过重写 (overrides) 修改已有的功能。另外，重写属性的 Getters 和 Setters 中的例子已经被替换为展示如何重写一个 `description` 属性。(而有关如何在子类的构造器中修改继承属性的默认值的例子，已经被移到 构造过程 篇章。)

- 更新 构造器的继承与重写 章节以标示：重写一个特定的构造器必须使用 `override` 修饰符。
- 更新 Required 构造器 章节以标示：`required` 修饰符现在需要出现在所有子类的 `required` 构造器的声明中，而 `required` 构造器的实现，现在可以仅从父类自动继承。
- 中置（Infix）的 运算符函数 不再需要 `@infix` 属性。
- 前置和后置运算符 的 `@prefix` 和 `@postfix` 属性，已变更为 `prefix` 和 `postfix` 声明修饰符。
- 新增一条提示，在 `Prefix` 和 `postfix` 运算符被作用于同一个操作数时 前置和后置运算符的执行顺序。
- 组合赋值运算符 的运算符函数不再使用 `@assignment` 属性来定义函数。
- 在定义 自定义操作符 时，修饰符（Modifiers）的出现顺序发生变化。比如现在，你该编写 `prefix operator`，而不是 `operator prefix`。
- 在 声明修饰符 章节新增了有关 `dynamic` 声明修饰符的内容。
- 新增有关 字面量 类型推导内容的内容。
- 新增更多有关柯里化函数的内容。
- 新增 权限控制 篇章。
- 更新 字符串和字符 章节，在 Swift 中现在 `Character` 类型代表的是扩展字符集（extended grapheme cluster）中的一个 Unicode，为此，新增了 Extended Grapheme Clusters 章节。同时，Unicode 标量 和 字符串比较 章节新增了更多内容。
- 更新 字符串字面量 章节，在一个字符串中，Unicode 标量（Unicode scalars）以 `\u{n}` 的形式来表示，`n` 是一个最大可以有8位的16进制数。
- `NSString length` 属性已被映射到 Swift 的内建 `String` 类型。（注意，这两属性的类型是 `utf16Count`，而非 `utf16count`）。
- Swift 的内建 `String` 类型不再拥有 `uppercaseString` 和 `lowercaseString` 属性。在 字符串和字符 章节中删除了对应部分，并更新了各种对应的代码用例。
- 新增 没有外部名的构造器参数 章节。
- 新增 Required 构造器 章节。
- 新增 可选元组返回类型 章节。
- 更新 类型注解 章节，多个相关变量可以用“类型注解”在同一行中声明为同一类型。
- `@optional`，`@lazy`，`@final`，`@required` 等关键字被更新为 `optional`，`lazy`，`final`，`required` 参见 声明修饰符。
- 更新了整本书中有关 `.. 的引用，从半闭区间改为了 半开区间。`
- 更新 读取和修改字典 章节，`Dictionary` 现在增加了一个 Boolean 型的属性：`isEmpty`。
- 解释了哪些字符（集）可被用来定义 自定义操作符。
- `nil` 和布尔运算中的 `true` 和 `false` 现在被定义为 字面量。
- Swift 中的数组（`Array`）类型从现在起具备了完整的值语义。具体内容被更新到 集合的可变性 和 数组 两小节，以反映这个新的变化。此外，还解释了如何给 `Strings`，`Arrays` 和 `Dictionaries` 进行赋值和拷贝。
- 数组类型速记语法 从 `SomeType []` 更新为 `[SomeType]`。
- 新增 字典类型的速记语法 章节，现在书写格式为：`[KeyType: ValueType]`。

- 新增字典键类型的哈希值章节。
- 闭包表达式示例中使用新的全局函数 `sorted` 取代原先的全局函数 `sort` 去展示如何返回一个全新的数组。
- 更新结构体逐一成员构造器章节，即使结构体的成员 没有默认值，逐一成员构造器也可以自动获得。
- 半开区间运算符中 `..` 更新为 `..。`
- 新增泛型拓展的示例。

# 基础部分 · GitBook

---

 [runoob.com/manual/gitbook/swift5/source/\\_book/chapter2/01\\_The\\_Basics.html](https://runoob.com/manual/gitbook/swift5/source/_book/chapter2/01_The_Basics.html)

## 基础部分

---

Swift 是一门开发 iOS, macOS, watchOS 和 tvOS 应用的新语言。然而，如果你有 C 或者 Objective-C 开发经验的话，你会发现 Swift 的很多内容都是你熟悉的。

Swift 包含了 C 和 Objective-C 上所有基础数据类型，`Int` 表示整型值；`Double` 和 `Float` 表示浮点型值；`Bool` 是布尔型值；`String` 是文本型数据。Swift 还提供了三个基本的集合类型，`Array`、`Set` 和 `Dictionary`，详见 [集合类型](#)。

就像 C 语言一样，Swift 使用变量来进行存储并通过变量名来关联值。在 Swift 中，广泛的使用着值不可变的变量，它们就是常量，而且比 C 语言的常量更强大。在 Swift 中，如果你要处理的值不需要改变，那使用常量可以让你的代码更加安全并且更清晰地表达你的意图。

除了我们熟悉的类型，Swift 还增加了 Objective-C 中没有的高阶数据类型比如元组（Tuple）。元组可以让你创建或者传递一组数据，比如作为函数的返回值时，你可以用一个元组可以返回多个值。

Swift 还增加了可选（Optional）类型，用于处理值缺失的情况。可选表示“那儿有一个值，并且它等于 `x`”或者“那儿没有值”。可选有点像在 Objective-C 中使用 `nil`，但是它可以用在任何类型上，不仅仅是类。可选类型比 Objective-C 中的 `nil` 指针更加安全也更具表现力，它是 Swift 许多强大特性的重要组成部分。

Swift 是一门类型安全的语言，这意味着 Swift 可以让你清楚地知道值的类型。如果你的代码需要一个 `String`，类型安全会阻止你不小心传入一个 `Int`。同样的，如果你的代码需要一个 `String`，类型安全会阻止你意外传入一个可选的 `String`。类型安全可以帮助你在开发阶段尽早发现并修正错误。

## 常量和变量

---

常量和变量把一个名字（比如 `maximumNumberOfLoginAttempts` 或者 `welcomeMessage`）和一个指定类型的值（比如数字 `10` 或者字符串 `"Hello"`）关联起来。常量的值一旦设定就不能改变，而变量的值可以随意更改。

## 声明常量和变量

---

常量和变量必须在使用前声明，用 `let` 来声明常量，用 `var` 来声明变量。下面的例子展示了如何用常量和变量来记录用户尝试登录的次数：

```
let maximumNumberOfLoginAttempts = 10
var currentLoginAttempt = 0
```

这两行代码可以被理解为：

“声明一个名字是 `maximumNumberOfLoginAttempts` 的新常量，并给它一个值 `10`。然后，声明一个名字是 `currentLoginAttempt` 的变量并将它的值初始化为 `0`。”

在这个例子中，允许的最大尝试登录次数被声明为一个常量，因为这个值不会改变。当前尝试登录次数被声明为一个变量，因为每次尝试登录失败的时候都需要增加这个值。

你可以在一行中声明多个常量或者多个变量，用逗号隔开：

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

### 注意

如果你的代码中有不需要改变的值，请使用 `let` 关键字将它声明为常量。只将需要改变的值声明为变量。

## 类型注解

当你声明常量或者变量的时候可以加上 **类型注解** (*type annotation*)，说明常量或者变量中要存储的值的类型。如果要添加类型注解，需要在常量或者变量名后面加上一个冒号和空格，然后加上类型名称。

这个例子给 `welcomeMessage` 变量添加了类型注解，表示这个变量可以存储 `String` 类型的值：

```
var welcomeMessage: String
```

声明中的冒号代表着“**是...类型**”，所以这行代码可以被理解为：

“声明一个类型为 `String`，名字为 `welcomeMessage` 的变量。”

“类型为 `String`”的意思是“可以存储任意 `String` 类型的值。”

`welcomeMessage` 变量现在可以被设置成任意字符串：

```
welcomeMessage = "Hello"
```

你可以在一行中定义多个同样类型的变量，用逗号分割，并在最后一个变量名之后添加类型注解：

```
var red, green, blue: Double
```

### 注意

一般来说你很少需要写类型注解。如果你在声明常量或者变量的时候赋了一个初始值，Swift 可以推断出这个常量或者变量的类型，请参考 [类型安全和类型推断](#)。在上面的例子中，没有给 `welcomeMessage` 赋初始值，所以变量 `welcomeMessage` 的类型是通过一个类型注解指定的，而不是通过初始值推断的。

## 常量和变量的命名

常量和变量名可以包含任何字符，包括 Unicode 字符：

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

常量与变量名不能包含数学符号，箭头，保留的（或者非法的）Unicode 码位，连线与制表符。也不能以数字开头，但是可以在常量与变量名的其他地方包含数字。

一旦你将常量或者变量声明为确定的类型，你就不能使用相同的名字再次进行声明，或者改变其存储的值的类型。同时，你也不能将常量与变量进行互转。

### 注意

如果你需要使用与 Swift 保留关键字相同的名称作为常量或者变量名，你可以使用反引号 (`) 将关键字包围的方式将其作为名字使用。无论如何，你应当避免使用关键字作为常量或变量名，除非你别无选择。

你可以更改现有的变量值为其他同类型的值，在下面的例子中，`friendlyWelcome` 的值从 `"Hello!"` 改为了 `"Bonjour!"`：

```
var friendlyWelcome = "Hello!"
friendlyWelcome = "Bonjour!"
// friendlyWelcome 现在是 "Bonjour!"
```

与变量不同，常量的值一旦被确定就不能更改了。尝试这样做会导致编译时报错：

```
let languageName = "Swift"
languageName = "Swift++"
// 这会报编译时错误 - languageName 不可改变
```

## 输出常量和变量

你可以用 `print(_:separator:terminator:)` 函数来输出当前常量或变量的值：

```
print(friendlyWelcome)
// 输出“Bonjour!”
```

`print(_:separator:terminator:)` 是一个用来输出一个或多个值到适当输出区的全局函数。如果你用 Xcode，`print(_:separator:terminator:)` 将会输出内容到“console”面板上。`separator` 和 `terminator` 参数具有默认值，因此你调用这个函数的时候可以忽略它们。默认情况下，该函数通过添加换行符来结束当前行。如果不想换行，可以传递一个空字符串给 `terminator` 参数--例如，`print(someValue, terminator:"")`。关于参数默认值的更多信息，请参考 [默认参数值](#)。

Swift 用字符串插值 (*string interpolation*) 的方式把常量名或者变量名当做占位符加入到长字符串中，Swift 会用当前常量或变量的值替换这些占位符。将常量或变量名放入圆括号中，并在开括号前使用反斜杠将其转义：

```
print("The current value of friendlyWelcome is \(friendlyWelcome)")
// 输出“The current value of friendlyWelcome is Bonjour!”
```

## 注意

字符串插值所有可用的选项，请参考 [字符串插值](#)。

## 注释

请将你的代码中的非执行文本注释成提示或者笔记以方便你将来阅读。Swift 的编译器将会在编译代码时自动忽略掉注释部分。

Swift 中的注释与 C 语言的注释非常相似。单行注释以双正斜杠（`//`）作为起始标记：

```
// 这是一个注释
```

你也可以进行多行注释，其起始标记为单个正斜杠后跟随一个星号（`/*`），终止标记为一个星号后跟随单个正斜杠（`*/`）：

```
/* 这也是一个注释，  
但是是多行的 */
```

与 C 语言多行注释不同，Swift 的多行注释可以嵌套在其它的多行注释之中。你可以先生成一个多行注释块，然后在这个注释块之中再嵌套成第二个多行注释。终止注释时先插入第二个注释块的终止标记，然后再插入第一个注释块的终止标记：

```
/* 这是第一个多行注释的开头  
/* 这是第二个被嵌套的多行注释 */  
这是第一个多行注释的结尾 */
```

通过运用嵌套多行注释，你可以快速方便的注释掉一大段代码，即使这段代码之中已经有了多行注释块。

## 分号

与其他大部分编程语言不同，Swift 并不强制要求你在每条语句的结尾处使用分号（`;`），当然，你也可以按照你自己的习惯添加分号。有一种情况下必须用分号，即你打算在同一行内写多条独立的语句：

```
let cat = "🐱"; print(cat)  
// 输出“🐱”
```

## 整数

整数就是没有小数部分的数字，比如 `42` 和 `-23`。整数可以是 **有符号**（正、负、零）或者 **无符号**（正、零）。

Swift 提供了 8、16、32 和 64 位的有符号和无符号整数类型。这些整数类型和 C 语言的命名方式很像，比如 8 位无符号整数类型是 `UInt8`，32 位有符号整数类型是 `Int32`。就像 Swift 的其他类型一样，整数类型采用大写命名法。

## 整数范围

你可以访问不同整数类型的 `min` 和 `max` 属性来获取对应类型的最小值和最大值：

```
let minValue = UInt8.min // minValue 为 0, 是 UInt8 类型  
let maxValue = UInt8.max // maxValue 为 255, 是 UInt8 类型
```

`min` 和 `max` 所传回值的类型，正是其所对的整数类型（如上例 `UInt8`, 所传回的类型是 `UInt8`），可用在表达式中相同类型值旁。

## Int

---

一般来说，你不需要专门指定整数的长度。Swift 提供了一个特殊的整数类型 `Int`，长度与当前平台的原生字长相同：

- 在32位平台上，`Int` 和 `Int32` 长度相同。
- 在64位平台上，`Int` 和 `Int64` 长度相同。

除非你需要特定长度的整数，一般来说使用 `Int` 就够了。这可以提高代码一致性和可复用性。即使是在32位平台上，`Int` 可以存储的整数范围也可以达到 `-2,147,483,648 ~ 2,147,483,647`，大多数时候这已经足够大了。

## UInt

---

Swift 也提供了一个特殊的无符号类型 `UInt`，长度与当前平台的原生字长相同：

- 在32位平台上，`UInt` 和 `UInt32` 长度相同。
- 在64位平台上，`UInt` 和 `UInt64` 长度相同。

### 注意

尽量不要使用 `UInt`，除非你真的需要存储一个和当前平台原生字长相同的无符号整数。除了这种情况，最好使用 `Int`，即使你要存储的值已知是非负的。统一使用 `Int` 可以提高代码的可复用性，避免不同类型数字之间的转换，并且匹配数字的类型推断，请参考 [类型安全和类型推断](#)。

## 浮点数

---

浮点数是有小数部分的数字，比如 `3.14159`、`0.1` 和 `-273.15`。

浮点类型比整数类型表示的范围更大，可以存储比 `Int` 类型更大或者更小的数字。Swift 提供了两种有符号浮点数类型：

- `Double` 表示64位浮点数。当你需要存储很大或者很高精度的浮点数时请使用此类型。
- `Float` 表示32位浮点数。精度要求不高的话可以使用此类型。

## 注意

`Double` 精确度很高，至少有15位数字，而 `Float` 只有6位数字。选择哪个类型取决于你的代码需要处理的值的范围，在两种类型都匹配的情况下，将优先选择 `Double`。

## 类型安全和类型推断

Swift 是一个类型安全 (*type safe*) 的语言。类型安全的语言可以让你清楚地知道代码要处理的值的类型。如果你的代码需要一个 `String`，你绝对不可能不小心传进去一个 `Int`。

由于 Swift 是类型安全的，所以它会在编译你的代码时进行类型检查 (*type checks*)，并把不匹配的类型标记为错误。这可以让你在开发的时候尽早发现并修复错误。

当你要处理不同类型的值时，类型检查可以帮你避免错误。然而，这并不是说你每次声明常量和变量的时候都需要显式指定类型。如果你没有显式指定类型，Swift 会使用类型推断 (*type inference*) 来选择合适的类型。有了类型推断，编译器可以在编译代码的时候自动推断出表达式的类型。原理很简单，只要检查你赋的值即可。

因为有类型推断，和 C 或者 Objective-C 比起来 Swift 很少需要声明类型。常量和变量虽然需要明确类型，但是大部分工作并不需要你自己来完成。

当你声明常量或者变量并赋初值的时候类型推断非常有用。当你在声明常量或者变量的时候赋给它们一个字面量 (literal value 或 literal) 即可触发类型推断。（字面量就是会直接出现在你代码中的值，比如 `42` 和 `3.14159`。）

例如，如果你给一个新常量赋值 `42` 并且没有标明类型，Swift 可以推断出常量类型是 `Int`，因为你给它赋的初始值看起来像一个整数：

```
let meaningOfLife = 42
// meaningOfLife 会被推断为 Int 类型
```

同理，如果你没有给浮点字面量标明类型，Swift 会推断你想要的是 `Double`：

```
let pi = 3.14159
// pi 会被推断为 Double 类型
```

当推断浮点数的类型时，Swift 总是会选择 `Double` 而不是 `Float`。

如果表达式中同时出现了整数和浮点数，会被推断为 `Double` 类型：

```
let anotherPi = 3 + 0.14159
// anotherPi 会被推断为 Double 类型
```

原始值 `3` 没有显式声明类型，而表达式中出现了一个浮点字面量，所以表达式会被推断为 `Double` 类型。

## 数值型字面量

整数字面量可以被写作：

- 一个十进制数，没有前缀
- 一个二进制数，前缀是 `0b`
- 一个八进制数，前缀是 `0o`
- 一个十六进制数，前缀是 `0x`

下面的所有整数字面量的十进制值都是 `17`：

```
let decimalInteger = 17
let binaryInteger = 0b10001      // 二进制的17
let octalInteger = 0o21         // 八进制的17
let hexadecimalInteger = 0x11    // 十六进制的17
```

浮点字面量可以是十进制（没有前缀）或者是十六进制（前缀是 `0x`）。小数点两边必须有至少一个十进制数字（或者是十六进制的数字）。十进制浮点数也可以有一个可选的指数（exponent），通过大写或者小写的 `e` 来指定；十六进制浮点数必须有一个指数，通过大写或者小写的 `p` 来指定。

如果一个十进制数的指数为 `exp`，那这个数相当于基数和 $10^{\text{exp}}$  的乘积：

- `1.25e2` 表示  $1.25 \times 10^2$ ，等于 `125.0`。
- `1.25e-2` 表示  $1.25 \times 10^{-2}$ ，等于 `0.0125`。

如果一个十六进制数的指数为 `exp`，那这个数相当于基数和 $2^{\text{exp}}$  的乘积：

- `0xFp2` 表示  $15 \times 2^2$ ，等于 `60.0`。
- `0xFp-2` 表示  $15 \times 2^{-2}$ ，等于 `3.75`。

下面的这些浮点字面量都等于十进制的 `12.1875`：

```
let decimalDouble = 12.1875
let exponentDouble = 1.21875e1
let hexadecimalDouble = 0xC.3p0
```

数值类字面量可以包括额外的格式来增强可读性。整数和浮点数都可以添加额外的零并且包含下划线，并不会影响字面量：

```
let paddedDouble = 000123.456
let oneMillion = 1_000_000
let justOverOneMillion = 1_000_000.000_000_1
```

## 数值型类型转换

通常来讲，即使代码中的整数常量和变量已知非负，也请使用 `Int` 类型。总是使用默认的整数类型可以保证你的整数常量和变量可以直接被复用并且可以匹配整数类字面量的类型推断。

只有在必要的时候才使用其他整数类型，比如要处理外部的长度明确的数据或者为了优化性能、内存占用等等。使用显式指定长度的类型可以及时发现值溢出并且可以暗示正在处理特殊数据。

## 整数转换

---

不同整数类型的变量和常量可以存储不同范围的数字。`Int8` 类型的常量或者变量可以存储的数字范围是 `-128 ~ 127`，而 `UInt8` 类型的常量或者变量能存储的数字范围是 `0 ~ 255`。如果数字超出了常量或者变量可存储的范围，编译的时候会报错：

```
let cannotBeNegative: UInt8 = -1
// UInt8 类型不能存储负数，所以会报错
let tooBig: Int8 = Int8.max + 1
// Int8 类型不能存储超过最大值的数，所以会报错
```

由于每种整数类型都可以存储不同范围的值，所以你必须根据不同情况选择性使用数值型类型转换。这种选择性使用的方式，可以预防隐式转换的错误并让你的代码中的类型转换意图变得清晰。

要将一种数字类型转换成另一种，你要用当前值来初始化一个期望类型的新数字，这个数字的类型就是你的目标类型。在下面的例子中，常量 `twoThousand` 是 `UInt16` 类型，然而常量 `one` 是 `UInt8` 类型。它们不能直接相加，因为它们类型不同。所以要调用 `UInt16(one)` 来创建一个新的 `UInt16` 数字并用 `one` 的值来初始化，然后使用这个新数字来计算：

```
let twoThousand: UInt16 = 2_000
let one: UInt8 = 1
let twoThousandAndOne = twoThousand + UInt16(one)
```

现在两个数字的类型都是 `UInt16`，可以进行相加。目标常量 `twoThousandAndOne` 的类型被推断为 `UInt16`，因为它是两个 `UInt16` 值的和。

`SomeType(ofInitialValue)` 是调用 Swift 构造器并传入一个初始值的默认方法。在语言内部，`UInt16` 有一个构造器，可以接受一个 `UInt8` 类型的值，所以这个构造器可以用现有的 `UInt8` 来创建一个新的 `UInt16`。注意，你并不能传入任意类型的值，只能传入 `UInt16` 内部有对应构造器的值。不过你可以扩展现有的类型来让它可以接收其他类型的值（包括自定义类型），请参考 扩展。

## 整数和浮点数转换

---

整数和浮点数的转换必须显式指定类型：

```
let three = 3
let pointOneFourOneFiveNine = 0.14159
let pi = Double(three) + pointOneFourOneFiveNine
// pi 等于 3.14159，所以被推断为 Double 类型
```

这个例子中，常量 `three` 的值被用来创建一个 `Double` 类型的值，所以加号两边的数据类型须相同。如果不进行转换，两者无法相加。

浮点数到整数的反向转换同样行，整数类型可以用 `Double` 或者 `Float` 类型来初始化：

```
let integerPi = Int(pi)  
// integerPi 等于 3，所以被推测为 Int 类型
```

当用这种方式来初始化一个新的整数值时，浮点值会被截断。也就是说 `4.75` 会变成 `4`，`-3.9` 会变成 `-3`。

### 注意

结合数字类常量和变量不同于结合数字类字面量。字面量 `3` 可以直接和字面量 `0.14159` 相加，因为数字字面量本身没有明确的类型。它们的类型只在编译器需要值的时候被推测。

## 类型别名

类型别名 (*type aliases*) 就是给现有类型定义另一个名字。你可以使用 `typealias` 关键字来定义类型别名。

当你想要给现有类型起一个更有意义的名字时，类型别名非常有用。假设你正在处理特定长度的外部资源的数据：

```
typealias AudioSample = UInt16
```

定义了一个类型别名之后，你可以在任何使用原始名的地方使用别名：

```
var maxAmplitudeFound = AudioSample.min  
// maxAmplitudeFound 现在是 0
```

本例中，`AudioSample` 被定义为 `UInt16` 的一个别名。因为它是别名，`AudioSample.min` 实际上是 `UInt16.min`，所以会给 `maxAmplitudeFound` 赋一个初值 `0`。

## 布尔值

Swift 有一个基本的布尔 (*Boolean*) 类型，叫做 `Bool`。布尔值指逻辑上的值，因为它们只能是真或者假。Swift 有两个布尔常量，`true` 和 `false`：

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

`orangesAreOrange` 和 `turnipsAreDelicious` 的类型会被推断为 `Bool`，因为它们的初值是布尔字面量。就像之前提到的 `Int` 和 `Double` 一样，如果你创建变量的时候给它们赋值 `true` 或者 `false`，那你不需要将常量或者变量声明为 `Bool` 类型。初始化常量或者变量的时候如果所赋的值类型已知，就可以触发类型推断，这让 Swift 代码更加简洁并且可读性更高。

当你编写条件语句比如 `if` 语句的时候，布尔值非常有用：

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

条件语句，例如 `if`，请参考 [控制流](#)。

如果你在需要使用 `Bool` 类型的地方使用了非布尔值，Swift 的类型安全机制会报错。下面的例子会报告一个编译时错误：

```
let i = 1  
if i {  
    // 这个例子不会通过编译，会报错  
}
```

然而，下面的例子是合法的：

```
let i = 1  
if i == 1 {  
    // 这个例子会编译成功  
}
```

`i == 1` 的比较结果是 `Bool` 类型，所以第二个例子可以通过类型检查。类似 `i == 1` 这样的比较，请参考 [基本操作符](#)。

和 Swift 中的其他类型安全的例子一样，这个方法可以避免错误并保证这块代码的意图总是清晰的。

## 元组

---

元组 (*tuples*) 把多个值组合成一个复合值。元组内的值可以是任意类型，并不要求是相同类型。

下面这个例子中，`(404, "Not Found")` 是一个描述 HTTP 状态码 (*HTTP status code*) 的元组。HTTP 状态码是当你请求网页的时候 web 服务器返回的一个特殊值。如果你请求的网页不存在就会返回一个 `404 Not Found` 状态码。

```
let http404Error = (404, "Not Found")  
// http404Error 的类型是 (Int, String)，值是 (404, "Not Found")
```

`(404, "Not Found")` 元组把一个 `Int` 值和一个 `String` 值组合起来表示 HTTP 状态码的两个部分：一个数字和一个人类可读的描述。这个元组可以被描述为“一个类型为 `(Int, String)` 的元组”。

你可以把任意顺序的类型组合成一个元组，这个元组可以包含所有类型。只要你想，你可以创建一个类型为 `(Int, Int, Int)` 或者 `(String, Bool)` 或者其他任何你想要的组合的元组。

你可以将一个元组的内容分解 (decompose) 成单独的常量和变量，然后你就可以正常使用它们了：

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

如果你只需要一部分元组值，分解的时候可以把要忽略的部分用下划线 (\_ ) 标记：

```
let (justTheStatusCode, _) = http404Error
print("The status code is \(justTheStatusCode)")
// 输出"The status code is 404"
```

此外，你还可以通过下标来访问元组中的单个元素，下标从零开始：

```
print("The status code is \(http404Error.0)")
// 输出"The status code is 404"
print("The status message is \(http404Error.1)")
// 输出"The status message is Not Found"
```

你可以在定义元组的时候给单个元素命名：

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

给元组中的元素命名后，你可以通过名字来获取这些元素的值：

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

作为函数返回值时，元组非常有用。一个用来获取网页的函数可能会返回一个 ([Int](#), [String](#)) 元组来描述是否获取成功。和只能返回一个类型的值比较起来，一个包含两个不同类型值的元组可以让函数的返回信息更有用。请参考 [函数参数与返回值](#)。

### 注意

当遇到一些相关值的简单分组时，元组是很有用的。元组不适合用来创建复杂的数据结构。如果你的数据结构比较复杂，不要使用元组，用类或结构体去建模。欲获得更多信息，请参考 [结构体和类](#)。

## 可选类型

使用 [可选类型](#) (*optionals*) 来处理值可能缺失的情况。可选类型表示两种可能：或者有值，你可以解析可选类型访问这个值，或者根本没有值。

## 注意

C 和 Objective-C 中并没有可选类型这个概念。最接近的是 Objective-C 中的一个特性，一个方法要不返回一个对象要不返回 `nil`，`nil` 表示“缺少一个合法的对象”。然而，这仅对对象起作用——对于结构体，基本的 C 类型或者枚举类型不起作用。对于这些类型，Objective-C 方法一般会返回一个特殊值（比如 `NSNotFound`）来暗示值缺失。这种方法假设方法的调用者知道并记得对特殊值进行判断。然而，Swift 的可选类型可以让你暗示任意类型的值缺失，并不需要一个特殊值。

来看一个例子。Swift 的 `Int` 类型有一种构造器，作用是将一个 `String` 值转换成一个 `Int` 值。然而，并不是所有的字符串都可以转换成一个整数。字符串 `"123"` 可以被转换成数字 `123`，但是字符串 `"hello, world"` 不行。

下面的例子使用这种构造器来尝试将一个 `String` 转换成 `Int`：

```
let possibleNumber = "123"
let convertedNumber = Int(possibleNumber)
// convertedNumber 被推测为类型 "Int?", 或者类型 "optional Int"
```

因为该构造器可能会失败，所以它返回一个可选类型（optional）`Int`，而不是一个 `Int`。一个可选的 `Int` 被写作 `Int?` 而不是 `Int`。问号暗示包含的值是可选类型，也就是说可能包含 `Int` 值也可能不包含值。（不能包含其他任何值比如 `Bool` 值或者 `String` 值。只能是 `Int` 或者什么都没有。）

## nil

你可以给可选变量赋值为 `nil` 来表示它没有值：

```
var serverResponseCode: Int? = 404
// serverResponseCode 包含一个可选的 Int 值 404
serverResponseCode = nil
// serverResponseCode 现在不包含值
```

## 注意

`nil` 不能用于非可选的常量和变量。如果你的代码中有常量或者变量需要处理值缺失的情况，请把它们声明成对应的可选类型。

如果你声明一个可选常量或者变量但是没有赋值，它们会自动被设置为 `nil`：

```
var surveyAnswer: String?
// surveyAnswer 被自动设置为 nil
```

## 注意

Swift 的 `nil` 和 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针。在 Swift 中，`nil` 不是指针——它是一个确定的值，用来表示值缺失。任何类型的可选状态都可以被设置为 `nil`，不只是对象类型。

## if 语句以及强制解析

你可以使用 `if` 语句和 `nil` 比较来判断一个可选值是否包含值。你可以使用“相等”( `==` )或“不等”( `!=` )来执行比较。

如果可选类型有值，它将不等于 `nil`：

```
if convertedNumber != nil {  
    print("convertedNumber contains some integer value.")  
}  
// 输出“convertedNumber contains some integer value.”
```

当你确定可选类型确实包含值之后，你可以在可选的名字后面加一个感叹号 ( `!` ) 来获取值。这个惊叹号表示“我知道这个可选有值，请使用它。”这被称为可选值的强制解析 (*forced unwrapping*)：

```
if convertedNumber != nil {  
    print("convertedNumber has an integer value of \(convertedNumber!).")  
}  
// 输出“convertedNumber has an integer value of 123.”
```

更多关于 `if` 语句的内容，请参考 [控制流](#)。

### 注意

使用 `!` 来获取一个不存在的可选值会导致运行时错误。使用 `!` 来强制解析值之前，一定要确定可选包含一个非 `nil` 的值。

## 可选绑定

使用 [可选绑定](#) (*optional binding*) 来判断可选类型是否包含值，如果包含就把值赋给一个临时常量或者变量。可选绑定可以用在 `if` 和 `while` 语句中，这条语句不仅可以用来判断可选类型中是否有值，同时可以将可选类型中的值赋给一个常量或者变量。`if` 和 `while` 语句，请参考 [控制流](#)。

像下面这样在 `if` 语句中写一个可选绑定：

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

你可以像上面这样使用可选绑定来重写在 [可选类型](#) 举出的 `possibleNumber` 例子：

```
if let actualNumber = Int(possibleNumber) {  
    print("\(possibleNumber)' has an integer value of \(actualNumber)")  
} else {  
    print("\(possibleNumber)' could not be converted to an integer")  
}  
// 输出“'123' has an integer value of 123”
```

这段代码可以被理解为：

“如果 `Int(possibleNumber)` 返回的可选 `Int` 包含一个值，创建一个叫做 `actualNumber` 的新常量并将可选包含的值赋给它。”

如果转换成功，`actualNumber` 常量可以在 `if` 语句的第一个分支中使用。它已经被可选类型包含的值初始化过，所以不需要再使用 `!` 后缀来获取它的值。在这个例子中，`actualNumber` 只被用来输出转换结果。

你可以在可选绑定中使用常量和变量。如果你想在 `if` 语句的第一个分支中操作 `actualNumber` 的值，你可以改成 `if var actualNumber`，这样可选类型包含的值就会被赋给一个变量而非常量。

你可以包含多个可选绑定或多个布尔条件在一个 `if` 语句中，只要使用逗号分开就行。只要有任意一个可选绑定的值为 `nil`，或者任意一个布尔条件为 `false`，则整个 `if` 条件判断为 `false`，这时你就需要使用嵌套 `if` 条件语句来处理，如下所示：

```
if let firstNumber = Int("4"), let secondNumber = Int("42"), firstNumber < secondNumber &&
secondNumber < 100 {
    print("\(firstNumber) < \(secondNumber) < 100")
}
// 输出“4 < 42 < 100”

if let firstNumber = Int("4") {
    if let secondNumber = Int("42") {
        if firstNumber < secondNumber && secondNumber < 100 {
            print("\(firstNumber) < \(secondNumber) < 100")
        }
    }
}
// 输出“4 < 42 < 100”
```

### 注意

在 `if` 条件语句中使用常量和变量来创建一个可选绑定，仅在 `if` 语句的句中（`body`）中才能获取到值。相反，在 `guard` 语句中使用常量和变量来创建一个可选绑定，仅在 `guard` 语句外且在语句后才能获取到值，请参考 [提前退出](#)。

## 隐式解析可选类型

如上所述，可选类型暗示了常量或者变量可以“没有值”。可选可以通过 `if` 语句来判断是否有值，如果有值的话可以通过可选绑定来解析值。

有时候在程序架构中，第一次被赋值之后，可以确定一个可选类型总会有值。在这种情况下，每次都要判断和解析可选值是非常低效的，因为可以确定它总会有值。

这种类型的可选状态被定义为隐式解析可选类型（implicitly unwrapped optionals）。把想要用作可选的类型的后面的问号（`String?`）改成感叹号（`String!`）来声明一个隐式解析可选类型。

当可选类型被第一次赋值之后就可以确定之后一直有值的时候，隐式解析可选类型非常有用。隐式解析可选类型主要被用在 Swift 中类的构造过程中，请参考 [无主引用以及隐式解析可选属性](#)。

一个隐式解析可选类型其实就是一个普通的可选类型，但是可以被当做非可选类型来使用，并不需要每次都使用解析来获取可选值。下面的例子展示了可选类型 `String` 和隐式解析可选类型 `String` 之间的区别：

```
let possibleString: String? = "An optional string."  
let forcedString: String = possibleString! // 需要感叹号来获取值  
  
let assumedString: String! = "An implicitly unwrapped optional string."  
let implicitString: String = assumedString // 不需要感叹号
```

你可以把隐式解析可选类型当做一个可以自动解析的可选类型。你要做的只是声明的时候把感叹号放到类型的结尾，而不是每次取值的可选名字的结尾。

### 注意

如果你在隐式解析可选类型没有值的时候尝试取值，会触发运行时错误。和你在没有值的普通可选类型后面加一个惊叹号一样。

你仍然可以把隐式解析可选类型当做普通可选类型来判断它是否包含值：

```
if assumedString != nil {  
    print(assumedString!)  
}  
// 输出“An implicitly unwrapped optional string.”
```

你也可以在可选绑定中使用隐式解析可选类型来检查并解析它的值：

```
if let definiteString = assumedString {  
    print(definiteString)  
}  
// 输出“An implicitly unwrapped optional string.”
```

### 注意

如果一个变量之后可能变成 `nil` 的话请不要使用隐式解析可选类型。如果你需要在变量的生命周期中判断是否是 `nil` 的话，请使用普通可选类型。

## 错误处理

你可以使用 [错误处理 \(error handling\)](#) 来应对程序执行中可能会遇到的错误条件。

相对于可选中运用值的存在与缺失来表达函数的成功与失败，错误处理可以推断失败的原因，并传播至程序的其他部分。

当一个函数遇到错误条件，它能报错。调用函数的地方能抛出错误消息并合理处理。

```
func canThrowAnError() throws {
    // 这个函数有可能抛出错误
}
```

一个函数可以通过在声明中添加 `throws` 关键词来抛出错误消息。当你的函数能抛出错误消息时，你应该在表达式中前置 `try` 关键词。

```
do {
    try canThrowAnError()
    // 没有错误消息抛出
} catch {
    // 有一个错误消息抛出
}
```

一个 `do` 语句创建了一个新的包含作用域，使得错误能被传播到一个或多个 `catch` 从句。

这里有一个错误处理如何用来应对不同错误条件的例子。

```
func makeASandwich() throws {
    // ...
}

do {
    try makeASandwich()
    eatASandwich()
} catch SandwichError.outOfCleanDishes {
    washDishes()
} catch SandwichError.missingIngredients(let ingredients) {
    buyGroceries(ingredients)
}
```

在此例中，`makeASandwich()`（做一个三明治）函数会抛出一个错误消息如果没有干净的盘子或者某个原料缺失。因为 `makeASandwich()` 抛出错误，函数调用被包裹在 `try` 表达式中。将函数包裹在一个 `do` 语句中，任何被抛出的错误会被传播到提供的 `catch` 从句中。

如果没有错误被抛出，`eatASandwich()` 函数会被调用。如果一个匹配 `SandwichError.outOfCleanDishes` 的错误被抛出，`washDishes()` 函数会被调用。如果一个匹配 `SandwichError.missingIngredients` 的错误被抛出，`buyGroceries(_)` 函数会被调用，并且使用 `catch` 所捕捉到的关联值 `[String]` 作为参数。

抛出，捕捉，以及传播错误会在 [错误处理](#) 章节详细说明。

## 断言和先决条件

断言和先决条件是在运行时所做的检查。你可以用他们来检查在执行后续代码之前是否一个必要的条件已经被满足了。如果断言或者先决条件中的布尔条件评估的结果为 `true`（真），则代码像往常一样继续执行。如果布尔条件评估结果为 `false`（假），程序的当前状态是无效的，则代码执行结束，应用程序中止。

你使用断言和先决条件来表达你所做的假设和你在编码时候的期望。你可以将这些包含在你的代码中。断言帮助你在开发阶段找到错误和不正确的假设，先决条件帮助你在生产环境中探测到存在的问题。

除了在运行时验证你的期望值，断言和先决条件也变成了一个在你的代码中的有用文档形式。和在上面讨论过的 错误处理 不同，断言和先决条件并不是用来处理可以恢复的或者可预期的错误。因为一个断言失败表明了程序正处于一个无效的状态，没有办法去捕获一个失败的断言。

使用断言和先决条件不是一个能够避免出现程序出现无效状态的编码方法。然而，如果一个无效状态程序产生了，断言和先决条件可以强制检查你的数据和程序状态，使得你的程序可预测的中止（译者：不是系统强制的，被动的中止），并帮助使这个问题更容易调试。一旦探测到无效的状态，执行则被中止，防止无效的状态导致的进一步对于系统的伤害。

断言和先决条件的不同点是，他们什么时候进行状态检测：断言仅在调试环境运行，而先决条件则在调试环境和生产环境中运行。在生产环境中，断言的条件将不会进行评估。这个意味着你可以使用很多断言在你的开发阶段，但是这些断言在生产环境中不会产生任何影响。

## 使用断言进行调试

---

你可以调用 Swift 标准库的 `assert(_:_:file:line:)` 函数来写一个断言。向这个函数传入一个结果为 `true` 或者 `false` 的表达式以及一条信息，当表达式的结果为 `false` 的时候这条信息会被显示：

```
let age = -3
assert(age >= 0, "A person's age cannot be less than zero")
// 因为 age < 0, 所以断言会触发
```

在这个例子中，只有 `age >= 0` 为 `true` 时，即 `age` 的值非负的时候，代码才会继续执行。如果 `age` 的值是负数，就像代码中那样，`age >= 0` 为 `false`，断言被触发，终止应用。

如果不需要断言信息，可以就像这样忽略掉：

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

如果代码已经检查了条件，你可以使用 `assertionFailure(_:_:file:line:)` 函数来表明断言失败了，例如：

```
if age > 10 {
    print("You can ride the roller-coaster or the ferris wheel.")
} else if age > 0 {
    print("You can ride the ferris wheel.")
} else {
    assertionFailure("A person's age can't be less than zero.")
}
```

## 强制执行先决条件

---

当一个条件可能为假，但是继续执行代码要求条件必须为真的时候，需要使用先决条件。例如使用先决条件来检查是否下标越界，或者来检查是否将一个正确的参数传给函数。

你可以使用全局 `precondition(_:_:file:line:)` 函数来写一个先决条件。向这个函数传入一个结果为 `true` 或者 `false` 的表达式以及一条信息，当表达式的结果为 `false` 的时候这条信息会被显示：

```
// 在一个下标的实现里...
precondition(index > 0, "Index must be greater than zero.")
```

你可以调用 `preconditionFailure(_:_:file:line:)` 方法来表明出现了一个错误，例如，`switch` 进入了 `default` 分支，但是所有的有效值应该被任意一个其他分支（非 `default` 分支）处理。

### 注意

如果你使用 `unchecked` 模式 (`-Ounchecked`) 编译代码，先决条件将不会进行检查。编译器假设所有的先决条件总是为 `true` (真)，他将优化你的代码。然而，`fatalError(_:_:file:line:)` 函数总是中断执行，无论你怎么进行优化设定。

你能使用 `fatalError(_:_:file:line:)` 函数在设计原型和早期开发阶段，这个阶段只有方法的声明，但是没有具体实现，你可以在方法体中写上 `fatalError("Unimplemented")` 作为具体实现。因为 `fatalError` 不会像断言和先决条件那样被优化掉，所以你可以确保当代码执行到一个没有被实现的方法时，程序会被中断。

# 基本运算符 · GitBook

 [runoob.com/manual/gitbook/swift5/source/\\_book/chapter2/02\\_Basic\\_Operators.html](https://runoob.com/manual/gitbook/swift5/source/_book/chapter2/02_Basic_Operators.html)

## 基本运算符

运算符是检查、改变、合并值的特殊符号或短语。例如，加号（`+`）将两个数相加（如 `let i = 1 + 2`）。更复杂的运算例子包括逻辑与运算符 `&&`（如 `if enteredDoorCode && passedRetinaScan`）。

Swift 支持大部分标准 C 语言的运算符，且为了减少常见编码错误做了部分改进。如：赋值符（`=`）不再有返回值，这样就消除了手误将判等运算符（`==`）写成赋值符导致代码错误的缺陷。算术运算符（`+`，`-`，`*`，`/`，`%` 等）的结果会被检测并禁止值溢出，以此来避免保存变量时由于变量大于或小于其类型所能承载的范围时导致的异常结果。当然允许你使用 Swift 的溢出运算符来实现溢出。详情参见 [溢出运算符](#)。

Swift 还提供了 C 语言没有的区间运算符，例如 `a..<b` 或 `a...b`，这方便我们表达一个区间内的数值。

本章节只描述了 Swift 中的基本运算符，[高级运算符](#) 这章会包含 Swift 中的高级运算符，及如何自定义运算符，及如何进行自定义类型的运算符重载。

## 术语

运算符分为一元、二元和三元运算符：

- **一元运算符**对单一操作对象操作（如 `-a`）。一元运算符分前置运算符和后置运算符，前置运算符需紧跟在操作对象之前（如 `!b`），后置运算符需紧跟在操作对象之后（如 `c!`）。
- **二元运算符**操作两个操作对象（如 `2 + 3`），是中置的，因为它们出现在两个操作对象之间。
- **三元运算符**操作三个操作对象，和 C 语言一样，Swift 只有一个三元运算符，就是三目运算符（`a ? b : c`）。

受运算符影响的值叫**操作数**，在表达式 `1 + 2` 中，加号 `+` 是二元运算符，它的两个操作数是值 `1` 和 `2`。

## 赋值运算符

赋值运算符（`a = b`），表示用 `b` 的值来初始化或更新 `a` 的值：

```
let b = 10
var a = 5
a = b
// a 现在等于 10
```

如果赋值的右边是一个多元组，它的元素可以马上被分解成多个常量或变量：

```
let (x, y) = (1, 2)
// 现在 x 等于 1, y 等于 2
```

与 C 语言和 Objective-C 不同，Swift 的赋值操作并不返回任何值。所以下面语句是无效的：

```
if x = y {
    // 此句错误，因为 x = y 并不返回任何值
}
```

通过将 `if x = y` 标记为无效语句，Swift 能帮你避免把 `( == )` 错写成 `( = )` 这类错误的出现。

## 算术运算符

---

Swift 中所有数值类型都支持了基本的四则算术运算符：

- 加法 (`+`)
- 减法 (`-`)
- 乘法 (`*`)
- 除法 (`/`)

```
1 + 2      // 等于 3
5 - 3      // 等于 2
2 * 3      // 等于 6
10.0 / 2.5 // 等于 4.0
```

与 C 语言和 Objective-C 不同的是，Swift 默认情况下不允许在数值运算中出现溢出情况。但是你可以使用 Swift 的溢出运算符来实现溢出运算（如 `a &+ b`）。详情参见 [溢出运算符](#)。

加法运算符也可用于 `String` 的拼接：

```
"hello, " + "world" // 等于 "hello, world"
```

## 求余运算符

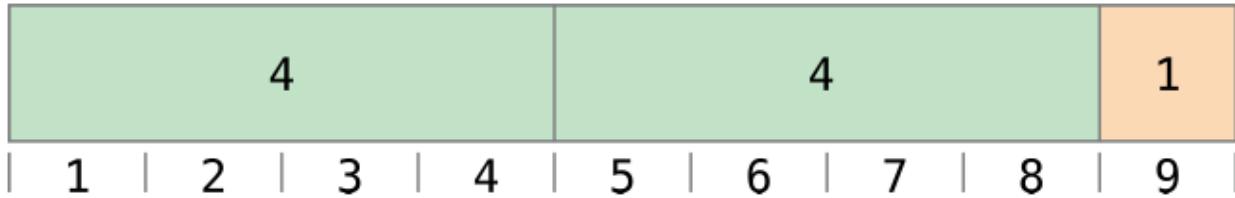
---

求余运算符 (`a % b`) 是计算 `b` 的多少倍刚刚好可以容入 `a`，返回多出来的那部分（余数）。

注意

求余运算符 (`%`) 在其他语言也叫取模运算符。但是严格说来，我们看该运算符对负数的操作结果，「求余」比「取模」更合适些。

我们来谈谈取余是怎么回事，计算 `9 % 4`，你先计算出 `4` 的多少倍会刚好可以放入 `9` 中：



你可以在 9 中放入两个 4 , 那余数是 1 (用橙色标出)。

在 Swift 中可以表达为 :

`9 % 4 // 等于 1`

为了得到 `a % b` 的结果 , `%` 计算了以下等式 , 并输出 余数 作为结果 :

$$a = (b \times \text{倍数}) + \text{余数}$$

当 倍数 取最大值的时候 , 就会刚好可以容入 a 中。

把 9 和 4 代入等式中 , 我们得 1 :

$$9 = (4 \times 2) + 1$$

同样的方法 , 我们来计算 `-9 % 4` :

`-9 % 4 // 等于 -1`

把 -9 和 4 代入等式 , -2 是取到的最大整数 :

$$-9 = (4 \times -2) + -1$$

余数是 -1 。

在对负数 b 求余时 , b 的符号会被忽略。这意味着 `a % b` 和 `a % -b` 的结果是相同的。

## 一元负号运算符

数值的正负号可以使用前缀 `-` (即一元负号符) 来切换 :

```
let three = 3
let minusThree = -three    // minusThree 等于 -3
let plusThree = -minusThree // plusThree 等于 3, 或 "负负3"
```

一元负号符 (`-`) 写在操作数之前 , 中间没有空格。

## 一元正号运算符

一元正号符 (`+`) 不做任何改变地返回操作数的值 :

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

虽然一元正号符什么都不会改变，但当你在使用一元负号来表达负数时，你可以使用一元正号来表达正数，如此你的代码会具有对称美。

## 组合赋值运算符

如同 C 语言，Swift 也提供把其他运算符和赋值运算（`=`）组合的组合赋值运算符，组合加运算（`+=`）是其中一个例子：

```
var a = 1  
a += 2  
// a 现在是 3
```

表达式 `a += 2` 是 `a = a + 2` 的简写，一个组合加运算就是把加法运算和赋值运算组合成进一个运算符里，同时完成两个运算任务。

注意

复合赋值运算没有返回值，`let b = a += 2` 这类代码是错误。这不同于上面提到的自增和自减运算符。

更多 Swift 标准库运算符的信息，请看 [运算符声明](#)。

## 比较运算符 (Comparison Operators)

所有标准 C 语言中的比较运算符都可以在 Swift 中使用：

- 等于（`a == b`）
- 不等于（`a != b`）
- 大于（`a > b`）
- 小于（`a < b`）
- 大于等于（`a >= b`）
- 小于等于（`a <= b`）

注意

Swift 也提供恒等（`==`）和不恒等（`!=`）这两个比较符来判断两个对象是否引用同一个对象实例。更多细节在 [类与结构](#) 章节的 **Identity Operators** 部分。

每个比较运算都返回了一个标识表达式是否成立的布尔值：

```
1 == 1 // true, 因为 1 等于 1  
2 != 1 // true, 因为 2 不等于 1  
2 > 1 // true, 因为 2 大于 1  
1 < 2 // true, 因为 1 小于 2  
1 >= 1 // true, 因为 1 大于等于 1  
2 <= 1 // false, 因为 2 并不小于等于 1
```

比较运算多用于条件语句，如 `if` 条件：

```
let name = "world"
if name == "world" {
    print("hello, world")
} else {
    print("I'm sorry \((name), but I don't recognize you")
}
// 输出"hello, world", 因为 `name` 就是等于 "world"
```

关于 `if` 语句，请看 [控制流](#)。

如果两个元组的元素相同，且长度相同的话，元组就可以被比较。比较元组大小会按照从左到右、逐值比较的方式，直到发现有两个值不等时停止。如果所有的值都相等，那么这一对元组我们就称它们是相等的。例如：

```
(1, "zebra") < (2, "apple") // true, 因为 1 小于 2
(3, "apple") < (3, "bird") // true, 因为 3 等于 3, 但是 apple 小于 bird
(4, "dog") == (4, "dog") // true, 因为 4 等于 4, dog 等于 dog
```

在上面的例子中，你可以看到，在第一行中从左到右的比较行为。因为 `1` 小于 `2`，所以 `(1, "zebra")` 小于 `(2, "apple")`，不管元组剩下的值如何。所以 `"zebra"` 大于 `"apple"` 对结果没有任何影响，因为元组的比较结果已经被第一个元素决定了。不过，当元组的第一个元素相同时时候，第二个元素将会用作比较-第二行和第三行代码就发生了这样的比较。

当元组中的元素都可以被比较时，你也可以使用这些运算符来比较它们的大小。例如，像下面展示的代码，你可以比较两个类型为 `(String, Int)` 的元组，因为 `Int` 和 `String` 类型的值可以比较。相反，`Bool` 不能被比较，也意味着存有布尔类型的元组不能被比较。

```
("blue", -1) < ("purple", 1) // 正常, 比较的结果为 true
("blue", false) < ("purple", true) // 错误, 因为 < 不能比较布尔类型
```

### 注意

Swift 标准库只能比较七个以内元素的元组比较函数。如果你的元组元素超过七个时，你需要自己实现比较运算符。

## 三元运算符 (Ternary Conditional Operator)

三元运算符的特殊在于它是有三个操作数的运算符，它的形式是 `问题 ? 答案 1 : 答案 2`。它简洁地表达根据 `问题` 成立与否作出二选一的操作。如果 `问题` 成立，返回 `答案 1` 的结果；反之返回 `答案 2` 的结果。

三元运算符是以下代码的缩写形式：

```
if question {
    answer1
} else {
    answer2
}
```

这里有个计算表格行高的例子。如果有表头，那行高应比内容高度要高出 50 点；如果没有表头，只需高出 20 点：

```
let contentHeight = 40
let hasHeader = true
let rowHeight = contentHeight + (hasHeader ? 50 : 20)
// rowHeight 现在是 90
```

上面的写法比下面的代码更简洁：

```
let contentHeight = 40
let hasHeader = true
var rowHeight = contentHeight
if hasHeader {
    rowHeight = rowHeight + 50
} else {
    rowHeight = rowHeight + 20
}
// rowHeight 现在是 90
```

第一段代码例子使用了三元运算，所以一行代码就能让我们得到正确答案。这比第二段代码简洁得多，无需将 `rowHeight` 定义成变量，因为它的值无需在 `if` 语句中改变。

三元运算为二选一场景提供了一个非常便捷的表达形式。不过需要注意的是，滥用三元运算符会降低代码可读性。所以我们应避免在一个复合语句中使用多个三元运算符。

## 空合运算符（Nil Coalescing Operator）

空合运算符 (`a ?? b`) 将对可选类型 `a` 进行空判断，如果 `a` 包含一个值就进行解包，否则就返回一个默认值 `b`。表达式 `a` 必须是 `Optional` 类型。默认值 `b` 的类型必须要和 `a` 存储值的类型保持一致。

空合运算符是对以下代码的简短表达方法：

```
a != nil ? a! : b
```

上述代码使用了三元运算符。当可选类型 `a` 的值不为空时，进行强制解封 (`a!`)，访问 `a` 中的值；反之返回默认值 `b`。无疑空合运算符 (`??`) 提供了一种更为优雅的方式去封装条件判断和解封两种行为，显得简洁以及更具可读性。

注意

如果 `a` 为非空值 (`non-nil`)，那么值 `b` 将不会被计算。这也就是所谓的短路求值。

下文例子采用空合运算符，实现了在默认颜色名和可选自定义颜色名之间抉择：

```
let defaultColorName = "red"
var userDefinedColorName: String? //默认值为 nil

var colorNameToUse = userDefinedColorName ?? defaultColorName
// userDefinedColorName 的值为空，所以 colorNameToUse 的值为 "red"
```

`userDefinedColorName` 变量被定义为一个可选的 `String` 类型，默认值为 `nil`。由于 `userDefinedColorName` 是一个可选类型，我们可以使用空合运算符去判断其值。在上一个例子中，通过空合运算符为一个名为 `colorNameToUse` 的变量赋予一个字符串类型初始值。由于 `userDefinedColorName` 值为空，因此表达式 `userDefinedColorName ?? defaultColorName` 返回 `defaultColorName` 的值，即 `red`。

如果你分配一个非空值（`non-nil`）给 `userDefinedColorName`，再次执行空合运算，运算结果为封包在 `userDefaultColorName` 中的值，而非默认值。

```
userDefinedColorName = "green"
colorNameToUse = userDefinedColorName ?? defaultColorName
// userDefinedColorName 非空，因此 colorNameToUse 的值为 "green"
```

## 区间运算符 (Range Operators)

---

Swift 提供了几种方便表达一个区间的值的区间运算符。

### 闭区间运算符

---

闭区间运算符 (`a...b`) 定义一个包含从 `a` 到 `b` (包括 `a` 和 `b`) 的所有值的区间。`a` 的值不能超过 `b`。

闭区间运算符在迭代一个区间的所有值时是非常有用的，如在 `for-in` 循环中：

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

关于 `for-in` 循环，请看 [控制流](#)。

### 半开区间运算符

---

半开区间运算符 (`a..`) 定义一个从 `a` 到 `b` 但不包括 `b` 的区间。之所以称为半开区间，是因为该区间包含第一个值而不包括最后的值。

半开区间的实用性在于当你使用一个从 `0` 开始的列表（如数组）时，非常方便地从 `0` 数到列表的长度。

```
let names = ["Anna", "Alex", "Brian", "Jack"]
let count = names.count
for i in 0..<count {
    print("第 \((i + 1) 个人叫 \(names[i])")
}
// 第 1 个人叫 Anna
// 第 2 个人叫 Alex
// 第 3 个人叫 Brian
// 第 4 个人叫 Jack
```

数组有 4 个元素，但 `0..<count` 只数到 3（最后一个元素的下标），因为它是半开区间。关于数组，请查阅 [数组](#)。

## 单侧区间

---

闭区间操作符有另一个表达形式，可以表达往一侧无限延伸的区间——例如，一个包含了数组从索引 2 到结尾的所有值的区间。在这些情况下，你可以省略掉区间操作符一侧的值。这种区间叫做单侧区间，因为操作符只有一侧有值。例如：

```
for name in names[2...] {
    print(name)
}
// Brian
// Jack

for name in names[...2] {
    print(name)
}
// Anna
// Alex
// Brian
```

半开区间操作符也有单侧表达形式，附带上它的最终值。就像你使用区间去包含一个值，最终值并不会落在区间内。例如：

```
for name in names[..<2] {
    print(name)
}
// Anna
// Alex
```

单侧区间不止可以在下标里使用，也可以在别的情境下使用。你不能遍历省略了初始值的单侧区间，因为遍历的开端并不明显。你可以遍历一个省略最终值的单侧区间；然而，由于这种区间无限延伸的特性，请保证你在循环里有一个结束循环的分支。你也可以查看一个单侧区间是否包含某个特定的值，就像下面展示的那样。

```
let range = ...5
range.contains(7) // false
range.contains(4) // true
range.contains(-1) // true
```

## 逻辑运算符（Logical Operators）

---

逻辑运算符的操作对象是逻辑布尔值。Swift 支持基于 C 语言的三个标准逻辑运算。

- 逻辑非 ( `!a` )
- 逻辑与 ( `a && b` )
- 逻辑或 ( `a || b` )

## 逻辑非运算符

---

逻辑非运算符 ( `!a` ) 对一个布尔值取反，使得 `true` 变 `false`，`false` 变 `true`。

它是一个前置运算符，需紧跟在操作数之前，且不加空格。读作 `非 a`，例子如下：

```
let allowedEntry = false
if !allowedEntry {
    print("ACCESS DENIED")
}
// 输出“ACCESS DENIED”
```

`if !allowedEntry` 语句可以读作「如果非 `allowedEntry`」，接下一行代码只有在「非 `allowedEntry`」为 `true`，即 `allowEntry` 为 `false` 时被执行。

在示例代码中，小心地选择布尔常量或变量有助于代码的可读性，并且避免使用双重逻辑非运算，或混乱的逻辑语句。

## 逻辑与运算符 #`{logical_and_operator}`

---

逻辑与运算符 ( `a && b` ) 表达了只有 `a` 和 `b` 的值都为 `true` 时，整个表达式的值才会是 `true`。

只要任意一个值为 `false`，整个表达式的值就为 `false`。事实上，如果第一个值为 `false`，那么是不去计算第二个值的，因为它已经不可能影响整个表达式的结果了。这被称做短路计算 (*short-circuit evaluation*)。

以下例子，只有两个 `Bool` 值都为 `true` 的时候才允许进入 if：

```
let enteredDoorCode = true
let passedRetinaScan = false
if enteredDoorCode && passedRetinaScan {
    print("Welcome!")
} else {
    print("ACCESS DENIED")
}
// 输出“ACCESS DENIED”
```

## 逻辑或运算符 #`{logical_or_operator}`

---

逻辑或运算符 ( `a || b` ) 是一个由两个连续的 | 组成的中置运算符。它表示了两个逻辑表达式的其中一个为 `true`，整个表达式就为 `true`。

同逻辑与运算符类似，逻辑或也是「短路计算」的，当左端的表达式为 `true` 时，将不计算右边的表达式了，因为它不可能改变整个表达式的值了。

以下示例代码中，第一个布尔值（`hasDoorKey`）为 `false`，但第二个值（`knowsOverridePassword`）为 `true`，所以整个表达是 `true`，于是允许进入：

```
let hasDoorKey = false
let knowsOverridePassword = true
if hasDoorKey || knowsOverridePassword {
    print("Welcome!")
} else {
    print("ACCESS DENIED")
}
// 输出“Welcome!”
```

## 逻辑运算符组合计算

---

我们可以组合多个逻辑运算符来表达一个复合逻辑：

```
if enteredDoorCode && passedRetinaScan || hasDoorKey || knowsOverridePassword {
    print("Welcome!")
} else {
    print("ACCESS DENIED")
}
// 输出“Welcome!”
```

这个例子使用了含多个 `&&` 和 `||` 的复合逻辑。但无论怎样，`&&` 和 `||` 始终只能操作两个值。所以这实际是三个简单逻辑连续操作的结果。我们来解读一下：

如果我们输入了正确的密码并通过了视网膜扫描，或者我们有一把有效的钥匙，又或者我们知道紧急情况下重置的密码，我们就能把门打开进入。

前两种情况，我们都不能满足，所以前两个简单逻辑的结果是 `false`，但是我们知道紧急情况下重置的密码的，所以整个复杂表达式的值还是 `true`。

注意

Swift 逻辑操作符 `&&` 和 `||` 是左结合的，这意味着拥有多元逻辑操作符的复合表达式优先计算最左边的子表达式。

## 使用括号来明确优先级

---

为了一个复杂表达式更容易读懂，在合适的地方使用括号来明确优先级是很有效的，虽然它并非必要的。在上个关于门的权限的例子中，我们给第一个部分加个括号，使它看起来逻辑更明确：

```
if (enteredDoorCode && passedRetinaScan) || hasDoorKey || knowsOverridePassword {  
    print("Welcome!")  
} else {  
    print("ACCESS DENIED")  
}  
// 输出“Welcome!”
```

这括号使得前两个值被看成整个逻辑表达中独立的一个部分。虽然有括号和没括号的输出结果是一样的，但对于读代码的人来说有括号的代码更清晰。可读性比简洁性更重要，请在可以让你代码变清晰的地方加个括号吧！

# 字符串和字符 · GitBook

 [runoob.com/manual/gitbook/swift5/source/\\_book/chapter2/03.Strings\\_and\\_Characters.html](https://runoob.com/manual/gitbook/swift5/source/_book/chapter2/03.Strings_and_Characters.html)

## 字符串和字符

字符串是一系列字符的集合，例如 `"hello, world"`，`"albatross"`。Swift 的字符串通过 `String` 类型来表示。而 `String` 内容的访问方式有多种，例如以 `Character` 值的集合。

Swift 的 `String` 和 `Character` 类型提供了一种快速且兼容 Unicode 的方式来处理代码中的文本内容。创建和操作字符串的语法与 C 语言中字符串操作相似，轻量并且易读。通过 `+` 符号就可以非常简单的实现两个字符串的拼接操作。与 Swift 中其他值一样，能否更改字符串的值，取决于其被定义为常量还是变量。你可以在已有字符串中插入常量、变量、字面量和表达式从而形成更长的字符串，这一过程也被成为字符串插值。尤其是在为显示、存储和打印创建自定义字符串值时，字符串插值操作尤其有用。

尽管语法简易，但 Swift 中的 `String` 类型的实现却很快速和现代化。每一个字符串都是由编码无关的 Unicode 字符组成，并支持访问字符的多种 Unicode 表示形式。

### 注意

Swift 的 `String` 类型与 Foundation `NSString` 类进行了无缝桥接。Foundation 还对 `String` 进行扩展使其可以访问 `NSString` 类型中定义的方法。这意味着调用那些 `NSString` 的方法，你无需进行任何类型转换。

更多关于在 Foundation 和 Cocoa 中使用 `String` 的信息请查看 [Bridging Between String and NSString](#)。

## 字符串字面量

你可以在代码里使用一段预定义的字符串值作为字符串字面量。字符串字面量是由一对双引号包裹着的具有固定顺序的字符集。

字符串字面量可以用于为常量和变量提供初始值：

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

注意，Swift 之所以推断 `someString` 常量为字符串类型，是因为它使用了字面量方式进行初始化。

## 多行字符串字面量

如果你需要一个字符串是跨越多行的，那就使用多行字符串字面量 – 由一对三个双引号包裹着的具有固定顺序的文本字符集：

```
let quotation = """
The White Rabbit put on his spectacles. "Where shall I begin,
please your Majesty?" he asked.
```

"Begin at the beginning," the King said gravely, "and go on
till you come to the end; then stop."

""

一个多行字符串字面量包含了所有的在开启和关闭引号（`"""`）中的行。这个字符从开启引号（`"""`）之后的第一行开始，到关闭引号（`"""`）之前为止。这就意味着字符串开启引号之后（`"""`）或者结束引号（`"""`）之前都没有换行符号。（译者：下面两个字符串其实是一样的，虽然第二个使用了多行字符串的形式）

```
let singleLineString = "These are the same."
let multilineString = """
These are the same.
"""


```

如果你的代码中，多行字符串字面量包含换行符的话，则多行字符串字面量中也会包含换行符。如果你想换行，以便加强代码的可读性，但是你又不想在你的多行字符串字面量中出现换行符的话，你可以用在行尾写一个反斜杠（`\`）作为续行符。

```
let softWrappedQuotation = """
The White Rabbit put on his spectacles. "Where shall I begin, \
please your Majesty?" he asked.
```

"Begin at the beginning," the King said gravely, "and go on \
till you come to the end; then stop."

""

为了让一个多行字符串字面量开始和结束于换行符，请将换行写在第一行和最后一行，例如：

```
let lineBreaks = """
This string starts with a line break.
It also ends with a line break.

"""


```

一个多行字符串字面量能够缩进来匹配周围的代码。关闭引号（`"""`）之前的空白字符串告诉 Swift 编译器其他各行多少空白字符串需要忽略。然而，如果你在某行的前面写的空白字符串超出了关闭引号（`"""`）之前的空白字符串，则超出部分将被包含在多行字符串字面量中。

```

let linesWithIndentation = """
    This line doesn't begin with whitespace.
Space ignored      This line begins with four spaces.
Appears in string  This line doesn't begin with whitespace.
                    """

```

在上面的例子中，尽管整个多行字符串字面量都是缩进的（源代码缩进），第一行和最后一行没有以空白字符串开始（实际的变量值）。中间一行的缩进用空白字符串（源代码缩进）比关闭引号（`"""`）之前的空白字符串多，所以，它的行首将有4个空格。

## 字符串字面量的特殊字符

---

字符串字面量可以包含以下特殊字符：

- 转义字符 `\0` (空字符)、`\\"` (反斜线)、`\t` (水平制表符)、`\n` (换行符)、`\r` (回车符)、`\\"` (双引号)、`\'` (单引号)。
- Unicode 标量，写成 `\u{n}` (`n` 为小写)，其中 `n` 为任意一到八位十六进制数且可用的 Unicode 位码。

下面的代码为各种特殊字符的使用示例。`wiseWords` 常量包含了两个双引号。

`dollarSign`、`blackHeart` 和 `sparklingHeart` 常量演示了三种不同格式的 Unicode 标量：

```

let wiseWords = "\"Imagination is more important than knowledge\" - Einstein"
// "Imageination is more important than knowledge" - Enistein
let dollarSign = "\u{24}"          // $, Unicode 标量 U+0024
let blackHeart = "\u{2665}"        // ❤, Unicode 标量 U+2665
let sparklingHeart = "\u{1F496}"    // 💫, Unicode 标量 U+1F496

```

由于多行字符串字面量使用了三个双引号，而不是一个，所以你可以在多行字符串字面量里直接使用双引号（`"`）而不必加上转义符（`\`）。要在多行字符串字面量中使用`"""` 的话，就需要使用至少一个转义符（`\`）：

```

let threeDoubleQuotes = """
Escaping the first quote \""""
Escaping all three quotes \"\""""
"""

```

## 扩展字符串分隔符

---

您可以将字符串文字放在扩展分隔符中，这样字符串中的特殊字符将会被直接包含而非转义后的效果。将字符串放在引号（`"`）中并用数字符号（`#`）括起来。例如，打印字符串文字 `# "Line 1 \ nLine 2" #` 打印换行符转义序列（`\n`）而不是进行换行打印。

如果需要字符串文字中字符的特殊效果，请匹配转义字符（`\`）后面添加与起始位置个数相匹配的 `#` 符。例如，如果您的字符串是 `# "Line 1 \ nLine 2" #` 并且您想要换行，则可以使用 `# “Line 1 \ #nLine 2” #` 来代替。同样，`###"Line1 \ ###`

**nLine2"###** 也可以实现换行效果。

扩展分隔符创建的字符串文字也可以是多行字符串文字。您可以使用扩展分隔符在多行字符串中包含文本 """"，覆盖原有的结束文字的默认行为。例如：

```
let threeMoreDoubleQuotationMarks = #"""
Here are three more double quotes: """
"""#
```

## 初始化空字符串

要创建一个空字符串作为初始值，可以将空的字符串字面量赋值给变量，也可以初始化一个新的 `String` 实例：

```
var emptyString = ""           // 空字符串字面量
var anotherEmptyString = String() // 初始化方法
// 两个字符串均为空并等价。
```

你可以通过检查 `Bool` 类型的 `isEmpty` 属性来判断该字符串是否为空：

```
if emptyString.isEmpty {
    print("Nothing to see here")
}
// 打印输出：“Nothing to see here”
```

## 字符串可变性

你可以通过将一个特定字符串分配给一个变量来对其进行修改，或者分配给一个常量来保证其不会被修改：

```
var variableString = "Horse"
variableString += " and carriage"
// variableString 现在为 "Horse and carriage"

let constantString = "Highlander"
constantString += " and another Highlander"
// 这会报告一个编译错误 (compile-time error) - 常量字符串不可以被修改。
```

注意

在 Objective-C 和 Cocoa 中，需要通过选择两个不同的类（`NSString` 和 `NSMutableString`）来指定字符串是否可以被修改。

## 字符串是值类型

在 Swift 中 `String` 类型是值类型。如果你创建了一个新的字符串，那么当对其进行常量、变量赋值操作，或在函数/方法中传递时，会进行值拷贝。在前述任一情况下，都会对已有字符串值创建新副本，并对该新副本而非原始字符串进行传递或赋值操作。值类型在 [结构体和枚举是值类型](#) 中进行了详细描述。

Swift 默认拷贝字符串的行为保证了在函数/方法向你传递的字符串所属权属于你，无论

该值来自于哪里。你可以确信传递的字符串不会被修改，除非你自己去修改它。

在实际编译时，Swift 编译器会优化字符串的使用，使实际的复制只发生在绝对必要的情况下，这意味着你将字符串作为值类型的同时可以获得极高的性能。

## 使用字符

---

你可通过 `for-in` 循环来遍历字符串，获取字符串中每一个字符的值：

```
for character in "Dog!%@", {
    print(character)
}
// D
// o
// g
// !
// %
```

`for-in` 循环在 [For 循环](#) 中进行了详细描述。

另外，通过标明一个 `Character` 类型并用字符字面量进行赋值，可以建立一个独立的字符常量或变量：

```
let exclamationMark: Character = "!"
```

字符串可以通过传递一个值类型为 `Character` 的数组作为自变量来初始化：

```
let catCharacters: [Character] = ["C", "a", "t", "!", "🐱"]
let catString = String(catCharacters)
print(catString)
// 打印输出：“Cat!🐱”
```

## 连接字符串和字符

---

字符串可以通过加法运算符（`+`）相加在一起（或称“连接”）创建一个新的字符串：

```
let string1 = "hello"
let string2 = " there"
var welcome = string1 + string2
// welcome 现在等于 "hello there"
```

你也可以通过加法赋值运算符（`+=`）将一个字符串添加到一个已经存在字符串变量上：

```
var instruction = "look over"
instruction += string2
// instruction 现在等于 "look over there"
```

你可以用 `append()` 方法将一个字符附加到一个字符串变量的尾部：

```
let exclamationMark: Character = "!"  
welcome.append(exclamationMark)  
// welcome 现在等于 "hello there!"
```

### 注意

你不能将一个字符串或者字符添加到一个已经存在的字符变量上，因为字符变量只能包含一个字符。

如果你需要使用多行字符串字面量来拼接字符串，并且你需要字符串每一行都以换行符结尾，包括最后一行：

```
let badStart = """  
one  
two  
"""
```

```
let end = """  
three  
"""  
print(badStart + end)  
// 打印两行:  
// one  
// twothree
```

```
let goodStart = """  
one  
two  
"""  
print(goodStart + end)
```

```
// 打印三行:  
// one  
// two  
// three
```

上面的代码，把 `badStart` 和 `end` 拼接起来的字符串非我们想要的结果。因为 `badStart` 最后一行没有换行符，它与 `end` 的第一行结合到了一起。相反的，`goodStart` 的每一行都以换行符结尾，所以它与 `end` 拼接的字符串总共有三行，正如我们期望的那样。

## 字符串插值

字符串插值是一种构建新字符串的方式，可以在其中包含常量、变量、字面量和表达式。字符串字面量和多行字符串字面量都可以使用字符串插值。你插入的字符串字面量的每一项都在以反斜线为前缀的圆括号中：

```
let multiplier = 3  
let message = "\n(multiplier) times 2.5 is \n(Double(multiplier) * 2.5)"  
// message 是 "3 times 2.5 is 7.5"
```

在上面的例子中，`multiplier` 作为 `\(multiplier)` 被插入到一个字符串常量量中。当创建字符串执行插值计算时此占位符会被替换为 `multiplier` 实际的值。

`multiplier` 的值也作为字符串中后面表达式的一部分。该表达式计算 `Double(multiplier) * 2.5` 的值并将结果（`7.5`）插入到字符串中。在这个例子中，表达式写为 `\(Double(multiplier) * 2.5)` 并包含在字符串字面量中。

### 注意

插值字符串中写在括号中的表达式不能包含非转义反斜杠（`\`），并且不能包含回车或换行符。不过，插值字符串可以包含其他字面量。

## Unicode

*Unicode*是一个用于在不同书写系统中对文本进行编码、表示和处理的国际标准。它使你可以用标准格式表示来自任意语言几乎所有的字符，并能够对文本文件或网页这样的外部资源中的字符进行读写操作。Swift 的 `String` 和 `Character` 类型是完全兼容 *Unicode* 标准的。

### Unicode 标量

Swift 的 `String` 类型是基于 *Unicode* 标量建立的。*Unicode* 标量是对应字符或者修饰符的唯一的 21 位数字，例如 `U+0061` 表示小写的拉丁字母（`LATIN SMALL LETTER A`）（"a"），`U+1F425` 表示小鸡表情（`FRONT-FACING BABY CHICK`）（"🐥"）。

请注意，并非所有 21 位 *Unicode* 标量值都分配给字符，某些标量被保留用于将来分配或用于 UTF-16 编码。已分配的标量值通常也有一个名称，例如上面示例中的 `LATIN SMALL LETTER A` 和 `FRONT-FACING BABY CHICK`。

### 可扩展的字形群集

每一个 Swift 的 `Character` 类型代表一个可扩展的字形群。而一个可扩展的字形群构成了人类可读的单个字符，它由一个或多个（当组合时）*Unicode* 标量的序列组成。

举个例子，字母 `é` 可以用单一的 *Unicode* 标量 `é`（`LATIN SMALL LETTER E WITH ACUTE`，或者 `U+00E9`）来表示。然而一个标准的字母 `e`（`LATIN SMALL LETTER E` 或者 `U+0065`）加上一个急促重音（`COMBINING ACTUE ACCENT`）的标量（`U+0301`），这样一对标量就表示了同样的字母 `é`。这个急促重音的标量形象的将 `e` 转换成了 `é`。

在这两种情况下，字母 `é` 代表了一个单一的 Swift 的 `Character` 值，同时代表了一个可扩展的字形群。在第一种情况，这个字形群包含一个单一标量；而在第二种情况，它是包含两个标量的字形群：

```
let eAcute: Character = "\u{E9}"           // é
let combinedEAcute: Character = "\u{65}\u{301}" // e 后面加上 ·
// eAcute 是 é, combinedEAcute 是 é
```

可扩展的字形集是一个将许多复杂的脚本字符表示为单个字符值的灵活方式。例如，来自朝鲜语字母表的韩语音节能表示为组合或分解的有序排列。在 Swift 都会表示为同一个单一的 `Character` 值：

```
let precomposed: Character = "\u{D55C}"           // 한
let decomposed: Character = "\u{1112}\u{1161}\u{11AB}" // ㅎ,ㅏ,ㄴ
// precomposed 是 한, decomposed 是 한
```

可拓展的字符群集可以使包围记号（例如 **COMBINING ENCLOSING CIRCLE** 或者 **U+20DD**）的标量包围其他 Unicode 标量，作为一个单一的 **Character** 值：

```
let enclosedEAcute: Character = "\u{E9}\u{20DD}"
// enclosedEAcute 是 é○
```

地域性指示符号的 Unicode 标量可以组合成一个单一的 **Character** 值，例如 **REGIONAL INDICATOR SYMBOL LETTER U** (**U+1F1FA**) 和 **REGIONAL INDICATOR SYMBOL LETTER S** (**U+1F1F8**)：

```
let regionalIndicatorForUS: Character = "\u{1F1FA}\u{1F1F8}"
// regionalIndicatorForUS 是 🇺🇸
```

## 计算字符数量

---

如果想要获得一个字符串中 **Character** 值的数量，可以使用 **count** 属性：

```
let unusualMenagerie = "Koala 🐾, Snail 🐌, Penguin 🐧, Dromedary 🐫"
print("unusualMenagerie has \(unusualMenagerie.count) characters")
// 打印输出“unusualMenagerie has 40 characters”
```

注意在 Swift 中，使用可拓展的字符群集作为 **Character** 值来连接或改变字符串时，并不一定会更改字符串的字符数量。

例如，如果你用四个字符的单词 **cafe** 初始化一个新的字符串，然后添加一个 **COMBINING ACTUE ACCENT** (**U+0301**) 作为字符串的结尾。最终这个字符串的字符数量仍然是 **4**，因为第四个字符是 **é**，而不是 **e**：

```
var word = "cafe"
print("the number of characters in \(word) is \(word.count)")
// 打印输出“the number of characters in cafe is 4”

word += "\u{301}" // 拼接一个重音，U+0301

print("the number of characters in \(word) is \(word.count)")
// 打印输出“the number of characters in café is 4”
```

## 注意

可扩展的字形群可以由多个 Unicode 标量组成。这意味着不同的字符以及相同字符的不同表示方式可能需要不同数量的内存空间来存储。所以 Swift 中的字符在一个字符串中并不一定占用相同的内存空间数量。因此在没有获取字符串的可扩展的字符群的范围时候，就不能计算出字符串的字符数量。如果你正在处理一个长字符串，需要注意 `count` 属性必须遍历全部的 Unicode 标量，来确定字符串的字符数量。

另外需要注意的是通过 `count` 属性返回的字符数量并不总是与包含相同字符的 `NSString` 的 `length` 属性相同。`NSString` 的 `length` 属性是利用 UTF-16 表示的十六位代码单元数字，而不是 Unicode 可扩展的字符群集。

## 访问和修改字符串

你可以通过字符串的属性和方法来访问和修改它，当然也可以用下标语法完成。

### 字符串索引

每一个 `String` 值都有一个关联的索引 (`index`) 类型，`String.Index`，它对应着字符串中的每一个 `Character` 的位置。

前面提到，不同的字符可能会占用不同数量的内存空间，所以要知道 `Character` 的确定位置，就必须从 `String` 开头遍历每一个 Unicode 标量直到结尾。因此，Swift 的字符串不能用整数 (integer) 做索引。

使用 `startIndex` 属性可以获取一个 `String` 的第一个 `Character` 的索引。使用 `endIndex` 属性可以获取最后一个 `Character` 的后一个位置的索引。因此，`endIndex` 属性不能作为一个字符串的有效下标。如果 `String` 是空串，`startIndex` 和 `endIndex` 是相等的。

通过调用 `String` 的 `index(before:)` 或 `index(after:)` 方法，可以立即得到前面或后面的一个索引。你还可以通过调用 `index(_:offsetBy:)` 方法来获取对应偏移量的索引，这种方式可以避免多次调用 `index(before:)` 或 `index(after:)` 方法。

你可以使用下标语法来访问 `String` 特定索引的 `Character`。

```
let greeting = "Guten Tag!"  
greeting[greeting.startIndex]  
// G  
greeting[greeting.index(before: greeting.endIndex)]  
// !  
greeting[greeting.index(after: greeting.startIndex)]  
// u  
let index = greeting.index(greeting.startIndex, offsetBy: 7)  
greeting[index]  
// a
```

试图获取越界索引对应的 `Character`，将引发一个运行时错误。

```
greeting[greeting.endIndex] // error  
greeting.index(after: endIndex) // error
```

使用 `indices` 属性会创建一个包含全部索引的范围（`Range`），用来在一个字符串中访问单个字符。

```
for index in greeting.indices {  
    print("\(greeting[index]) ", terminator: "")  
}  
// 打印输出“G u t e n  T a g ! ”
```

### 注意

你可以使用 `startIndex` 和 `endIndex` 属性或者 `index(before:)`、`index(after:)` 和 `index(_:offsetBy:)` 方法在任意一个确认的并遵循 `Collection` 协议的类型里面，如上文所示是使用在 `String` 中，你也可以使用在 `Array`、`Dictionary` 和 `Set` 中。

## 插入和删除

调用 `insert(_:at:)` 方法可以在一个字符串的指定索引插入一个字符，调用 `insert(contentsOf:at:)` 方法可以在一个字符串的指定索引插入一个段字符串。

```
var welcome = "hello"  
welcome.insert("!", at: welcome.endIndex)  
// welcome 变量现在等于 "hello!"
```

```
welcome.insert(contentsOf: " there", at: welcome.index(before: welcome.endIndex))  
// welcome 变量现在等于 "hello there!"
```

调用 `remove(at:)` 方法可以在一个字符串的指定索引删除一个字符，调用 `removeSubrange(_)` 方法可以在一个字符串的指定索引删除一个子字符串。

```
welcome.remove(at: welcome.index(before: welcome.endIndex))  
// welcome 现在等于 "hello there"
```

```
let range = welcome.index(welcome.endIndex, offsetBy: -6)..<welcome.endIndex  
welcome.removeSubrange(range)  
// welcome 现在等于 "hello"
```

### 注意

你可以使用 `insert(_:at:)`、`insert(contentsOf:at:)`、`remove(at:)` 和 `removeSubrange(_)` 方法在任意一个确认的并遵循 `RangeReplaceableCollection` 协议的类型里面，如上文所示是使用在 `String` 中，你也可以使用在 `Array`、`Dictionary` 和 `Set` 中。

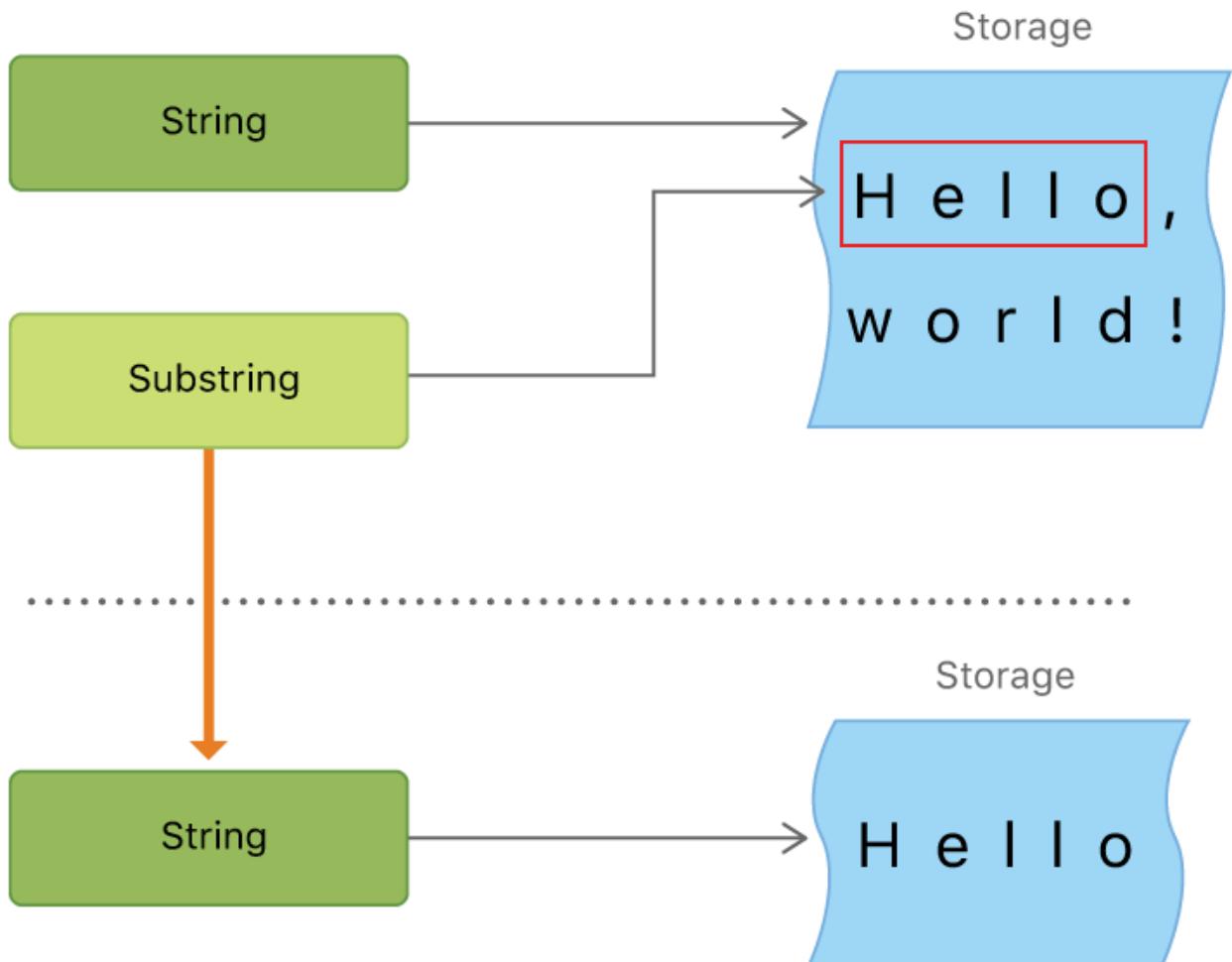
## 子字符串

当你从字符串中获取一个子字符串 —— 例如，使用下标或者 `prefix(_:)` 之类的方法 —— 就可以得到一个 `SubString` 的实例，而非另外一个 `String`。Swift 里的 `SubString` 绝大部分函数都跟 `String` 一样，意味着你可以使用同样的方式去操作 `SubString` 和 `String`。然而，跟 `String` 不同的是，你只有在短时间内需要操作字符串时，才会使用 `SubString`。当你需要长时间保存结果时，就把 `SubString` 转化为 `String` 的实例：

```
let greeting = "Hello, world!"  
let index = greeting.firstIndex(of: ",") ?? greeting.endIndex  
let beginning = greeting[..// beginning 的值为 "Hello"  
  
// 把结果转化为 String 以便长期存储。  
let newString = String(beginning)
```

就像 `String`，每一个 `SubString` 都会在内存里保存字符集。而 `String` 和 `SubString` 的区别在于性能优化上，`SubString` 可以重用原 `String` 的内存空间，或者另一个 `SubString` 的内存空间（`String` 也有同样的优化，但如果两个 `String` 共享内存的话，它们就会相等）。这一优化意味着你在修改 `String` 和 `SubString` 之前都不需要消耗性能去复制内存。就像前面说的那样，`SubString` 不适合长期存储 —— 因为它重用了原 `String` 的内存空间，原 `String` 的内存空间必须保留直到它的 `SubString` 不再被使用为止。

上面的例子，`greeting` 是一个 `String`，意味着它在内存里有一片空间保存字符集。而由于 `beginning` 是 `greeting` 的 `SubString`，它重用了 `greeting` 的内存空间。相反，`newString` 是一个 `String` —— 它是使用 `SubString` 创建的，拥有一片自己的内存空间。下面的图展示了他们之间的关系：



注意

String 和 SubString 都遵循 `StringProtocol</apple_ref/swift/intf/s:s14StringProtocolP>` 协议，这意味着操作字符串的函数使用 `StringProtocol` 会更加方便。你可以传入 String 或 SubString 去调用函数。

## 比较字符串

Swift 提供了三种方式来比较文本值：字符串字符相等、前缀相等和后缀相等。

### 字符串/字符相等

字符串/字符可以用等于操作符 (`==`) 和不等于操作符 (`!=`)，详细描述在 [比较运算符](#)：

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

如果两个字符串（或者两个字符）的可扩展的字形群集是标准相等，那就认为它们是相等的。只要可扩展的字形群集有同样的语言意义和外观则认为它们标准相等，即使它们是由不同的 Unicode 标量构成。

例如，`LATIN SMALL LETTER E WITH ACUTE` (`U+00E9`) 就是标准相等于 `LATIN SMALL LETTER E` (`U+0065`) 后面加上 `COMBINING ACUTE ACCENT` (`U+0301`)。这两个字符群集都是表示字符 `é` 的有效方式，所以它们被认为是标准相等的：

```
// "Voulez-vous un café?" 使用 LATIN SMALL LETTER E WITH ACUTE
let eAcuteQuestion = "Voulez-vous un caf\u{E9}?"

// "Voulez-vous un café?" 使用 LATIN SMALL LETTER E and COMBINING ACUTE ACCENT
let combinedEAcuteQuestion = "Voulez-vous un caf\u{65}\u{301}?"

if eAcuteQuestion == combinedEAcuteQuestion {
    print("These two strings are considered equal")
}
// 打印输出"These two strings are considered equal"
```

相反，英语中的 `LATIN CAPITAL LETTER A` (`U+0041`，或者 `A`) 不等于俄语中的 `CYRILLIC CAPITAL LETTER A` (`U+0410`，或者 `А`)。两个字符看着是一样的，但却有不同的语言意义：

```
let latinCapitalLetterA: Character = "\u{41}"

let cyrillicCapitalLetterA: Character = "\u{0410}"

if latinCapitalLetterA != cyrillicCapitalLetterA {
    print("These two characters are not equivalent")
}
// 打印"These two characters are not equivalent"
```

注意

在 Swift 中，字符串和字符并不区分地域（not locale-sensitive）。

## 前缀/后缀相等

通过调用字符串的 `hasPrefix(_)` / `hasSuffix(_)` 方法来检查字符串是否拥有特定前缀/后缀，两个方法均接收一个 `String` 类型的参数，并返回一个布尔值。

下面的例子以一个字符串数组表示莎士比亚话剧《罗密欧与朱丽叶》中前两场的场景位置：

```
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"
]
```

你可以调用 `hasPrefix(_)` 方法来计算话剧中第一幕的场景数：

```
var act1SceneCount = 0
for scene in romeoAndJuliet {
    if scene.hasPrefix("Act 1 ") {
        act1SceneCount += 1
    }
}
print("There are \act1SceneCount scenes in Act 1")
// 打印输出“There are 5 scenes in Act 1”
```

相似地，你可以用 `hasSuffix(_)` 方法来计算发生在不同地方的场景数：

```
var mansionCount = 0
var cellCount = 0
for scene in romeoAndJuliet {
    if scene.hasSuffix("Capulet's mansion") {
        mansionCount += 1
    } else if scene.hasSuffix("Friar Lawrence's cell") {
        cellCount += 1
    }
}
print("\mansionCount mansion scenes; \cellCount cell scenes")
// 打印输出“6 mansion scenes; 2 cell scenes”
```

注意

`hasPrefix(_)` 和 `hasSuffix(_)` 方法都是在每个字符串中逐字符比较其可扩展的字符群集是否标准相等，详细描述在 [字符串/字符相等](#)。

## 字符串的 Unicode 表示形式

当一个 Unicode 字符串被写进文本文件或者其他储存时，字符串中的 Unicode 标量会用 Unicode 定义的几种 [编码格式](#) (encoding forms) 编码。每一个字符串中的小块编码都被称 [代码单元](#) (code units)。这些包括 UTF-8 编码格式 (编码字符串为 8 位的代码单元)，UTF-16 编码格式 (编码字符串为 16 位的代码单元)，以及 UTF-32 编码格式 (编码字符串为 32 位的代码单元)。

Swift 提供了几种不同的方式来访问字符串的 Unicode 表示形式。你可以利用 `for-in` 来对字符串进行遍历，从而以 Unicode 可扩展的字符群集的方式访问每一个 `Character` 值。该过程在 [使用字符](#) 中进行了描述。

另外，能够以其他三种 Unicode 兼容的方式访问字符串的值：

- UTF-8 代码单元集合（利用字符串的 `utf8` 属性进行访问）
- UTF-16 代码单元集合（利用字符串的 `utf16` 属性进行访问）
- 21 位的 Unicode 标量值集合，也就是字符串的 UTF-32 编码格式（利用字符串的 `unicodeScalars` 属性进行访问）

下面由 `D` , `o` , `g` , `!!` ( `DOUBLE EXCLAMATION MARK` , Unicode 标量 `U+203C` )和 `⌚` ( `DOG FACE` , Unicode 标量为 `U+1F436` )组成的字符串中的每一个字符代表着一种不同的表示：

```
let dogString = "Dog!!⌚"
```

## UTF-8 表示

你可以通过遍历 `String` 的 `utf8` 属性来访问它的 `UTF-8` 表示。其为 `String.UTF8View` 类型的属性，`UTF8View` 是无符号 8 位（`UInt8`）值的集合，每一个 `UInt8` 值都是一个字符的 `UTF-8` 表示：

Character	<code>D</code>	<code>o</code>	<code>g</code>	<code>!!</code>	<code>⌚</code>					
UTF-8 Code Unit	68	111	103	226 128 188	240 159 144 182					
Position	0	1	2	3	4	5	6	7	8	9

```
for codeUnit in dogString.utf8 {  
    print("\(codeUnit) ", terminator: "")  
}  
print("")  
// 68 111 103 226 128 188 240 159 144 182
```

上面的例子中，前三个 `10` 进制 `codeUnit` 值（`68`、`111`、`103`）代表了字符 `D`、`o` 和 `g`，它们的 `UTF-8` 表示与 ASCII 表示相同。接下来的三个 `10` 进制 `codeUnit` 值（`226`、`128`、`188`）是 `DOUBLE EXCLAMATION MARK` 的 3 字节 `UTF-8` 表示。最后的四个 `codeUnit` 值（`240`、`159`、`144`、`182`）是 `DOG FACE` 的 4 字节 `UTF-8` 表示。

## UTF-16 表示

你可以通过遍历 `String` 的 `utf16` 属性来访问它的 `UTF-16` 表示。其为 `String.UTF16View` 类型的属性，`UTF16View` 是无符号16位（`UInt16`）值的集合，每一个 `UInt16` 都是一个字符的 `UTF-16` 表示：

Character	D U+0044	o U+006F	g U+0067	!! U+203C	⌚ U+1F436	
UTF-16 Code Unit	68	111	103	8252	55357	56374
Position	0	1	2	3	4	5

```
for codeUnit in dogString.utf16 {  
    print("\u2028(codeUnit) ", terminator: "")  
}  
print("")  
// 68 111 103 8252 55357 56374
```

同样，前三个 `codeUnit` 值（`68`、`111`、`103`）代表了字符 `D`、`o` 和 `g`，它们的 `UTF-16` 代码单元和 `UTF-8` 完全相同（因为这些 `Unicode` 标量表示 `ASCII` 字符）。

第四个 `codeUnit` 值（`8252`）是一个等于十六进制 `203C` 的十进制值。这个代表了 `DOUBLE EXCLAMATION MARK` 字符的 `Unicode` 标量值 `U+203C`。这个字符在 `UTF-16` 中可以用一个代码单元表示。

第五和第六个 `codeUnit` 值（`55357` 和 `56374`）是 `DOG FACE` 字符的 `UTF-16` 表示。第一个值为 `U+D83D`（十进制值为 `55357`），第二个值为 `U+DC36`（十进制值为 `56374`）。

## Unicode 标量表示

你可以通过遍历 `String` 值的 `unicodeScalars` 属性来访问它的 `Unicode` 标量表示。其为 `UnicodeScalarView` 类型的属性，`UnicodeScalarView` 是 `UnicodeScalar` 类型的值的集合。

每一个 `UnicodeScalar` 拥有一个 `value` 属性，可以返回对应的 21 位数值，用 `UInt32` 来表示：

Character	D U+0044	o U+006F	g U+0067	!! U+203C	⌚ U+1F436	
Value	68	111	103	8252	55357	56374

Unicode Scalar Code Unit	68	111	103	8252	128054
Position	0	1	2	3	4

```
for scalar in dogString.unicodeScalars {  
    print("\(scalar.value) ", terminator: "")  
}  
print("")  
// 68 111 103 8252 128054
```

前三个 `UnicodeScalar` 值（`68`、`111`、`103`）的 `value` 属性仍然代表字符 `D`、`o` 和 `g`。

第四个 `codeUnit` 值（`8252`）仍然是一个等于十六进制 `203C` 的十进制值。这个代表了 `DOUBLE EXCLAMATION MARK` 字符的 Unicode 标量 `U+203C`。

第五个 `UnicodeScalar` 值的 `value` 属性，`128054`，是一个十六进制 `1F436` 的十进制表示。其等同于 `DOG FACE` 的 Unicode 标量 `U+1F436`。

作为查询它们的 `value` 属性的一种替代方法，每个 `UnicodeScalar` 值也可以用来构建一个新的 `String` 值，比如在字符串插值中使用：

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

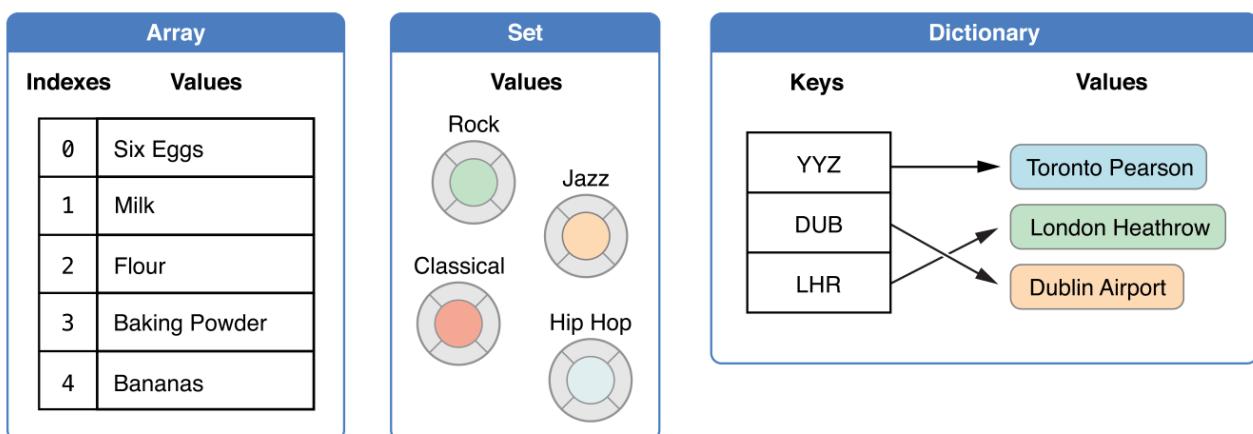
# 集合类型 · GitBook



[runoob.com/manual/gitbook/swift5/source/\\_book/chapter2/04\\_Collection\\_Types.html](http://runoob.com/manual/gitbook/swift5/source/_book/chapter2/04_Collection_Types.html)

## 集合类型

Swift 语言提供 `Arrays`、`Sets` 和 `Dictionaries` 三种基本的集合类型用来存储集合数据。数组 (`Arrays`) 是有序数据的集。集合 (`Sets`) 是无序无重复数据的集。字典 (`Dictionaries`) 是无序的键值对的集。



Swift 语言中的 `Arrays`、`Sets` 和 `Dictionaries` 中存储的数据值类型必须明确。这意味着我们不能把错误的数据类型插入其中。同时这也说明你完全可以对取回值的类型非常放心。

### 注意

Swift 的 `Arrays`、`Sets` 和 `Dictionaries` 类型被实现为泛型集合。更多关于泛型类型和集合，参见 [泛型](#) 章节。

## 集合的可变性

如果创建一个 `Arrays`、`Sets` 或 `Dictionaries` 并且把它分配成一个变量，这个集合将会是可变的。这意味着你可以在创建之后添加更多或移除已存在的数据项，或者改变集合中的数据项。如果我们把 `Arrays`、`Sets` 或 `Dictionaries` 分配成常量，那么它就是不可变的，它的大小和内容都不能被改变。

### 注意

在我们不需要改变集合的时候创建不可变集合是很好的实践。如此 Swift 编译器可以优化我们创建的集合。

## 数组 (`Arrays`)

数组使用有序列表存储同一类型的多个值。相同的值可以多次出现在一个数组的不同位置中。

## 注意

Swift 的 `Array` 类型被桥接到 `Foundation` 中的 `NSArray` 类。更多关于在 `Foundation` 和 `Cocoa` 中使用 `Array` 的信息，参见 [Using Swift with Cocoa and Objective-C\(Swift 4.1\)](#) 中 使用 Cocoa 数据类型 部分。

## 数组的简单语法

写 Swift 数组应该遵循像 `Array<Element>` 这样的形式，其中 `Element` 是这个数组中唯一允许存在的数据类型。我们也可以使用像 `[Element]` 这样的简单语法。尽管两种形式在功能上是一样的，但是推荐较短的那种，而且在本文中都会使用这种形式来使用数组。

## 创建一个空数组

我们可以使用构造语法来创建一个由特定数据类型构成的空数组：

```
var someInts = [Int]()
print("someInts is of type [Int] with \(someInts.count) items.")
// 打印“someInts is of type [Int] with 0 items.”
```

注意，通过构造函数的类型，`someInts` 的值类型被推断为 `[Int]`。

或者，如果代码上下文中已经提供了类型信息，例如一个函数参数或者一个已经定义好类型的常量或者变量，我们可以使用空数组语句创建一个空数组，它的写法很简单：`[]`（一对空方括号）：

```
someInts.append(3)
// someInts 现在包含一个 Int 值
someInts = []
// someInts 现在是空数组，但是仍然是 [Int] 类型的。
```

## 创建一个带有默认值的数组

Swift 中的 `Array` 类型还提供一个可以创建特定大小并且所有数据都被默认的构造方法。我们可以把准备加入新数组的数据项数量（`count`）和适当类型的初始值（`repeating`）传入数组构造函数：

```
var threeDoubles = Array(repeating: 0.0, count: 3)
// threeDoubles 是一种 [Double] 数组，等价于 [0.0, 0.0, 0.0]
```

## 通过两个数组相加创建一个数组

我们可以使用加法操作符（`+`）来组合两种已存在的相同类型数组。新数组的数据类型会被从两个数组的数据类型中推断出来：

```
var anotherThreeDoubles = Array(repeating: 2.5, count: 3)
// anotherThreeDoubles 被推断为 [Double]，等价于 [2.5, 2.5, 2.5]
```

```
var sixDoubles = threeDoubles + anotherThreeDoubles
// sixDoubles 被推断为 [Double]，等价于 [0.0, 0.0, 0.0, 2.5, 2.5, 2.5]
```

## 用数组字面量构造数组

---

我们可以使用数组字面量来进行数组构造，这是一种用一个或者多个数值构造数组的简单方法。数组字面量是一系列由逗号分割并由方括号包含的数值：

[value 1, value 2, value 3]。

下面这个例子创建了一个叫做 `shoppingList` 并且存储 `String` 的数组：

```
var shoppingList: [String] = ["Eggs", "Milk"]
// shoppingList 已经被构造并且拥有两个初始项。
```

`shoppingList` 变量被声明为“字符串值类型的数组”，记作 `[String]`。因为这个数组被规定只有 `String` 一种数据结构，所以只有 `String` 类型可以在其中被存取。在这里，`shoppingList` 数组由两个 `String` 值（`"Eggs"` 和 `"Milk"`）构造，并且由数组字面量定义。

注意

`shoppingList` 数组被声明为变量（`var` 关键字创建）而不是常量（`let` 创建）是因为以后可能会有更多的数据项被插入其中。

在这个例子中，字面量仅仅包含两个 `String` 值。匹配了该数组的变量声明（只能包含 `String` 的数组），所以这个字面量的分配过程可以作为用两个初始项来构造 `shoppingList` 的一种方式。

由于 Swift 的类型推断机制，当我们用字面量构造只拥有相同类型值数组的时候，我们不必把数组的类型定义清楚。`shoppingList` 的构造也可以这样写：

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

因为所有数组字面量中的值都是相同的类型，Swift 可以推断出 `[String]` 是 `shoppingList` 中变量的正确类型。

## 访问和修改数组

---

我们可以通过数组的方法和属性来访问和修改数组，或者使用下标语法。

可以使用数组的只读属性 `count` 来获取数组中的数据项数量：

```
print("The shopping list contains \(shoppingList.count) items.")
// 输出"The shopping list contains 2 items." (这个数组有2个项)
```

使用布尔属性 `isEmpty` 作为一个缩写形式去检查 `count` 属性是否为 `0`：

```
if shoppingList.isEmpty {
    print("The shopping list is empty.")
} else {
    print("The shopping list is not empty.")
}
// 打印"The shopping list is not empty." (shoppinglist 不是空的)
```

也可以使用 `append(_:)` 方法在数组后面添加新的数据项：

```
shoppingList.append("Flour")
// shoppingList 现在有3个数据项，有人在摊煎饼
```

除此之外，使用加法赋值运算符（`+=`）也可以直接在数组后面添加一个或多个拥有相同类型的数据项：

```
shoppingList += ["Baking Powder"]
// shoppingList 现在有四项了
shoppingList += ["Chocolate Spread", "Cheese", "Butter"]
// shoppingList 现在有七项了
```

可以直接使用下标语法来获取数组中的数据项，把我们需要的数据项的索引值放在直接放在数组名称的方括号中：

```
var firstItem = shoppingList[0]
// 第一项是“Eggs”
```

注意

第一项在数组中的索引值是 `0` 而不是 `1`。Swift 中的数组索引总是从零开始。

我们也可以用下标来改变某个已有索引值对应的数据值：

```
shoppingList[0] = "Six eggs"
// 其中的第一项现在是“Six eggs”而不是“Eggs”
```

还可以利用下标来一次改变一系列数据值，即使新数据和原有数据的数量是不一样的。

下面的例子把 `"Chocolate Spread"`、`"Cheese"` 和 `"Butter"` 替换为 `"Bananas"` 和 `"Apples"`：

```
shoppingList[4...6] = ["Bananas", "Apples"]
// shoppingList 现在有6项
```

注意

不可以用下标访问的形式去在数组尾部添加新项。

调用数组的 `insert(_:at:)` 方法来在某个具体索引值之前添加数据项：

```
shoppingList.insert("Maple Syrup", at: 0)
// shoppingList 现在有7项
// 现在是这个列表中的第一项是“Maple Syrup”
```

这次 `insert(_:at:)` 方法调用把值为 `"Maple Syrup"` 的新数据项插入列表的最开始位置，并且使用 `0` 作为索引值。

类似的我们可以使用 `remove(at:)` 方法来移除数组中的某一项。这个方法把数组在特定索引值中存储的数据项移除并且返回这个被移除的数据项（我们不需要的时候就可以无视它）：

```
let mapleSyrup = shoppingList.remove(at: 0)
// 索引值为0的数据项被移除
// shoppingList 现在只有6项，而且不包括 Maple Syrup
// mapleSyrup 常量的值等于被移除数据项“Maple Syrup”的值
```

### 注意

如果我们试着对索引越界的数组进行检索或者设置新值的操作，会引发一个运行期错误。我们可以使用索引值和数组的 `count` 属性进行比较来在使用某个索引之前先检验是否有效。除了当 `count` 等于 0 时（说明这是个空数组），最大索引值一直是 `count - 1`，因为数组都是零起索引。

数据项被移除后数组中的空出项会被自动填补，所以现在索引值为 0 的数据项的值再次等于 “Six eggs”：

```
firstItem = shoppingList[0]
// firstItem 现在等于“Six eggs”
```

如果我们只想把数组中的最后一项移除，可以使用 `removeLast()` 方法而不是 `remove(at:)` 方法来避免我们需要获取数组的 `count` 属性。就像后者一样，前者也会返回被移除的数据项：

```
let apples = shoppingList.removeLast()
// 数组的最后一项被移除了
// shoppingList 现在只有5项，不包括 Apples
// apples 常量的值现在等于“Apples”字符串
```

## 数组的遍历

我们可以使用 `for-in` 循环来遍历所有数组中的数据项：

```
for item in shoppingList {
    print(item)
}
// Six eggs
// Milk
// Flour
// Baking Powder
// Bananas
```

如果我们同时需要每个数据项的值和索引值，可以使用 `enumerated()` 方法来进行数组遍历。`enumerated()` 返回一个由每一个数据项索引值和数据值组成的元组。我们可以把这个元组分解成临时常量或者变量来进行遍历：

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

更多关于 `for-in` 循环的介绍请参见 [For 循环](#)。

## 集合 (Sets)

集合 (`Set`) 用来存储相同类型并且没有确定顺序的值。当集合元素顺序不重要时或者希望确保每个元素只出现一次时可以使用集合而不是数组。

注意 Swift 的 `Set` 类型被桥接到 `Foundation` 中的 `NSSet` 类。

关于使用 `Foundation` 和 `Cocoa` 中 `Set` 的知识，参见 [\*Using Swift with Cocoa and Objective-C\(Swift 4.1\)\*](#) 中使用 `Cocoa` 数据类型部分。

### 集合类型的哈希值

一个类型为了存储在集合中，该类型必须是可哈希化的——也就是说，该类型必须提供一个方法来计算它的哈希值。一个哈希值是 `Int` 类型的，相等的对象哈希值必须相同，比如 `a==b`，因此必须 `a.hashValue == b.hashValue`。

Swift 的所有基本类型（比如 `String`、`Int`、`Double` 和 `Bool`）默认都是可哈希化的，可以作为集合的值的类型或者字典的键的类型。没有关联值的枚举成员值（在 [枚举](#) 有讲述）默认也是可哈希化的。

注意

你可以使用你自定义的类型作为集合的值的类型或者是字典的键的类型，但你需要使你的自定义类型遵循 Swift 标准库中的 `Hashable` 协议。遵循 `Hashable` 协议的类型需要提供一个类型为 `Int` 的可读属性 `hashValue`。由类型的 `hashValue` 属性返回的值不需要在同一程序的不同执行周期或者不同程序之间保持相同。

因为 `Hashable` 协议遵循 `Equatable` 协议，所以遵循该协议的类型也必须提供一个“是否相等”运算符（`==`）的实现。这个 `Equatable` 协议要求任何遵循 `==` 实现的实例间都是一种相等的关系。也就是说，对于 `a,b,c` 三个值来说，`==` 的实现必须满足下面三种情况：

- `a == a` (自反性)
- `a == b` 意味着 `b == a` (对称性)
- `a == b && b == c` 意味着 `a == c` (传递性)

关于遵循协议的更多信息，请看 [协议](#)。

### 集合类型语法

Swift 中的 `Set` 类型被写为 `Set<Element>`，这里的 `Element` 表示 `Set` 中允许存储的类型，和数组不同的是，集合没有等价的简化形式。

### 创建和构造一个空的集合

你可以通过构造器语法创建一个特定类型的空集合：

```
var letters = Set<Character>()
print("letters is of type Set<Character> with \(letters.count) items.")
// 打印“letters is of type Set<Character> with 0 items.”
```

注意

通过构造器，这里的 `letters` 变量的类型被推断为 `Set<Character>`。

此外，如果上下文提供了类型信息，比如作为函数的参数或者已知类型的变量或常量，我们可以通过一个空的数组字面量创建一个空的 `Set`：

```
letters.insert("a")
// letters 现在含有1个 Character 类型的值
letters = []
// letters 现在是一个空的 Set，但是它依然是 Set<Character> 类型
```

## 用数组字面量创建集合

你可以使用数组字面量来构造集合，并且可以使用简化形式写一个或者多个值作为集合元素。

下面的例子创建一个称之为 `favoriteGenres` 的集合来存储 `String` 类型的值：

```
var favoriteGenres: Set<String> = ["Rock", "Classical", "Hip hop"]
// favoriteGenres 被构造成含有三个初始值的集合
```

这个 `favoriteGenres` 变量被声明为“一个 `String` 值的集合”，写为 `Set<String>`。由于这个特定的集合含有指定 `String` 类型的值，所以它只允许存储 `String` 类型值。这里的 `favoriteGenres` 变量有三个 `String` 类型的初始值（`"Rock"`，`"Classical"` 和 `"Hip hop"`），并以数组字面量的方式出现。

注意

`favoriteGenres` 被声明为一个变量（拥有 `var` 标示符）而不是一个常量（拥有 `let` 标示符），因为它里面的元素将会在下面的例子中被增加或者移除。

一个 `Set` 类型不能从数组字面量中被单独推断出来，因此 `Set` 类型必须显式声明。然而，由于 Swift 的类型推断功能，如果你想使用一个数组字面量构造一个 `Set` 并且该数组字面量中的所有元素类型相同，那么你无须写出 `Set` 的具体类型。`favoriteGenres` 的构造形式可以采用简化的方式代替：

```
var favoriteGenres: Set = ["Rock", "Classical", "Hip hop"]
```

由于数组字面量中的所有元素类型相同，Swift 可以推断出 `Set<String>` 作为 `favoriteGenres` 变量的正确类型。

## 访问和修改一个集合

你可以通过 `Set` 的属性和方法来访问和修改一个 `Set`。

为了找出一个 `Set` 中元素的数量，可以使用其只读属性 `count`：

```
print("I have \(favoriteGenres.count) favorite music genres.")  
// 打印"I have 3 favorite music genres."
```

使用布尔属性 `isEmpty` 作为一个缩写形式去检查 `count` 属性是否为 `0` :

```
if favoriteGenres.isEmpty {  
    print("As far as music goes, I'm not picky.")  
} else {  
    print("I have particular music preferences.")  
}  
// 打印"I have particular music preferences."
```

你可以通过调用 `Set` 的 `insert(_)` 方法来添加一个新元素 :

```
favoriteGenres.insert("Jazz")  
// favoriteGenres 现在包含4个元素
```

你可以通过调用 `Set` 的 `remove(_)` 方法去删除一个元素 , 如果该值是该 `Set` 的一个元素则删除该元素并且返回被删除的元素值 , 否则如果该 `Set` 不包含该值 , 则返回 `nil` 。另外 , `Set` 中的所有元素可以通过它的 `removeAll()` 方法删除。

```
if let removedGenre = favoriteGenres.remove("Rock") {  
    print("\(removedGenre)? I'm over it.")  
} else {  
    print("I never much cared for that.")  
}  
// 打印"Rock? I'm over it."
```

使用 `contains(_)` 方法去检查 `Set` 中是否包含一个特定的值 :

```
if favoriteGenres.contains("Funk") {  
    print("I get up on the good foot.")  
} else {  
    print("It's too funky in here.")  
}  
// 打印"It's too funky in here."
```

## 遍历一个集合

---

你可以在一个 `for-in` 循环中遍历一个 `Set` 中的所有值。

```
for genre in favoriteGenres {  
    print("\(genre)")  
}  
// Classical  
// Jazz  
// Hip hop
```

更多关于 `for-in` 循环的信息 , 参见 [For 循环](#)。

Swift 的 `Set` 类型没有确定的顺序 , 为了按照特定顺序来遍历一个 `Set` 中的值可以使用 `sorted()` 方法 , 它将返回一个有序数组 , 这个数组的元素排列顺序由操作符'<'对元素进行比较的结果来确定。

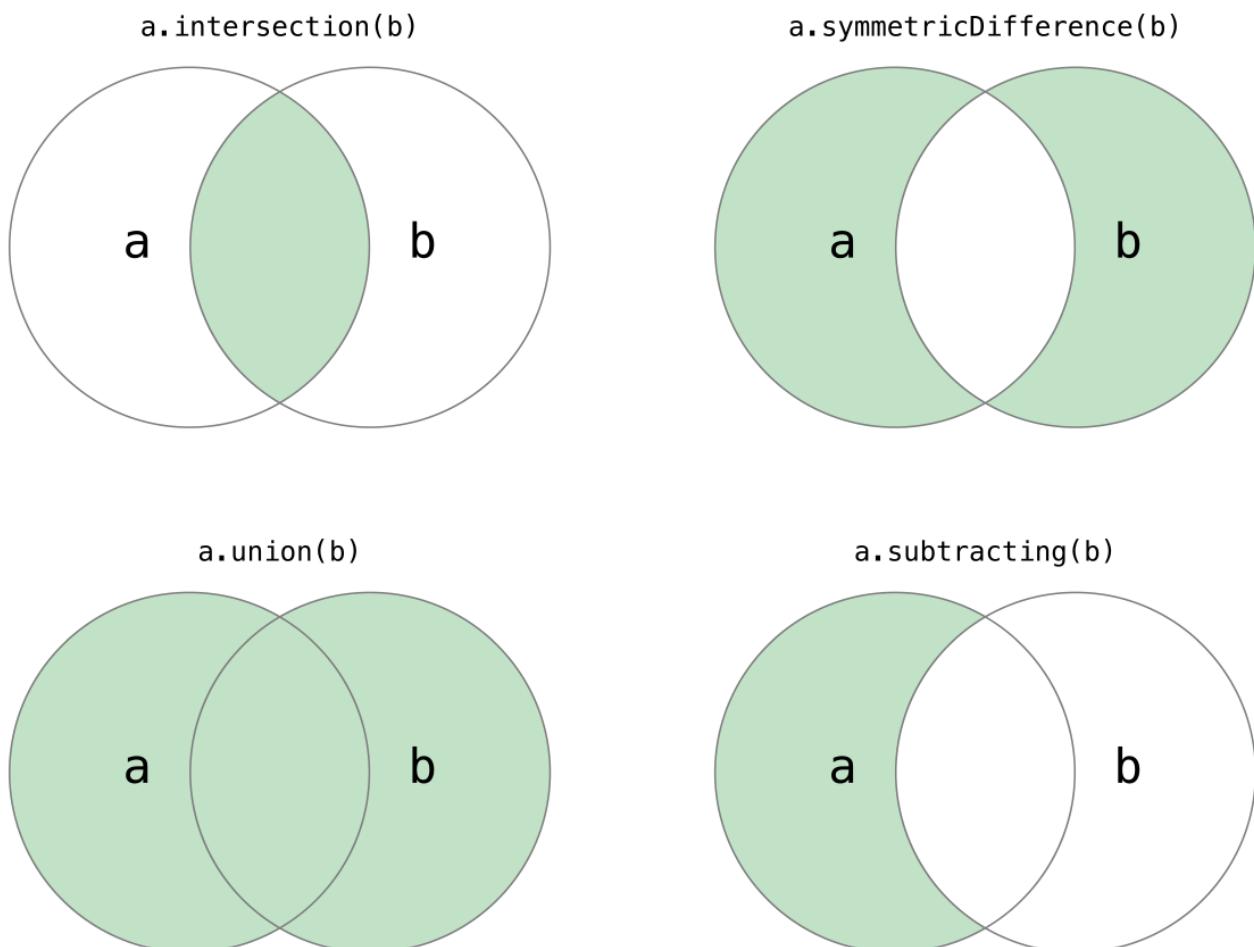
```
for genre in favoriteGenres.sorted() {  
    print("\(genre)")  
}  
// Classical  
// Hip hop  
// Jazz
```

## 集合操作

你可以高效地完成 `Set` 的一些基本操作，比如把两个集合组合到一起，判断两个集合共有元素，或者判断两个集合是否全包含，部分包含或者不相交。

### 基本集合操作

下面的插图描述了两个集合 `a` 和 `b`，以及通过阴影部分的区域显示集合各种操作的结果。



- 使用 `intersection(_:)` 方法根据两个集合中都包含的值创建一个新的集合。
- 使用 `symmetricDifference(_:)` 方法根据在一个集合中但不在两个集合中的值创建一个新的集合。
- 使用 `union(_:)` 方法根据两个集合的值创建一个新的集合。
- 使用 `subtracting(_:)` 方法根据不在该集合中的值创建一个新的集合。

```

let oddDigits: Set = [1, 3, 5, 7, 9]
let evenDigits: Set = [0, 2, 4, 6, 8]
let singleDigitPrimeNumbers: Set = [2, 3, 5, 7]

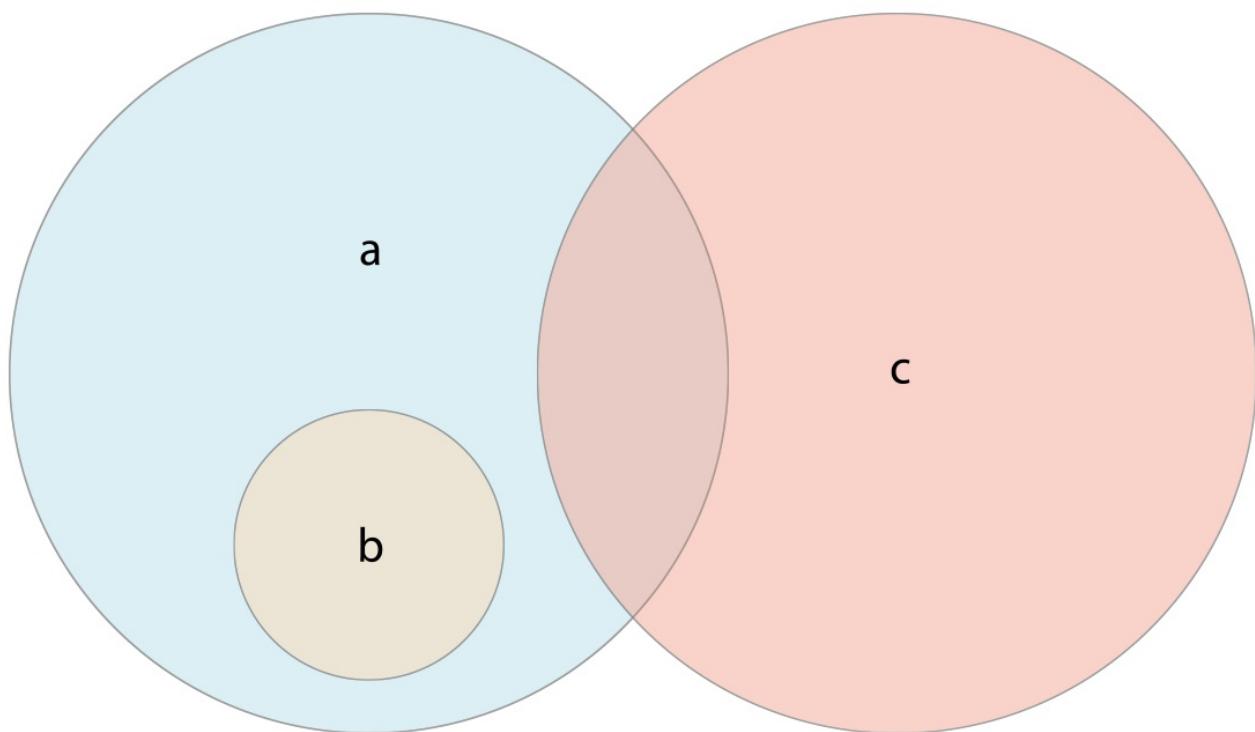
oddDigits.union(evenDigits).sorted()
// [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
oddDigits.intersection(evenDigits).sorted()
// []
oddDigits.subtracting(singleDigitPrimeNumbers).sorted()
// [1, 9]
oddDigits.symmetricDifference(singleDigitPrimeNumbers).sorted()
// [1, 2, 9]

```

## 集合成员关系和相等

---

下面的插图描述了三个集合 `a`、`b` 和 `c`，以及通过重叠区域表述集合间共享的元素。集合 `a` 是集合 `b` 的父集合，因为 `a` 包含了 `b` 中所有的元素，相反的，集合 `b` 是集合 `a` 的子集合，因为属于 `b` 的元素也被 `a` 包含。集合 `b` 和集合 `c` 彼此不关联，因为它们之间没有共同的元素。



- 使用“是否相等”运算符（`==`）来判断两个集合是否包含全部相同的值。
- 使用 `isSubset(of:)` 方法来判断一个集合中的值是否也被包含在另外一个集合中。
- 使用 `isSuperset(of:)` 方法来判断一个集合中包含另一个集合中所有的值。
- 使用 `isStrictSubset(of:)` 或者 `isStrictSuperset(of:)` 方法来判断一个集合是否是另外一个集合的子集合或者父集合并且两个集合并不相等。
- 使用 `isDisjoint(with:)` 方法来判断两个集合是否不含有相同的值（是否没有交集）。

```
let houseAnimals: Set = ["🐶", "🐱"]
let farmAnimals: Set = ["🐮", "🐔", "🐑", "🐹", "🐰"]
let cityAnimals: Set = ["🐀", "🐦"]

houseAnimals.isSubset(of: farmAnimals)
// true
farmAnimals.isSuperset(of: houseAnimals)
// true
farmAnimals.isDisjoint(with: cityAnimals)
// true
```

## 字典

字典是一种存储多个相同类型的值的容器。每个值 (value) 都关联唯一的键 (key) ，键作为字典中的这个值数据的标识符。和数组中的数据项不同，字典中的数据项并没有具体顺序。我们在需要通过标识符 (键) 访问数据的时候使用字典，这种方法很大程度上和我们在现实世界中使用字典查字义的方法一样。

注意

Swift 的 `Dictionary` 类型被桥接到 `Foundation` 的 `NSDictionary` 类。

更多关于在 `Foundation` 和 `Cocoa` 中使用 `Dictionary` 类型的信息，参见 [Using Swift with Cocoa and Objective-C\(Swift 4.1\)](#) 中 使用 Cocoa 数据类型 部分。

## 字典类型简化语法

Swift 的字典使用 `Dictionary<Key, Value>` 定义，其中 `Key` 是字典中键的数据类型，`Value` 是字典中对应于这些键所存储值的数据类型。

注意

一个字典的 `Key` 类型必须遵循 `Hashable` 协议，就像 `Set` 的值类型。

我们也可以用 `[Key: Value]` 这样简化的形式去创建一个字典类型。虽然这两种形式功能上相同，但是后者是首选，并且这本指导书涉及到字典类型时通篇采用后者。

## 创建一个空字典

我们可以像数组一样使用构造语法创建一个拥有确定类型的空字典：

```
var namesOfIntegers = [Int: String]()
// namesOfIntegers 是一个空的 [Int: String] 字典
```

这个例子创建了一个 `[Int: String]` 类型的空字典来储存整数的英语命名。它的键是 `Int` 型，值是 `String` 型。

如果上下文已经提供了类型信息，我们可以使用空字典字面量来创建一个空字典，记作 `[]` (中括号中放一个冒号)：

```
namesOfIntegers[16] = "sixteen"
// namesOfIntegers 现在包含一个键值对
namesOfIntegers = [:]
// namesOfIntegers 又成为了一个 [Int: String] 类型的空字典
```

## 用字典字面量创建字典

我们可以使用字典字面量来构造字典，这和我们刚才介绍过的数组字面量拥有相似语法。字典字面量是一种将一个或多个键值对写作 `Dictionary` 集合的快捷途径。

一个键值对是一个 `key` 和一个 `value` 的结合体。在字典字面量中，每一个键值对的键和值都由冒号分割。这些键值对构成一个列表，其中这些键值对由方括号包含、由逗号分割：

```
[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称的字典。在这个字典中键是三个字母的国际航空运输相关代码，值是机场名称：

```
var airports: [String: String] = ["YYZ": "Toronto Pearson", "DUB": "Dublin"]
```

`airports` 字典被声明为一种 `[String: String]` 类型，这意味着这个字典的键和值都是 `String` 类型。

注意

`airports` 字典被声明为变量（用 `var` 关键字）而不是常量（`let` 关键字）因为后来更多的机场信息会被添加到这个示例字典中。

`airports` 字典使用字典字面量初始化，包含两个键值对。第一对的键是 `YYZ`，值是 `Toronto Pearson`。第二对的键是 `DUB`，值是 `Dublin`。

这个字典语句包含了两个 `String: String` 类型的键值对。它们对应 `airports` 变量声明的类型（一个只有 `String` 键和 `String` 值的字典）所以这个字典字面量的任务是构造拥有两个初始数据项的 `airport` 字典。

和数组一样，我们在用字典字面量构造字典时，如果它的键和值都有各自一致的类型，那么就不必写出字典的类型。`airports` 字典也可以用这种简短方式定义：

```
var airports = ["YYZ": "Toronto Pearson", "DUB": "Dublin"]
```

因为这个语句中所有的键和值都各自拥有相同的数据类型，Swift 可以推断出 `Dictionary<String, String>` 是 `airports` 字典的正确类型。

## 访问和修改字典

我们可以通过字典的方法和属性来访问和修改字典，或者通过使用下标语法。

和数组一样，我们可以通过字典的只读属性 `count` 来获取某个字典的数据项数量：

```
print("The dictionary of airports contains \(airports.count) items.")
// 打印"The dictionary of airports contains 2 items." (这个字典有两个数据项)
```

使用布尔属性 `isEmpty` 作为一个缩写形式去检查 `count` 属性是否为 `0` :

```
if airports.isEmpty {
    print("The airports dictionary is empty.")
} else {
    print("The airports dictionary is not empty.")
}
// 打印"The airports dictionary is not empty."
```

我们也可以在字典中使用下标语法来添加新的数据项。可以使用一个恰当类型的键作为下标索引，并且分配恰当类型的新值：

```
airports["LHR"] = "London"
// airports 字典现在有三个数据项
```

我们也可以使用下标语法来改变特定键对应的值：

```
airports["LHR"] = "London Heathrow"
// "LHR"对应的值被改为"London Heathrow"
```

作为另一种下标方法，字典的 `updateValue(_:forKey:)` 方法可以设置或者更新特定键对应的值。就像上面所示的下标示例，`updateValue(_:forKey:)` 方法在这个键不存在对应值的时候会设置新值或者在存在时更新已存在的值。和上面的下标方法不同的，`updateValue(_:forKey:)` 这个方法返回更新值之前的原值。这样使得我们可以检查更新是否成功。

`updateValue(_:forKey:)` 方法会返回对应值的类型的可选值。举例来说：对于存储 `String` 值的字典，这个函数会返回一个 `String?` 或者“可选 `String`”类型的值。

如果有值存在于更新前，则这个可选值包含了旧值，否则它将会是 `nil`。

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

我们也可以使用下标语法来在字典中检索特定键对应的值。因为有可能请求的键没有对应的值存在，字典的下标访问会返回对应值的类型的可选值。如果这个字典包含请求键所对应的值，下标会返回一个包含这个存在值的可选值，否则将返回 `nil`：

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

我们还可以使用下标语法来通过给某个键的对应值赋值为 `nil` 来从字典里移除一个键值对：

```
airports["APL"] = "Apple International"
// "Apple International"不是真的 APL 机场，删除它
airports["APL"] = nil
// APL 现在被移除了
```

此外，`removeValue(forKey:)` 方法也可以用来在字典中移除键值对。这个方法在键值对存在的情况下会移除该键值对并且返回被移除的值或者在没有值的情况下返回 `nil`：

```
if let removedValue = airports.removeValue(forKey: "DUB") {
    print("The removed airport's name is \(removedValue).")
} else {
    print("The airports dictionary does not contain a value for DUB.")
}
// 打印"The removed airport's name is Dublin Airport."
```

## 字典遍历

---

我们可以使用 `for-in` 循环来遍历某个字典中的键值对。每一个字典中的数据项都以 `(key, value)` 元组形式返回，并且我们可以使用临时常量或者变量来分解这些元组：

```
for (airportCode, airportName) in airports {
    print("\(airportCode): \(airportName)")
}
// YYZ: Toronto Pearson
// LHR: London Heathrow
```

更多关于 `for-in` 循环的信息，参见 [For 循环](#)。

通过访问 `keys` 或者 `values` 属性，我们也可以遍历字典的键或者值：

```
for airportCode in airports.keys {
    print("Airport code: \(airportCode)")
}
// Airport code: YYZ
// Airport code: LHR

for airportName in airports.values {
    print("Airport name: \(airportName)")
}
// Airport name: Toronto Pearson
// Airport name: London Heathrow
```

如果我们只是需要使用某个字典的键集合或者值集合来作为某个接受 `Array` 实例的 API 的参数，可以直接使用 `keys` 或者 `values` 属性构造一个新数组：

```
let airportCodes = [String](airports.keys)
// airportCodes 是 ["YYZ", "LHR"]

let airportNames = [String](airports.values)
// airportNames 是 ["Toronto Pearson", "London Heathrow"]
```

Swift 的字典类型是无序集合类型。为了以特定的顺序遍历字典的键或值，可以对字典的 `keys` 或 `values` 属性使用 `sorted()` 方法。



# 控制流 · GitBook

---



[runoob.com/manual/gitbook/swift5/source/\\_book/chapter2/05\\_Control\\_Flow.html](https://runoob.com/manual/gitbook/swift5/source/_book/chapter2/05_Control_Flow.html)

## 控制流

---

Swift 提供了多种流程控制结构，包括可以多次执行任务的 `while` 循环，基于特定条件选择执行不同代码分支的 `if`、`guard` 和 `switch` 语句，还有控制流程跳转到其他代码位置的 `break` 和 `continue` 语句。

Swift 还提供了 `for-in` 循环，用来更简单地遍历数组（Array），字典（Dictionary），区间（Range），字符串（String）和其他序列类型。

Swift 的 `switch` 语句比许多类 C 语言要更加强大。`case` 还可以匹配很多不同的模式，包括范围匹配，元组（tuple）和特定类型匹配。`switch` 语句的 `case` 中匹配的值可以声明为临时常量或变量，在 `case` 作用域内使用，也可以配合 `where` 来描述更复杂的匹配条件。

## For-In 循环

---

你可以使用 `for-in` 循环来遍历一个集合中的所有元素，例如数组中的元素、范围内的数字或者字符串中的字符。

以下例子使用 `for-in` 遍历一个数组所有元素：

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

你也可以通过遍历一个字典来访问它的键值对。遍历字典时，字典的每项元素会以 `(key, value)` 元组的形式返回，你可以在 `for-in` 循环中使用显式的常量名称来解读 `(key, value)` 元组。下面的例子中，字典的键声明会为 `animalName` 常量，字典的值会声明为 `legCount` 常量：

```
let numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
for (animalName, legCount) in numberOfLegs {
    print("\(animalName)s have \(legCount) legs")
}
// cats have 4 legs
// ants have 6 legs
// spiders have 8 legs
```

字典的内容理论上是无序的，遍历元素时的顺序是无法确定的。将元素插入字典的顺序并不会决定它们被遍历的顺序。关于数组和字典的细节，参见 [集合类型](#)。

`for-in` 循环还可以使用数字范围。下面的例子用来输出乘法表的一部分内容：

```
for index in 1...5 {  
    print("\(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
```

例子中用来进行遍历的元素是使用闭区间操作符（`...`）表示的从 `1` 到 `5` 的数字区间。`index` 被赋值为闭区间中的第一个数字（`1`），然后循环中的语句被执行一次。在本例中，这个循环只包含一个语句，用来输出当前 `index` 值所对应的乘 `5` 乘法表的结果。该语句执行后，`index` 的值被更新为闭区间中的第二个数字（`2`），之后 `print(_:separator:terminator:)` 函数会再执行一次。整个过程会进行到闭区间结尾为止。

上面的例子中，`index` 是一个每次循环遍历开始时被自动赋值的常量。这种情况下，`index` 在使用前不需要声明，只需要将它包含在循环的声明中，就可以对其进行隐式声明，而无需使用 `let` 关键字声明。

如果你不需要区间序列内每一项的值，你可以使用下划线（`_`）替代变量名来忽略这个值：

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

这个例子计算 `base` 这个数的 `power` 次幂（本例中，是 `3` 的 `10` 次幂），从 `1`（`3` 的 `0` 次幂）开始做 `3` 的乘法，进行 `10` 次，使用 `1` 到 `10` 的闭区间循环。这个计算并不需要知道每一次循环中计数器具体的值，只需要执行了正确的循环次数即可。下划线符号 `_`（替代循环中的变量）能够忽略当前值，并且不提供循环遍历时对值的访问。

在某些情况下，你可能不想使用包括两个端点的闭区间。想象一下，你在一个手表上绘制分钟的刻度线。总共 `60` 个刻度，从 `0` 分开始。使用半开区间运算符（`.. <`）来表示一个左闭右开的区间。有关区间的更多信息，请参阅 [区间运算符](#)。

```
let minutes = 60  
for tickMark in 0..<minutes {  
    // 每一分钟都渲染一个刻度线（60次）  
}
```

一些用户可能在其 UI 中可能需要较少的刻度。他们可以每 5 分钟作为一个刻度。使用 `stride(from:to:by:)` 函数跳过不需要的标记。

```
let minuteInterval = 5
for tickMark in stride(from: 0, to: minutes, by: minuteInterval) {
    // 每5分钟渲染一个刻度线 (0, 5, 10, 15 ... 45, 50, 55)
}
```

可以在闭区间使用 `stride(from:through:by:)` 起到同样作用：

```
let hours = 12
let hourInterval = 3
for tickMark in stride(from: 3, through: hours, by: hourInterval) {
    // 每3小时渲染一个刻度线 (3, 6, 9, 12)
}
```

## While 循环

---

`while` 循环会一直运行一段语句直到条件变成 `false`。这类循环适合使用在第一次迭代前，迭代次数未知的情况下。Swift 提供两种 `while` 循环形式：

- `while` 循环，每次在循环开始时计算条件是否符合；
- `repeat-while` 循环，每次在循环结束时计算条件是否符合。

## While

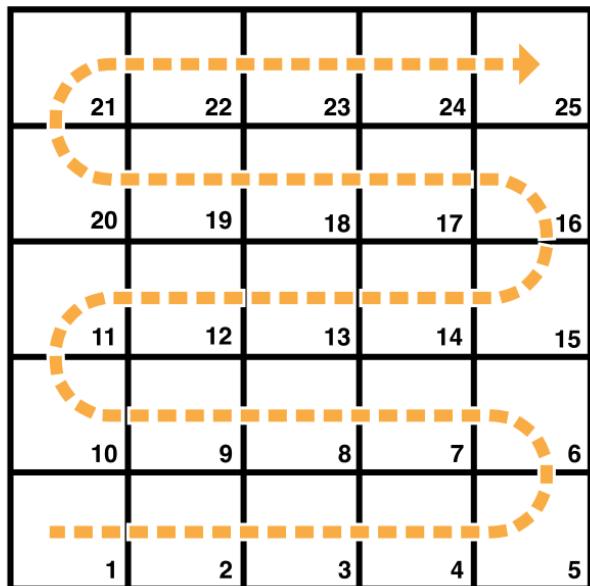
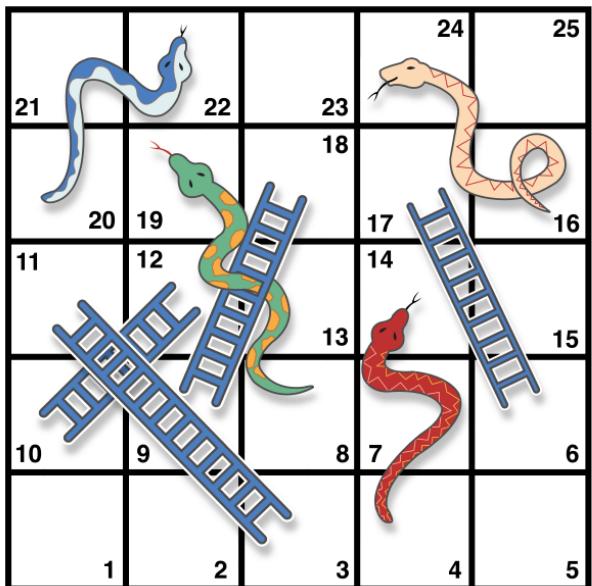
---

`while` 循环从计算一个条件开始。如果条件为 `true`，会重复运行一段语句，直到条件变为 `false`。

下面是 `while` 循环的一般格式：

```
while condition {
    statements
}
```

下面的例子来玩一个叫做蛇和梯子（也叫做滑道和梯子）的小游戏：



游戏的规则如下：

- 游戏盘面包括 25 个方格，游戏目标是达到或者超过第 25 个方格；
- 每一轮，你通过掷一个六面体骰子来确定你移动方块的步数，移动的路线由上图中横向的虚线所示；
- 如果在某轮结束，你移动到了梯子的底部，可以顺着梯子爬上去；
- 如果在某轮结束，你移动到了蛇的头部，你会顺着蛇的身体滑下去。

游戏盘面可以使用一个 Int 数组来表达。数组的长度由一个 finalSquare 常量储存，用来初始化数组和检测最终胜利条件。游戏盘面由 26 个 Int 0 值初始化，而不是 25 个（由 0 到 25，一共 26 个）：

```
let finalSquare = 25
var board = [Int](repeating: 0, count: finalSquare + 1)
```

一些方格被设置成特定的值来表示有蛇或者梯子。梯子底部的方格是一个正值，使你可以向上移动，蛇头处的方格是一个负值，会让你向下移动：

```
board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
```

3 号方格是梯子的底部，会让你向上移动到 11 号方格，我们使用 `board[03]` 等于 `+08`（来表示 `11` 和 `3` 之间的差值）。为了对齐语句，这里使用了一元正运算符（`+i`）和一元负运算符（`-i`），并且小于 10 的数字都使用 0 补齐（这些语法的技巧不是必要的，只是为了让代码看起来更加整洁）。

玩家由左下角空白处编号为 0 的方格开始游戏。玩家第一次掷骰子后才会进入游戏盘面：

```
var square = 0
var diceRoll = 0
while square < finalSquare {
    // 掷骰子
    diceRoll += 1
    if diceRoll == 7 { diceRoll = 1 }
    // 根据点数移动
    square += diceRoll
    if square < board.count {
        // 如果玩家还在棋盘上，顺着梯子爬上去或者顺着蛇滑下去
        square += board[square]
    }
}
print("Game over!")
```

本例中使用了最简单的方法来模拟掷骰子。`diceRoll` 的值并不是一个随机数，而是以 `0` 为初始值，之后每一次 `while` 循环，`diceRoll` 的值增加 `1`，然后检测是否超出了最大值。当 `diceRoll` 的值等于 `7` 时，就超过了骰子的最大值，会被重置为 `1`。所以 `diceRoll` 的取值顺序会一直是 `1, 2, 3, 4, 5, 6, 1, 2` 等。

掷完骰子后，玩家向前移动 `diceRoll` 个方格，如果玩家移动超过了第 `25` 个方格，这个时候游戏将会结束，为了应对这种情况，代码会首先判断 `square` 的值是否小于 `board` 的 `count` 属性，只有小于才会在 `board[square]` 上增加 `square`，来向前或向后移动（遇到了梯子或者蛇）。

### 注意

如果没有这个检测（`square < board.count`），`board[square]` 可能会越界访问 `board` 数组，导致运行时错误。

当本轮 `while` 循环运行完毕，会再检测循环条件是否需要再运行一次循环。如果玩家移动到或者超过第 `25` 个方格，循环条件结果为 `false`，此时游戏结束。

`while` 循环比较适合本例中的这种情况，因为在 `while` 循环开始时，我们并不知道游戏要跑多久，只有在达成指定条件时循环才会结束。

## Repeat-While

`while` 循环的另外一种形式是 `repeat-while`，它和 `while` 的区别是在判断循环条件之前，先执行一次循环的代码块。然后重复循环直到条件为 `false`。

### 注意

Swift 语言的 `repeat-while` 循环和其他语言中的 `do-while` 循环是类似的。

下面是 `repeat-while` 循环的一般格式：

```
repeat {
    statements
} while condition
```

还是蛇和梯子的游戏，使用 `repeat-while` 循环来替代 `while` 循环。`finalSquare`、`board`、`square` 和 `diceRoll` 的值初始化同 `while` 循环时一样：

```
let finalSquare = 25
var board = [Int](repeating: 0, count: finalSquare + 1)
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
```

`repeat-while` 的循环版本，循环中第一步就需要去检测是否在梯子或者蛇的方块上。没有梯子会让玩家直接上到第 25 个方格，所以玩家不会通过梯子直接赢得游戏。这样在循环开始时先检测是否踩在梯子或者蛇上是安全的。

游戏开始时，玩家在第 0 个方格上，`board[0]` 一直等于 0，不会有什影响：

```
repeat {
    // 顺着梯子爬上去或者顺着蛇滑下去
    square += board[square]
    // 掷骰子
    diceRoll += 1
    if diceRoll == 7 { diceRoll = 1 }
    // 根据点数移动
    square += diceRoll
} while square < finalSquare
print("Game over!")
```

检测完玩家是否踩在梯子或者蛇上之后，开始掷骰子，然后玩家向前移动 `diceRoll` 个方格，本轮循环结束。

循环条件（`while square < finalSquare`）和 `while` 方式相同，但是只会在循环结束后进行计算。在这个游戏中，`repeat-while` 表现得比 `while` 循环更好。`repeat-while` 方式会在条件判断 `square` 没有超出后直接运行 `square += board[square]`，这种方式可以比起前面 `while` 循环的版本，可以省去数组越界的检查。

## 条件语句

---

根据特定的条件执行特定的代码通常是十分有用的。当错误发生时，你可能想运行额外的代码；或者，当值太大或太小时，向用户显示一条消息。要实现这些功能，你就需要使用条件语句。

Swift 提供两种类型的条件语句：`if` 语句和 `switch` 语句。通常，当条件较为简单且可能的情况很少时，使用 `if` 语句。而 `switch` 语句更适用于条件较复杂、有更多排列组合的时候。并且 `switch` 在需要用到模式匹配（pattern-matching）的情况下会更有用。

### If

---

`if` 语句最简单的形式就是只包含一个条件，只有该条件为 `true` 时，才执行相关代码：

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

上面的例子会判断温度是否小于等于 32 华氏度（水的冰点）。如果是，则打印一条消息；否则，不打印任何消息，继续执行 `if` 块后面的代码。

当然，`if` 语句允许二选一执行，叫做 `else` 从句。也就是当条件为 `false` 时，执行 `else` 语句：

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

显然，这两条分支中总有一条会被执行。由于温度已升至 40 华氏度，不算太冷，没必要再围围巾。因此，`else` 分支就被触发了。

你可以把多个 `if` 语句链接在一起，来实现更多分支：

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

在上面的例子中，额外的 `if` 语句用于判断是不是特别热。而最后的 `else` 语句被保留了下来，用于打印既不冷也不热时的消息。

实际上，当不需要完整判断情况的时候，最后的 `else` 语句是可选的：

```
temperatureInFahrenheit = 72
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
} else if temperatureInFahrenheit >= 86 {
    print("It's really warm. Don't forget to wear sunscreen.")
}
```

在这个例子中，由于既不冷也不热，所以不会触发 `if` 或 `else if` 分支，也就不会打印任何消息。

## Switch

---

`switch` 语句会尝试把某个值与若干个模式（pattern）进行匹配。根据第一个匹配成功的模式，`switch` 语句会执行对应的代码。当有可能的情况较多时，通常用 `switch` 语句替换 `if` 语句。

`switch` 语句最简单的形式就是把某个值与一个或若干个相同类型的值作比较：

```
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  
}
```

`switch` 语句由多个 `case` 构成，每个由 `case` 关键字开始。为了匹配某些更特定的值，Swift 提供了几种方法来进行更复杂的模式匹配，这些模式将在本节的稍后部分提到。

与 `if` 语句类似，每一个 `case` 都是代码执行的一条分支。`switch` 语句会决定哪一条分支应该被执行，这个流程被称作根据给定的值切换 (*switching*)。

`switch` 语句必须是完备的。这就是说，每一个可能的值都必须至少有一个 `case` 分支与之对应。在某些不可能涵盖所有值的情况下，你可以使用默认 (`default`) 分支来涵盖其它所有没有对应的值，这个默认分支必须在 `switch` 语句的最后面。

下面的例子使用 `switch` 语句来匹配一个名为 `someCharacter` 的小写字符：

```
let someCharacter: Character = "z"  
switch someCharacter {  
    case "a":  
        print("The first letter of the alphabet")  
    case "z":  
        print("The last letter of the alphabet")  
    default:  
        print("Some other character")  
}  
// 输出"The last letter of the alphabet"
```

在这个例子中，第一个 `case` 分支用于匹配第一个英文字母 `a`，第二个 `case` 分支用于匹配最后一个字母 `z`。因为 `switch` 语句必须有一个 `case` 分支用于覆盖所有可能的字符，而不仅仅是所有的英文字母，所以 `switch` 语句使用 `default` 分支来匹配除了 `a` 和 `z` 外的所有值，这个分支保证了 `switch` 语句的完备性。

## 不存在隐式的贯穿

与 C 和 Objective-C 中的 `switch` 语句不同，在 Swift 中，当匹配的 `case` 分支中的代码执行完毕后，程序会终止 `switch` 语句，而不会继续执行下一个 `case` 分支。这也就是说，不需要在 `case` 分支中显式地使用 `break` 语句。这使得 `switch` 语句更安全、更容易用，也避免了漏写 `break` 语句导致多个语言被执行的错误。

## 注意

虽然在 Swift 中 `break` 不是必须的，但你依然可以在 `case` 分支中的代码执行完毕前使用 `break` 跳出，详情请参见 [Switch 语句中的 break](#)。

每一个 `case` 分支都必须包含至少一条语句。像下面这样书写代码是无效的，因为第一个 `case` 分支是空的：

```
let anotherCharacter: Character = "a"
switch anotherCharacter {
    case "a": // 无效，这个分支下面没有语句
    case "A":
        print("The letter A")
    default:
        print("Not the letter A")
}
// 这段代码会报编译错误
```

不像 C 语言里的 `switch` 语句，在 Swift 中，`switch` 语句不会一起匹配 `"a"` 和 `"A"`。相反的，上面的代码会引起编译期错误：`case "a": 不包含任何可执行语句` ——这就避免了意外地从一个 `case` 分支贯穿到另外一个，使得代码更安全、也更直观。

为了让单个 `case` 同时匹配 `a` 和 `A`，可以将这个两个值组合成一个复合匹配，并且用逗号分开：

```
let anotherCharacter: Character = "a"
switch anotherCharacter {
    case "a", "A":
        print("The letter A")
    default:
        print("Not the letter A")
}
// 输出"The letter A"
```

为了可读性，复合匹配可以写成多行形式，详情请参考 [复合匹配](#)

## 注意

如果想要显式贯穿 `case` 分支，请使用 `fallthrough` 语句，详情请参见 [贯穿](#)。

## 区间匹配

`case` 分支的模式也可以是一个值的区间。下面的例子展示了如何使用区间匹配来输出任意数字对应的自然语言格式：

```

let approximateCount = 62
let countedThings = "moons orbiting Saturn"
let naturalCount: String
switch approximateCount {
case 0:
    naturalCount = "no"
case 1..<5:
    naturalCount = "a few"
case 5..<12:
    naturalCount = "several"
case 12..<100:
    naturalCount = "dozens of"
case 100..<1000:
    naturalCount = "hundreds of"
default:
    naturalCount = "many"
}
print("There are \n(naturalCount) \n(countedThings).")
// 输出"There are dozens of moons orbiting Saturn."

```

在上例中，`approximateCount` 在一个 `switch` 声明中被评估。每一个 `case` 都与之进行比较。因为 `approximateCount` 落在了 12 到 100 的区间，所以 `naturalCount` 等于 `"dozens of"` 值，并且此后的执行跳出了 `switch` 语句。

## 元组

---

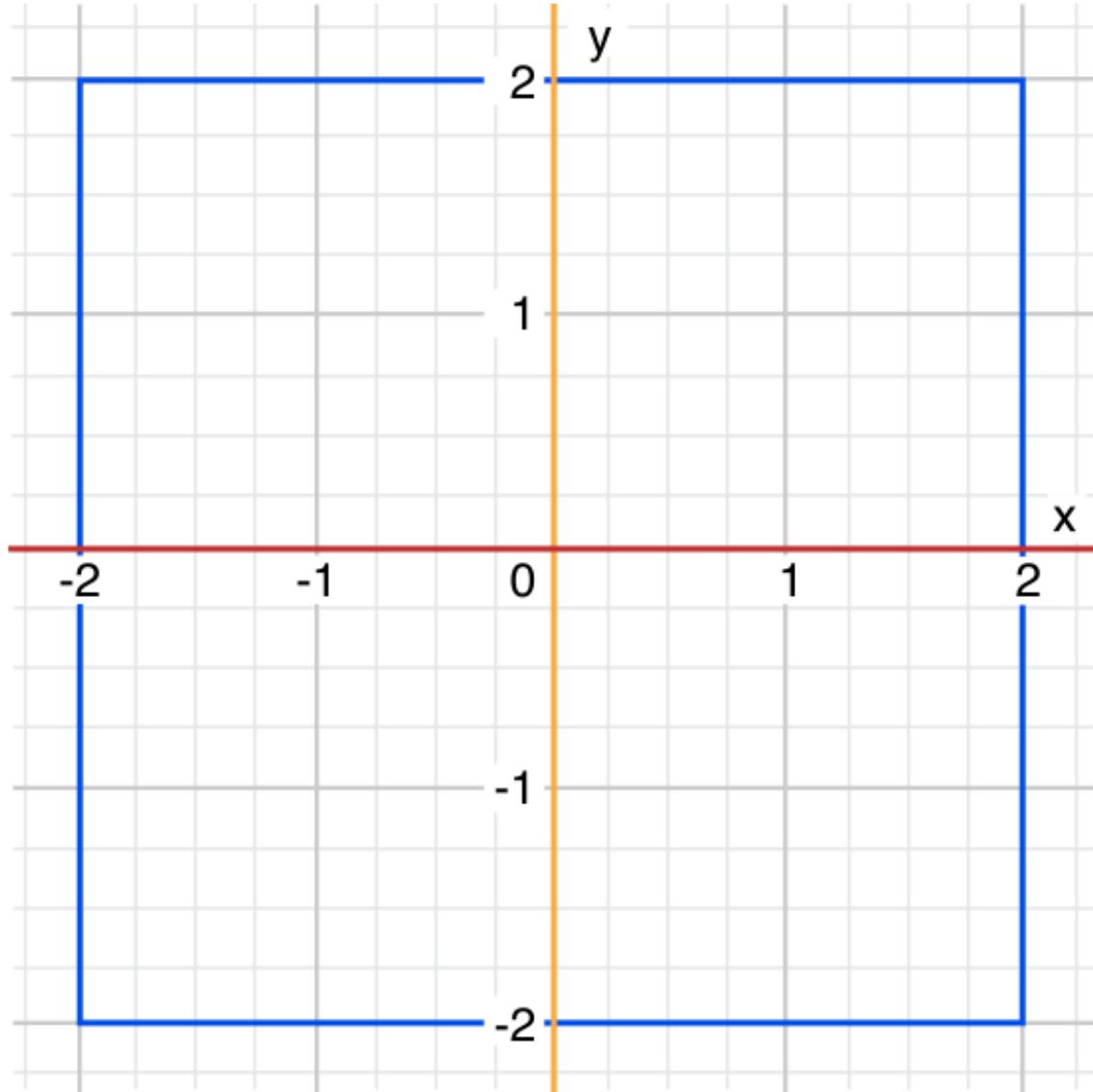
我们可以使用元组在同一个 `switch` 语句中测试多个值。元组中的元素可以是值，也可以是区间。另外，使用下划线（`_`）来匹配所有可能的值。

下面的例子展示了如何使用一个 `(Int, Int)` 类型的元组来分类下图中的点  $(x, y)$ ：

```

let somePoint = (1, 1)
switch somePoint {
case (0, 0):
    print("\n(somePoint) is at the origin")
case (_, 0):
    print("\n(somePoint) is on the x-axis")
case (0, _):
    print("\n(somePoint) is on the y-axis")
case (-2...2, -2...2):
    print("\n(somePoint) is inside the box")
default:
    print("\n(somePoint) is outside of the box")
}
// 输出"(1, 1) is inside the box"

```



在上面的例子中，`switch` 语句会判断某个点是否是原点  $(0, 0)$ ，是否在红色的  $x$  轴上，是否在橘黄色的  $y$  轴上，是否在一个以原点为中心的  $4 \times 4$  的蓝色矩形里，或者在这个矩形外面。

不像 C 语言，Swift 允许多个 `case` 匹配同一个值。实际上，在这个例子中，点  $(0, 0)$  可以匹配所有四个 `case`。但是，如果存在多个匹配，那么只会执行第一个被匹配到的 `case` 分支。考虑点  $(0, 0)$  会首先匹配 `case (0, 0)`，因此剩下的能够匹配的分支都会被忽视掉。

## 值绑定 (Value Bindings)

`case` 分支允许将匹配的值声明为临时常量或变量，并且在 `case` 分支体内使用 — 这种行为被称为 **值绑定** (value binding)，因为匹配的值在 `case` 分支体内，与临时的常量或变量绑定。

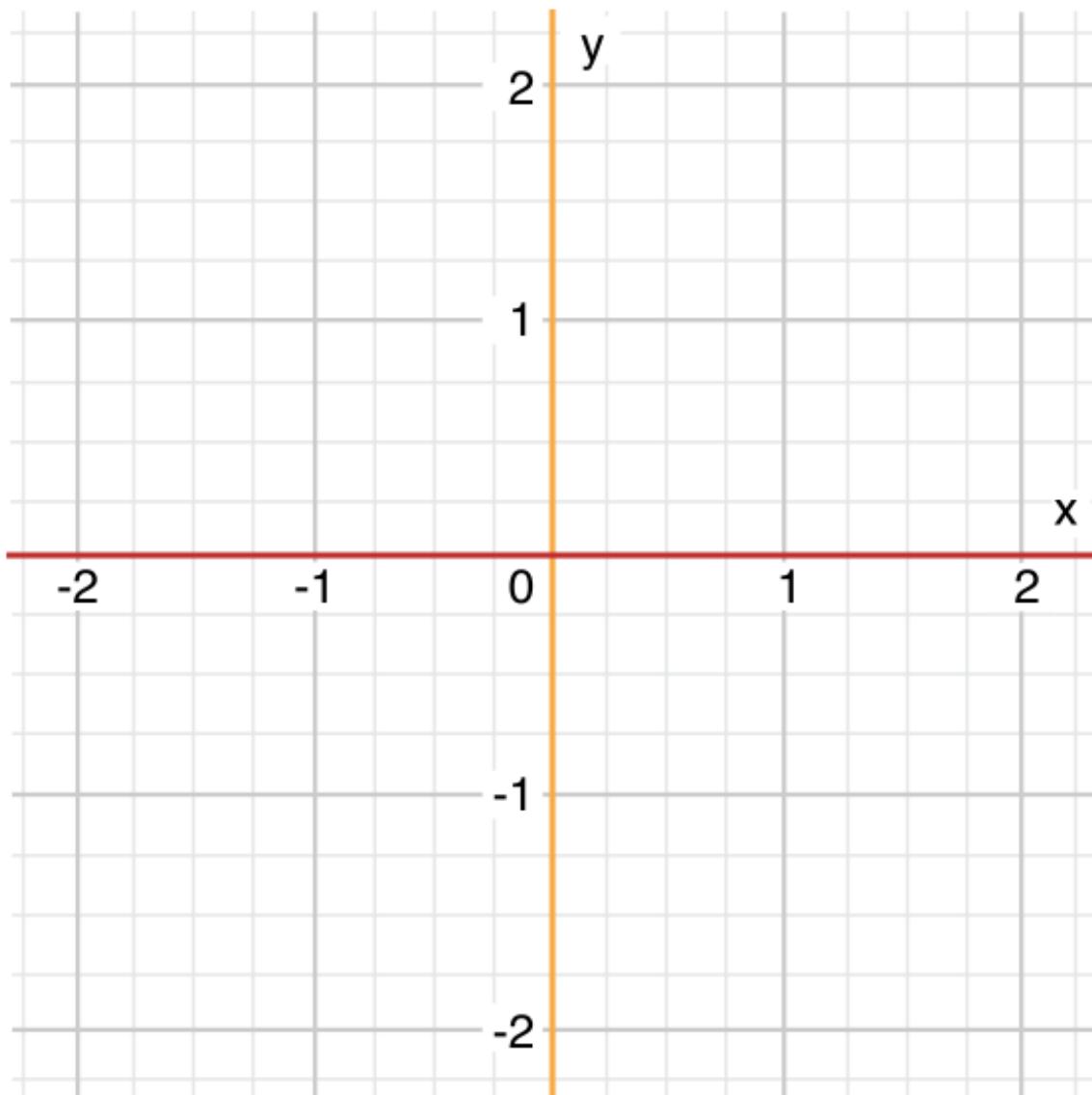
下面的例子将下图中的点  $(x, y)$ ，使用 `(Int, Int)` 类型的元组表示，然后分类表示：

```

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

}
// 输出“on the x-axis with an x value of 2”

```



在上面的例子中，`switch` 语句会判断某个点是否在红色的 x 轴上，是否在橘黄色的 y 轴上，或者不在坐标轴上。

这三个 case 都声明了常量 `x` 和 `y` 的占位符，用于临时获取元组 `anotherPoint` 的一个或两个值。第一个 case —— `case (let x, 0)` 将匹配一个纵坐标为 0 的点，并把这个点的横坐标赋给临时的常量 `x`。类似的，第二个 case —— `case (0, let y)` 将匹配一个横坐标为 0 的点，并把这个点的纵坐标赋给临时的常量 `y`。

一旦声明了这些临时的常量，它们就可以在其对应的 case 分支里使用。在这个例子中，它们用于打印给定点的类型。

请注意，这个 `switch` 语句不包含默认分支。这是因为最后一个 case —— `case let(x, y)` 声明了一个可以匹配余下所有值的元组。这使得 `switch` 语句已经完备了，因此不需要再书写默认分支。

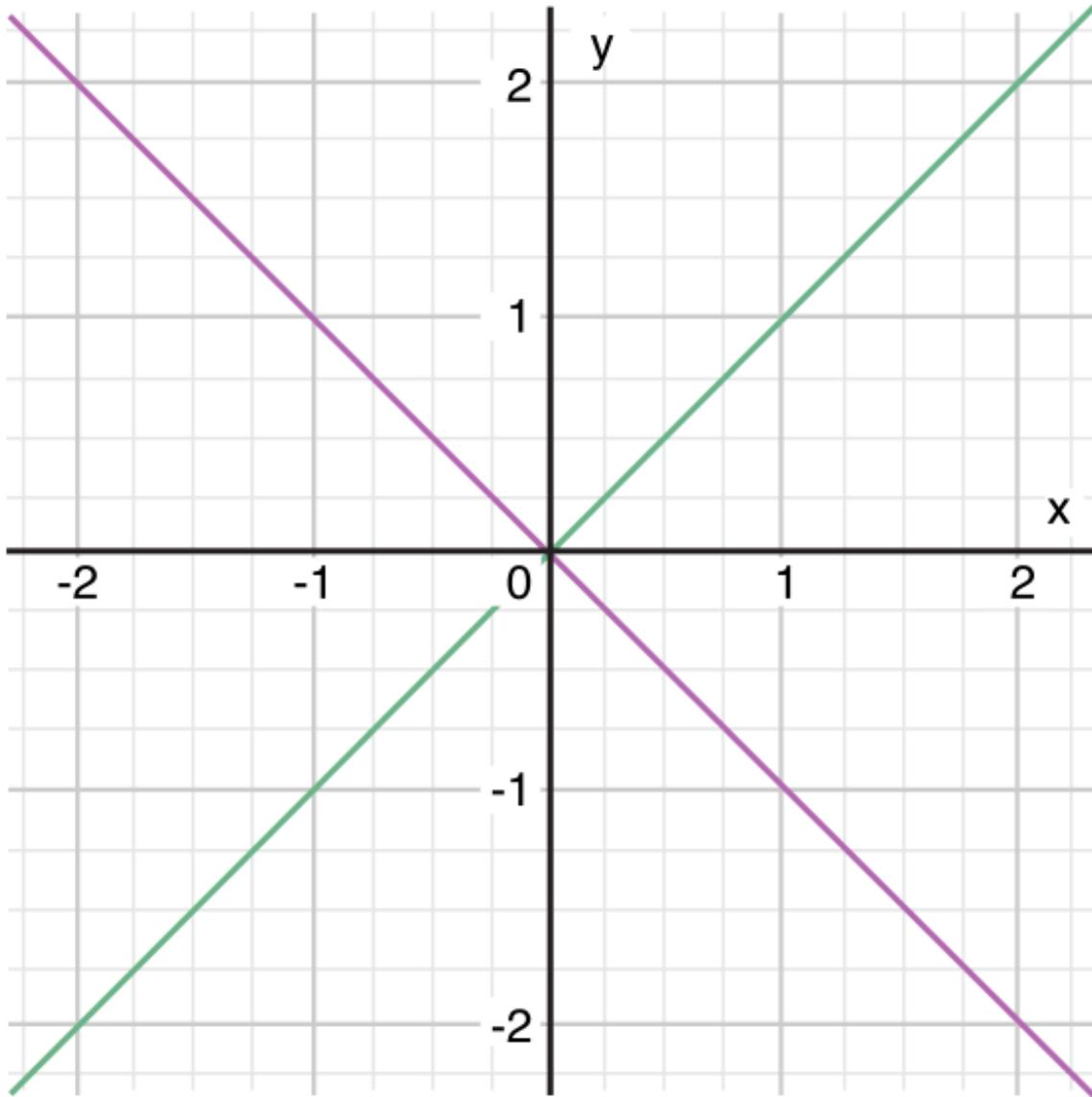
## Where

---

case 分支的模式可以使用 `where` 语句来判断额外的条件。

下面的例子把下图中的点  $(x, y)$  进行了分类：

```
let yetAnotherPoint = (1, -1)
switch yetAnotherPoint {
  case let (x, y) where x == y:
    print("(x, y) is on the line x == y")
  case let (x, y) where x == -y:
    print("(x, y) is on the line x == -y")
  case let (x, y):
    print("(x, y) is just some arbitrary point")
}
// 输出“(1, -1) is on the line x == -y”
```



在上面的例子中，`switch` 语句会判断某个点是否在绿色的对角线 `x == y` 上，是否在紫色的对角线 `x == -y` 上，或者不在对角线上。

这三个 case 都声明了常量 `x` 和 `y` 的占位符，用于临时获取元组 `yetAnotherPoint` 的两个值。这两个常量被用作 `where` 语句的一部分，从而创建一个动态的过滤器 (filter)。当且仅当 `where` 语句的条件为 `true` 时，匹配到的 case 分支才会被执行。

就像是值绑定中的例子，由于最后一个 case 分支匹配了余下所有可能的值，`switch` 语句就已经完备了，因此不需要再书写默认分支。

## 复合型 Cases

---

当多个条件可以使用同一种方法来处理时，可以将这几种可能放在同一个 `case` 后面，并且用逗号隔开。当 `case` 后面的任意一种模式匹配的时候，这条分支就会被匹配。并且，如果匹配列表过长，还可以分行书写：

```

let someCharacter: Character = "e"
switch someCharacter {
    case "a", "e", "i", "o", "u":
        print("\(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":
        print("\(someCharacter) is a consonant")
    default:
        print("\(someCharacter) is not a vowel or a consonant")
}
// 输出“e is a vowel”

```

这个 `switch` 语句中的第一个 `case`，匹配了英语中的五个小写元音字母。相似的，第二个 `case` 匹配了英语中所有的小写辅音字母。最终，`default` 分支匹配了其它所有字符。

复合匹配同样可以包含值绑定。复合匹配里所有的匹配模式，都必须包含相同的值绑定。并且每一个绑定都必须获取到相同类型的值。这保证了，无论复合匹配中的哪个模式发生了匹配，分支体内的代码，都能获取到绑定的值，并且绑定的值都有一样的类型。

```

let stillAnotherPoint = (9, 0)
switch stillAnotherPoint {
    case (let distance, 0), (0, let distance):
        print("On an axis, \(distance) from the origin")
    default:
        print("Not on an axis")
}
// 输出“On an axis, 9 from the origin”

```

上面的 `case` 有两个模式：`(let distance, 0)` 匹配了在 x 轴上的值，`(0, let distance)` 匹配了在 y 轴上的值。两个模式都绑定了 `distance`，并且 `distance` 在两种模式下，都是整型——这意味着分支体内的代码，只要 `case` 匹配，都可以获取到 `distance` 值。

## 控制转移语句

---

控制转移语句改变你代码的执行顺序，通过它可以实现代码的跳转。Swift 有五种控制转移语句：

- `continue`
- `break`
- `fallthrough`
- `return`
- `throw`

我们将会在下面讨论 `continue`、`break` 和 `fallthrough` 语句。`return` 语句将会在 [函数](#) 章节讨论，`throw` 语句会在 [错误抛出](#) 章节讨论。

### Continue

---

`continue` 语句告诉一个循环体立刻停止本次循环，重新开始下次循环。就好像在说“本次循环我已经执行完了”，但是并不会离开整个循环体。

下面的例子把一个小写字符串中的元音字母和空格字符移除，生成了一个含义模糊的短句：

```
let puzzleInput = "great minds think alike"
var puzzleOutput = ""
for character in puzzleInput {
    switch character {
        case "a", "e", "i", "o", "u", " ":
            continue
        default:
            puzzleOutput.append(character)
    }
}
print(puzzleOutput)
// 输出“grtmndsthnlk”
```

在上面的代码中，只要匹配到元音字母或者空格字符，就调用 `continue` 语句，使本次循环结束，重新开始下次循环。这种行为使 `switch` 匹配到元音字母和空格字符时不做处理，而不是让每一个匹配到的字符都被打印。

## Break

`break` 语句会立刻结束整个控制流的执行。`break` 可以在 `switch` 或循环语句中使用，用来提前结束 `switch` 或循环语句。

### 循环语句中的 break

当在一个循环体中使用 `break` 时，会立刻中断该循环体的执行，然后跳转到表示循环体结束的大括号（`}`）后的第一行代码。不会再有本次循环的代码被执行，也不会再有下次的循环产生。

### Switch 语句中的 break

当在一个 `switch` 代码块中使用 `break` 时，会立即中断该 `switch` 代码块的执行，并且跳转到表示 `switch` 代码块结束的大括号（`}`）后的第一行代码。

这种特性可以被用来匹配或者忽略一个或多个分支。因为 Swift 的 `switch` 需要包含所有的分支而且不允许有为空的分支，有时为了使你的意图更明显，需要特意匹配或者忽略某个分支。那么当你想忽略某个分支时，可以在该分支内写上 `break` 语句。当那个分支被匹配到时，分支内的 `break` 语句立即结束 `switch` 代码块。

#### 注意

当一个 `switch` 分支仅仅包含注释时，会被报编译时错误。注释不是代码语句而且也不能让 `switch` 分支达到被忽略的效果。你应该使用 `break` 来忽略某个分支。

下面的例子通过 `switch` 来判断一个 `Character` 值是否代表下面四种语言之一。为了简洁，多个值被包含在了同一个分支情况中。

```
let numberSymbol: Character = "三" // 简体中文里的数字 3
var possibleIntegerValue: Int?
switch numberSymbol {
    case "1", "\u{1f1f4}", "\u{1f1f6}", "\u{1f1f8}":
        possibleIntegerValue = 1
    case "2", "\u{1f1f4}\u{1f1f5}", "\u{1f1f6}\u{1f1f5}", "\u{1f1f8}\u{1f1f5}":
        possibleIntegerValue = 2
    case "3", "\u{1f1f4}\u{1f1f7}", "\u{1f1f6}\u{1f1f7}", "\u{1f1f8}\u{1f1f7}":
        possibleIntegerValue = 3
    case "4", "\u{1f1f4}\u{1f1f8}", "\u{1f1f6}\u{1f1f8}", "\u{1f1f8}\u{1f1f8}":
        possibleIntegerValue = 4
    default:
        break
}
if let integerValue = possibleIntegerValue {
    print("The integer value of \(numberSymbol) is \(integerValue).")
} else {
    print("An integer value could not be found for \(numberSymbol).")
}
// 输出"The integer value of 三 is 3."
```

这个例子检查 `numberSymbol` 是否是拉丁，阿拉伯，中文或者泰语中的 1 到 4 之一。如果被匹配到，该 `switch` 分支语句给 `Int?` 类型变量 `possibleIntegerValue` 设置一个整数值。

当 `switch` 代码块执行完后，接下来的代码通过使用可选绑定来判断 `possibleIntegerValue` 是否曾经被设置过值。因为是可选类型的缘故，`possibleIntegerValue` 有一个隐式的初始值 `nil`，所以仅仅当 `possibleIntegerValue` 曾被 `switch` 代码块的前四个分支中的某个设置过一个值时，可选的绑定才会被判定为成功。

在上面的例子中，想要把 `Character` 所有的的可能性都枚举出来是不现实的，所以使用 `default` 分支来包含所有上面没有匹配到字符的情况。由于这个 `default` 分支不需要执行任何动作，所以它只写了一条 `break` 语句。一旦落入到 `default` 分支中后，`break` 语句就完成了该分支的所有代码操作，代码继续向下，开始执行 `if let` 语句。

## 贯穿 (Fallthrough)

在 Swift 里，`switch` 语句不会从上一个 `case` 分支跳转到下一个 `case` 分支中。相反，只要第一个匹配到的 `case` 分支完成了它需要执行的语句，整个 `switch` 代码块完成了它的执行。相比之下，C 语言要求你显式地插入 `break` 语句到每个 `case` 分支的末尾来阻止自动落入到下一个 `case` 分支中。Swift 的这种避免默认落入到下一个分支中的特性意味着它的 `switch` 功能要比 C 语言的更加清晰和可预测，可以避免无意识地执行多个 `case` 分支从而引发的错误。

如果你确实需要 C 风格的贯穿的特性，你可以在每个需要该特性的 `case` 分支中使用 `fallthrough` 关键字。下面的例子使用 `fallthrough` 来创建一个数字的描述语句。

```
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."
}
print(description)
// 输出"The number 5 is a prime number, and also an integer."
```

这个例子定义了一个 `String` 类型的变量 `description` 并且给它设置了一个初始值。函数使用 `switch` 逻辑来判断 `integerToDescribe` 变量的值。当 `integerToDescribe` 的值属于列表中的质数之一时，该函数在 `description` 后添加一段文字，来表明这个数字是一个质数。然后它使用 `fallthrough` 关键字来“贯穿”到 `default` 分支中。`default` 分支在 `description` 的最后添加一段额外的文字，至此 `switch` 代码块执行完了。

如果 `integerToDescribe` 的值不属于列表中的任何质数，那么它不会匹配到第一个 `switch` 分支。而这里没有其他特别的分支情况，所以 `integerToDescribe` 匹配到 `default` 分支中。

当 `switch` 代码块执行完后，使用 `print(_:separator:terminator:)` 函数打印该数字的描述。在这个例子中，数字 `5` 被准确的识别为了一个质数。

### 注意

`fallthrough` 关键字不会检查它下一个将会落入执行的 `case` 中的匹配条件。`fallthrough` 简单地使代码继续连接到下一个 `case` 中的代码，这和 C 语言标准中的 `switch` 语句特性是一样的。

## 带标签的语句

在 Swift 中，你可以在循环体和条件语句中嵌套循环体和条件语句来创造复杂的控制流结构。并且，循环体和条件语句都可以使用 `break` 语句来提前结束整个代码块。因此，显式地指明 `break` 语句想要终止的是哪个循环体或者条件语句，会很有用。类似地，如果你有许多嵌套的循环体，显式指明 `continue` 语句想要影响哪一个循环体也会非常有用。

为了实现这个目的，你可以使用标签 (*statement label*) 来标记一个循环体或者条件语句，对于一个条件语句，你可以使用 `break` 加标签的方式，来结束这个被标记的语句。对于一个循环语句，你可以使用 `break` 或者 `continue` 加标签，来结束或者继续这条被标记语句的执行。

声明一个带标签的语句是通过在该语句的关键词的同一行前面放置一个标签，作为这个语句的前导关键字 (introducer keyword)，并且该标签后面跟随一个冒号。下面是一个针对 `while` 循环体的标签语法，同样的规则适用于所有的循环体和条件语句。

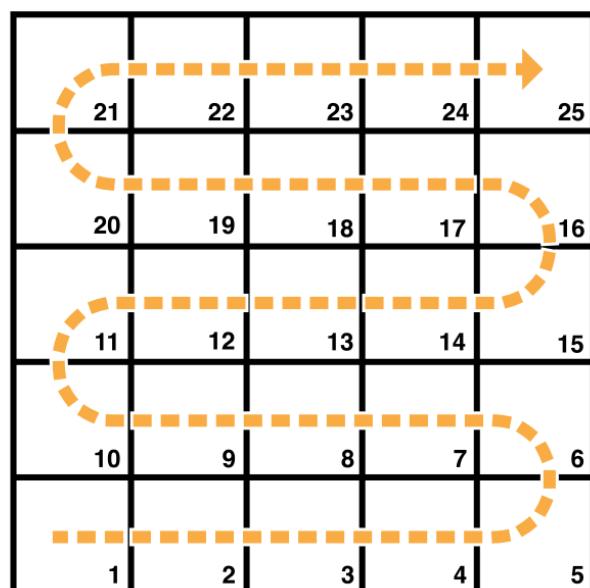
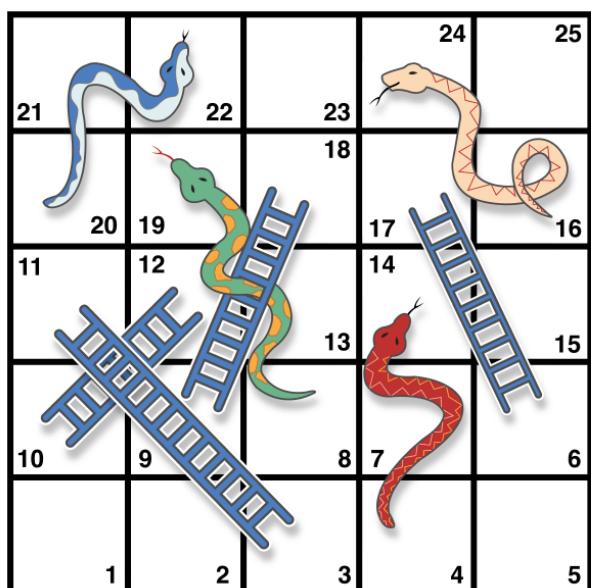
```
label name: while condition {  
    statements  
}
```

下面的例子是前面章节中蛇和梯子的适配版本，在此版本中，我们将使用一个带有标签的 `while` 循环体中调用 `break` 和 `continue` 语句。这次，游戏增加了一条额外的规则：

为了获胜，你必须刚好落在第 25 个方块中。

如果某次掷骰子使你的移动超出第 25 个方块，你必须重新掷骰子，直到你掷出的骰子数刚好使你能落在第 25 个方块中。

游戏的棋盘和之前一样：



`finalSquare`、`board`、`square` 和 `diceRoll` 值被和之前一样的方式初始化：

```
let finalSquare = 25  
var board = [Int](repeating: 0, count: finalSquare + 1)  
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
```

这个版本的游戏使用 `while` 循环和 `switch` 语句来实现游戏的逻辑。`while` 循环有一个标签名 `gameLoop`，来表明它是游戏的主循环。

该 `while` 循环体的条件判断语句是 `while square !=finalSquare`，这表明你必须刚好落在方格25中。

```

gameLoop: while square != finalSquare {
    diceRoll += 1
    if diceRoll == 7 { diceRoll = 1 }
    switch square + diceRoll {
        case finalSquare:
            // 骰子数刚好使玩家移动到最终的方格里，游戏结束。
            break gameLoop
        case let newSquare where newSquare > finalSquare:
            // 骰子数将会使玩家的移动超出最后的方格，那么这种移动是不合法的，玩家需要重新掷骰子
            continue gameLoop
        default:
            // 合法移动，做正常的处理
            square += diceRoll
            square += board[square]
    }
}
print("Game over!")

```

每次循环迭代开始时掷骰子。与之前玩家掷完骰子就立即移动不同，这里使用了 `switch` 语句来考虑每次移动可能产生的结果，从而决定玩家本次是否能够移动。

- 如果骰子数刚好使玩家移动到最终的方格里，游戏结束。`break gameLoop` 语句跳转控制去执行 `while` 循环体后的第一行代码，意味着游戏结束。
- 如果骰子数将会使玩家的移动超出最后的方格，那么这种移动是不合法的，玩家需要重新掷骰子。`continue gameLoop` 语句结束本次 `while` 循环，开始下一次循环。
- 在剩余的所有情况中，骰子数产生的都是合法的移动。玩家向前移动 `diceRoll` 个方格，然后游戏逻辑再处理玩家当前是否处于蛇头或者梯子的底部。接着本次循环结束，控制跳转到 `while` 循环体的条件判断语句处，再决定是否需要继续执行下次循环。

### 注意

如果上述的 `break` 语句没有使用 `gameLoop` 标签，那么它将会中断 `switch` 语句而不是 `while` 循环。使用 `gameLoop` 标签清晰的表明了 `break` 想要中断的是哪个代码块。

同时请注意，当调用 `continue gameLoop` 去跳转到下一次循环迭代时，这里使用 `gameLoop` 标签并不是严格必须的。因为在这个游戏中，只有一个循环体，所以 `continue` 语句会影响到哪个循环体是没有歧义的。然而，`continue` 语句使用 `gameLoop` 标签也是没有危害的。这样做符合标签的使用规则，同时参照旁边的 `break gameLoop`，能够使游戏的逻辑更加清晰和易于理解。

## 提前退出

像 `if` 语句一样，`guard` 的执行取决于一个表达式的布尔值。我们可以使用 `guard` 语句来要求条件必须为真时，以执行 `guard` 语句后的代码。不同于 `if` 语句，一个 `guard` 语句总是有一个 `else` 从句，如果条件不为真则执行 `else` 从句中的代码。

```

func greet(person: [String: String]) {
    guard let name = person["name"] else {
        return
    }

    print("Hello \(name)!")

    guard let location = person["location"] else {
        print("I hope the weather is nice near you.")
        return
    }

    print("I hope the weather is nice in \(location).")
}

greet(person: ["name": "John"])
// 输出“Hello John!”
// 输出“I hope the weather is nice near you.”
greet(person: ["name": "Jane", "location": "Cupertino"])
// 输出“Hello Jane!”
// 输出“I hope the weather is nice in Cupertino.”
```

如果 `guard` 语句的条件被满足，则继续执行 `guard` 语句大括号后的代码。将变量或者常量的可选绑定作为 `guard` 语句的条件，都可以保护 `guard` 语句后面的代码。

如果条件不被满足，在 `else` 分支上的代码就会被执行。这个分支必须转移控制以退出 `guard` 语句出现的代码段。它可以用控制转移语句如 `return`、`break`、`continue` 或者 `throw` 做这件事，或者调用一个不返回的方法或函数，例如 `fatalError()`。

相比于可以实现同样功能的 `if` 语句，按需使用 `guard` 语句会提升我们代码的可读性。它可以使你的代码连贯的被执行而不需要将它包在 `else` 块中，它可以使你在紧邻条件判断的地方，处理违规的情况。

## 检测 API 可用性

---

Swift 内置支持检查 API 可用性，这可以确保我们不会在当前部署机器上，不小心地使用了不可用的 API。

编译器使用 SDK 中的可用信息来验证我们的代码中使用的所有 API 在项目指定的部署目标上是否可用。如果我们尝试使用一个不可用的 API，Swift 会在编译时报错。

我们在 `if` 或 `guard` 语句中使用 **可用性条件 (availability condition)** 去有条件的执行一段代码，来在运行时判断调用的 API 是否可用。编译器使用从可用性条件语句中获取的信息去验证，在这个代码块中调用的 API 是否可用。

```

if #available(iOS 10, macOS 10.12, *) {
    // 在 iOS 使用 iOS 10 的 API, 在 macOS 使用 macOS 10.12 的 API
} else {
    // 使用先前版本的 iOS 和 macOS 的 API
}
```

以上可用性条件指定，`if` 语句的代码块仅仅在 iOS 10 或 macOS 10.12 及更高版本才运行。最后一个参数，`*`，是必须的，用于指定在所有其它平台中，如果版本号高于你的设备指定的最低版本，`if` 语句的代码块将会运行。

在它一般的形式中，可用性条件使用了一个平台名字和版本的列表。平台名字可以是 `iOS`，`macOS`，`watchOS` 和 `tvOS` ——请访问 [声明属性](#) 来获取完整列表。除了指定像 iOS 8 或 macOS 10.10 的大版本号，也可以指定像 iOS 11.2.6 以及 macOS 10.13.3 的小版本号。

```
if #available(平台名称 版本号, ..., *) {  
    APIs 可用，语句将执行  
} else {  
    APIs 不可用，语句将不执行  
}
```

## 函数

函数是一段完成特定任务的独立代码片段。你可以通过给函数命名来标识某个函数的功能，这个名字可以被用来在需要的时候“调用”这个函数来完成它的任务。

Swift 统一的函数语法非常的灵活，可以用来表示任何函数，包括从最简单的没有参数名字的 C 风格函数，到复杂的带局部和外部参数名的 Objective-C 风格函数。参数可以提供默认值，以简化函数调用。参数也可以既当做传入参数，也当做传出参数，也就是说，一旦函数执行结束，传入的参数值将被修改。

在 Swift 中，每个函数都有一个由函数的参数值类型和返回值类型组成的类型。你可以把函数类型当做任何其他普通变量类型一样处理，这样就可以更简单地把函数当做别的函数的参数，也可以从其他函数中返回函数。函数的定义可以写在其他函数定义中，这样可以在嵌套函数范围内实现功能封装。

## 函数的定义与调用

当你定义一个函数时，你可以定义一个或多个有名字和类型的值，作为函数的输入，称为参数，也可以定义某种类型的值作为函数执行结束时的输出，称为返回类型。

每个函数有个函数名，用来描述函数执行的任务。要使用一个函数时，用函数名来“调用”这个函数，并传给它匹配的输入值（称作实参）。函数的实参必须与函数参数表里参数的顺序一致。

下面例子中的函数的名字是 `greet(person:)`，之所以叫这个名字，是因为这个函数用一个人的名字当做输入，并返回向这个人问候的语句。为了完成这个任务，你需要定义一个输入参数——一个叫做 `person` 的 `String` 值，和一个包含给这个人问候语的 `String` 类型的返回值：

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

所有的这些信息汇总起来成为函数的定义，并以 `func` 作为前缀。指定函数返回类型时，用返回箭头 `->`（一个连字符后跟一个右尖括号）后跟返回类型的名称的方式来表示。

该定义描述了函数的功能，它期望接收什么作为参数和执行结束时它返回的结果是什么类型。这样的定义使得函数可以在别的地方以一种清晰的方式被调用：

```
print(greet(person: "Anna"))
// 打印“Hello, Anna!”
print(greet(person: "Brian"))
// 打印“Hello, Brian!”
```

调用 `greet(person:)` 函数时，在圆括号中传给它一个 `String` 类型的实参，例如 `greet(person: "Anna")`。正如上面所示，因为这个函数返回一个 `String` 类型的值，所以 `greet` 可以被包含在 `print(_:separator:terminator:)` 的调用中，用来输出这个函数的返回值。

### 注意

`print(_:separator:terminator:)` 函数的第一个参数并没有设置一个标签，而其他的参数因为已经有了默认值，因此是可选的。关于这些函数语法上的变化详见下方关于函数参数标签和参数名以及默认参数值。

在 `greet(person:)` 的函数体中，先定义了一个新的名为 `greeting` 的 `String` 常量，同时，把对 `personName` 的问候消息赋值给了 `greeting`。然后用 `return` 关键字把这个问候返回出去。一旦 `return greeting` 被调用，该函数结束它的执行并返回 `greeting` 的当前值。

你可以用不同的输入值多次调用 `greet(person:)`。上面的例子展示的是用 `"Anna"` 和 `"Brian"` 调用的结果，该函数分别返回了不同的结果。

为了简化这个函数的定义，可以将问候消息的创建和返回写成一句：

```
func greetAgain(person: String) -> String {
    return "Hello again, " + person + "!"
}
print(greetAgain(person: "Anna"))
// 打印“Hello again, Anna!”
```

## 函数参数与返回值

函数参数与返回值在 Swift 中非常的灵活。你可以定义任何类型的函数，包括从只带一个未名参数的简单函数到复杂的带有表达性参数名和不同参数选项的复杂函数。

## 无参数函数

函数可以没有参数。下面这个函数就是一个无参数函数，当被调用时，它返回固定的 `String` 消息：

```
func sayHelloWorld() -> String {
    return "hello, world"
}
print(sayHelloWorld())
// 打印“hello, world”
```

尽管这个函数没有参数，但是定义中在函数名后还是需要一对圆括号。当被调用时，也需要在函数名后写一对圆括号。

## 多参数函数

---

函数可以有多种输入参数，这些参数被包含在函数的括号之中，以逗号分隔。

下面这个函数用一个人名和是否已经打过招呼作为输入，并返回对这个人的适当问候语：

```
func greet(person: String, alreadyGreeted: Bool) -> String {  
    if alreadyGreeted {  
        return greetAgain(person: person)  
    } else {  
        return greet(person: person)  
    }  
}  
print(greet(person: "Tim", alreadyGreeted: true))  
// 打印“Hello again, Tim!”
```

你可以通过在括号内使用逗号分隔来传递一个 `String` 参数值和一个标识为 `alreadyGreeted` 的 `Bool` 值，来调用 `greet(person:alreadyGreeted:)` 函数。注意这个函数和上面 `greet(person:)` 是不同的。虽然它们都有着同样的名字 `greet`，但是 `greet(person:alreadyGreeted:)` 函数需要两个参数，而 `greet(person:)` 只需要一个参数。

## 无返回值函数

---

函数可以没有返回值。下面是 `greet(person:)` 函数的另一个版本，这个函数直接打印一个 `String` 值，而不是返回它：

```
func greet(person: String) {  
    print("Hello, \(person)!")  
}  
greet(person: "Dave")  
// 打印“Hello, Dave!”
```

因为这个函数不需要返回值，所以这个函数的定义中没有返回箭头 (`->`) 和返回类型。

注意

严格地说，即使没有明确定义返回值，该 `greet(Person:)` 函数仍然返回一个值。没有明确定义返回类型的函数的返回一个 `Void` 类型特殊值，该值为一个空元组，写成 `()`。

调用函数时，可以忽略该函数的返回值：

```
func printAndCount(string: String) -> Int {  
    print(string)  
    return string.count  
}  
func printWithoutCounting(string: String) {  
    let _ = printAndCount(string: string)  
}  
printAndCount(string: "hello, world")  
// 打印“hello, world”，并且返回值 12  
printWithoutCounting(string: "hello, world")  
// 打印“hello, world”，但是没有返回任何值
```

第一个函数 `printAndCount(string:)`，输出一个字符串并返回 `Int` 类型的字符数。第二个函数 `printWithoutCounting(string:)` 调用了第一个函数，但是忽略了它的返回值。当第二个函数被调用时，消息依然会由第一个函数输出，但是返回值不会被用到。

### 注意

返回值可以被忽略，但定义了有返回值的函数必须返回一个值，如果在函数定义底部没有返回任何值，将导致编译时错误。

## 多重返回值函数

你可以用元组 (tuple) 类型让多个值作为一个复合值从函数中返回。

下例中定义了一个名为 `minMax(array:)` 的函数，作用是在一个 `Int` 类型的数组中找出最小值与最大值。

```
func minMax(array: [Int]) -> (min: Int, max: Int) {  
    var currentMin = array[0]  
    var currentMax = array[0]  
    for value in array[1..<array.count] {  
        if value < currentMin {  
            currentMin = value  
        } else if value > currentMax {  
            currentMax = value  
        }  
    }  
    return (currentMin, currentMax)  
}
```

`minMax(array:)` 函数返回一个包含两个 `Int` 值的元组，这些值被标记为 `min` 和 `max`，以便查询函数的返回值时可以通过名字访问它们。

在 `minMax(array:)` 的函数体中，在开始的时候设置两个工作变量 `currentMin` 和 `currentMax` 的值为数组中的第一个数。然后函数会遍历数组中剩余的值并检查该值是否比 `currentMin` 和 `currentMax` 更小或更大。最后数组中的最小值与最大值作为一个包含两个 `Int` 值的元组返回。

因为元组的成员值已被命名，因此可以通过 `.` 语法来检索找到的最小值与最大值：

```
let bounds = minMax(array: [8, -6, 2, 109, 3, 71])
print("min is \(bounds.min) and max is \(bounds.max)")
// 打印“min is -6 and max is 109”
```

需要注意的是，元组的成员不需要在元组从函数中返回时命名，因为它们的名字已经在函数返回类型中指定了。

## 可选元组返回类型

---

如果函数返回的元组类型有可能整个元组都“没有值”，你可以使用可选的元组返回类型反映整个元组可以是 `nil` 的事实。你可以通过在元组类型的右括号后放置一个问号来定义一个可选元组，例如 `(Int, Int)?` 或 `(String, Int, Bool)?`

注意

可选元组类型如 `(Int, Int)?` 与元组包含可选类型如 `(Int?, Int?)` 是不同的。可选的元组类型，整个元组是可选的，而不只是元组中的每个元素值。

前面的 `minMax(array:)` 函数返回了一个包含两个 `Int` 值的元组。但是函数不会对传入的数组执行任何安全检查，如果 `array` 参数是一个空数组，如上定义的 `minMax(array:)` 在试图访问 `array[0]` 时会触发一个运行时错误。

为了安全地处理这个“空数组”问题，将 `minMax(array:)` 函数改写为使用可选元组返回类型，并且当数组为空时返回 `nil`：

```
func minMax(array: [Int]) -> (min: Int, max: Int)? {
    if array.isEmpty { return nil }
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..
```

你可以使用可选绑定来检查 `minMax(array:)` 函数返回的是一个存在的元组值还是 `nil`：

```
if let bounds = minMax(array: [8, -6, 2, 109, 3, 71]) {
    print("min is \(bounds.min) and max is \(bounds.max)")
}
// 打印“min is -6 and max is 109”
```

## 隐式返回的函数

---

如果一个函数的整个函数体是一个单行表达式，这个函数可以隐式地返回这个表达式。举个例子，以下的函数有着同样的作用：

```
func greeting(for person: String) -> String {
    "Hello, " + person + "!"
}
print(greeting(for: "Dave"))
// 打印 "Hello, Dave!"

func anotherGreeting(for person: String) -> String {
    return "Hello, " + person + "!"
}
print(anotherGreeting(for: "Dave"))
// 打印 "Hello, Dave!"
```

`greeting(for:)` 函数的完整定义是打招呼内容的返回，这就意味着它能使用隐式返回这样更简短的形式。`anotherGreeting(for:)` 函数返回同样的内容，却因为 `return` 关键字显得函数更长。任何一个可以被写成一行 `return` 语句的函数都可以忽略 `return`。

正如你将会在 简略的 Getter 声明 里看到的，一个属性的 getter 也可以使用隐式返回的形式。

## 函数参数标签和参数名称

---

每个函数参数都有一个参数标签 (*argument label*) 以及一个参数名称 (*parameter name*)。参数标签在调用函数的时候使用；调用的时候需要将函数的参数标签写在对应的参数前面。参数名称在函数的实现中使用。默认情况下，函数参数使用参数名称来作为它们的参数标签。

```
func someFunction(firstParameterName: Int, secondParameterName: Int) {
    // 在函数体内，firstParameterName 和 secondParameterName 代表参数中的第一个和第二个参数值
}
someFunction(firstParameterName: 1, secondParameterName: 2)
```

所有的参数都必须有一个独一无二的名字。虽然多个参数拥有同样的参数标签是可能的，但是一个唯一的函数标签能够使你的代码更具可读性。

## 指定参数标签

---

你可以在参数名称前指定它的参数标签，中间以空格分隔：

```
func someFunction(argumentLabel parameterName: Int) {
    // 在函数体内，parameterName 代表参数值
}
```

这个版本的 `greet(person:)` 函数，接收一个人的名字和他的家乡，并且返回一句问候：

```
func greet(person: String, from hometown: String) -> String {
    return "Hello \(person)! Glad you could visit from \(hometown)."
}
print(greet(person: "Bill", from: "Cupertino"))
// 打印 "Hello Bill! Glad you could visit from Cupertino."
```

参数标签的使用能够让一个函数在调用时更有表达力，更类似自然语言，并且仍保持了

函数内部的可读性以及清晰的意图。

## 忽略参数标签

---

如果你不希望为某个参数添加一个标签，可以使用一个下划线（`_`）来代替一个明确的参数标签。

```
func someFunction(_ firstParameterName: Int, secondParameterName: Int) {  
    // 在函数体内，firstParameterName 和 secondParameterName 代表参数中的第一个和第二个参数  
    // 值  
}  
someFunction(1, secondParameterName: 2)
```

如果一个参数有一个标签，那么在调用的时候必须使用标签来标记这个参数。

## 默认参数值

---

你可以在函数体中通过给参数赋值来为任意一个参数定义默认值 (*Default Value*)。当默认值被定义后，调用这个函数时可以忽略这个参数。

```
func someFunction(parameterWithoutDefault: Int, parameterWithDefault: Int = 12) {  
    // 如果你在调用时候不传第二个参数，parameterWithDefault 会值为 12 传入到函数体中。  
}  
someFunction(parameterWithoutDefault: 3, parameterWithDefault: 6) // parameterWithDefault  
= 6  
someFunction(parameterWithoutDefault: 4) // parameterWithDefault = 12
```

将不带有默认值的参数放在函数参数列表的最前。一般来说，没有默认值的参数更加的重要，将不带默认值的参数放在最前保证在函数调用时，非默认参数的顺序是一致的，同时也使得相同的函数在不同情况下调用时显得更为清晰。

## 可变参数

---

一个可变参数 (*variadic parameter*) 可以接受零个或多个值。函数调用时，你可以用可变参数来指定函数参数可以被传入不确定数量的输入值。通过在变量类型名后面加入（`...`）的方式来定义可变参数。

可变参数的传入值在函数体中变为此类型的一个数组。例如，一个叫做 `numbers` 的 `Double...` 型可变参数，在函数体内可以当做一个叫 `numbers` 的 `[Double]` 型的数组常量。

下面的这个函数用来计算一组任意长度数字的 算术平均数 (*arithmetic mean*)：

```
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)
// 返回 3.0, 是这 5 个数的平均数。
arithmeticMean(3, 8.25, 18.75)
// 返回 10.0, 是这 3 个数的平均数。
```

注意

一个函数最多只能拥有一个可变参数。

## 输入输出参数

函数参数默认是常量。试图在函数体中更改参数值将会导致编译错误。这意味着你不能错误地更改参数值。如果你想要一个函数可以修改参数的值，并且想要在这些修改在函数调用结束后仍然存在，那么就应该把这个参数定义为输入输出参数 (*In-Out Parameters*)。

定义一个输入输出参数时，在参数定义前加 `inout` 关键字。一个 **输入输出参数** 有传入函数的值，这个值被函数修改，然后被传出函数，替换原来的值。想获取更多的关于输入输出参数的细节和相关的编译器优化，请查看 [输入输出参数](#) 一节。

你只能传递变量给输入输出参数。你不能传入常量或者字面量，因为这些量是不能被修改的。当传入的参数作为输入输出参数时，需要在参数名前加 `&` 符，表示这个值可以被函数修改。

注意

输入输出参数不能有默认值，而且可变参数不能用 `inout` 标记。

下例中，`swapTwoInts(_:_)` 函数有两个分别叫做 `a` 和 `b` 的输入输出参数：

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

`swapTwoInts(_:_)` 函数简单地交换 `a` 与 `b` 的值。该函数先将 `a` 的值存到一个临时常量 `temporaryA` 中，然后将 `b` 的值赋给 `a`，最后将 `temporaryA` 赋值给 `b`。

你可以用两个 `Int` 型的变量来调用 `swapTwoInts(_:_)`。需要注意的是，`someInt` 和 `anotherInt` 在传入 `swapTwoInts(_:_)` 函数前，都加了 `&` 的前缀：

```
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
// 打印“someInt is now 107, and anotherInt is now 3”
```

从上面这个例子中，我们可以看到 `someInt` 和 `anotherInt` 的原始值在 `swapTwoInts(_:_:)` 函数中被修改，尽管它们的定义在函数体外。

### 注意

输入输出参数和返回值是不一样的。上面的 `swapTwoInts` 函数并没有定义任何返回值，但仍然修改了 `someInt` 和 `anotherInt` 的值。输入输出参数是函数对函数体外产生影响的另一种方式。

## 函数类型

每个函数都有种特定的 **函数类型**，函数的类型由函数的参数类型和返回类型组成。

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简单的数学函数：`addTwoInts` 和 `multiplyTwoInts`。这两个函数都接受两个 `Int` 值，返回一个 `Int` 值。

这两个函数的类型是 `(Int, Int) -> Int`，可以解读为：

“这个函数类型有两个 `Int` 型的参数并返回一个 `Int` 型的值”。

下面是另一个例子，一个没有参数，也没有返回值的函数：

```
func printHelloWorld() {
    print("hello, world")
}
```

这个函数的类型是：`() -> Void`，或者叫“没有参数，并返回 `Void` 类型的函数”。

## 使用函数类型

在 Swift 中，使用函数类型就像使用其他类型一样。例如，你可以定义一个类型为函数的常量或变量，并将适当的函数赋值给它：

```
var mathFunction: (Int, Int) -> Int = addTwoInts
```

这段代码可以被解读为：

”定义一个叫做 `mathFunction` 的变量，类型是‘一个有两个 `Int` 型的参数并返回一个 `Int` 型的值的函数’，并让这个新变量指向 `addTwoInts` 函数”。

`addTwoInts` 和 `mathFunction` 有同样的类型，所以这个赋值过程在 Swift 类型检查 (type-check) 中是允许的。

现在，你可以用 `mathFunction` 来调用被赋值的函数了：

```
print("Result: \(mathFunction(2, 3))")  
// Prints "Result: 5"
```

有相同匹配类型的不同函数可以被赋值给同一个变量，就像非函数类型的变量一样：

```
mathFunction = multiplyTwoInts  
print("Result: \(mathFunction(2, 3))")  
// Prints "Result: 6"
```

就像其他类型一样，当赋值一个函数给常量或变量时，你可以让 Swift 来推断其函数类型：

```
let anotherMathFunction = addTwoInts  
// anotherMathFunction 被推断为 (Int, Int) -> Int 类型
```

## 函数类型作为参数类型

---

你可以用 `(Int, Int) -> Int` 这样的函数类型作为另一个函数的参数类型。这样你可以将函数的一部分实现留给函数的调用者来提供。

下面是另一个例子，正如上面的函数一样，同样是输出某种数学运算结果：

```
func printMathResult(_ mathFunction: (Int, Int) -> Int, _ a: Int, _ b: Int) {  
    print("Result: \(mathFunction(a, b))")  
}  
printMathResult(addTwoInts, 3, 5)  
// 打印"Result: 8"
```

这个例子定义了 `printMathResult(_:_:_)` 函数，它有三个参数：第一个参数叫 `mathFunction`，类型是 `(Int, Int) -> Int`，你可以传入任何这种类型的函数；第二个和第三个参数叫 `a` 和 `b`，它们的类型都是 `Int`，这两个值作为已给出的函数的输入值。

当 `printMathResult(_:_:_)` 被调用时，它被传入 `addTwoInts` 函数和整数 `3` 和 `5`。它用传入 `3` 和 `5` 调用 `addTwoInts`，并输出结果：`8`。

`printMathResult(_:_:_)` 函数的作用就是输出另一个适当类型的数学函数的调用结果。它不关心传入函数是如何实现的，只关心传入的函数是不是一个正确的类型。这使得 `printMathResult(_:_:_)` 能以一种类型安全 (type-safe) 的方式将一部分功能转给调用者实现。

## 函数类型作为返回类型

---

你可以用函数类型作为另一个函数的返回类型。你需要做的是在返回箭头 ( $\rightarrow$ ) 后写一个完整的函数类型。

下面的这个例子中定义了两个简单函数，分别是 `stepForward(_)` 和 `stepBackward(_)`。`stepForward(_)` 函数返回一个比输入值大 `1` 的值。`stepBackward(_)` 函数返回一个比输入值小 `1` 的值。这两个函数的类型都是 `(Int) -> Int`：

```
func stepForward(_ input: Int) -> Int {  
    return input + 1  
}  
func stepBackward(_ input: Int) -> Int {  
    return input - 1  
}
```

如下名为 `chooseStepFunction(backward:)` 的函数，它的返回类型是 `(Int) -> Int` 类型的函数。`chooseStepFunction(backward:)` 根据布尔值 `backward` 来返回 `stepForward(_)` 函数或 `stepBackward(_)` 函数：

```
func chooseStepFunction(backward: Bool) -> (Int) -> Int {  
    return backward ? stepBackward : stepForward  
}
```

你现在可以用 `chooseStepFunction(backward:)` 来获得两个函数其中的一个：

```
var currentValue = 3  
let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)  
// moveNearerToZero 现在指向 stepBackward() 函数。
```

上面这个例子中计算出从 `currentValue` 逐渐接近到 `0` 是需要向正数走还是向负数走。`currentValue` 的初始值是 `3`，这意味着 `currentValue > 0` 为真 (true)，这将使得 `chooseStepFunction(_)` 返回 `stepBackward(_)` 函数。一个指向返回的函数的引用保存在了 `moveNearerToZero` 常量中。

现在，`moveNearerToZero` 指向了正确的函数，它可以被用来数到零：

```
print("Counting to zero:")  
// Counting to zero:  
while currentValue != 0 {  
    print("\(currentValue)... ")  
    currentValue = moveNearerToZero(currentValue)  
}  
print("zero!")  
// 3...  
// 2...  
// 1...  
// zero!
```

## 嵌套函数

---

到目前为止本章中你所见到的所有函数都叫 **全局函数** (*global functions*)，它们定义在全局域中。你也可以把函数定义在别的函数体中，称作 **嵌套函数** (*nested functions*)。

默认情况下，嵌套函数是对外界不可见的，但是可以被它们的外围函数 (enclosing function) 调用。一个外围函数也可以返回它的某一个嵌套函数，使得这个函数可以在其他域中被使用。

你可以用返回嵌套函数的方式重写 `chooseStepFunction(backward:)` 函数：

```
func chooseStepFunction(backward: Bool) -> (Int) -> Int {  
    func stepForward(input: Int) -> Int { return input + 1 }  
    func stepBackward(input: Int) -> Int { return input - 1 }  
    return backward ? stepBackward : stepForward  
}  
var currentValue = -4  
let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)  
// moveNearerToZero now refers to the nested stepForward() function  
while currentValue != 0 {  
    print("\(currentValue)... ")  
    currentValue = moveNearerToZero(currentValue)  
}  
print("zero!")  
// -4...  
// -3...  
// -2...  
// -1...  
// zero!
```

# 闭包 · GitBook

---



[runoob.com/manual/gitbook/swift5/source/\\_book/chapter2/07\\_Closures.html](https://runoob.com/manual/gitbook/swift5/source/_book/chapter2/07_Closures.html)

## 闭包

---

闭包是自包含的函数代码块，可以在代码中被传递和使用。Swift 中的闭包与 C 和 Objective-C 中的代码块 (blocks) 以及其他一些编程语言中的匿名函数 (Lambdas) 比较相似。

闭包可以捕获和存储其所在上下文中任意常量和变量的引用。被称为 **包裹常量和变量**。Swift 会为你管理在捕获过程中涉及到的所有内存操作。

### 注意

如果你不熟悉捕获 (capturing) 这个概念也不用担心，在 [值捕获](#) 章节有它更详细的介绍。

在 [函数](#) 章节中介绍的全局和嵌套函数实际上也是特殊的闭包，闭包采用如下三种形式之一：

- 全局函数是一个有名字但不会捕获任何值的闭包
- 嵌套函数是一个有名字并可以捕获其封闭函数域内值的闭包
- 闭包表达式是一个利用轻量级语法所写的可以捕获其上下文中变量或常量值的匿名闭包

Swift 的闭包表达式拥有简洁的风格，并鼓励在常见场景中进行语法优化，主要优化如下：

- 利用上下文推断参数和返回值类型
- 隐式返回单表达式闭包，即单表达式闭包可以省略 `return` 关键字
- 参数名称缩写
- 尾随闭包语法

## 闭包表达式

---

[嵌套函数](#) 作为复杂函数的一部分时，它自包含代码块式的定义和命名形式在使用上带来了方便。当然，编写未完整声明和没有函数名的类函数结构代码是很有用的，尤其是在编码中涉及到函数作为参数的那些方法时。

闭包表达式是一种构建内联闭包的方式，它的语法简洁。在保证不丢失它语法清晰明了的同时，闭包表达式提供了几种优化的语法简写形式。下面通过对 `sorted(by:)` 这一个案例的多次迭代改进来展示这个过程，每次迭代都使用了更加简明的方式描述了相同功能。。

## 排序方法

---

Swift 标准库提供了名为 `sorted(by:)` 的方法，它会基于你提供的排序闭包表达式的判断结果对数组中的值（类型确定）进行排序。一旦它完成排序过程，`sorted(by:)` 方法会返回一个与旧数组类型大小相同类型的新数组，该数组的元素有着正确的排序顺序。原数组不会被 `sorted(by:)` 方法修改。

下面的闭包表达式示例使用 `sorted(by:)` 方法对一个 `String` 类型的数组进行字母逆序排序。以下是初始数组：

```
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
```

`sorted(by:)` 方法接受一个闭包，该闭包函数需要传入与数组元素类型相同的两个值，并返回一个布尔类型值来表明当排序结束后传入的第一个参数排在第二个参数前面还是后面。如果第一个参数值出现在第二个参数值前面，排序闭包函数需要返回 `true`，反之返回 `false`。

该例子对一个 `String` 类型的数组进行排序，因此排序闭包函数类型需为 `(String, String) -> Bool`。

提供排序闭包函数的一种方式是撰写一个符合其类型要求的普通函数，并将其作为 `sorted(by:)` 方法的参数传入：

```
func backward(_ s1: String, _ s2: String) -> Bool {  
    return s1 > s2  
}  
var reversedNames = names.sorted(by: backward)  
// reversedNames 为 ["Ewa", "Daniella", "Chris", "Barry", "Alex"]
```

如果第一个字符串（`s1`）大于第二个字符串（`s2`），`backward(_:_:_)` 函数会返回 `true`，表示在新的数组中 `s1` 应该出现在 `s2` 前。对于字符串中的字符来说，“大于”表示“按照字母顺序较晚出现”。这意味着字母 “B” 大于字母 “A”，字符串 “Tom” 大于字符串 “Tim”。该闭包将进行字母逆序排序，“Barry” 将会排在 “Alex” 之前。

然而，以这种方式来编写一个实际上很简单的表达式（`a > b`），确实太过繁琐了。对于这个例子来说，利用闭包表达式语法可以更好地构造一个内联排序闭包。

## 闭包表达式语法

闭包表达式语法有如下的一般形式：

```
{ (parameters) -> return type in  
    statements  
}
```

闭包表达式参数可以是 `in-out` 参数，但不能设定默认值。如果你命名了可变参数，也可以使用此可变参数。元组也可以作为参数和返回值。

下面的例子展示了之前 `backward(_:_:_)` 函数对应的闭包表达式版本的代码：

```
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in
    return s1 > s2
})
```

需要注意的是内联闭包参数和返回值类型声明与 `backward(_:_)` 函数类型声明相同。在这两种方式中，都写成了 `(s1: String, s2: String) -> Bool`。然而在内联闭包表达式中，函数和返回值类型都写在大括号内，而不是大括号外。

闭包的函数体部分由关键字 `in` 引入。该关键字表示闭包的参数和返回值类型定义已经完成，闭包函数体即将开始。

由于这个闭包的函数体部分如此短，以至于可以将其改写成一行代码：

```
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in return s1 > s2 } )
```

该例中 `sorted(by:)` 方法的整体调用保持不变，一对圆括号仍然包裹住了方法的整个参数。然而，参数现在变成了内联闭包。

## 根据上下文推断类型

---

因为排序闭包函数是作为 `sorted(by:)` 方法的参数传入的，Swift 可以推断其参数和返回值的类型。`sorted(by:)` 方法被一个字符串数组调用，因此其参数必须是 `(String, String) -> Bool` 类型的函数。这意味着 `(String, String)` 和 `Bool` 类型并不需要作为闭包表达式定义的一部分。因为所有的类型都可以被正确推断，返回箭头 (`->`) 和围绕在参数周围的括号也可以被省略：

```
reversedNames = names.sorted(by: { s1, s2 in return s1 > s2 } )
```

实际上，通过内联闭包表达式构造的闭包作为参数传递给函数或方法时，总是能够推断出闭包的参数和返回值类型。这意味着闭包作为函数或者方法的参数时，你几乎不需要利用完整格式构造内联闭包。

尽管如此，你仍然可以明确写出有着完整格式的闭包。如果完整格式的闭包能够提高代码的可读性，则我们更鼓励采用完整格式的闭包。而在 `sorted(by:)` 方法这个例子里，显然闭包的目的就是排序。由于这个闭包是为了处理字符串数组的排序，因此读者能够推测出这个闭包是用于字符串处理的。

## 单表达式闭包的隐式返回

---

单行表达式闭包可以通过省略 `return` 关键字来隐式返回单行表达式的结果，如上版本的例子可以改写为：

```
reversedNames = names.sorted(by: { s1, s2 in s1 > s2 } )
```

在这个例子中，`sorted(by:)` 方法的参数类型明确了闭包必须返回一个 `Bool` 类型值。因为闭包函数体只包含了一个单一表达式 (`s1 > s2`)，该表达式返回 `Bool` 类型值，因此这里没有歧义，`return` 关键字可以省略。

## 参数名称缩写

---

Swift 自动为内联闭包提供了参数名称缩写功能，你可以直接通过 `$0` , `$1` , `$2` 来顺序调用闭包的参数，以此类推。

如果你在闭包表达式中使用参数名称缩写，你可以在闭包定义中省略参数列表，并且对应参数名称缩写的类型会通过函数类型进行推断。`in` 关键字也同样可以被省略，因为此时闭包表达式完全由闭包函数体构成：

```
reversedNames = names.sorted(by: { $0 > $1 } )
```

在这个例子中，`$0` 和 `$1` 表示闭包中第一个和第二个 `String` 类型的参数。

## 运算符方法

---

实际上还有一种更简短的方式来编写上面例子中的闭包表达式。Swift 的 `String` 类型定义了关于大于号 (`>`) 的字符串实现，其作为一个函数接受两个 `String` 类型的参数并返回 `Bool` 类型的值。而这正好与 `sorted(by:)` 方法的参数需要的函数类型相符合。因此，你可以简单地传递一个大于号，Swift 可以自动推断找到系统自带的那个字符串函数的实现：

```
reversedNames = names.sorted(by: >)
```

更多关于运算符方法的内容请查看 [运算符方法](#)。

## 尾随闭包

---

如果你需要将一个很长的闭包表达式作为最后一个参数传递给函数，将这个闭包替换成为尾随闭包的形式很有用。尾随闭包是一个书写在函数圆括号之后的闭包表达式，函数支持将其作为最后一个参数调用。在使用尾随闭包时，你不用写出它的参数标签：

```
func someFunctionThatTakesAClosure(closure: () -> Void) {  
    // 函数体部分  
}
```

```
// 以下是不使用尾随闭包进行函数调用  
someFunctionThatTakesAClosure(closure: {  
    // 闭包主体部分  
})
```

```
// 以下是使用尾随闭包进行函数调用  
someFunctionThatTakesAClosure() {  
    // 闭包主体部分  
}
```

在 [闭包表达式语法](#) 上章节中的字符串排序闭包可以作为尾随包的形式改写在 `sorted(by:)` 方法圆括号的外面：

```
reversedNames = names.sorted() { $0 > $1 }
```

如果闭包表达式是函数或方法的唯一参数，则当你使用尾随闭包时，你甚至可以把 `()` 省略掉：

```
reversedNames = names.sorted { $0 > $1 }
```

当闭包非常长以至于不能在一行中进行书写时，尾随闭包变得非常有用。举例来说，Swift 的 `Array` 类型有一个 `map(_:)` 方法，这个方法获取一个闭包表达式作为其唯一参数。该闭包函数会为数组中的每一个元素调用一次，并返回该元素所映射的值。具体的映射方式和返回值类型由闭包来指定。

当提供给数组的闭包应用于每个数组元素后，`map(_:)` 方法将返回一个新的数组，数组中包含了与原数组中的元素一一对应的映射后的值。

下例介绍了如何在 `map(_:)` 方法中使用尾随闭包将 `Int` 类型数组 `[16, 58, 510]` 转换为包含对应 `String` 类型的值的数组 `["OneSix", "FiveEight", "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]
```

如上代码创建了一个整型数位和它们英文版本名字相映射的字典。同时还定义了一个准备转换为字符串数组的整型数组。

你现在可以通过传递一个尾随闭包给 `numbers` 数组的 `map(_:)` 方法来创建对应的字符串版本数组：

```
let strings = numbers.map {
    (number) -> String in
    var number = number
    var output = ""
    repeat {
        output = digitNames[number % 10]! + output
        number /= 10
    } while number > 0
    return output
}
// strings 常量被推断为字符串类型数组，即 [String]
// 其值为 ["OneSix", "FiveEight", "FiveOneZero"]
```

`map(_:)` 为数组中每一个元素调用了一次闭包表达式。你不需要指定闭包的输入参数 `number` 的类型，因为可以通过要映射的数组类型进行推断。

在该例中，局部变量 `number` 的值由闭包中的 `number` 参数获得，因此可以在闭包函数体内对其进行修改，(闭包或者函数的参数总是常量)，闭包表达式指定了返回类型为 `String`，以表明存储映射值的新数组类型为 `String`。

闭包表达式在每次被调用的时候创建了一个叫做 `output` 的字符串并返回。其使用求余运算符 (`number % 10`) 计算最后一位数字并利用 `digitNames` 字典获取所映射的字符串。这个闭包能够用于创建任意正整数的字符串表示。

## 注意

字典 `digitNames` 下标后跟着一个叹号（`!`），因为字典下标返回一个可选值（optional value），表明该键不存在时会查找失败。在上例中，由于可以确定 `number % 10` 总是 `digitNames` 字典的有效下标，因此叹号可以用于强制解包（force-unwrap）存储在下标的可选类型的返回值中的 `String` 类型的值。

从 `digitNames` 字典中获取的字符串被添加到 `output` 的前部，逆序建立了一个字符串版本的数字。（在表达式 `number % 10` 中，如果 `number` 为 `16`，则返回 `6`，`58` 返回 `8`，`510` 返回 `0`。）

`number` 变量之后除以 `10`。因为其是整数，在计算过程中未除尽部分被忽略。因此 `16` 变成了 `1`，`58` 变成了 `5`，`510` 变成了 `51`。

整个过程重复进行，直到 `number /= 10` 为 `0`，这时闭包会将字符串 `output` 返回，而 `map(_:)` 方法则会将字符串添加到映射数组中。

在上面的例子中，通过尾随闭包语法，优雅地在函数后封装了闭包的具体功能，而不再需要将整个闭包包裹在 `map(_:)` 方法的括号内。

## 值捕获

闭包可以在其被定义的上下文中捕获常量或变量。即使定义这些常量和变量的原作用域已经不存在，闭包仍然可以在闭包函数体内引用和修改这些值。

Swift 中，可以捕获值的闭包的最简单形式是嵌套函数，也就是定义在其他函数的函数体内的函数。嵌套函数可以捕获其外部函数所有的参数以及定义的常量和变量。

举个例子，这有一个叫做 `makeIncrementer` 的函数，其包含了一个叫做 `incrementer` 的嵌套函数。嵌套函数 `incrementer()` 从上下文中捕获了两个值，`runningTotal` 和 `amount`。捕获这些值之后，`makeIncrementer` 将 `incrementer` 作为闭包返回。每次调用 `incrementer` 时，其会以 `amount` 作为增量增加 `runningTotal` 的值。

```
func makeIncrementer(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementer() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementer
}
```

`makeIncrementer` 返回类型为 `() -> Int`。这意味着其返回的是一个函数，而非一个简单类型的值。该函数在每次调用时不接受参数，只返回一个 `Int` 类型的值。关于函数返回其他函数的内容，请查看 [函数类型作为返回类型](#)。

`makeIncrementer(forIncrement:)` 函数定义了一个初始值为 `0` 的整型变量 `runningTotal`，用来存储当前总计数值。该值为 `incrementer` 的返回值。

`makeIncrementer(forIncrement:)` 有一个 `Int` 类型的参数，其外部参数名为 `forIncrement`，内部参数名为 `amount`，该参数表示每次 `incrementer` 被调用时 `runningTotal` 将要增加的量。`makeIncrementer` 函数还定义了一个嵌套函数 `incrementer`，用来执行实际的增加操作。该函数简单地使 `runningTotal` 增加 `amount`，并将其返回。

如果我们单独考虑嵌套函数 `incrementer()`，会发现它有些不同寻常：

```
func incrementer() -> Int {  
    runningTotal += amount  
    return runningTotal  
}
```

`incrementer()` 函数并没有任何参数，但是在函数体内访问了 `runningTotal` 和 `amount` 变量。这是因为它从外围函数捕获了 `runningTotal` 和 `amount` 变量的引用。捕获引用保证了 `runningTotal` 和 `amount` 变量在调用完 `makeIncrementer` 后不会消失，并且保证了在下一次执行 `incrementer` 函数时，`runningTotal` 依旧存在。

### 注意

为了优化，如果一个值不会被闭包改变，或者在闭包创建后不会改变，Swift 可能会改为捕获并保存一份对值的拷贝。

Swift 也会负责被捕获变量的所有内存管理工作，包括释放不再需要的变量。

下面是一个使用 `makeIncrementer` 的例子：

```
let incrementByTen = makeIncrementer(forIncrement: 10)
```

该例子定义了一个叫做 `incrementByTen` 的常量，该常量指向一个每次调用会将其 `runningTotal` 变量增加 `10` 的 `incrementer` 函数。调用这个函数多次可以得到以下结果：

```
incrementByTen()  
// 返回的值为10  
incrementByTen()  
// 返回的值为20  
incrementByTen()  
// 返回的值为30
```

如果你创建了另一个 `incrementer`，它会有属于自己的引用，指向一个全新、独立的 `runningTotal` 变量：

```
let incrementBySeven = makeIncrementer(forIncrement: 7)  
incrementBySeven()  
// 返回的值为7
```

再次调用原来的 `incrementByTen` 会继续增加它自己的 `runningTotal` 变量，该变量和 `incrementBySeven` 中捕获的变量没有任何联系：

```
incrementByTen()  
// 返回的值为40
```

### 注意

如果你将闭包赋值给一个类实例的属性，并且该闭包通过访问该实例或其成员而捕获了该实例，你将在闭包和该实例间创建一个循环强引用。Swift 使用捕获列表来打破这种循环强引用。更多信息，请参考 [闭包引起的循环强引用](#)。

## 闭包是引用类型

上面的例子中，`incrementBySeven` 和 `incrementByTen` 都是常量，但是这些常量指向的闭包仍然可以增加其捕获的变量的值。这是因为函数和闭包都是引用类型。

无论你将函数或闭包赋值给一个常量还是变量，你实际上都是将常量或变量的值设置为对应函数或闭包的引用。上面的例子中，指向闭包的引用 `incrementByTen` 是一个常量，而并非闭包内容本身。

这也意味着如果你将闭包赋值给了两个不同的常量或变量，两个值都会指向同一个闭包：

```
let alsoIncrementByTen = incrementByTen  
alsoIncrementByTen()  
// 返回的值为50
```

## 逃逸闭包

当一个闭包作为参数传到一个函数中，但是这个闭包在函数返回之后才被执行，我们称该闭包从函数中逃逸。当你定义接受闭包作为参数的函数时，你可以在参数名之前标注 `@escaping`，用来指明这个闭包是允许“逃逸”出这个函数的。

一种能使闭包“逃逸”出函数的方法是，将这个闭包保存在一个函数外部定义的变量中。举个例子，很多启动异步操作的函数接受一个闭包参数作为 completion handler。这类函数会在异步操作开始之后立刻返回，但是闭包直到异步操作结束后才会被调用。在这种情况下，闭包需要“逃逸”出函数，因为闭包需要在函数返回之后被调用。例如：

```
var completionHandlers: [() -> Void] = []  
func someFunctionWithEscapingClosure(completionHandler: @escaping () -> Void) {  
    completionHandlers.append(completionHandler)  
}
```

`someFunctionWithEscapingClosure(_)` 函数接受一个闭包作为参数，该闭包被添加到一个函数外定义的数组中。如果你不将这个参数标记为 `@escaping`，就会得到一个编译错误。

将一个闭包标记为 `@escaping` 意味着你必须在闭包中显式地引用 `self`。比如说，在下面的代码中，传递到 `someFunctionWithEscapingClosure(_)` 中的闭包是一个逃逸闭包，这意味着它需要显式地引用 `self`。相对的，传递到

`someFunctionWithNonEscapingClosure(_)` 中的闭包是一个非逃逸闭包，这意味着它可以隐式引用 `self`。

```
func someFunctionWithNonEscapingClosure(closure: () -> Void) {
    closure()
}

class SomeClass {
    var x = 10
    func doSomething() {
        someFunctionWithEscapingClosure { self.x = 100 }
        someFunctionWithNonEscapingClosure { x = 200 }
    }
}

let instance = SomeClass()
instance.doSomething()
print(instance.x)
// 打印出“200”

completionHandlers.first?()
print(instance.x)
// 打印出“100”
```

## 自动闭包

---

自动闭包是一种自动创建的闭包，用于包装传递给函数作为参数的表达式。这种闭包不接受任何参数，当它被调用的时候，会返回被包装在其中的表达式的值。这种便利语法让你能够省略闭包的花括号，用一个普通的表达式来代替显式的闭包。

我们经常会调用采用自动闭包的函数，但是很少去实现这样的函数。举个例子来说，`assert(condition:message:file:line:)` 函数接受自动闭包作为它的 `condition` 参数和 `message` 参数；它的 `condition` 参数仅会在 debug 模式下被求值，它的 `message` 参数仅当 `condition` 参数为 `false` 时被计算求值。

自动闭包让你能够延迟求值，因为直到你调用这个闭包，代码段才会被执行。延迟求值对于那些有副作用（Side Effect）和高计算成本的代码来说是很有益处的，因为它使得你能控制代码的执行时机。下面的代码展示了闭包如何延时求值。

```
var customersInLine = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
print(customersInLine.count)
// 打印出“5”

let customerProvider = { customersInLine.remove(at: 0) }
print(customersInLine.count)
// 打印出“5”

print("Now serving \(customerProvider())!")
// Prints "Now serving Chris!"
print(customersInLine.count)
// 打印出“4”
```

尽管在闭包的代码中，`customersInLine` 的第一个元素被移除了，不过在闭包被调用之前，这个元素是不会被移除的。如果这个闭包永远不被调用，那么在闭包里面的表达式将永远不会执行，那意味着列表中的元素永远不会被移除。请注意，`customerProvider` 的类型不是 `String`，而是 `() -> String`，一个没有参数且返回值为 `String` 的函数。

将闭包作为参数传递给函数时，你能获得同样的延时求值行为。

```
// customersInLine is ["Alex", "Ewa", "Barry", "Daniella"]
func serve(customer customerProvider: () -> String) {
    print("Now serving \((customerProvider())!)")
}
serve(customer: { customersInLine.remove(at: 0) } )
// 打印出“Now serving Alex!”
```

上面的 `serve(customer:)` 函数接受一个返回顾客名字的显式的闭包。下面这个版本的 `serve(customer:)` 完成了相同的操作，不过它并没有接受一个显式的闭包，而是通过将参数标记为 `@autoclosure` 来接收一个自动闭包。现在你可以将该函数当作接受 `String` 类型参数（而非闭包）的函数来调用。`customerProvider` 参数将自动转化为一个闭包，因为该参数被标记了 `@autoclosure` 特性。

```
// customersInLine is ["Ewa", "Barry", "Daniella"]
func serve(customer customerProvider: @autoclosure () -> String) {
    print("Now serving \((customerProvider())!)")
}
serve(customer: customersInLine.remove(at: 0))
// 打印“Now serving Ewa!”
```

### 注意

过度使用 `autoclosures` 会让你的代码变得难以理解。上下文和函数名应该能够清晰地表明求值是被延迟执行的。

如果你想让一个自动闭包可以“逃逸”，则应该同时使用 `@autoclosure` 和 `@escaping` 属性。`@escaping` 属性的讲解见上面的 [逃逸闭包](#)。

```
// customersInLine i= ["Barry", "Daniella"]
var customerProviders: [() -> String] = []
func collectCustomerProviders(_ customerProvider: @autoclosure @escaping () -> String) {
    customerProviders.append(customerProvider)
}
collectCustomerProviders(customersInLine.remove(at: 0))
collectCustomerProviders(customersInLine.remove(at: 0))

print("Collected \(customerProviders.count) closures.")
// 打印“Collected 2 closures.”
for customerProvider in customerProviders {
    print("Now serving \((customerProvider())!)")
}
// 打印“Now serving Barry!”
// 打印“Now serving Daniella!”
```

在上面的代码中，`collectCustomerProviders(_)` 函数并没有调用传入的 `customerProvider` 闭包，而是将闭包追加到了 `customerProviders` 数组中。这个数组定义在函数作用域范围外，这意味着数组内的闭包能够在函数返回之后被调用。因此，`customerProvider` 参数必须允许“逃逸”出函数作用域。

# 枚举 · GitBook

---



[runoob.com/manual/gitbook/swift5/source/\\_book/chapter2/08\\_Enumerations.html](https://runoob.com/manual/gitbook/swift5/source/_book/chapter2/08_Enumerations.html)

## 枚举

---

枚举为一组相关的值定义了一个共同的类型，使你可以在你的代码中以类型安全的方式来使用这些值。

如果你熟悉 C 语言，你会知道在 C 语言中，枚举会为一组整型值分配相关联的名称。Swift 中的枚举更加灵活，不必给每一个枚举成员提供一个值。如果给枚举成员提供一个值（称为原始值），则该值的类型可以是字符串、字符，或是一个整型值或浮点数。

此外，枚举成员可以指定任意类型的关联值存储到枚举成员中，就像其他语言中的联合体（unions）和变体（variants）。你可以在一个枚举中定义一组相关的枚举成员，每一个枚举成员都可以有适当类型的关联值。

在 Swift 中，枚举类型是一等（first-class）类型。它们采用了很多在传统上只被类（class）所支持的特性，例如计算属性（computed properties），用于提供枚举值的附加信息，实例方法（instance methods），用于提供和枚举值相关联的功能。枚举也可以定义构造函数（initializers）来提供一个初始值；可以在原始实现的基础上扩展它们的功能；还可以遵循协议（protocols）来提供标准的功能。

想了解更多相关信息，请参见 [属性](#)，[方法](#)，[构造过程](#)，[扩展](#) 和 [协议](#)。

## 枚举语法

---

使用 `enum` 关键词来创建枚举并且把它们的整个定义放在一对大括号内：

```
enum SomeEnumeration {  
    // 枚举定义放在这里  
}
```

下面是用枚举表示指南针四个方向的例子：

```
enum CompassPoint {  
    case north  
    case south  
    case east  
    case west  
}
```

枚举中定义的值（如 `north`，`south`，`east` 和 `west`）是这个枚举的成员值（或成员）。你可以使用 `case` 关键字来定义一个新的枚举成员值。

## 注意

与 C 和 Objective-C 不同，Swift 的枚举成员在被创建时不会被赋予一个默认的整型值。在上面的 `CompassPoint` 例子中，`north`，`south`，`east` 和 `west` 不会被隐式地赋值为 `0`，`1`，`2` 和 `3`。相反，这些枚举成员本身就是完备的值，这些值的类型是已经明确定义好的 `CompassPoint` 类型。

多个成员值可以出现在同一行上，用逗号隔开：

```
enum Planet {  
    case mercury, venus, earth, mars, jupiter, saturn, uranus, neptune  
}
```

每个枚举定义了一个全新的类型。像 Swift 中其他类型一样，它们的名字（例如 `CompassPoint` 和 `Planet`）以一个大写字母开头。给枚举类型起一个单数名字而不是复数名字，以便于：

```
var directionToHead = CompassPoint.west
```

`directionToHead` 的类型可以在它被 `CompassPoint` 的某个值初始化时推断出来。一旦 `directionToHead` 被声明为 `CompassPoint` 类型，你可以使用更简短的点语法将其设置为另一个 `CompassPoint` 的值：

```
directionToHead = .east
```

当 `directionToHead` 的类型已知时，再次为其赋值可以省略枚举类型名。在使用具有显式类型的枚举值时，这种写法让代码具有更好的可读性。

## 使用 Switch 语句匹配枚举值

你可以使用 `switch` 语句匹配单个枚举值：

```
directionToHead = .south  
switch directionToHead {  
case .north:  
    print("Lots of planets have a north")  
case .south:  
    print("Watch out for penguins")  
case .east:  
    print("Where the sun rises")  
case .west:  
    print("Where the skies are blue")  
}  
// 打印“Watch out for penguins”
```

你可以这样理解这段代码：

“判断 `directionToHead` 的值。当它等于 `.north`，打印 `“Lots of planets have a north”`。当它等于 `.south`，打印 `“Watch out for penguins”`。”

.....以此类推。

正如在 [控制流](#) 中介绍的那样，在判断一个枚举类型的值时，`switch` 语句必须穷举所有情况。如果忽略了 `.west` 这种情况，上面那段代码将无法通过编译，因为它没有考虑到 `CompassPoint` 的全部成员。强制穷举确保了枚举成员不会被意外遗漏。

当不需要匹配每个枚举成员的时候，你可以提供一个 `default` 分支来涵盖所有未明确处理的枚举成员：

```
let somePlanet = Planet.earth
switch somePlanet {
case .earth:
    print("Mostly harmless")
default:
    print("Not a safe place for humans")
}
// 打印“Mostly harmless”
```

## 枚举成员的遍历

---

在一些情况下，你会需要得到一个包含枚举所有成员的集合。可以通过如下代码实现：

令枚举遵循 `Caselterable` 协议。Swift 会生成一个 `allCases` 属性，用于表示一个包含枚举所有成员的集合。下面是一个例子：

```
enum Beverage: Caselterable {
    case coffee, tea, juice
}
let numberOfChoices = Beverage.allCases.count
print("\(numberOfChoices) beverages available")
// 打印“3 beverages available”
```

在前面的例子中，通过 `Beverage.allCases` 可以访问到包含 `Beverage` 枚举所有成员的集合。`allCases` 的使用方法和其它一般集合一样——集合中的元素是枚举类型的实例，所以在上面的情况下，这些元素是 `Beverage` 值。在前面的例子中，统计了总共有多少个枚举成员。而在下面的例子中，则使用 `for` 循环来遍历所有枚举成员。

```
for beverage in Beverage.allCases {
    print(beverage)
}
// coffee
// tea
// juice
```

在前面的例子中，使用的语法表明这个枚举遵循 `CaseIterable` 协议。想了解 protocols 相关信息，请参见 [协议](#)。

## 关联值

---

枚举语法那一小节的例子演示了如何定义和分类枚举的成员。你可以为 `Planet.earth` 设置一个常量或者变量，并在赋值之后查看这个值。然而，有时候把其他类型的值和成员值一起存储起来会很有用。这额外的信息称为 **关联值**，并且你每次在代码中使用该枚举

成员时，还可以修改这个关联值。

你可以定义 Swift 枚举来存储任意类型的关联值，如果需要的话，每个枚举成员的关联值类型可以各不相同。枚举的这种特性跟其他语言中的可识别联合（discriminated unions），标签联合（tagged unions），或者变体（variants）相似。

例如，假设一个库存跟踪系统需要利用两种不同类型的条形码来跟踪商品。有些商品上标有使用 0 到 9 的数字的 UPC 格式的一维条形码。每一个条形码都有一个代表数字系统的数字，该数字后接五位代表厂商代码的数字，接下来是五位代表“产品代码”的数字。最后一个数字是检查位，用来验证代码是否被正确扫描：

其他商品上标有 QR 码格式的二维码，它可以使  
用任何 ISO 8859-1 字符，并且可以编码一个最多  
拥有 2,953 个字符的字符串：

这便于库存跟踪系统用包含四个整型值的元组存  
储 UPC 码，以及用任意长度的字符串储存 QR  
码。



在 Swift 中，使用如下方式定义表示两种商品条形码的枚举：

```
enum Barcode {  
    case upc(Int, Int, Int, Int)  
    case qrCode(String)  
}
```

以上代码可以这么理解：



“定义一个名为 `Barcode` 的枚举类型，它的一个成员值是具有 `(Int, Int, Int, Int)` 类型关联值的 `upc`，另一个成员值是具有 `String` 类型关联值的 `qrCode`。”

这个定义不提供任何 `Int` 或 `String` 类型的关联值，它只是定义了，当 `Barcode` 常量和变量等于 `Barcode.upc` 或 `Barcode.qrCode` 时，可以存储的关联值的类型。

然后你可以使用任意一种条形码类型创建新的条形码，例如：

```
var productBarcode = Barcode.upc(8, 85909, 51226, 3)
```

上面的例子创建了一个名为 `productBarcode` 的变量，并将 `Barcode.upc` 赋值给它，  
关联的元组值为 `(8, 85909, 51226, 3)`。

同一个商品可以被分配一个不同类型的条形码，例如：

```
productBarcode = .qrCode("ABCDEFGHIJKLMNP")
```

这时，原始的 `Barcode.upc` 和其整数关联值被新的 `Barcode.qrCode` 和其字符串关联值所替代。`Barcode` 类型的常量和变量可以存储一个 `.upc` 或者一个 `.qrCode`（连同它们的关联值），但是在同一时间只能存储这两个值中的一个。

你可以使用一个 switch 语句来检查不同的条形码类型，和之前使用 Switch 语句来匹配枚举值的例子一样。然而，这一次，关联值可以被提取出来作为 switch 语句的一部分。你可以在 `switch` 的 case 分支代码中提取每个关联值作为一个常量（用 `let` 前缀）或者作为一个变量（用 `var` 前缀）来使用：

```
switch productBarcode {  
    case .upc(let numberSystem, let manufacturer, let product, let check):  
        print("UPC: \(numberSystem), \(manufacturer), \(product), \(check).")  
    case .qrCode(let productCode):  
        print("QR code: \(productCode).")  
}  
// 打印“QR code: ABCDEFGHIJKLMNOP.”
```

如果一个枚举成员的所有关联值都被提取为常量，或者都被提取为变量，为了简洁，你可以只在成员名称前标注一个 `let` 或者 `var`：

```
switch productBarcode {  
    case let .upc(numberSystem, manufacturer, product, check):  
        print("UPC: \(numberSystem), \(manufacturer), \(product), \(check).")  
    case let .qrCode(productCode):  
        print("QR code: \(productCode).")  
}  
// 打印“QR code: ABCDEFGHIJKLMNOP.”
```

## 原始值

---

在 [关联值](#) 小节的条形码例子中，演示了如何声明存储不同类型关联值的枚举成员。作为关联值的替代选择，枚举成员可以被默认值（称为 **原始值**）预填充，这些原始值的类型必须相同。

这是一个使用 ASCII 码作为原始值的枚举：

```
enum ASCIIControlCharacter: Character {  
    case tab = "\t"  
    case lineFeed = "\n"  
    case carriageReturn = "\r"  
}
```

枚举类型 `ASCIIControlCharacter` 的原始值类型被定义为 `Character`，并设置了一些比较常见的 ASCII 控制字符。`Character` 的描述详见 [字符串和字符](#) 部分。

原始值可以是字符串、字符，或者任意整型值或浮点型值。每个原始值在枚举声明中必须是唯一的。

### 注意

原始值和关联值是不同的。原始值是在定义枚举时被预先填充的值，像上述三个 ASCII 码。对于一个特定的枚举成员，它的原始值始终不变。关联值是创建一个基于枚举成员的常量或变量时才设置的值，枚举成员的关联值可以变化。

## 原始值的隐式赋值

---

在使用原始值为整数或者字符串类型的枚举时，不需要显式地为每一个枚举成员设置原始值，Swift 将会自动为你赋值。

例如，当使用整数作为原始值时，隐式赋值的值依次递增 1。如果第一个枚举成员没有设置原始值，其原始值将为 0。

下面的枚举是对之前 `Planet` 这个枚举的一个细化，利用整型的原始值来表示每个行星在太阳系中的顺序：

```
enum Planet: Int {  
    case mercury = 1, venus, earth, mars, jupiter, saturn, uranus, neptune  
}
```

在上面的例子中，`Plant.mercury` 的显式原始值为 1，`Planet.venus` 的隐式原始值为 2，依次类推。

当使用字符串作为枚举类型的原始值时，每个枚举成员的隐式原始值为该枚举成员的名称。

下面的例子是 `CompassPoint` 枚举的细化，使用字符串类型的原始值来表示各个方向的名称：

```
enum CompassPoint: String {  
    case north, south, east, west  
}
```

上面例子中，`CompassPoint.south` 拥有隐式原始值 `south`，依次类推。

使用枚举成员的 `rawValue` 属性可以访问该枚举成员的原始值：

```
let earthsOrder = Planet.earth.rawValue  
// earthsOrder 值为 3  
  
let sunsetDirection = CompassPoint.west.rawValue  
// sunsetDirection 值为 "west"
```

## 使用原始值初始化枚举实例

---

如果在定义枚举类型的时候使用了原始值，那么将会自动获得一个初始化方法，这个方法接收一个叫做 `rawValue` 的参数，参数类型即为原始值类型，返回值则是枚举成员或 `nil`。你可以使用这个初始化方法来创建一个新的枚举实例。

这个例子利用原始值 7 创建了枚举成员 `Uranus`：

```
let possiblePlanet = Planet(rawValue: 7)  
// possiblePlanet 类型为 Planet? 值为 Planet.uranus
```

然而，并非所有 `Int` 值都可以找到一个匹配的行星。因此，原始值构造器总是返回一个可选的枚举成员。在上面的例子中，`possiblePlanet` 是 `Planet?` 类型，或者说“可选的 `Planet`”。

## 注意

原始值构造器是一个可失败构造器，因为并不是每一个原始值都有与之对应的枚举成员。更多信息请参见 [可失败构造器](#)

如果你试图寻找一个位置为 `11` 的行星，通过原始值构造器返回的可选 `Planet` 值将是 `nil`：

```
let positionToFind = 11
if let somePlanet = Planet(rawValue: positionToFind) {
    switch somePlanet {
        case .earth:
            print("Mostly harmless")
        default:
            print("Not a safe place for humans")
    }
} else {
    print("There isn't a planet at position \(positionToFind)")
}
// 打印"There isn't a planet at position 11"
```

这个例子使用了可选绑定（optional binding），试图通过原始值 `11` 来访问一个行星。`if let somePlanet = Planet(rawValue: 11)` 语句创建了一个可选 `Planet`，如果可选 `Planet` 的值存在，就会赋值给 `somePlanet`。在这个例子中，无法检索到位置为 `11` 的行星，所以 `else` 分支被执行。

## 递归枚举

递归枚举是一种枚举类型，它有一个或多个枚举成员使用该枚举类型的实例作为关联值。使用递归枚举时，编译器会插入一个间接层。你可以在枚举成员前加上 `indirect` 来表示该成员可递归。

例如，下面的例子中，枚举类型存储了简单的算术表达式：

```
enum ArithmeticExpression {
    case number(Int)
    indirect case addition(ArithmeticExpression, ArithmeticExpression)
    indirect case multiplication(ArithmeticExpression, ArithmeticExpression)
}
```

你也可以在枚举类型开头加上 `indirect` 关键字来表明它的所有成员都是可递归的：

```
indirect enum ArithmeticExpression {
    case number(Int)
    case addition(ArithmeticExpression, ArithmeticExpression)
    case multiplication(ArithmeticExpression, ArithmeticExpression)
}
```

上面定义的枚举类型可以存储三种算术表达式：纯数字、两个表达式相加、两个表达式相乘。枚举成员 `addition` 和 `multiplication` 的关联值也是算术表达式——这些关联值使得嵌套表达式成为可能。例如，表达式 `(5 + 4) * 2`，乘号右边是一个数字，左边则是另

一个表达式。因为数据是嵌套的，因而用来存储数据的枚举类型也需要支持这种嵌套——这意味着枚举类型需要支持递归。下面的代码展示了使用 `ArithmeticExpression` 这个递归枚举创建表达式 `(5 + 4) * 2`

```
let five = ArithmeticExpression.number(5)
let four = ArithmeticExpression.number(4)
let sum = ArithmeticExpression.addition(five, four)
let product = ArithmeticExpression.multiplication(sum, ArithmeticExpression.number(2))
```

要操作具有递归性质的数据结构，使用递归函数是一种直截了当的方式。例如，下面是一个对算术表达式求值的函数：

```
func evaluate(_ expression: ArithmeticExpression) -> Int {
    switch expression {
        case let .number(value):
            return value
        case let .addition(left, right):
            return evaluate(left) + evaluate(right)
        case let .multiplication(left, right):
            return evaluate(left) * evaluate(right)
    }
}

print(evaluate(product))
// 打印“18”
```

该函数如果遇到纯数字，就直接返回该数字的值。如果遇到的是加法或乘法运算，则分别计算左边表达式和右边表达式的值，然后相加或相乘。

# 类和结构体 · GitBook

 [runoob.com/manual/gitbook/swift5/source/\\_book/chapter2/09\\_Structures\\_And\\_Classes.html](https://runoob.com/manual/gitbook/swift5/source/_book/chapter2/09_Structures_And_Classes.html)

## 结构体和类

结构体和类作为一种通用而又灵活的结构，成为了人们构建代码的基础。你可以使用定义常量、变量和函数的语法，为你的结构体和类定义属性、添加方法。

与其他编程语言所不同的是，Swift 并不要求你为自定义的结构体和类的接口与实现代码分别创建文件。你只需在单一的文件中定义一个结构体或者类，系统将会自动生成面向其它代码的外部接口。

### 注意

通常一个类的实例被称为对象。然而相比其他语言，Swift 中结构体和类的功能更加相近，本章中所讨论的大部分功能都可以用在结构体或者类上。因此，这里会使用实例这个更通用的术语。

## 结构体和类对比

Swift 中结构体和类有很多共同点。两者都可以：

- 定义属性用于存储值
- 定义方法用于提供功能
- 定义下标操作用于通过下标语法访问它们的值
- 定义构造器用于设置初始值
- 通过扩展以增加默认实现之外的功能
- 遵循协议以提供某种标准功能

更多信息请参见 [属性](#)、[方法](#)、[下标](#)、[构造过程](#)、[扩展](#) 和 [协议](#)。

与结构体相比，类还有如下的附加功能：

- 继承允许一个类继承另一个类的特征
- 类型转换允许在运行时检查和解释一个类实例的类型
- 析构器允许一个类实例释放任何其所被分配的资源
- 引用计数允许对一个类的多次引用

更多信息请参见 [继承](#)、[类型转换](#)、[析构过程](#) 和 [自动引用计数](#)。

类支持的附加功能是以增加复杂性为代价的。作为一般准则，优先使用结构体，因为它们更容易理解，仅在适当或必要时才使用类。实际上，这意味着你的大多数自定义数据类型都会是结构体和枚举。更多详细的比较参见 [在结构和类之间进行选择](#)。

## 类型定义的语法

结构体和类有着相似的定义方式。你通过 `struct` 关键字引入结构体，通过 `class` 关键字引入类，并将它们的具体定义放在一对大括号中：

```
struct SomeStructure {  
    // 在这里定义结构体  
}  
class SomeClass {  
    // 在这里定义类  
}
```

### 注意

每当你定义一个新的结构体或者类时，你都是定义了一个新的 Swift 类型。请使用 `UpperCamelCase` 这种方式来命名类型（如这里的 `SomeClass` 和 `SomeStructure`），以便符合标准 Swift 类型的大写命名风格（如 `String`，`Int` 和 `Bool`）。请使用 `lowerCamelCase` 这种方式来命名属性和方法（如 `framerate` 和 `incrementCount`），以便和类型名区分。

以下是定义结构体和定义类的示例：

```
struct Resolution {  
    var width = 0  
    var height = 0  
}  
class VideoMode {  
    var resolution = Resolution()  
    var interlaced = false  
    var frameRate = 0.0  
    var name: String?  
}
```

在上面的示例中定义了一个名为 `Resolution` 的结构体，用来描述基于像素的分辨率。这个结构体包含了名为 `width` 和 `height` 的两个存储属性。存储属性是与结构体或者类绑定的，并存储在其中的常量或变量。当这两个属性被初始化为整数 `0` 的时候，它们会被推断为 `Int` 类型。

在上面的示例还定义了一个名为 `VideoMode` 的类，用来描述视频显示器的某个特定视频模式。这个类包含了四个可变的存储属性。第一个，`resolution`，被初始化为一个新的 `Resolution` 结构体的实例，属性类型被推断为 `Resolution`。新 `VideoMode` 实例同时还会初始化其它三个属性，它们分别是初始值为 `false` 的 `interlaced`（意为“非隔行视频”），初始值为 `0.0` 的 `frameRate`，以及值为可选 `String` 的 `name`。因为 `name` 是一个可选类型，它会被自动赋予一个默认值 `nil`，意为“没有 `name` 值”。

## 结构体和类的实例

`Resolution` 结构体和 `VideoMode` 类的定义仅描述了什么是 `Resolution` 和 `VideoMode`。它们并没有描述一个特定的分辨率（`resolution`）或者视频模式（`video mode`）。为此，你需要创建结构体或者类的一个实例。

创建结构体和类实例的语法非常相似：

```
let someResolution = Resolution()  
let someVideoMode = VideoMode()
```

结构体和类都使用构造器语法来创建新的实例。构造器语法的最简单形式是在结构体或者类的类型名称后跟随一对空括号，如 `Resolution()` 或 `VideoMode()`。通过这种方式所创建的类或者结构体实例，其属性均会被初始化为默认值。[构造过程](#)章节会对类和结构体的初始化进行更详细的讨论。

## 属性访问

---

你可以通过使用点语法访问实例的属性。其语法规则是，实例名后面紧跟属性名，两者以点号（`.`）分隔，不带空格：

```
print("The width of someResolution is \(someResolution.width)")  
// 打印 "The width of someResolution is 0"
```

在上面的例子中，`someResolution.width` 引用 `someResolution` 的 `width` 属性，返回 `width` 的初始值 `0`。

你也可以访问子属性，如 `VideoMode` 中 `resolution` 属性的 `width` 属性：

```
print("The width of someVideoMode is \(someVideoMode.resolution.width)")  
// 打印 "The width of someVideoMode is 0"
```

你也可以使用点语法为可变属性赋值：

```
someVideoMode.resolution.width = 1280  
print("The width of someVideoMode is now \(someVideoMode.resolution.width)")  
// 打印 "The width of someVideoMode is now 1280"
```

## 结构体类型的成员逐一构造器

---

所有结构体都有一个自动生成的成员逐一构造器，用于初始化新结构体实例中成员的属性。新实例中各个属性的初始值可以通过属性的名称传递到成员逐一构造器之中：

```
let vga = Resolution(width: 640, height: 480)
```

与结构体不同，类实例没有默认的成员逐一构造器。[构造过程](#)章节会对构造器进行更详细的讨论。

## 结构体和枚举是值类型

---

值类型是这样一种类型，当它被赋值给一个变量、常量或者被传递给一个函数的时候，其值会被拷贝。

在之前的章节中，你已经大量使用了值类型。实际上，Swift 中所有的基本类型：整数（integer）、浮点数（floating-point number）、布尔值（boolean）、字符串（string）、数组（array）和字典（dictionary），都是值类型，其底层也是使用结构体实现的。

Swift 中所有的结构体和枚举类型都是值类型。这意味着它们的实例，以及实例中所包含的任何值类型的属性，在代码中传递的时候都会被复制。

### 注意

标准库定义的集合，例如数组，字典和字符串，都对复制进行了优化以降低性能成本。新集合不会立即复制，而是跟原集合共享同一份内存，共享同样的元素。在集合的某个副本要被修改前，才会复制它的元素。而你在代码中看起来就像是立即发生了复制。

请看下面这个示例，其使用了上一个示例中的 `Resolution` 结构体：

```
let hd = Resolution(width: 1920, height: 1080)
var cinema = hd
```

在以上示例中，声明了一个名为 `hd` 的常量，其值为一个初始化为全高清视频分辨率（`1920` 像素宽，`1080` 像素高）的 `Resolution` 实例。

然后示例中又声明了一个名为 `cinema` 的变量，并将 `hd` 赋值给它。因为 `Resolution` 是一个结构体，所以会先创建一个现有实例的副本，然后将副本赋值给 `cinema`。尽管 `hd` 和 `cinema` 有着相同的宽（`width`）和高（`height`），但是在幕后它们是两个完全不同的实例。

下面，为了符合数码影院放映的需求（`2048` 像素宽，`1080` 像素高），`cinema` 的 `width` 属性被修改为稍微宽一点的 `2K` 标准：

```
cinema.width = 2048
```

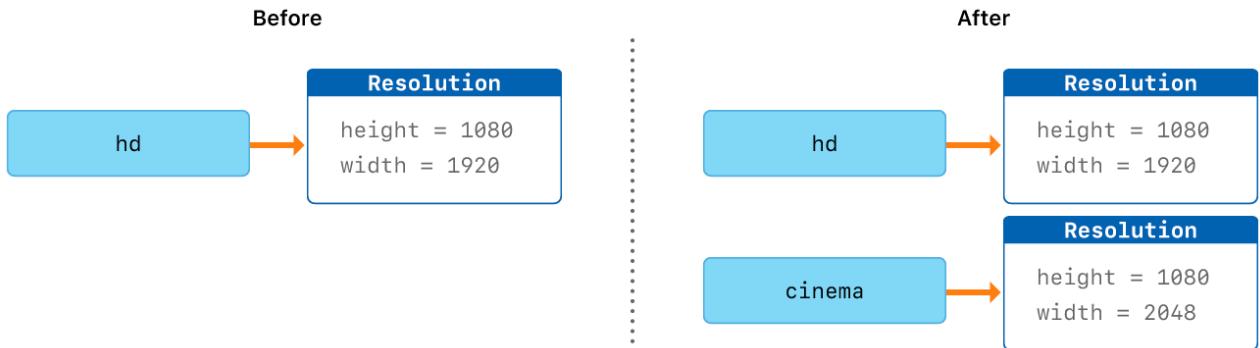
查看 `cinema` 的 `width` 属性，它的值确实改为了 `2048`：

```
print("cinema is now \(cinema.width) pixels wide")
// 打印 "cinema is now 2048 pixels wide"
```

然而，初始的 `hd` 实例中 `width` 属性还是 `1920`：

```
print("hd is still \(hd.width) pixels wide")
// 打印 "hd is still 1920 pixels wide"
```

将 `hd` 赋值给 `cinema` 时，`hd` 中所存储的值会拷贝到新的 `cinema` 实例中。结果就是两个完全独立的实例包含了相同的数值。由于两者相互独立，因此将 `cinema` 的 `width` 修改为 `2048` 并不会影响 `hd` 中的 `width` 的值，如下图所示：



枚举也遵循相同的行为准则：

```
enum CompassPoint {
    case north, south, east, west
    mutating func turnNorth() {
        self = .north
    }
}
var currentDirection = CompassPoint.west
let rememberedDirection = currentDirection
currentDirection.turnNorth()

print("The current direction is \(currentDirection)")
print("The remembered direction is \(rememberedDirection)")
// 打印 "The current direction is north"
// 打印 "The remembered direction is west"
```

当 `rememberedDirection` 被赋予了 `currentDirection` 的值，实际上它被赋予的是值的一个拷贝。赋值过程结束后再修改 `currentDirection` 的值并不影响 `rememberedDirection` 所储存的原始值的拷贝。

## 类是引用类型

与值类型不同，引用类型在被赋予到一个变量、常量或者被传递到一个函数时，其值不会被拷贝。因此，使用的是已存在实例的引用，而不是其拷贝。

请看下面这个示例，其使用了之前定义的 `VideoMode` 类：

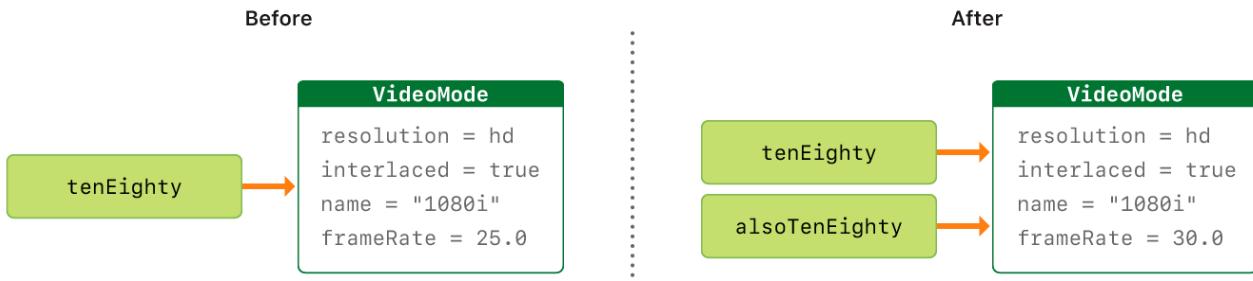
```
let tenEighty = VideoMode()
tenEighty.resolution = hd
tenEighty.interlaced = true
tenEighty.name = "1080i"
tenEighty.frameRate = 25.0
```

以上示例中，声明了一个名为 `tenEighty` 的常量，并让其引用一个 `VideoMode` 类的新实例。它的视频模式 (video mode) 被赋值为之前创建的 HD 分辨率 (`1920 * 1080`) 的一个拷贝。然后将它设置为隔行视频，名字设为 “`1080i`”，并将帧率设置为 `25.0` 帧每秒。

接下来，将 `tenEighty` 赋值给一个名为 `alsoTenEighty` 的新常量，并修改 `alsoTenEighty` 的帧率：

```
let alsoTenEighty = tenEighty  
alsoTenEighty.frameRate = 30.0
```

因为类是引用类型，所以 `tenEighty` 和 `alsoTenEighty` 实际上引用的是同一个 `VideoMode` 实例。换句话说，它们是同一个实例的两种叫法，如下图所示：



通过查看 `tenEighty` 的 `frameRate` 属性，可以看到它正确地显示了底层的 `VideoMode` 实例的新帧率 30.0：

```
print("The frameRate property of tenEighty is now \(tenEighty.frameRate)")  
// 打印 "The frameRate property of theEighty is now 30.0"
```

这个例子也显示了为何引用类型更加难以理解。如果 `tenEighty` 和 `alsoTenEighty` 在你代码中的位置相距很远，那么就很难找到所有修改视频模式的地方。无论在哪使用 `tenEighty`，你都要考虑使用 `alsoTenEighty` 的代码，反之亦然。相反，值类型就更容易理解了，因为你的源码中与同一个值交互的代码都很近。

需要注意的是 `tenEighty` 和 `alsoTenEighty` 被声明为常量而不是变量。然而你依然可以改变 `tenEighty.frameRate` 和 `alsoTenEighty.frameRate`，这是因为 `tenEighty` 和 `alsoTenEighty` 这两个常量的值并未改变。它们并不“存储”这个 `VideoMode` 实例，而仅是对 `VideoMode` 实例的引用。所以，改变的是底层 `VideoMode` 实例的 `frameRate` 属性，而不是指向 `VideoMode` 的常量引用的值。

## 恒等运算符

因为类是引用类型，所以多个常量和变量可能在幕后同时引用同一个类实例。（对于结构体和枚举来说，这并不成立。因为它们作为值类型，在被赋予到常量、变量或者传递到函数时，其值总是会被拷贝。）

判定两个常量或者变量是否引用同一个类实例有时很有用。为了达到这个目的，Swift 提供了两个恒等运算符：

- 相同 (`==`)
- 不相同 (`!=`)

使用这两个运算符检测两个常量或者变量是否引用了同一个实例：

```
if tenEighty === alsoTenEighty {  
    print("tenEighty and alsoTenEighty refer to the same VideoMode instance.")  
}  
// 打印 "tenEighty and alsoTenEighty refer to the same VideoMode instance."
```

请注意，“相同”（用三个等号表示，`====`）与“等于”（用两个等号表示，`==`）的不同。“相同”表示两个类类型（class type）的常量或者变量引用同一个类实例。“等于”表示两个实例的值“相等”或“等价”，判定时要遵照设计者定义的评判标准。

当在定义你的自定义结构体和类的时候，你有义务来决定判定两个实例“相等”的标准。在章节[等价操作符](#)中将会详细介绍实现自定义`==`和`!=`运算符的流程。

## 指针

---

如果你有 C，C++ 或者 Objective-C 语言的经验，那么你也许会知道这些语言使用指针来引用内存中的地址。Swift 中引用了某个引用类型实例的常量或变量，与 C 语言中的指针类似，不过它并不直接指向某个内存地址，也不要求你使用星号（`*`）来表明你在创建一个引用。相反，Swift 中引用的定义方式与其它的常量或变量的一样。如果需要直接与指针交互，你可以使用标准库提供的指针和缓冲区类型——参见[手动管理内存](#)。

# 属性 · GitBook

---



[runoob.com/manual/gitbook/swift5/source/\\_book/chapter2/10\\_Properties.html](https://runoob.com/manual/gitbook/swift5/source/_book/chapter2/10_Properties.html)

## 属性

---

属性将值与特定的类、结构体或枚举关联。存储属性会将常量和变量存储为实例的一部分，而计算属性则是直接计算（而不是存储）值。计算属性可以用于类、结构体和枚举，而存储属性只能用于类和结构体。

存储属性和计算属性通常与特定类型的实例关联。但是，属性也可以直接与类型本身关联，这种属性称为类型属性。

另外，还可以定义属性观察器来监控属性值的变化，以此来触发自定义的操作。属性观察器可以添加到类本身定义的存储属性上，也可以添加到从父类继承的属性上。

## 存储属性

---

简单来说，一个存储属性就是存储在特定类或结构体实例里的一个常量或变量。存储属性可以是变量存储属性（用关键字 `var` 定义），也可以是常量存储属性（用关键字 `let` 定义）。

可以在定义存储属性的时候指定默认值，请参考 [默认构造器](#) 一节。也可以在构造过程中设置或修改存储属性的值，甚至修改常量存储属性的值，请参考 [构造过程中常量属性的修改](#) 一节。

下面的例子定义了一个名为 `FixedLengthRange` 的结构体，该结构体用于描述整数的区间，且这个范围值在被创建后不能被修改。

```
struct FixedLengthRange {  
    var firstValue: Int  
    let length: Int  
}  
var rangeOfThreeItems = FixedLengthRange(firstValue: 0, length: 3)  
// 该区间表示整数 0, 1, 2  
rangeOfThreeItems.firstValue = 6  
// 该区间现在表示整数 6, 7, 8
```

`FixedLengthRange` 的实例包含一个名为 `firstValue` 的变量存储属性和一个名为 `length` 的常量存储属性。在上面的例子中，`length` 在创建实例的时候被初始化，且之后无法修改它的值，因为它是一个常量存储属性。

## 常量结构体实例的存储属性

---

如果创建了一个结构体实例并将其赋值给一个常量，则无法修改该实例的任何属性，即使被声明为可变属性也不行：

```
let rangeOfFourItems = FixedLengthRange(firstValue: 0, length: 4)
// 该区间表示整数 0, 1, 2, 3
rangeOfFourItems.firstValue = 6
// 尽管 firstValue 是个可变属性，但这里还是会报错
```

因为 `rangeOfFourItems` 被声明成了常量（用 `let` 关键字），所以即使 `firstValue` 是一个可变属性，也无法再修改它了。

这种行为是由于结构体属于值类型。当值类型的实例被声明为常量的时候，它的所有属性也就成了常量。

属于引用类型的类则不一样。把一个引用类型的实例赋给一个常量后，依然可以修改该实例的可变属性。

## 延时加载存储属性

---

延时加载存储属性是指当第一次被调用的时候才会计算其初始值的属性。在属性声明前使用 `lazy` 来标示一个延时加载存储属性。

### 注意

必须将延时加载属性声明成变量（使用 `var` 关键字），因为属性的初始值可能在实例构造完成之后才会得到。而常量属性在构造过程完成之前必须要有初始值，因此无法声明成延时加载。

当属性的值依赖于一些外部因素且这些外部因素只有在构造过程结束之后才会知道的时候，延时加载属性就会很有用。或者当获得属性的值因为需要复杂或者大量的计算，而需要采用需要的时候再计算的方式，延时加载属性也会很有用。

下面的例子使用了延时加载存储属性来避免复杂类中不必要的初始化工作。例子中定义了 `DataImporter` 和 `DataManager` 两个类，下面是部分代码：

```
class DataImporter {
    /*
    DataImporter 是一个负责将外部文件中的数据导入的类。
    这个类的初始化会消耗不少时间。
    */
    var fileName = "data.txt"
    // 这里会提供数据导入功能
}

class DataManager {
    lazy var importer = DataImporter()
    var data = [String]()
    // 这里会提供数据管理功能
}

let manager = DataManager()
manager.data.append("Some data")
manager.data.append("Some more data")
// DataImporter 实例的 importer 属性还没有被创建
```

`DataManager` 类包含一个名为 `data` 的存储属性，初始值是一个空的字符串数组。这里没有给出全部代码，只需知道 `DataManager` 类的目的是管理和提供对这个字符串数组的访问即可。

`DataManager` 的一个功能是从文件中导入数据。这个功能由 `DataImporter` 类提供，`DataImporter` 完成初始化需要消耗不少时间：因为它的实例在初始化时可能需要打开文件并读取文件中的内容到内存中。

`DataManager` 管理数据时也可能不从文件中导入数据。所以当 `DataManager` 的实例被创建时，没必要创建一个 `DataImporter` 的实例，更明智的做法是第一次用到 `DataImporter` 的时候才去创建它。

由于使用了 `lazy`，`DataImporter` 的实例 `importer` 属性只有在第一次被访问的时候才被创建。比如访问它的属性 `fileName` 时：

```
print(manager.importer.fileName)
// DataImporter 实例的 importer 属性现在被创建了
// 输出“data.txt”
```

注意

如果一个被标记为 `lazy` 的属性在没有初始化时就同时被多个线程访问，则无法保证该属性只会被初始化一次。

## 存储属性和实例变量

如果您有过 Objective-C 经验，应该知道 Objective-C 为类实例存储值和引用提供两种方法。除了属性之外，还可以使用实例变量作为一个备份存储将变量值赋值给属性。

Swift 编程语言中把这些理论统一用属性来实现。Swift 中的属性没有对应的实例变量，属性的备份存储也无法直接访问。这就避免了不同场景下访问方式的困扰，同时也将属性的定义简化成一个语句。属性的全部信息——包括命名、类型和内存管理特征——作为类型定义的一部分，都定义在一个地方。

## 计算属性

除存储属性外，类、结构体和枚举可以定义计算属性。计算属性不直接存储值，而是提供一个 getter 和一个可选的 setter，来间接获取和设置其他属性或变量的值。

```

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)
        }
    }
}
var square = Rect(origin: Point(x: 0.0, y: 0.0),
    size: Size(width: 10.0, height: 10.0))
let initialSquareCenter = square.center
square.center = Point(x: 15.0, y: 15.0)
print("square.origin is now at (\(square.origin.x), \(square.origin.y))")
// 打印“square.origin is now at (10.0, 10.0)”

```

这个例子定义了 3 个结构体来描述几何形状：

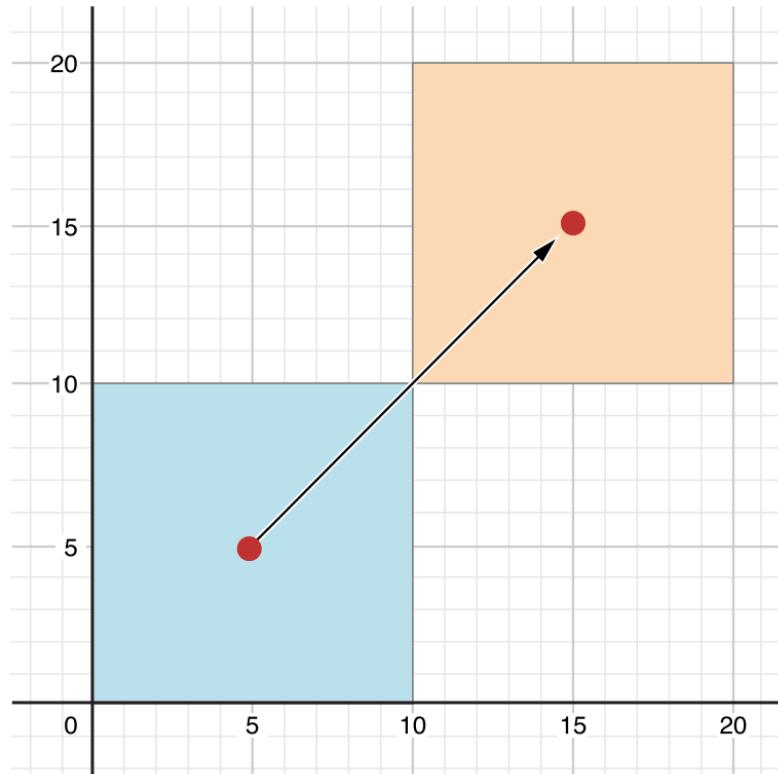
- `Point` 封装了一个 `(x, y)` 的坐标
- `Size` 封装了一个 `width` 和一个 `height`
- `Rect` 表示一个有原点和尺寸的矩形

`Rect` 也提供了一个名为 `center` 的计算属性。一个 `Rect` 的中心点可以从 `origin` (原点) 和 `size` (大小) 算出，所以不需要将中心点以 `Point` 类型的值来保存。`Rect` 的计算属性 `center` 提供了自定义的 getter 和 setter 来获取和设置矩形的中心点，就像它有一个存储属性一样。

上述例子中创建了一个名为 `square` 的 `Rect` 实例，初始值原点是 `(0, 0)`，宽度高度都是 `10`。如下图中蓝色正方形所示。

`square` 的 `center` 属性可以通过点运算符 (`square.center`) 来访问，这会调用该属性的 getter 来获取它的值。跟直接返回已经存在的值不同，getter 实际上通过计算然后返回一个新的 `Point` 来表示 `square` 的中心点。如代码所示，它正确返回了中心点 `(5, 5)`。

`center` 属性之后被设置了一个新的值 `(15, 15)`，表示向右上方移动正方形到如下图橙色正方形所示的位置。设置属性 `center` 的值会调用它的 setter 来修改属性 `origin` 的 `x` 和 `y` 的值，从而实现移动正方形到新的位置。



## 简化 Setter 声明

如果计算属性的 setter 没有定义表示新值的参数名，则可以使用默认名称 `newValue`。下面是使用了简化 setter 声明的 `Rect` 结构体代码：

```
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)
        }
    }
}
```

## 简化 Getter 声明

如果整个 getter 是单一表达式，getter 会隐式地返回这个表达式结果。下面是另一个版本的 `Rect` 结构体，用到了简化的 getter 和 setter 声明：

```
struct CompactRect {  
    var origin = Point()  
    var size = Size()  
    var center: Point {  
        get {  
            Point(x: origin.x + (size.width / 2),  
                  y: origin.y + (size.height / 2))  
        }  
        set {  
            origin.x = newValue.x - (size.width / 2)  
            origin.y = newValue.y - (size.height / 2)  
        }  
    }  
}
```

在 `getter` 中忽略 `return` 与在函数中忽略 `return` 的规则相同，请参考 [隐式返回的函数](#)。

## 只读计算属性

只有 `getter` 没有 `setter` 的计算属性叫 **只读计算属性**。只读计算属性总是返回一个值，可以通过点运算符访问，但不能设置新的值。

注意

必须使用 `var` 关键字定义计算属性，包括只读计算属性，因为它们的值不是固定的。`let` 关键字只用来声明常量属性，表示初始化后再也无法修改的值。

只读计算属性的声明可以去掉 `get` 关键字和花括号：

```
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)  
print("the volume of fourByFiveByTwo is \(fourByFiveByTwo.volume)")  
// 打印“the volume of fourByFiveByTwo is 40.0”
```

这个例子定义了一个名为 `Cuboid` 的结构体，表示三维空间的立方体，包含 `width`、`height` 和 `depth` 属性。结构体还有一个名为 `volume` 的只读计算属性用来返回立方体的体积。为 `volume` 提供 `setter` 毫无意义，因为无法确定如何修改 `width`、`height` 和 `depth` 三者的值来匹配新的 `volume`。然而，`Cuboid` 提供一个只读计算属性来让外部用户直接获取体积是很有用的。

## 属性观察器

属性观察器监控和响应属性值的变化，每次属性被设置值的时候都会调用属性观察器，即使新值和当前值相同的时候也不例外。

你可以为除了延时加载存储属性之外的其他存储属性添加属性观察器，你也可以在子类中通过重写属性的方式为继承的属性（包括存储属性和计算属性）添加属性观察器。你不必为非重写的计算属性添加属性观察器，因为你可以直接通过它的 setter 监控和响应值的变化。属性重写请参考 [重写](#)。

可以为属性添加其中一个或两个观察器：

- `willSet` 在新的值被设置之前调用
- `didSet` 在新的值被设置之后调用

`willSet` 观察器会将新的属性值作为常量参数传入，在 `willSet` 的实现代码中可以为这个参数指定一个名称，如果不指定则参数仍然可用，这时使用默认名称 `newValue` 表示。

同样，`didSet` 观察器会将旧的属性值作为参数传入，可以为该参数指定一个名称或者使用默认参数名 `oldValue`。如果在 `didSet` 方法中再次对该属性赋值，那么新值会覆盖旧的值。

### 注意

在父类初始化方法调用之后，在子类构造器中给父类的属性赋值时，会调用父类属性的 `willSet` 和 `didSet` 观察器。而在父类初始化方法调用之前，给子类的属性赋值时不会调用子类属性的观察器。

有关构造器代理的更多信息，请参考 [值类型的构造器代理](#) 和 [类的构造器代理](#)。

下面是一个 `willSet` 和 `didSet` 实际运用的例子，其中定义了一个名为 `StepCounter` 的类，用来统计一个人步行时的总步数。这个类可以跟计步器或其他日常锻炼的统计装置的输入数据配合使用。

```

class StepCounter {
    var totalSteps: Int = 0 {
        willSet(newTotalSteps) {
            print("将 totalSteps 的值设置为 \(newTotalSteps)")
        }
        didSet {
            if totalSteps > oldValue {
                print("增加了 \(totalSteps - oldValue) 步")
            }
        }
    }
}

let stepCounter = StepCounter()
stepCounter.totalSteps = 200
// 将 totalSteps 的值设置为 200
// 增加了 200 步
stepCounter.totalSteps = 360
// 将 totalSteps 的值设置为 360
// 增加了 160 步
stepCounter.totalSteps = 896
// 将 totalSteps 的值设置为 896
// 增加了 536 步

```

`StepCounter` 类定义了一个叫 `totalSteps` 的 `Int` 类型的属性。它是一个存储属性，包含 `willSet` 和 `didSet` 观察器。

当 `totalSteps` 被设置新值的时候，它的 `willSet` 和 `didSet` 观察器都会被调用，即使新值和当前值完全相同时也会被调用。

例子中的 `willSet` 观察器将表示新值的参数自定义为 `newTotalSteps`，这个观察器只是简单的将新的值输出。

`didSet` 观察器在 `totalSteps` 的值改变后被调用，它把新值和旧值进行对比，如果总步数增加了，就输出一个消息表示增加了多少步。`didSet` 没有为旧值提供自定义名称，所以默认值 `oldValue` 表示旧值的参数名。

### 注意

如果将带有观察器的属性通过 `in-out` 方式传入函数，`willSet` 和 `didSet` 也会调用。这是因为 `in-out` 参数采用了拷入拷出内存模式：即在函数内部使用的是参数的 `copy`，函数结束后，又对参数重新赋值。关于 `in-out` 参数详细的介绍，请参考 [输入输出参数](#)

## 全局变量和局部变量

计算属性和观察属性所描述的功能也可以用于全局变量和局部变量。全局变量是在函数、方法、闭包或任何类型之外定义的变量。局部变量是在函数、方法或闭包内部定义的变量。

前面章节提到的全局或局部变量都属于 `存储型变量`，跟存储属性类似，它为特定类型的值提供存储空间，并允许读取和写入。

另外，在全局或局部范围都可以定义计算型变量和为存储型变量定义观察器。计算型变量跟计算属性一样，返回一个计算结果而不是存储值，声明格式也完全一样。

### 注意

全局的常量或变量都是延迟计算的，跟 延时加载存储属性 相似，不同的地方在于，全局的常量或变量不需要标记 `lazy` 修饰符。

局部范围的常量和变量从不延迟计算。

## 类型属性

实例属性属于一个特定类型的实例，每创建一个实例，实例都拥有属于自己的一套属性值，实例之间的属性相互独立。

你也可以为类型本身定义属性，无论创建了多少个该类型的实例，这些属性都只有唯一一份。这种属性就是**类型属性**。

类型属性用于定义某个类型所有实例共享的数据，比如所有实例都能用的一个常量（就像 C 语言中的静态常量），或者所有实例都能访问的一个变量（就像 C 语言中的静态变量）。

存储型类型属性可以是变量或常量，计算型类型属性跟实例的计算型属性一样只能定义成变量属性。

### 注意

跟实例的存储型属性不同，必须给存储型类型属性指定默认值，因为类型本身没有构造器，也就无法在初始化过程中使用构造器给类型属性赋值。

存储型类型属性是延迟初始化的，它们只有在第一次被访问的时候才会被初始化。即使它们被多个线程同时访问，系统也保证只会对其进行一次初始化，并且不需要对其使用 `lazy` 修饰符。

## 类型属性语法

在 C 或 Objective-C 中，与某个类型关联的静态常量和静态变量，是作为 *global* (全局) 静态变量定义的。但是在 Swift 中，类型属性是作为类型定义的一部分写在类型最外层的花括号内，因此它的作用范围也就在类型支持的范围内。

使用关键字 `static` 来定义类型属性。在为类定义计算型类型属性时，可以改用关键字 `class` 来支持子类对父类的实现进行重写。下面的例子演示了存储型和计算型类型属性的语法：

```
struct SomeStructure {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 1
    }
}
enum SomeEnumeration {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 6
    }
}
class SomeClass {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 27
    }
    class var overrideableComputedTypeProperty: Int {
        return 107
    }
}
```

### 注意

例子中的计算型类型属性是只读的，但也可以定义可读可写的计算型类型属性，跟计算型实例属性的语法相同。

## 获取和设置类型属性的值

跟实例属性一样，类型属性也是通过点运算符来访问。但是，类型属性是通过类型本身来访问，而不是通过实例。比如：

```
print(SomeStructure.storedTypeProperty)
// 打印“Some value.”
SomeStructure.storedTypeProperty = "Another value."
print(SomeStructure.storedTypeProperty)
// 打印“Another value.”
print(SomeEnumeration.computedTypeProperty)
// 打印“6”
print(SomeClass.computedTypeProperty)
// 打印“27”
```

下面的例子定义了一个结构体，使用两个存储型类型属性来表示两个声道的音量，每个声道具有 0 到 10 之间的整数音量。

下图展示了如何把两个声道结合起来模拟立体声的音量。当声道的音量是 0，没有一个灯会亮；当声道的音量是 10，所有灯点亮。本图中，左声道的音量是 9，右声道的音量是 7：

上面所描述的声音模型使用 `AudioChannel` 结构

体的实例来表示：

```
struct AudioChannel {  
    static let thresholdLevel = 10  
    static var maxInputLevelForAllChannels = 0  
    var currentLevel: Int = 0 {  
        didSet {  
            if currentLevel > AudioChannel.thresholdLevel  
            {  
                // 将当前音量限制在阈值之内  
                currentLevel =  
                    AudioChannel.thresholdLevel  
            }  
            if currentLevel >  
                AudioChannel.maxInputLevelForAllChannels {  
                    // 存储当前音量作为新的最大输入音量  
                    AudioChannel.maxInputLevelForAllChannels  
                    = currentLevel  
                }  
            }  
        }  
    }  
}
```



`AudioChannel` 结构定义了 2 个存储型类型属性来实现上述功能。第一个是 `thresholdLevel`，表示音量的最大上限阈值，它是一个值为 `10` 的常量，对所有实例都可见，如果音量高于 `10`，则取最大上限值 `10`（见后面描述）。

第二个类型属性是变量存储型属性 `maxInputLevelForAllChannels`，它用来表示所有 `AudioChannel` 实例的最大输入音量，初始值是 `0`。

`AudioChannel` 也定义了一个名为 `currentLevel` 的存储型实例属性，表示当前声道现在的音量，取值为 `0` 到 `10`。

属性 `currentLevel` 包含 `didSet` 属性观察器来检查每次设置后的属性值，它做如下两个检查：

- 如果 `currentLevel` 的新值大于允许的阈值 `thresholdLevel`，属性观察器将 `currentLevel` 的值限定为阈值 `thresholdLevel`。
- 如果修正后的 `currentLevel` 值大于静态类型属性 `maxInputLevelForAllChannels` 的值，属性观察器就将新值保存在 `maxInputLevelForAllChannels` 中。

### 注意

在第一个检查过程中，`didSet` 属性观察器将 `currentLevel` 设置成了不同的值，但这不会造成属性观察器被再次调用。

可以使用结构体 `AudioChannel` 创建两个声道 `leftChannel` 和 `rightChannel`，用以表示立体声系统的音量：

```
var leftChannel = AudioChannel()  
var rightChannel = AudioChannel()
```

如果将左声道的 `currentLevel` 设置成 `7`，类型属性 `maxInputLevelForAllChannels` 也会更新成 `7`：

```
leftChannel.currentLevel = 7  
print(leftChannel.currentLevel)  
// 输出“7”  
print(AudioChannel.maxInputLevelForAllChannels)  
// 输出“7”
```

如果试图将右声道的 `currentLevel` 设置成 `11`，它会被修正到最大值 `10`，同时 `maxInputLevelForAllChannels` 的值也会更新到 `10`：

```
rightChannel.currentLevel = 11  
print(rightChannel.currentLevel)  
// 输出“10”  
print(AudioChannel.maxInputLevelForAllChannels)  
// 输出“10”
```

# 方法 · GitBook

---



[runoob.com/manual/gitbook/swift5/source/\\_book/chapter2/11\\_Methods.html](https://runoob.com/manual/gitbook/swift5/source/_book/chapter2/11_Methods.html)

## 方法

---

方法是与某些特定类型相关联的函数。类、结构体、枚举都可以定义实例方法；实例方法为给定类型的实例封装了具体的任务与功能。类、结构体、枚举也可以定义类型方法；类型方法与类型本身相关联。类型方法与 Objective-C 中的类方法（class methods）相似。

结构体和枚举能够定义方法是 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。但在 Swift 中，你不仅能选择是否要定义一个类/结构体/枚举，还能灵活地在你创建的类型（类/结构体/枚举）上定义方法。

## 实例方法（Instance Methods）

---

实例方法是属于某个特定类、结构体或者枚举类型实例的方法。实例方法提供访问和修改实例属性的方法或提供与实例目的相关的功能，并以此来支撑实例的功能。实例方法的语法与函数完全一致，详情参见 [函数](#)。

实例方法要写在它所属的类型的前后大括号之间。实例方法能够隐式访问它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某个特定实例调用。实例方法不能脱离于现存的实例而被调用。

下面的例子，定义一个很简单的 `Counter` 类，`Counter` 能被用来对一个动作发生的次数进行计数：

```
class Counter {  
    var count = 0  
    func increment() {  
        count += 1  
    }  
    func increment(by amount: Int) {  
        count += amount  
    }  
    func reset() {  
        count = 0  
    }  
}
```

`Counter` 类定义了三个实例方法：

- `increment` 让计数器按一递增；
- `increment(by: Int)` 让计数器按一个指定的整数值递增；
- `reset` 将计数器重置为0。

`Counter` 这个类还声明了一个可变属性 `count`，用它来保持对当前计数器值的追踪。

和调用属性一样，用点语法（dot syntax）调用实例方法：

```
let counter = Counter()  
// 初始计数值是0  
counter.increment()  
// 计数值现在是1  
counter.increment(by: 5)  
// 计数值现在是6  
counter.reset()  
// 计数值现在是0
```

函数参数可以同时有一个局部名称（在函数体内部使用）和一个外部名称（在调用函数时使用），详情参见 [指定外部参数名](#)。方法参数也一样，因为方法就是函数，只是这个函数与某个类型相关联了。

## self 属性

---

类型的每一个实例都有一个隐含属性叫做 `self`，`self` 完全等同于该实例本身。你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。

上面例子中的 `increment` 方法还可以这样写：

```
func increment() {  
    self.count += 1  
}
```

实际上，你不必在你的代码里面经常写 `self`。不论何时，只要在一个方法中使用一个已知的属性或者方法名称，如果你没有明确地写 `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`）。

使用这条规则的主要场景是实例方法的某个参数名称与实例的某个属性名称相同的时候。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严格的方式。这时你可以使用 `self` 属性来区分参数名称和属性名称。

下面的例子中，`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 isToTheRightOf(x: Double) -> Bool {  
        return self.x > x  
    }  
}  
let somePoint = Point(x: 4.0, y: 5.0)  
if somePoint.isToTheRightOf(x: 1.0) {  
    print("This point is to the right of the line where x == 1.0")  
}  
// 打印"This point is to the right of the line where x == 1.0"
```

如果不使用 `self` 前缀，Swift 会认为 `x` 的两个用法都引用了名为 `x` 的方法参数。

## 在实例方法中修改值类型

---

结构体和枚举是值类型。默认情况下，值类型的属性不能在它的实例方法中被修改。

但是，如果你确实需要在某个特定的方法中修改结构体或者枚举的属性，你可以为这个方法选择 可变 (mutating) 行为，然后就可以从其方法内部改变它的属性；并且这个方法做的任何改变都会在方法执行结束时写回到原始结构中。方法还可以给它隐含的 `self` 属性赋予一个全新的实例，这个新实例在方法结束时会替换现存实例。

要使用 可变 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 moveBy(x deltaX: Double, y deltaY: Double) {  
        x += deltaX  
        y += deltaY  
    }  
}  
var somePoint = Point(x: 1.0, y: 1.0)  
somePoint.moveBy(x: 2.0, y: 3.0)  
print("The point is now at (\(somePoint.x), \(somePoint.y))")  
// 打印"The point is now at (3.0, 4.0)"
```

上面的 `Point` 结构体定义了一个可变方法 `moveBy (x:y:)` 来移动 `Point` 实例到给定的位置。该方法被调用时修改了这个点，而不是返回一个新的点。方法定义时加上了 `mutating` 关键字，从而允许修改属性。

注意，不能在结构体类型的常量（a constant of structure type）上调用可变方法，因为其属性不能被改变，即使属性是变量属性，详情参见 [常量结构体的存储属性](#)：

```
let fixedPoint = Point(x: 3.0, y: 3.0)  
fixedPoint.moveBy(x: 2.0, y: 3.0)  
// 这里将会报告一个错误
```

## 在可变方法中给 `self` 赋值

---

可变方法能够赋给隐含属性 `self` 一个全新的实例。上面 `Point` 的例子可以用下面的方式改写：

```
struct Point {  
    var x = 0.0, y = 0.0  
    mutating func moveBy(x deltaX: Double, y deltaY: Double) {  
        self = Point(x: x + deltaX, y: y + deltaY)  
    }  
}
```

新版的可变方法 `moveBy(x:y:)` 创建了一个新的结构体实例，它的 `x` 和 `y` 的值都被设定为目标值。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。

枚举的可变方法可以把 `self` 设置为同一枚举类型中不同的成员：

```

enum TriStateSwitch {
    case off, low, high
    mutating func next() {
        switch self {
        case .off:
            self = .low
        case .low:
            self = .high
        case .high:
            self = .off
        }
    }
}

var ovenLight = TriStateSwitch.low
ovenLight.next()
// ovenLight 现在等于 .high
ovenLight.next()
// ovenLight 现在等于 .off

```

上面的例子中定义了一个三态切换的枚举。每次调用 `next()` 方法时，开关在不同的电源状态（`off`, `low`, `high`）之间循环切换。

## 类型方法

---

实例方法是被某个类型的实例调用的方法。你也可以定义在类型本身上调用的方法，这种方法就叫做 **类型方法**。在方法的 `func` 关键字之前加上关键字 `static`，来指定类型方法。类还可以用关键字 `class` 来指定，从而允许子类重写父类该方法的实现。

注意

在 Objective-C 中，你只能为 Objective-C 的类类型（classes）定义类型方法（type-level methods）。在 Swift 中，你可以为所有的类、结构体和枚举定义类型方法。每一个类型方法都被它所支持的类型显式包含。

类型方法和实例方法一样用点语法调用。但是，你是在类型上调用这个方法，而不是在实例上调用。下面是如何在 `SomeClass` 类上调用类型方法的例子：

```

class SomeClass {
    class func someTypeMethod() {
        // 在这里实现类型方法
    }
}
SomeClass.someTypeMethod()

```

在类型方法的方法体（body）中，`self` 属性指向这个类型本身，而不是类型的某个实例。这意味着你可以用 `self` 来消除类型属性和类型方法参数之间的歧义（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）。

一般来说，在类型方法的方法体中，任何未限定的方法和属性名称，可以被本类中其他的类型方法和类型属性引用。一个类型方法可以直接通过类型方法的名称调用本类中的其它类型方法，而无需在方法名称前面加上类型名称。类似地，在结构体和枚举中，也

能够直接通过类型属性的名称访问本类中的类型属性，而不需要前面加上类型名称。

下面的例子定义了一个名为 `LevelTracker` 结构体。它监测玩家的游戏发展情况（游戏的不同层次或阶段）。这是一个单人游戏，但也可以存储多个玩家在同一设备上的游戏信息。

游戏初始时，所有的游戏等级（除了等级 1）都被锁定。每次有玩家完成一个等级，这个等级就对这个设备上的所有玩家解锁。`LevelTracker` 结构体用类型属性和方法监测游戏的哪个等级已经被解锁。它还监测每个玩家的当前等级。

```
struct LevelTracker {  
    static var highestUnlockedLevel = 1  
    var currentLevel = 1  
  
    static func unlock(_ level: Int) {  
        if level > highestUnlockedLevel { highestUnlockedLevel = level }  
    }  
  
    static func isUnlocked(_ level: Int) -> Bool {  
        return level <= highestUnlockedLevel  
    }  
  
    @discardableResult  
    mutating func advance(to level: Int) -> Bool {  
        if LevelTracker.isUnlocked(level) {  
            currentLevel = level  
            return true  
        } else {  
            return false  
        }  
    }  
}
```

`LevelTracker` 监测玩家已解锁的最高等级。这个值被存储在类型属性 `highestUnlockedLevel` 中。

`LevelTracker` 还定义了两个类型方法与 `highestUnlockedLevel` 配合工作。第一个类型方法是 `unlock(_)`，一旦新等级被解锁，它会更新 `highestUnlockedLevel` 的值。第二个类型方法是 `isUnlocked(_)`，如果某个给定的等级已经被解锁，它将返回 `true`。

（注意，尽管我们没有使用类似 `LevelTracker.highestUnlockedLevel` 的写法，这个类型方法还是能够访问类型属性 `highestUnlockedLevel`）

除了类型属性和类型方法，`LevelTracker` 还监测每个玩家的进度。它用实例属性 `currentLevel` 来监测每个玩家当前的等级。

为了便于管理 `currentLevel` 属性，`LevelTracker` 定义了实例方法 `advance(to:)`。这个方法会在更新 `currentLevel` 之前检查所请求的新等级是否已经解锁。`advance(to:)` 方法返回布尔值以指示是否能够设置 `currentLevel`。因为允许在调用 `advance(to:)` 时候忽略返回值，不会产生编译警告，所以函数被标注为 `@discardableResult` 属性，更多关于属性信息，请参考 [特性章节](#)。

下面，`Player` 类使用 `LevelTracker` 来监测和更新每个玩家的发展进度：

```
class Player {  
    var tracker = LevelTracker()  
    let playerName: String  
    func complete(level: Int) {  
        LevelTracker.unlock(level + 1)  
        tracker.advance(to: level + 1)  
    }  
    init(name: String) {  
        playerName = name  
    }  
}
```

`Player` 类创建一个新的 `LevelTracker` 实例来监测这个用户的进度。它提供了 `complete(level:)` 方法，一旦玩家完成某个指定等级就调用它。这个方法为所有玩家解锁下一等级，并且将当前玩家的进度更新为下一等级。（我们忽略了 `advance(to:)` 返回的布尔值，因为之前调用 `LevelTracker.unlock(_)` 时就知道了这个等级已经被解锁了）。

你还可以为一个新的玩家创建一个 `Player` 的实例，然后看这个玩家完成等级时发生了什么：

```
var player = Player(name: "Argyrios")  
player.complete(level: 1)  
print("highest unlocked level is now \(LevelTracker.highestUnlockedLevel)")  
// 打印“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，那么试图设置玩家当前等级将会失败：

```
player = Player(name: "Beto")  
if player.tracker.advance(to: 6) {  
    print("player is now on level 6")  
} else {  
    print("level 6 has not yet been unlocked")  
}  
// 打印“level 6 has not yet been unlocked”
```

# 下标 · GitBook

---

 [runoob.com/manual/gitbook/swift5/source/\\_book/chapter2/12\\_Subscripts.html](https://runoob.com/manual/gitbook/swift5/source/_book/chapter2/12_Subscripts.html)

## 下标

---

下标可以定义在类、结构体和枚举中，是访问集合、列表或序列中元素的快捷方式。可以使用下标的索引，设置和获取值，而不需要再调用对应的存取方法。举例来说，用下标访问一个 `Array` 实例中的元素可以写作 `someArray[index]`，访问 `Dictionary` 实例中的元素可以写作 `someDictionary[key]`。

一个类型可以定义多个下标，通过不同索引类型进行重载。下标不限于一维，你可以定义具有多个入参的下标满足自定义类型的需求。

## 下标语法

---

下标允许你通过在实例名称后面的方括号中传入一个或者多个索引值来对实例进行存取。语法类似于实例方法语法和计算型属性语法的混合。与定义实例方法类似，定义下标使用 `subscript` 关键字，指定一个或多个输入参数和返回类型；与实例方法不同的是，下标可以设定为读写或只读。这种行为由 `getter` 和 `setter` 实现，有点类似计算型属性：

```
subscript(index: Int) -> Int {  
    get {  
        // 返回一个适当的 Int 类型的值  
    }  
    set(newValue) {  
        // 执行适当的赋值操作  
    }  
}
```

`newValue` 的类型和下标的返回类型相同。如同计算型属性，可以不指定 `setter` 的参数（`newValue`）。如果不指定参数，`setter` 会提供一个名为 `newValue` 的默认参数。

如同只读计算型属性，可以省略只读下标的 `get` 关键字：

```
subscript(index: Int) -> Int {  
    // 返回一个适当的 Int 类型的值  
}
```

下面代码演示了只读下标的实现，这里定义了一个 `TimesTable` 结构体，用来表示传入整数的乘法表：

```
struct TimesTable {  
    let multiplier: Int  
    subscript(index: Int) -> Int {  
        return multiplier * index  
    }  
}  
let threeTimesTable = TimesTable(multiplier: 3)  
print("six times three is \(threeTimesTable[6])")  
// 打印“six times three is 18”
```

在上例中，创建了一个 `TimesTable` 实例，用来表示整数 `3` 的乘法表。数值 `3` 被传递给结构体的构造函数，作为实例成员 `multiplier` 的值。

你可以通过下标访问 `threeTimesTable` 实例，例如上面演示的 `threeTimesTable[6]`。这条语句查询了 `3` 的乘法表中的第六个元素，返回 `3` 的 `6` 倍即 `18`。

### 注意

`TimesTable` 例子基于一个固定的数学公式，对 `threeTimesTable[someIndex]` 进行赋值操作并不合适，因此下标定义为只读的。

## 下标用法

下标的确切含义取决于使用场景。下标通常作为访问集合，列表或序列中元素的快捷方式。你可以针对自己特定的类或结构体的功能来自由地以最恰当的方式实现下标。

例如，Swift 的 `Dictionary` 类型实现下标用于对其实例中储存的值进行存取操作。为字典设值时，在下标中使用和字典的键类型相同的键，并把一个和字典的值类型相同的值赋给这个下标：

```
var numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]  
numberOfLegs["bird"] = 2
```

上例定义一个名为 `numberOfLegs` 的变量，并用一个包含三对键值的字典字面量初始化它。`numberOfLegs` 字典的类型被推断为 `[String: Int]`。字典创建完成后，该例子通过下标将 `String` 类型的键 `bird` 和 `Int` 类型的值 `2` 添加到字典中。

更多关于 `Dictionary` 下标的信息请参考 [读取和修改字典](#)。

### 注意

Swift 的 `Dictionary` 类型的下标接受并返回可选类型的值。上例中的 `numberOfLegs` 字典通过下标返回的是一个 `Int?` 或者说“可选的 int”。`Dictionary` 类型之所以如此实现下标，是因为不是每个键都有个对应的值，同时这也提供了一种通过键删除对应值的方式，只需将键对应的值赋值为 `nil` 即可。

## 下标选项

下标可以接受任意数量的入参，并且这些入参可以是任意类型。下标的返回值也可以是

任意类型。下标可以使用可变参数，并且可以提供默认参数数值，但是不能使用输入输出参数。

一个类或结构体可以根据自身需要提供多个下标实现，使用下标时将通过入参的数量和类型进行区分，自动匹配合适的下标，这就是 **下标的重载**。

虽然接受单一入参的下标是最常见的，但也可以根据情况定义接受多个入参的下标。例如下例定义了一个 **Matrix** 结构体，用于表示一个 **Double** 类型的二维矩阵。**Matrix** 结构体的下标接受两个整型参数：

```
struct Matrix {  
    let rows: Int, columns: Int  
    var grid: [Double]  
    init(rows: Int, columns: Int) {  
        self.rows = rows  
        self.columns = columns  
        grid = Array(repeating: 0.0, count: rows * columns)  
    }  
    func indexIsValid(row: Int, column: Int) -> Bool {  
        return row >= 0 && row < rows && column >= 0 && column < columns  
    }  
    subscript(row: Int, column: Int) -> Double {  
        get {  
            assert(indexIsValid(row: row, column: column), "Index out of range")  
            return grid[(row * columns) + column]  
        }  
        set {  
            assert(indexIsValid(row: row, column: column), "Index out of range")  
            grid[(row * columns) + column] = newValue  
        }  
    }  
}
```

**Matrix** 提供了一个接受两个入参的构造方法，入参分别是 **rows** 和 **columns**，创建了一个足够容纳 **rows \* columns** 个 **Double** 类型的值的数组。通过传入数组长度和初始值 **0.0** 到数组的构造器，将矩阵中每个位置的值初始化为 **0.0**。关于数组的这种构造方法请参考 [创建一个带有默认值的数组](#)。

你可以通过传入合适的 **row** 和 **column** 的数量来构造一个新的 **Matrix** 实例：

```
var matrix = Matrix(rows: 2, columns: 2)
```

上例中创建了一个 **Matrix** 实例来表示两行两列的矩阵。该 **Matrix** 实例的 **grid** 数组按照从左上到右下的阅读顺序将矩阵扁平化存储：

```
grid = [0.0, 0.0, 0.0, 0.0]
```

		column	
		0	1
row	0	[0.0, 0.0,	
	1	0.0, 0.0 ]	

将 `row` 和 `column` 的值传入下标来为矩阵设值，下标的入参使用逗号分隔：

```
matrix[0, 1] = 1.5  
matrix[1, 0] = 3.2
```

上面两条语句分别调用下标的 `setter` 将矩阵右上角位置（即 `row` 为 0、`column` 为 1 的位置）的值设置为 1.5，将矩阵左下角位置（即 `row` 为 1、`column` 为 0 的位置）的值设置为 3.2：

`Matrix` 下标的 `getter` 和 `setter` 中都含有断言，用来检查下标入参 `row` 和 `column` 的值是否有效。为了方便进行断言，`Matrix` 包含了一个名为 `indexIsValid(row:column:)` 的便利方法，用来检查入参 `row` 和 `column` 的值是否在矩阵范围内：

```
func indexIsValid(row: Int, column: Int) -> Bool {  
    return row >= 0 && row < rows && column >= 0 &&  
    column < columns  
}
```

0.0	1.5
3.2	0.0

断言在下标越界时触发：

```
let someValue = matrix[2, 2]  
// 断言将会触发，因为 [2, 2] 已经超过了 matrix 的范围
```

## 类型下标

正如上节所述，实例下标是在特定类型的一个实例上调用的下标。你也可以定义一种在这个类型本身上调用的下标。这种下标的类型被称作类型下标。你可以通过在 `subscript` 关键字之前写下 `static` 关键字的方式来表示一个类型下标。类可以使用

`class` 关键字来允许子类重写父类中对那个下标的实现。下面的例子展示了如何定义和调用一个类型下标：

```
enum Planet: Int {  
    case mercury = 1, venus, earth, mars, jupiter, saturn, uranus, neptune  
    static subscript(n: Int) -> Planet {  
        return Planet(rawValue: n)!  
    }  
}  
let mars = Planet[4]  
print(mars)
```

# 继承 · GitBook

 [runoob.com/manual/gitbook/swift5/source/\\_book/chapter2/13\\_Inheritance.html](https://runoob.com/manual/gitbook/swift5/source/_book/chapter2/13_Inheritance.html)

## 继承

一个类可以继承另一个类的方法，属性和其它特性。当一个类继承其它类时，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，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 会检查你的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器，这样一来，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器，无论它原本被定义为存储型属性还是计算型属性。

## 定义一个基类

不继承于其它类的类，称之为基类。

注意

Swift 中的类并不是从一个通用的基类继承而来的。如果你不为自己定义的类指定一个超类的话，这个类就会自动成为基类。

下面的例子定义了一个叫 `Vehicle` 的基类。这个基类声明了一个名为 `currentSpeed`，默认值是 `0.0` 的存储型属性（属性类型推断为 `Double`）。`currentSpeed` 属性的值被一个 `String` 类型的只读计算型属性 `description` 使用，用来创建对于车辆的描述。

`Vehicle` 基类还定义了一个名为 `makeNoise` 的方法。这个方法实际上不为 `Vehicle` 实例做任何事，但之后将会被 `Vehicle` 的子类定制：

```
class Vehicle {  
    var currentSpeed = 0.0  
    var description: String {  
        return "traveling at \(currentSpeed) miles per hour"  
    }  
    func makeNoise() {  
        // 什么也不做——因为车辆不一定会有噪音  
    }  
}
```

可以用初始化语法创建一个 `Vehicle` 的新实例，即类名后面跟一个空括号：

```
let someVehicle = Vehicle()
```

现在已经创建了一个 `Vehicle` 的新实例，你可以访问它的 `description` 属性来打印车辆的当前速度：

```
print("Vehicle: \$(someVehicle.description)")  
// 打印“Vehicle: traveling at 0.0 miles per hour”
```

`Vehicle` 类定义了一个具有通用特性的车辆类，但实际上对于它本身来说没什么用处。为了让它变得更加有用，还需要进一步完善它，从而能够描述一个具体类型的车辆。

## 子类生成

---

子类生成指的是在一个已有类的基础上创建一个新的类。子类继承超类的特性，并且可以进一步完善。你还可以为子类添加新的特性。

为了指明某个类的超类，将超类名写在子类名的后面，用冒号分隔：

```
class SomeClass: SomeSuperclass {  
    // 这里是子类的定义  
}
```

下一个例子，定义了一个叫 `Bicycle` 的子类，继承自父类 `Vehicle`：

```
class Bicycle: Vehicle {  
    var hasBasket = false  
}
```

新的 `Bicycle` 类自动继承 `Vehicle` 类的所有特性，比如 `currentSpeed` 和 `description` 属性，还有 `makeNoise()` 方法。

除了所继承的特性，`Bicycle` 类还定义了一个默认值为 `false` 的存储型属性 `hasBasket`（属性推断为 `Bool`）。

默认情况下，你创建的所有新的 `Bicycle` 实例不会有篮子（即 `hasBasket` 属性默认为 `false`）。创建该实例之后，你可以为 `Bicycle` 实例设置 `hasBasket` 属性为 `true`：

```
let bicycle = Bicycle()  
bicycle.hasBasket = true
```

你还可以修改 `Bicycle` 实例所继承的 `currentSpeed` 属性，和查询实例所继承的 `description` 属性：

```
bicycle.currentSpeed = 15.0  
print("Bicycle: \$(bicycle.description)")  
// 打印“Bicycle: traveling at 15.0 miles per hour”
```

子类还可以继续被其它类继承，下面的示例为 `Bicycle` 创建了一个名为 `Tandem`（双人自行车）的子类：

```
class Tandem: Bicycle {  
    var currentNumberOfPassengers = 0  
}
```

`Tandem` 从 `Bicycle` 继承了所有的属性与方法，这又使它同时继承了 `Vehicle` 的所有属性与方法。`Tandem` 也增加了一个新的叫做 `currentNumberOfPassengers` 的存储型属性，默认值为 `0`。

如果你创建了一个 `Tandem` 的实例，你可以使用它所有的新属性和继承的属性，还能查询从 `Vehicle` 继承来的只读属性 `description`：

```
let tandem = Tandem()  
tandem.hasBasket = true  
tandem.currentNumberOfPassengers = 2  
tandem.currentSpeed = 22.0  
print("Tandem: \(tandem.description)")  
// 打印：“Tandem: traveling at 22.0 miles per hour”
```

## 重写

---

子类可以为继承来的实例方法，类方法，实例属性，类属性，或下标提供自己定制的实现。我们把这种行为叫重写。

如果要重写某个特性，你需要在重写定义的前面加上 `override` 关键字。这么做，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。意外的重写行为可能会导致不可预知的错误，任何缺少 `override` 关键字的重写都会在编译时被认定为错误。

`override` 关键字会提醒 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。这个检查可以确保你的重写定义是正确的。

## 访问超类的方法，属性及下标

---

当你在子类中重写超类的方法，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，你可以完善已有实现的行为，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：

- 在方法 `someMethod()` 的重写实现中，可以通过 `super.someMethod()` 来调用超类版本的 `someMethod()` 方法。
- 在属性 `someProperty` 的 getter 或 setter 的重写实现中，可以通过 `super.someProperty` 来访问超类版本的 `someProperty` 属性。
- 在下标的重写实现中，可以通过 `super[someIndex]` 来访问超类版本中的相同下标。

## 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。

下面的例子定义了 `Vehicle` 的一个新的子类，叫 `Train`，它重写了从 `Vehicle` 类继承来的 `makeNoise()` 方法：

```
class Train: Vehicle {  
    override func makeNoise() {  
        print("Choo Choo")  
    }  
}
```

如果你创建一个 `Train` 的新实例，并调用了它的 `makeNoise()` 方法，你就会发现 `Train` 版本的方法被调用：

```
let train = Train()  
train.makeNoise()  
// 打印“Choo Choo”
```

## 重写属性

---

你可以重写继承来的实例属性或类型属性，提供自己定制的 getter 和 setter，或添加属性观察器，使重写的属性可以观察到底层的属性值什么时候发生改变。

### 重写属性的 Getters 和 Setters

---

你可以提供定制的 getter（或 setter）来重写任何一个继承来的属性，无论这个属性是存储型还是计算型属性。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字和类型。你在重写一个属性时，必须将它的名字和类型都写出来。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的。

你可以将一个继承来的只读属性重写为一个读写属性，只需要在重写版本的属性里提供 getter 和 setter 即可。但是，你不可以将一个继承来的读写属性重写为一个只读属性。

注意

如果你在重写属性中提供了 setter，那么你也一定要提供 getter。如果你不想在重写版本中的 getter 里修改继承来的属性值，你可以直接通过 `super.someProperty` 来返回继承来的值，其中 `someProperty` 是你要重写的名字。

以下的例子定义了一个新类，叫 `Car`，它是 `Vehicle` 的子类。这个类引入了一个新的存储型属性叫做 `gear`，默认值为整数 `1`。`Car` 类重写了继承自 `Vehicle` 的 `description` 属性，提供包含当前档位的自定义描述：

```
class Car: Vehicle {  
    var gear = 1  
    override var description: String {  
        return super.description + " in gear \\"(gear)"  
    }  
}
```

重写的 `description` 属性首先要调用 `super.description` 返回 `Vehicle` 类的 `description` 属性。之后，`Car` 类版本的 `description` 在末尾增加了一些额外的文本来提供关于当前档位的信息。

如果你创建了 `Car` 的实例并且设置了它的 `gear` 和 `currentSpeed` 属性，你可以看到它的 `description` 返回了 `Car` 中的自定义描述：

```
let car = Car()
car.currentSpeed = 25.0
car.gear = 3
print("Car: \(car.description)")
// 打印“Car: traveling at 25.0 miles per hour in gear 3”
```

## 重写属性观察器

你可以通过重写属性为一个继承来的属性添加属性观察器。这样一来，无论被继承属性原本是如何实现的，当其属性值发生改变时，你就会被通知到。关于属性观察器的更多内容，请看 [属性观察器](#)。

### 注意

你不可以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器。这些属性的值是不可以被设置的，所以，为它们提供 `willSet` 或 `didSet` 实现也是不恰当。此外还要注意，你不可以同时提供重写的 `setter` 和重写的属性观察器。如果你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`，那么你在 `setter` 中就可以观察到任何值变化了。

下面的例子定义了一个新类叫 `AutomaticCar`，它是 `Car` 的子类。`AutomaticCar` 表示自动档汽车，它可以根据当前的速度自动选择合适的档位：

```
class AutomaticCar: Car {
    override var currentSpeed: Double {
        didSet {
            gear = Int(currentSpeed / 10.0) + 1
        }
    }
}
```

当你设置 `AutomaticCar` 的 `currentSpeed` 属性，属性的 `didSet` 观察器就会自动地设置 `gear` 属性，为新的速度选择一个合适的档位。具体来说就是，属性观察器将新的速度值除以 `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 `35.0` 时，档位为 `4`：

```
let automatic = AutomaticCar()
automatic.currentSpeed = 35.0
print("AutomaticCar: \(automatic.description)")
// 打印“AutomaticCar: traveling at 35.0 miles per hour in gear 4”
```

## 防止重写

你可以通过把方法、属性或下标标记为 `final` 来防止它们被重写，只需要在声明关键字前加上 `final` 修饰符即可（例如：`final var`、`final func`、`final class func` 以及 `final subscript`）。

任何试图对带有 `final` 标记的方法、属性或下标进行重写的代码，都会在编译时会报错。在类扩展中的方法，属性或下标也可以在扩展的定义里标记为 `final`。

可以通过在关键字 `class` 前添加 `final` 修饰符（`final class`）来将整个类标记为 `final`。这样的类是不可被继承的，试图继承这样的类会导致编译报错。

# 构造过程 · GitBook

---



[runoob.com/manual/gitbook/swift5/source/\\_book/chapter2/14\\_Initialization.html](https://runoob.com/manual/gitbook/swift5/source/_book/chapter2/14_Initialization.html)

## 构造过程

---

构造过程是使用类、结构体或枚举类型的实例之前的准备过程。在新实例使用前有个过程是必须的，它包括设置实例中每个存储属性的初始值和执行其他必须的设置或构造过程。

你要通过定义构造器来实现构造过程，它就像用来创建特定类型新实例的特殊方法。与 Objective-C 中的构造器不同，Swift 的构造器没有返回值。它们的主要任务是保证某种类型的新实例在第一次使用前完成正确的初始化。

类的实例也可以通过实现析构器来执行它释放之前自定义的清理工作。想了解更多关于析构器的内容，请参考 [析构过程](#)。

## 存储属性的初始赋值

---

类和结构体在创建实例时，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。

你可以在构造器中为存储型属性设置初始值，也可以在定义属性时分配默认值。以下小节将详细介绍这两种方法。

注意

当你为存储型属性分配默认值或者在构造器中为设置初始值时，它们的值是被直接设置的，不会触发任何属性观察者。

## 构造器

---

构造器在创建某个特定类型的新实例时被调用。它的最简形式类似于一个不带任何形参的实例方法，以关键字 `init` 命名：

```
init() {  
    // 在此处执行构造过程  
}
```

下面例子中定义了一个用来保存华氏温度的结构体 `Fahrenheit`，它拥有一个 `Double` 类型的存储型属性 `temperature`：

```
struct Fahrenheit {  
    var temperature: Double  
    init() {  
        temperature = 32.0  
    }  
}  
var f = Fahrenheit()  
print("The default temperature is \(f.temperature)° Fahrenheit")  
// 打印"The default temperature is 32.0° Fahrenheit"
```

这个结构体定义了一个不带形参的构造器 `init`，并在里面将存储型属性 `temperature` 的值初始化为 `32.0`（华氏温度下水的冰点）。

## 默认属性值

如前所述，你可以在构造器中为存储型属性设置初始值。同样，你也可以在属性声明时为其设置默认值。

### 注意

如果一个属性总是使用相同的初始值，那么为其设置一个默认值比每次都在构造器中赋值要好。两种方法的最终结果是一样的，只不过使用默认值让属性的初始化和声明结合得更紧密。它能让你的构造器更简洁、更清晰，且能通过默认值自动推导出属性的类型；同时，它也能让你充分利用默认构造器、构造器继承等特性，后续章节将讲到。

你可以通过在属性声明时为 `temperature` 提供默认值来使用更简单的方式定义结构体 `Fahrenheit`：

```
struct Fahrenheit {  
    var temperature = 32.0  
}
```

## 自定义构造过程

你可以通过输入形参和可选属性类型来自定义构造过程，也可以在构造过程中分配常量属性。这些都将在后面章节中提到。

## 形参的构造过程

自定义构造过程时，可以在定义中提供构造形参，指定其值的类型和名字。构造形参的功能和语法跟函数和方法的形参相同。

下面例子中定义了一个用来保存摄氏温度的结构体 `Celsius`。它定义了两个不同的构造器：`init(fromFahrenheit:)` 和 `init(fromKelvin:)`，二者分别通过接受不同温标下的温度值来创建新的实例：

```

struct Celsius {
    var temperatureInCelsius: Double
    init(fromFahrenheit fahrenheit: Double) {
        temperatureInCelsius = (fahrenheit - 32.0) / 1.8
    }
    init(fromKelvin kelvin: Double) {
        temperatureInCelsius = kelvin - 273.15
    }
}

let boilingPointOfWater = Celsius(fromFahrenheit: 212.0)
// boilingPointOfWater.temperatureInCelsius 是 100.0
let freezingPointOfWater = Celsius(fromKelvin: 273.15)
// freezingPointOfWater.temperatureInCelsius 是 0.0

```

第一个构造器拥有一个构造形参，其实参标签为 `fromFahrenheit`，形参命名为 `fahrenheit`；第二个构造器也拥有一个构造形参，其实参标签为 `fromKelvin`，形参命名为 `kelvin`。这两个构造器都将单一的实参转换成摄氏温度值，并保存在属性 `temperatureInCelsius` 中。

## 形参命名和实参标签

---

跟函数和方法形参相同，构造形参可以同时使用在构造器里使用的形参命名和一个外部调用构造器时使用的实参标签。

然而，构造器并不像函数和方法那样在括号前有一个可辨别的方法名。因此在调用构造器时，主要通过构造器中形参命名和类型来确定应该被调用的构造器。正因如此，如果你在定义构造器时没有提供实参标签，Swift 会为构造器的每个形参自动生成一个实参标签。

以下例子中定义了一个结构体 `Color`，它包含了三个常量：`red`、`green` 和 `blue`。这些属性可以存储 `0.0` 到 `1.0` 之间的值，用来表明颜色中红、绿、蓝成分的含量。

`Color` 提供了一个构造器，为红蓝绿提供三个合适 `Double` 类型的形参命名。`Color` 也提供了第二个构造器，它只包含名为 `white` 的 `Double` 类型的形参，它为三个颜色的属性提供相同的值。

```

struct Color {
    let red, green, blue: Double
    init(red: Double, green: Double, blue: Double) {
        self.red = red
        self.green = green
        self.blue = blue
    }
    init(white: Double) {
        red = white
        green = white
        blue = white
    }
}

```

两种构造器都能通过为每一个构造器形参提供命名值来创建一个新的 `Color` 实例：

```
let magenta = Color(red: 1.0, green: 0.0, blue: 1.0)
let halfGray = Color(white: 0.5)
```

注意，如果不通过实参标签传值，这个构造器是没法调用的。如果构造器定义了某个实参标签，就必须使用它，忽略它将导致编译期错误：

```
let veryGreen = Color(0.0, 1.0, 0.0)
// 报编译期错误-需要实参标签
```

## 不带实参标签的构造器形参

---

如果你不希望构造器的某个形参使用实参标签，可以使用下划线（`_`）来代替显式的实参标签来重写默认行为。

下面是之前 [形参的构造过程中](#) `Celsius` 例子的扩展，多了一个用已经的摄氏表示的 `Double` 类型值来创建新的 `Celsius` 实例的额外构造器：

```
struct Celsius {
    var temperatureInCelsius: Double
    init(fromFahrenheit fahrenheit: Double) {
        temperatureInCelsius = (fahrenheit - 32.0) / 1.8
    }
    init(fromKelvin kelvin: Double) {
        temperatureInCelsius = kelvin - 273.15
    }
    init(_ celsius: Double){
        temperatureInCelsius = celsius
    }
}

let bodyTemperature = Celsius(37.0)
// bodyTemperature.temperatureInCelsius 为 37.0
```

构造器调用 `Celsius(37.0)` 意图明确，不需要实参标签。因此适合使用 `init(_ celsius: Double)` 这样的构造器，从而可以通过提供未命名的 `Double` 值来调用构造器。

## 可选属性类型

---

如果你自定义的类型有一个逻辑上允许值为空的存储型属性——无论是因为它无法在初始化时赋值，还是因为它在之后某个时机可以赋值为空——都需要将它声明为 [可选类型](#)。

可选类型的属性将自动初始化为 `nil`，表示这个属性是特意在构造过程设置为空。

下面例子中定义了类 `SurveyQuestion`，它包含一个可选 `String` 属性 `response`：

```
class SurveyQuestion {  
    var text: String  
    var response: String?  
    init(text: String) {  
        self.text = text  
    }  
    func ask() {  
        print(text)  
    }  
}  
  
let cheeseQuestion = SurveyQuestion(text: "Do you like cheese?")  
cheeseQuestion.ask()  
// 打印"Do you like cheese?"  
cheeseQuestion.response = "Yes, I do like cheese."
```

调查问题的答案在询问前是无法确定的，因此我们将属性 `response` 声明为 `String?` 类型，或者说是“可选类型 `String`”。当 `SurveyQuestion` 的实例初始化时，它将自动赋值为 `nil`，表明“暂时还没有字符”。

## 构造过程中常量属性的赋值

你可以在构造过程中的任意时间点给常量属性赋值，只要在构造过程结束时它设置成确定的值。一旦常量属性被赋值，它将永远不可更改。

### 注意

对于类的实例来说，它的常量属性只能在定义它的类的构造过程中修改；不能在子类中修改。

你可以修改上面的 `SurveyQuestion` 示例，用常量属性替代变量属性 `text`，表示问题内容 `text` 在 `SurveyQuestion` 的实例被创建之后不会再被修改。尽管 `text` 属性现在是常量，我们仍然可以在类的构造器中设置它的值：

```
class SurveyQuestion {  
    let text: String  
    var response: String?  
    init(text: String) {  
        self.text = text  
    }  
    func ask() {  
        print(text)  
    }  
}  
let beetsQuestion = SurveyQuestion(text: "How about beets?")  
beetsQuestion.ask()  
// 打印"How about beets?"  
beetsQuestion.response = "I also like beets. (But not with cheese.)"
```

## 默认构造器

如果结构体或类为所有属性提供了默认值，又没有提供任何自定义的构造器，那么 Swift 会为这些结构体或类提供一个默认构造器。这个默认构造器将简单地创建一个所有属性值都设置为它们默认值的实例。

下面例子中定义了一个类 `ShoppingListItem`，它封装了购物清单中的某一物品的名字（`name`）、数量（`quantity`）和购买状态 `purchase state`：

```
class ShoppingListItem {  
    var name: String?  
    var quantity = 1  
    var purchased = false  
}  
var item = ShoppingListItem()
```

由于 `ShoppingListItem` 类中的所有属性都有默认值，且它是没有父类的基类，它将自动获得一个将为所有属性设置默认值的并创建实例的默认构造器（由于 `name` 属性是可选 `String` 类型，它将接收一个默认 `nil` 的默认值，尽管代码中没有写出这个值）。上面例子中使用默认构造器创造了一个 `ShoppingListItem` 类的实例（使用 `ShoppingListItem()` 形式的构造器语法），并将其赋值给变量 `item`。

## 结构体的逐一成员构造器

---

结构体如果没有定义任何自定义构造器，它们将自动获得一个逐一成员构造器 (*memberwise initializer*)。不像默认构造器，即使存储型属性没有默认值，结构体也能获得逐一成员构造器。

逐一成员构造器是用来初始化结构体新实例里成员属性的快捷方法。新实例的属性初始值可以通过名字传入逐一成员构造器中。

下面例子中定义了一个结构体 `Size`，它包含两个属性 `width` 和 `height`。根据这两个属性默认赋值为 `0.0`，它们的类型被推断出来为 `Double`。

结构体 `Size` 自动获得了一个逐一成员构造器 `init(width:height:)`。你可以用它来创建新的 `Size` 实例：

```
struct Size {  
    var width = 0.0, height = 0.0  
}  
let twoByTwo = Size(width: 2.0, height: 2.0)
```

当你调用一个逐一成员构造器 (*memberwise initializer*) 时，可以省略任何一个有默认值的属性。在上面这个例子中，`Size` 结构体的 `height` 和 `width` 属性各有一个默认值。你可以省略两者或两者之一，对于被省略的属性，构造器会使用默认值。举个例子：

```
let zeroByTwo = Size(height: 2.0)
print(zeroByTwo.width, zeroByTwo.height)
// 打印 "0.0 2.0"

let zeroByZero = Size()
print(zeroByZero.width, zeroByZero.height)
// 打印 "0.0 0.0"
```

## 值类型的构造器代理

构造器可以通过调用其它构造器来完成实例的部分构造过程。这一过程称为 **构造器代理**，它能避免多个构造器间的代码重复。

构造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，所以构造器代理的过程相对简单，因为它们只能代理给自己的其它构造器。类则不同，它可以继承自其它类（请参考 [继承](#)）。这意味着类有责任保证其所有继承的存储型属性在构造时也能正确的初始化。这些责任将在后续章节 [类的继承和构造过程](#) 中介绍。

对于值类型，你可以使用 `self.init` 在自定义的构造器中引用相同类型中的其它构造器。并且你只能在构造器内部调用 `self.init`。

请注意，如果你为某个值类型定义了一个自定义的构造器，你将无法访问到默认构造器（如果是结构体，还将无法访问逐一成员构造器）。这种限制避免了在一个更复杂的构造器中做了额外的重要设置，但有人不小心使用自动生成的构造器而导致错误的情况。

### 注意

假如你希望默认构造器、逐一成员构造器以及你自己的自定义构造器都能用来创建实例，可以将自定义的构造器写到扩展（[extension](#)）中，而不是写在值类型的原始定义中。想查看更多内容，请查看 [扩展章节](#)。

下面例子定义一个自定义结构体 `Rect`，用来代表几何矩形。这个例子需要两个辅助的结构体 `Size` 和 `Point`，它们各自为其所有的属性提供了默认初始值 `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 `Rect` 创建实例——使用含有默认值的 `origin` 和 `size` 属性来初始化；提供指定的 `origin` 和 `size` 实例来初始化；提供指定的 `center` 和 `size` 来初始化。在下面 `Rect` 结构体定义中，我们为这三种方式提供了三个自定义的构造器：

```

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` 构造器 `init()`，在功能上跟没有自定义构造器时自动获得的默认构造器是一样的。这个构造器是函数体是空的，使用一对大括号 `{}` 来表示。调用这个构造器将返回一个 `Rect` 实例，它的 `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 的 origin 是 (0.0, 0.0), size 是 (0.0, 0.0)

```

第二个 `Rect` 构造器 `init(origin:size:)`，在功能上跟结构体在没有自定义构造器时获得的逐一成员构造器是一样的。这个构造器只是简单地将 `origin` 和 `size` 的实参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 的 origin 是 (2.0, 2.0), size 是 (5.0, 5.0)

```

第三个 `Rect` 构造器 `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（或者说代理给）`init(origin:size:)` 构造器来将新的 `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 的 origin 是 (2.5, 2.5), size 是 (3.0, 3.0)

```

构造器 `init(center:size:)` 可以直接将 `origin` 和 `size` 的新值赋值到对应的属性中。然而，构造器 `init(center:size:)` 通过使用提供了相关功能的现有构造器将会更加便捷（而且意图更清晰）。

### 注意

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，请参考 [扩展](#)。

## 类的继承和构造过程

类里面的所有存储型属性——包括所有继承自父类的属性——都必须在构造过程中设置初始值。

Swift 为类类型提供了两种构造器来确保实例中所有存储型属性都能获得初始值，它们被称为指定构造器和便利构造器。

## 指定构造器和便利构造器

---

指定构造器是类中最主要的构造器。一个指定构造器将初始化类中提供的所有属性，并调用合适的父类构造器让构造过程沿着父类链继续往上进行。

类倾向于拥有极少的指定构造器，普遍的是一个类只拥有一个指定构造器。指定构造器像一个个“漏斗”放在构造过程发生的地方，让构造过程沿着父类链继续往上进行。

每一个类都必须至少拥有一个指定构造器。在某些情况下，许多类通过继承了父类中的指定构造器而满足了这个条件。具体内容请参考后续章节 [构造器的自动继承](#)。

便利构造器是类中比较次要的、辅助型的构造器。你可以定义便利构造器来调用同一个类中的指定构造器，并为部分形参提供默认值。你也可以定义便利构造器来创建一个特殊用途或特定输入值的实例。

你应当只在必要的时候为类提供便利构造器，比方说某种情况下通过使用便利构造器来快捷调用某个指定构造器，能够节省更多开发时间并让类的构造过程更清晰明了。

## 指定构造器和便利构造器的语法

---

类的指定构造器的写法跟值类型简单构造器一样：

```
init(parameters) {  
    statements  
}
```

便利构造器也采用相同样式的写法，但需要在 `init` 关键字之前放置 `convenience` 关键字，并使用空格将它们俩分开：

```
convenience init(parameters) {  
    statements  
}
```

## 类类型的构造器代理

---

为了简化指定构造器和便利构造器之间的调用关系，Swift 构造器之间的代理调用遵循以下三条规则：

规则 1

指定构造器必须调用其直接父类的指定构造器。

规则 2

便利构造器必须调用同类中定义的其它构造器。

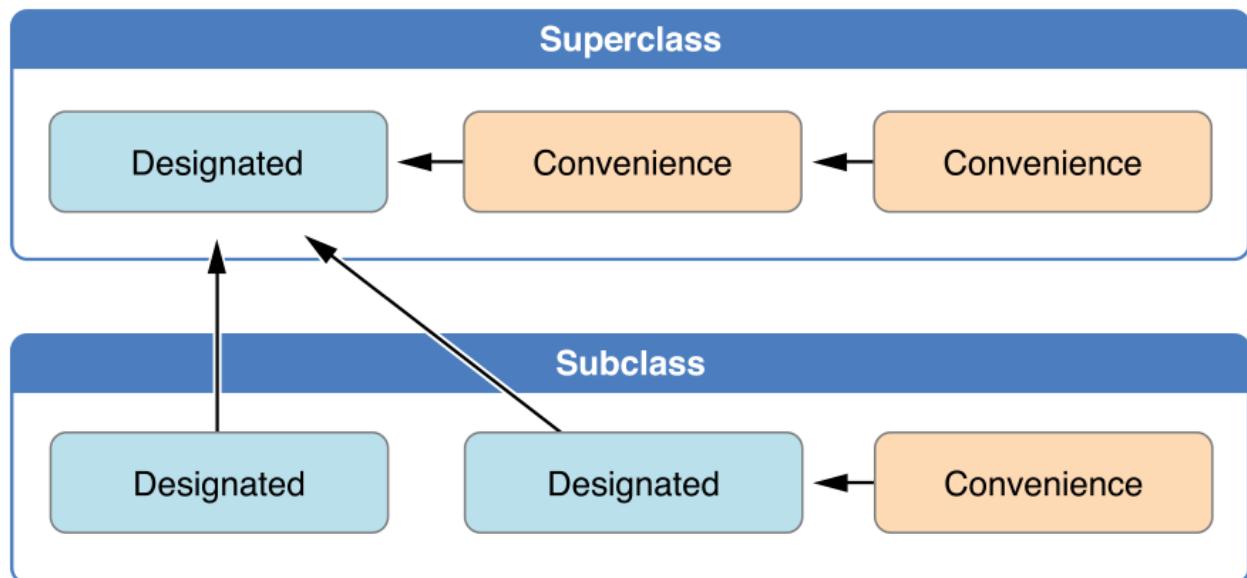
### 规则 3

便利构造器最后必须调用指定构造器。

一个更方便记忆的方法是：

- 指定构造器必须总是向上代理
- 便利构造器必须总是横向代理

这些规则可以通过下面图例来说明：



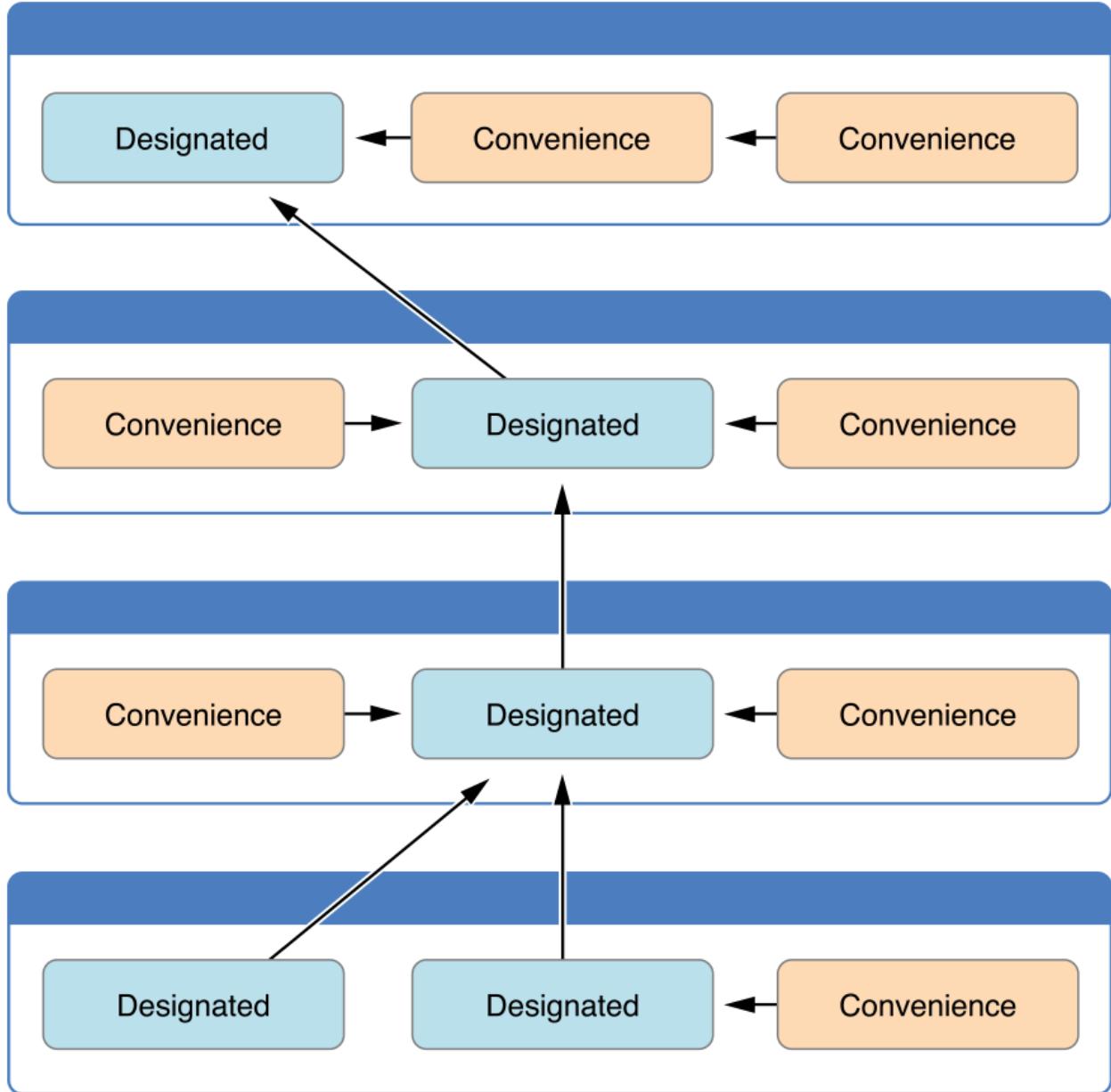
如图所示，父类中包含一个指定构造器和两个便利构造器。其中一个便利构造器调用了另外一个便利构造器，而后者又调用了唯一的指定构造器。这满足了上面提到的规则 2 和 3。这个父类没有自己的父类，所以规则 1 没有用到。

子类中包含两个指定构造器和一个便利构造器。便利构造器必须调用两个指定构造器中的任意一个，因为它只能调用同一个类里的其他构造器。这满足了上面提到的规则 2 和 3。而两个指定构造器必须调用父类中唯一的指定构造器，这满足了规则 1。

注意

这些规则不会影响类的实例如何创建。任何上图中展示的构造器都可以用来创建完全初始化的实例。这些规则只影响类的构造器如何实现。

下面图例中展示了一种涉及四个类的更复杂的类层级结构。它演示了指定构造器是如何在类层级中充当“漏斗”的作用，在类的构造器链上简化了类之间的相互关系。



## 两段式构造过程

Swift 中类的构造过程包含两个阶段。第一个阶段，类中的每个存储型属性赋一个初始值。当每个存储型属性的初始值被赋值后，第二阶段开始，它给每个类一次机会，在新实例准备使用之前进一步自定义它们的存储型属性。

两段式构造过程的使用让构造过程更安全，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构造器意外地赋予不同的值。

### 注意

Swift 的两段式构造过程跟 Objective-C 中的构造过程类似。最主要的区别在于阶段 1，Objective-C 给每一个属性赋值 `0` 或空值（比如说 `0` 或 `nil`）。Swift 的构造流程则更加灵活，它允许你设置定制的初始值，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 4 种有效安全检查，以确保两段式构造过程不出错地完成：

#### 安全检查 1

指定构造器必须保证它所在类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父类中的构造器。

如上所述，一个对象的内存只有在其所有存储型属性确定之后才能完全初始化。为了满足这一规则，指定构造器必须保证它所在类的属性在它往上代理之前先完成初始化。

#### 安全检查 2

指定构造器必须在为继承的属性设置新值之前向上代理调用父类构造器。如果没这么做，指定构造器赋予的新值将被父类中的构造器所覆盖。

#### 安全检查 3

便利构造器必须为任意属性（包括所有同类中定义的）赋新值之前代理调用其它构造器。如果没这么做，便利构造器赋予的新值将被该类的指定构造器所覆盖。

#### 安全检查 4

构造器在第一阶段构造完成之前，不能调用任何实例方法，不能读取任何实例属性的值，不能引用 `self` 作为一个值。

类的实例在第一阶段结束以前并不是完全有效的。只有第一阶段完成后，类的实例才是有效的，才能访问属性和调用方法。

以下是基于上述安全检查的两段式构造过程展示：

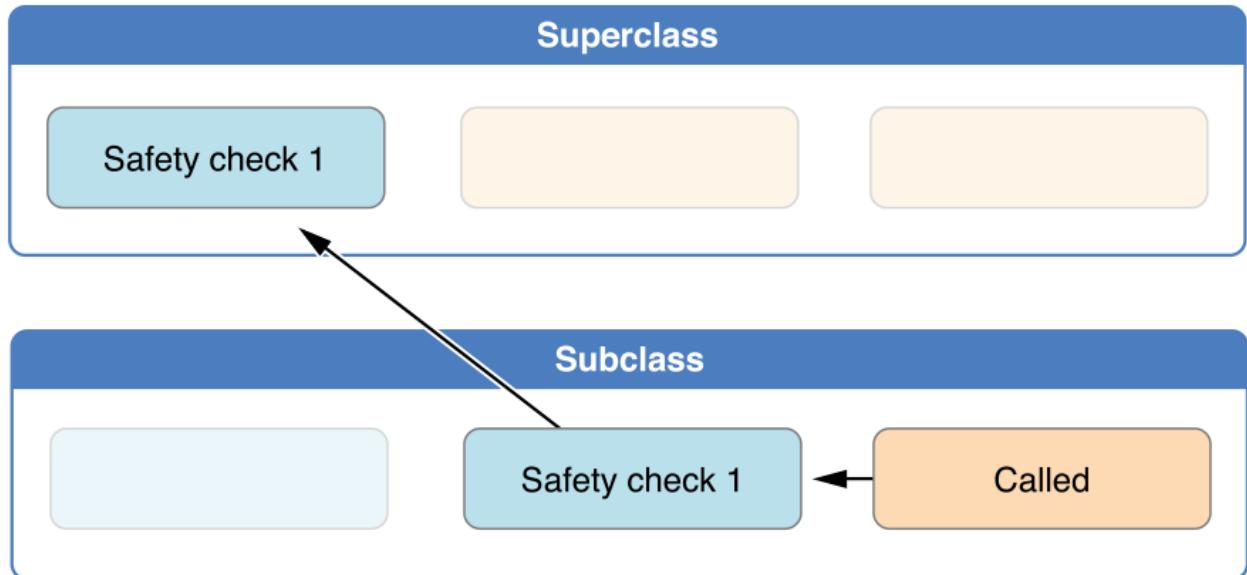
#### 阶段 1

- 类的某个指定构造器或便利构造器被调用。
- 完成类的新实例内存的分配，但此时内存还没有被初始化。
- 指定构造器确保其所在类引入的所有存储型属性都已赋初值。存储型属性所属的内存完成初始化。
- 指定构造器切换到父类的构造器，对其存储属性完成相同的任务。
- 这个过程沿着类的继承链一直往上执行，直到到达继承链的最顶部。
- 当到达了继承链最顶部，而且继承链的最后一个类已确保所有的存储型属性都已经赋值，这个实例的内存被认为已经完全初始化。此时阶段 1 完成。

#### 阶段 2

- 从继承链顶部往下，继承链中每个类的指定构造器都有机会进一步自定义实例。构造器此时可以访问 `self`、修改它的属性并调用实例方法等等。
- 最终，继承链中任意的便利构造器有机会自定义实例和使用 `self`。

下图展示了在假定的子类和父类之间的构造阶段 1：



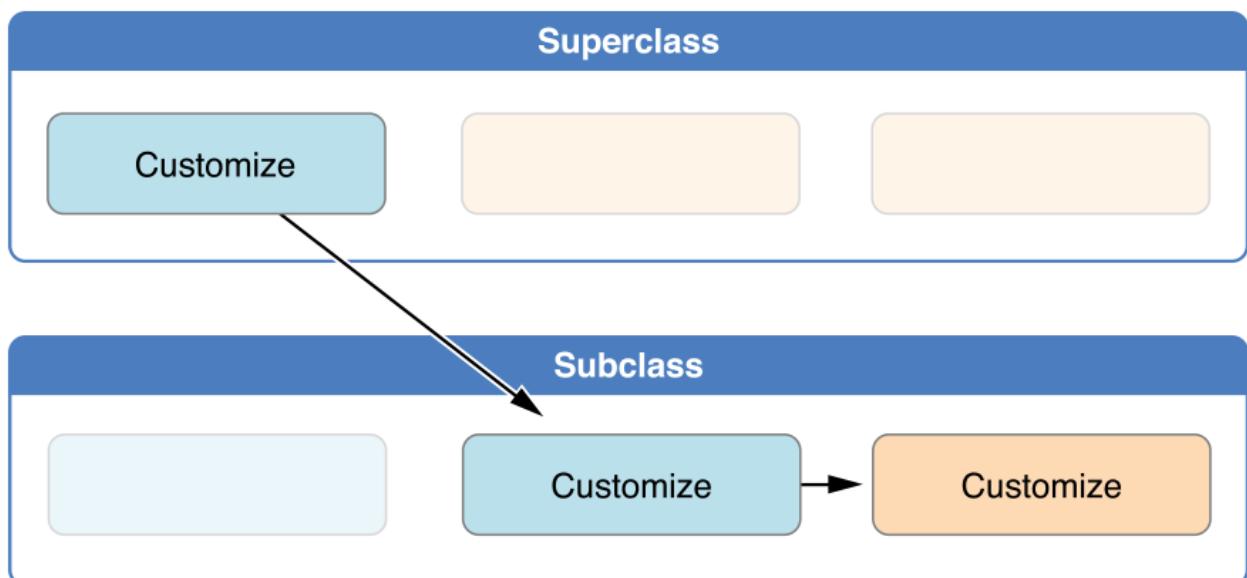
在这个例子中，构造过程从对子类中一个便利构造器的调用开始。这个便利构造器此时还不能修改任何属性，它会代理到该类中的指定构造器。

如安全检查 1 所示，指定构造器将确保所有子类的属性都有值。然后它将调用父类的指定构造器，并沿着继承链一直往上完成父类的构造过程。

父类中的指定构造器确保所有父类的属性都有值。由于没有更多的父类需要初始化，也就无需继续向上代理。

一旦父类中所有属性都有了初始值，实例的内存被认为是完全初始化，阶段 1 完成。

以下展示了相同构造过程的阶段 2：



父类中的指定构造器现在有机会进一步自定义实例（尽管这不是必须的）。

一旦父类中的指定构造器完成调用，子类中的指定构造器可以执行更多的自定义操作（这也不是必须的）。

最终，一旦子类的指定构造器完成调用，最开始被调用的便利构造器可以执行更多的自定义操作。

## 构造器的继承和重写

跟 Objective-C 中的子类不同，Swift 中的子类默认情况下不会继承父类的构造器。Swift 的这种机制可以防止一个父类的简单构造器被一个更精细的子类继承，而在用来创建子类时的新实例时没有完全或错误被初始化。

### 注意

父类的构造器仅会在安全和适当的某些情况下被继承。具体内容请参考后续章节 [构造器的自动继承](#)。

假如你希望自定义的子类中能提供一个或多个跟父类相同的构造器，你可以在子类中提供这些构造器的自定义实现。

当你在编写一个和父类中指定构造器相匹配的子类构造器时，你实际上是在重写父类的这个指定构造器。因此，你必须在定义子类构造器时带上 `override` 修饰符。即使你重写的是系统自动提供的默认构造器，也需要带上 `override` 修饰符，具体内容请参考 [默认构造器](#)。

正如重写属性，方法或者是下标，`override` 修饰符会让编译器去检查父类中是否有相匹配的指定构造器，并验证构造器参数是否被按预想中被指定。

### 注意

当你重写一个父类的指定构造器时，你总是需要写 `override` 修饰符，即使是为了实现子类的便利构造器。

相反，如果你编写了一个和父类便利构造器相匹配的子类构造器，由于子类不能直接调用父类的便利构造器（每个规则都在上文 [类的构造器代理规则](#) 有所描述），因此，严格意义上讲，你的子类并未对一个父类构造器提供重写。最后的结果就是，你在子类中“重写”一个父类便利构造器时，不需要加 `override` 修饰符。

在下面的例子中定义了一个叫 `Vehicle` 的基类。基类中声明了一个存储型属性 `numberOfWheels`，它是默认值为 `Int` 类型的 `0`。`numberOfWheels` 属性用在一个描述车辆特征 `String` 类型为 `description` 的计算型属性中：

```
class Vehicle {  
    var numberOfWheels = 0  
    var description: String {  
        return "\(numberOfWheels) wheel(s)"  
    }  
}
```

`Vehicle` 类只为存储型属性提供默认值，也没有提供自定义构造器。因此，它会自动获得一个默认构造器，具体内容请参考 [默认构造器](#)。默认构造器（如果有的话）总是类中的指定构造器，可以用于创建 `numberOfWheels` 为 `0` 的 `Vehicle` 实例：

```
let vehicle = Vehicle()
print("Vehicle: \(vehicle.description)")
// Vehicle: 0 wheel(s)
```

下面例子中定义了一个 `Vehicle` 的子类 `Bicycle` :

```
class Bicycle: Vehicle {
    override init() {
        super.init()
        numberOfWheels = 2
    }
}
```

子类 `Bicycle` 定义了一个自定义指定构造器 `init()`。这个指定构造器和父类的指定构造器相匹配，所以 `Bicycle` 中这个版本的构造器需要带上 `override` 修饰符。

`Bicycle` 的构造器 `init()` 以调用 `super.init()` 方法开始，这个方法的作用是调用 `Bicycle` 的父类 `Vehicle` 的默认构造器。这样可以确保 `Bicycle` 在修改属性之前，它所继承的属性 `numberOfWheels` 能被 `Vehicle` 类初始化。在调用 `super.init()` 之后，属性 `numberOfWheels` 的原值被新值 `2` 替换。

如果你创建一个 `Bicycle` 实例，你可以调用继承的 `description` 计算型属性去查看属性 `numberOfWheels` 是否有改变：

```
let bicycle = Bicycle()
print("Bicycle: \(bicycle.description)")
// 打印“Bicycle: 2 wheel(s)”
```

如果子类的构造器没有在阶段 2 过程中做自定义操作，并且父类有一个无参数的自定义构造器。你可以在所有父类的存储属性赋值之后省略 `super.init()` 的调用。

这个例子定义了另一个 `Vehicle` 的子类 `Hoverboard`，只设置它的 `color` 属性。这个构造器依赖隐式调用父类的构造器来完成，而不是显示调用 `super.init()`。

```
class Hoverboard: Vehicle {
    var color: String
    init(color: String) {
        self.color = color
        // super.init() 在这里被隐式调用
    }
    override var description: String {
        return "\(super.description) in a beautiful \(color)"
    }
}
```

`Hoverboard` 的实例用 `Vehicle` 构造器里默认的轮子数量。

```
let hoverboard = Hoverboard(color: "silver")
print("Hoverboard: \(hoverboard.description)")
// Hoverboard: 0 wheel(s) in a beautiful silver
```

注意

子类可以在构造过程修改继承来的变量属性，但是不能修改继承来的常量属性。

## 构造器的自动继承

如上所述，子类在默认情况下不会继承父类的构造器。但是如果满足特定条件，父类构造器是可以被自动继承的。事实上，这意味着对于许多常见场景你不必重写父类的构造器，并且可以在安全的情况下以最小的代价继承父类的构造器。

假设你为子类中引入的所有新属性都提供了默认值，以下 2 个规则将适用：

规则 1

如果子类没有定义任何指定构造器，它将自动继承父类所有的指定构造器。

规则 2

如果子类提供了所有父类指定构造器的实现——无论是通过规则 1 继承过来的，还是提供了自定义实现——它将自动继承父类所有的便利构造器。

即使你在子类中添加了更多的便利构造器，这两条规则仍然适用。

注意

子类可以将父类的指定构造器实现为便利构造器来满足规则 2。

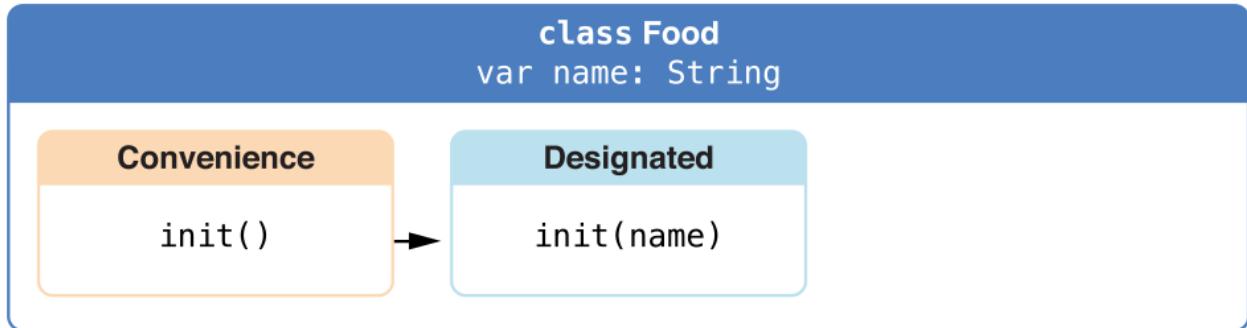
## 指定构造器和便利构造器实践

接下来的例子将在实践中展示指定构造器、便利构造器以及构造器的自动继承。这个例子定义了包含三个类 `Food`、`RecipeIngredient` 以及 `ShoppingListItem` 的层级结构，并将演示它们的构造器是如何相互作用的。

类层次中的基类是 `Food`，它是一个简单的用来封装食物名字的类。`Food` 类引入了一个叫做 `name` 的 `String` 类型的属性，并且提供了两个构造器来创建 `Food` 实例：

```
class Food {  
    var name: String  
    init(name: String) {  
        self.name = name  
    }  
  
    convenience init() {  
        self.init(name: "[Unnamed]")  
    }  
}
```

下图中展示了 `Food` 的构造器链：



类类型没有默认的逐一成员构造器，所以 `Food` 类提供了一个接受单一参数 `name` 的指定构造器。这个构造器可以使用一个特定的名字来创建新的 `Food` 实例：

```
let namedMeat = Food(name: "Bacon")
// namedMeat 的名字是 "Bacon"
```

`Food` 类中的构造器 `init(name: String)` 被定义为一个指定构造器，因为它能确保 `Food` 实例的所有存储型属性都被初始化。`Food` 类没有父类，所以 `init(name: String)` 构造器不需要调用 `super.init()` 来完成构造过程。

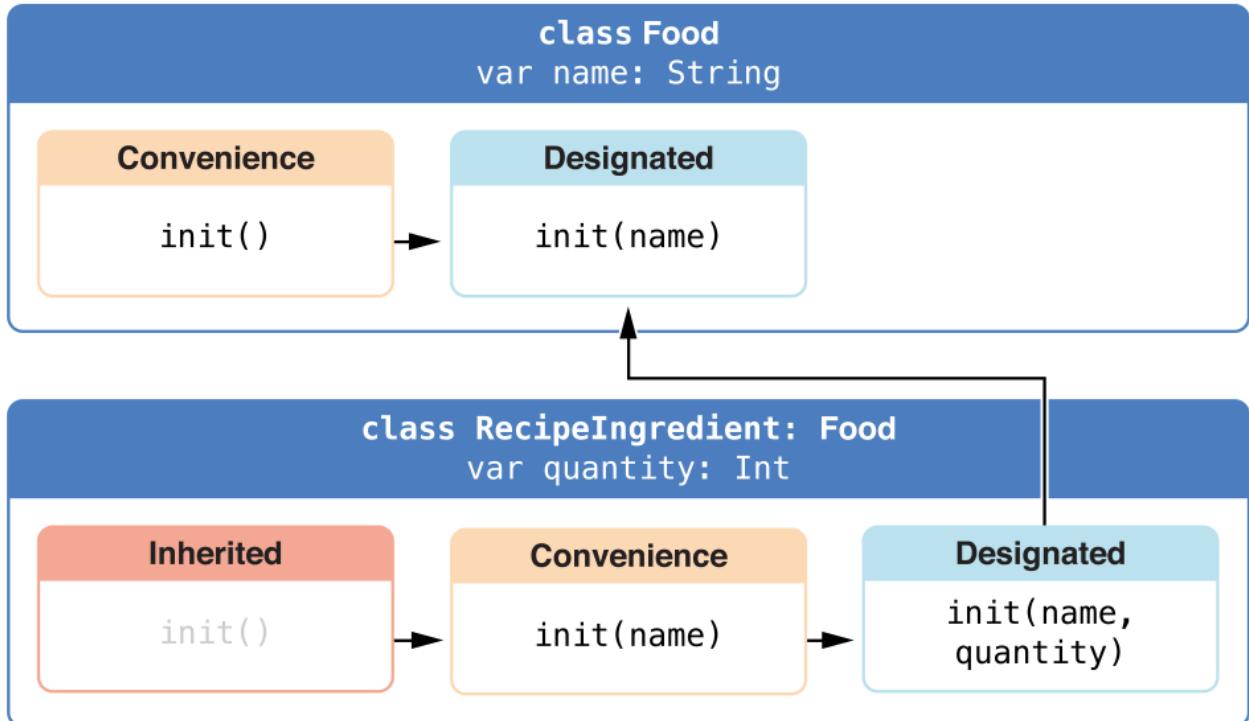
`Food` 类同样提供了一个没有参数的便利构造器 `init()`。这个 `init()` 构造器为新食物提供了一个默认的占位名字，通过横向代理到指定构造器 `init(name: String)` 并给参数 `name` 赋值为 `[Unnamed]` 来实现：

```
let mysteryMeat = Food()
// mysteryMeat 的名字是 [Unnamed]
```

层级中的第二个类是 `Food` 的子类 `Recipelngredient`。`Recipelngredient` 类用来表示食谱中的一项原料。它引入了 `Int` 类型的属性 `quantity`（以及从 `Food` 继承过来的 `name` 属性），并且定义了两个构造器来创建 `Recipelngredient` 实例：

```
class Recipelngredient: Food {
    var quantity: Int
    init(name: String, quantity: Int) {
        self.quantity = quantity
        super.init(name: name)
    }
    override convenience init(name: String) {
        self.init(name: name, quantity: 1)
    }
}
```

下图中展示了 `Recipelngredient` 类的构造器链：



`RecipeIngredient` 类拥有一个指定构造器 `init(name: String, quantity: Int)`，它可以用 来填充 `RecipeIngredient` 实例的所有属性值。这个构造器一开始先将传入的 `quantity` 实参赋值给 `quantity` 属性，这个属性也是唯一在 `RecipeIngredient` 中新引入的属性。随后，构造器向上代理到父类 `Food` 的 `init(name: String)`。这个过程满足 两段式构造 过程中的安全检查 1。

`RecipeIngredient` 也定义了一个便利构造器 `init(name: String)`，它只通过 `name` 来 创建 `RecipeIngredient` 的实例。这个便利构造器假设任意 `RecipeIngredient` 实例的 `quantity` 为 1，所以不需要显式的质量即可创建出实例。这个便利构造器的定义可以 更加方便和快捷地创建实例，并且避免了创建多个 `quantity` 为 1 的 `RecipeIngredient` 实例时的代码重复。这个便利构造器只是简单地横向代理到类中的 指定构造器，并为 `quantity` 参数传递 1。

`RecipeIngredient` 的便利构造器 `init(name: String)` 使用了跟 `Food` 中指定构造器 `init(name: String)` 相同的形参。由于这个便利构造器重写了父类的指定构造器 `init(name: String)`，因此必须在前面使用 `override` 修饰符（参见 [构造器的继承和重写](#)）。

尽管 `RecipeIngredient` 将父类的指定构造器重写为了便利构造器，但是它依然提供了父类的所有指定构造器的实现。因此，`RecipeIngredient` 会自动继承父类的所有便利构造器。

在这个例子中，`RecipeIngredient` 的父类是 `Food`，它有一个便利构造器 `init()`。这个便利构造器会被 `RecipeIngredient` 继承。这个继承版本的 `init()` 在功能上跟 `Food` 提供的版本是一样的，只是它会代理到 `RecipeIngredient` 版本的 `init(name: String)` 而不是 `Food` 提供的版本。

所有的这三种构造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，叫做 `ShoppingListItem`。这个类构建了购物单中出现的某一种食谱原料。

购物单中的每一项总是从未购买状态开始的。为了呈现这一事实，`ShoppingListItem` 引入了一个 Boolean (布尔类型) 的属性 `purchased`，它的默认值是 `false`。`ShoppingListItem` 还添加了一个计算型属性 `description`，它提供了关于 `ShoppingListItem` 实例的一些文字描述：

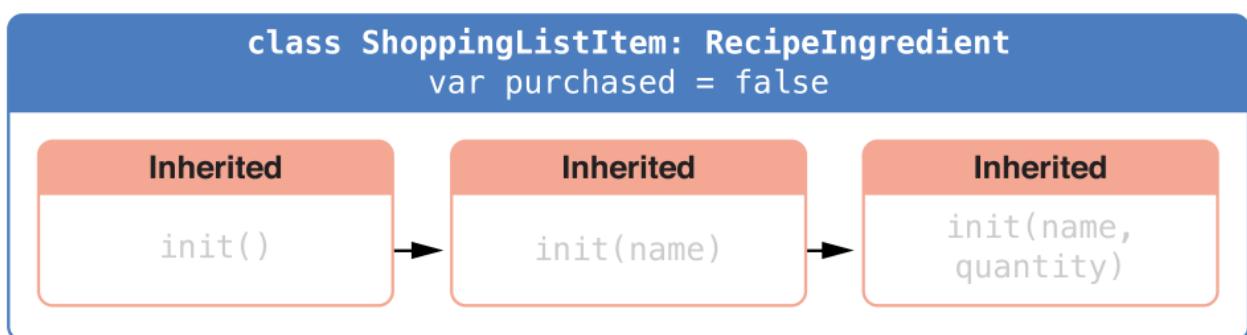
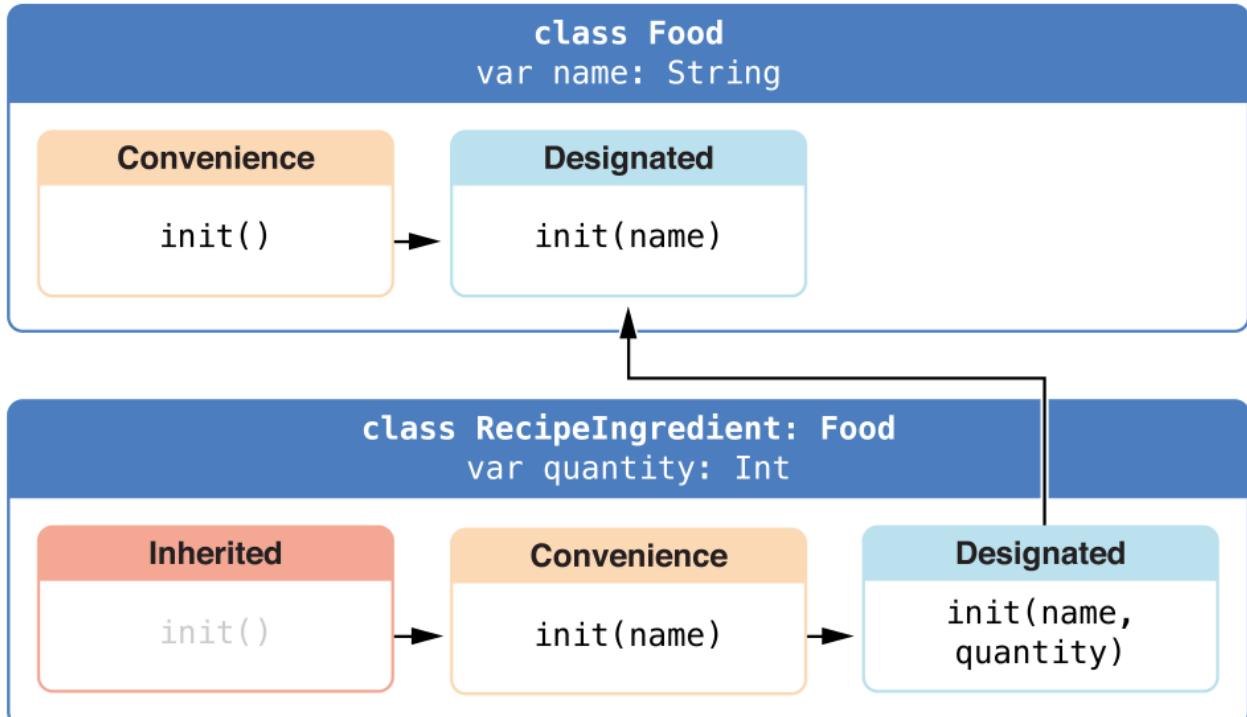
```
class ShoppingListItem: RecipeIngredient {
    var purchased = false
    var description: String {
        var output = "\u2022 (quantity) x \u2022 (name)"
        output += purchased ? " ✓" : " ✗"
        return output
    }
}
```

注意

`ShoppingListItem` 没有定义构造器来为 `purchased` 提供初始值，因为添加到购物单的物品的初始状态总是未购买。

因为它为自己引入的所有属性都提供了默认值，并且自己没有定义任何构造器，`ShoppingListItem` 将自动继承所有父类中的指定构造器和便利构造器。

下图展示了这三个类的构造器链：



你可以使用三个继承来的构造器来创建 `ShoppingListItem` 的新实例：

```

var breakfastList = [
    ShoppingListItem(),
    ShoppingListItem(name: "Bacon"),
    ShoppingListItem(name: "Eggs", quantity: 6),
]
breakfastList[0].name = "Orange juice"
breakfastList[0].purchased = true
for item in breakfastList {
    print(item.description)
}
// 1 x orange juice ✓
// 1 x bacon ✗
// 6 x eggs ✗
  
```

如上所述，例子中通过字面量方式创建了一个数组 `breakfastList`，它包含了三个 `ShoppingListItem` 实例，因此数组的类型也能被自动推导为 `[ShoppingListItem]`。在数组创建完之后，数组中第一个 `ShoppingListItem` 实例的名字从 `[Unnamed]` 更改为 `Orange juice`，并标记状态为已购买。打印数组中每个元素的描述显示了它们都已按照预期被赋值。

## 可失败构造器

有时，定义一个构造器可失败的类，结构体或者枚举是很有用的。这里所指的“失败”指的是，如给构造器传入无效的形参，或缺少某种所需的外部资源，又或是不满足某种必要的条件等。

为了妥善处理这种构造过程中可能会失败的情况。你可以在一个类，结构体或是枚举类型的定义中，添加一个或多个可失败构造器。其语法为在 `init` 关键字后面添加问号（`init?`）。

### 注意

可失败构造器的参数名和参数类型，不能与其它非可失败构造器的参数名，及其参数类型相同。

可失败构造器会创建一个类型为自身类型的可选类型的对象。你通过 `return nil` 语句来表明可失败构造器在何种情况下应该“失败”。

### 注意

严格来说，构造器都不支持返回值。因为构造器本身的作用，只是为了确保对象能被正确构造。因此你只是用 `return nil` 表明可失败构造器构造失败，而不要用关键字 `return` 来表明构造成功。

例如，实现针对数字类型转换的可失败构造器。确保数字类型之间的转换能保持精确的值，使用这个 `init(exactly:)` 构造器。如果类型转换不能保持值不变，则这个构造器构造失败。

```
let wholeNumber: Double = 12345.0
let pi = 3.14159

if let valueMaintained = Int(exactly: wholeNumber) {
    print("\(wholeNumber) conversion to Int maintains value of \(valueMaintained)")
}
// 打印“12345.0 conversion to Int maintains value of 12345”

let valueChanged = Int(exactly: pi)
// valueChanged 是 Int? 类型，不是 Int 类型

if valueChanged == nil {
    print("\(pi) conversion to Int does not maintain value")
}
// 打印“3.14159 conversion to Int does not maintain value”
```

下例中，定义了一个名为 `Animal` 的结构体，其中有一个名为 `species` 的 `String` 类型的常量属性。同时该结构体还定义了一个接受一个名为 `species` 的 `String` 类型形参的可失败构造器。这个可失败构造器检查传入的 `species` 值是否为一个空字符串。如果为空字符串，则构造失败。否则，`species` 属性被赋值，构造成功。

```
struct Animal {  
    let species: String  
    init?(species: String) {  
        if species.isEmpty {  
            return nil  
        }  
        self.species = species  
    }  
}
```

你可以通过该可失败构造器来尝试构建一个 `Animal` 的实例，并检查构造过程是否成功：

```
let someCreature = Animal(species: "Giraffe")  
// someCreature 的类型是 Animal? 而不是 Animal  
  
if let giraffe = someCreature {  
    print("An animal was initialized with a species of \(giraffe.species)")  
}  
// 打印“An animal was initialized with a species of Giraffe”
```

如果你给该可失败构造器传入一个空字符串到形参 `species`，则会导致构造失败：

```
let anonymousCreature = Animal(species: "")  
// anonymousCreature 的类型是 Animal?, 而不是 Animal  
  
if anonymousCreature == nil {  
    print("The anonymous creature could not be initialized")  
}  
// 打印“The anonymous creature could not be initialized”
```

### 注意

检查空字符串的值（如 `""`，而不是 `"Giraffe"`）和检查值为 `nil` 的可选类型的字符串是两个完全不同的概念。上例中的空字符串（`""`）其实是一个有效的，非可选类型的字符串。这里我们之所以让 `Animal` 的可失败构造器构造失败，只是因为对于 `Animal` 这个类的 `species` 属性来说，它更适合有一个具体的值，而不是空字符串。

## 枚举类型的可失败构造器

你可以通过一个带一个或多个形参的可失败构造器来获取枚举类型中特定的枚举成员。如果提供的形参无法匹配任何枚举成员，则构造失败。

下例中，定义了一个名为 `TemperatureUnit` 的枚举类型。其中包含了三个可能的枚举状态（`Kelvin`、`Celsius` 和 `Fahrenheit`），以及一个根据表示温度单位的 `Character` 值找出合适的枚举成员的可失败构造器：

```

enum TemperatureUnit {
    case Kelvin, Celsius, Fahrenheit
    init?(symbol: Character) {
        switch symbol {
            case "K":
                self = .Kelvin
            case "C":
                self = .Celsius
            case "F":
                self = .Fahrenheit
            default:
                return nil
        }
    }
}

```

你可以利用该可失败构造器在三个枚举成员中选择合适的枚举成员，当形参不能和任何枚举成员相匹配时，则构造失败：

```

let fahrenheitUnit = TemperatureUnit(symbol: "F")
if fahrenheitUnit != nil {
    print("This is a defined temperature unit, so initialization succeeded.")
}
// 打印"This is a defined temperature unit, so initialization succeeded."

let unknownUnit = TemperatureUnit(symbol: "X")
if unknownUnit == nil {
    print("This is not a defined temperature unit, so initialization failed.")
}
// 打印"This is not a defined temperature unit, so initialization failed."

```

## 带原始值的枚举类型的可失败构造器

---

带原始值的枚举类型会自带一个可失败构造器 `init?(rawValue:)`，该可失败构造器有一个合适的原始值类型的 `rawValue` 形参，选择找到的相匹配的枚举成员，找不到则构造失败。

因此上面的 `TemperatureUnit` 的例子可以用原始值类型的 `Character` 和进阶的 `init?(rawValue:)` 构造器重写为：

```

enum TemperatureUnit: Character {
    case Kelvin = "K", Celsius = "C", Fahrenheit = "F"
}

let fahrenheitUnit = TemperatureUnit(rawValue: "F")
if fahrenheitUnit != nil {
    print("This is a defined temperature unit, so initialization succeeded.")
}
// 打印"This is a defined temperature unit, so initialization succeeded."

let unknownUnit = TemperatureUnit(rawValue: "X")
if unknownUnit == nil {
    print("This is not a defined temperature unit, so initialization failed.")
}
// 打印"This is not a defined temperature unit, so initialization failed."

```

## 构造失败的传递

---

类、结构体、枚举的可失败构造器可以横向代理到它们自己其他的可失败构造器。类似的，子类的可失败构造器也能向上代理到父类的可失败构造器。

无论是向上代理还是横向代理，如果你代理到的其他可失败构造器触发构造失败，整个构造过程将立即终止，接下来的任何构造代码不会再被执行。

### 注意

可失败构造器也可以代理到其它的不可失败构造器。通过这种方式，你可以增加一个可能的失败状态到现有的构造过程中。

下面这个例子，定义了一个名为 `CartItem` 的 `Product` 类的子类。这个类建立了一个在线购物车中的物品的模型，它有一个名为 `quantity` 的常量存储型属性，并确保该属性的值至少为 `1`：

```

class Product {
    let name: String
    init?(name: String) {
        if name.isEmpty { return nil }
        self.name = name
    }
}

class CartItem: Product {
    let quantity: Int
    init?(name: String, quantity: Int) {
        if quantity < 1 { return nil }
        self.quantity = quantity
        super.init(name: name)
    }
}

```

`CartItem` 可失败构造器首先验证接收的 `quantity` 值是否大于等于 1。倘若 `quantity` 值无效，则立即终止整个构造过程，返回失败结果，且不再执行余下代码。同样地，`Product` 的可失败构造器首先检查 `name` 值，假如 `name` 值为空字符串，则构造器立即执行失败。

如果你通过传入一个非空字符串 `name` 以及一个值大于等于 1 的 `quantity` 来创建一个 `CartItem` 实例，那么构造方法能够成功被执行：

```
if let twoSocks = CartItem(name: "sock", quantity: 2) {  
    print("Item: \(twoSocks.name), quantity: \(twoSocks.quantity)")  
}  
// 打印“Item: sock, quantity: 2”
```

倘若你以一个值为 0 的 `quantity` 来创建一个 `CartItem` 实例，那么将导致 `CartItem` 构造器失败：

```
if let zeroShirts = CartItem(name: "shirt", quantity: 0) {  
    print("Item: \(zeroShirts.name), quantity: \(zeroShirts.quantity)")  
} else {  
    print("Unable to initialize zero shirts")  
}  
// 打印“Unable to initialize zero shirts”
```

同样地，如果你尝试传入一个值为空字符串的 `name` 来创建一个 `CartItem` 实例，那么将导致父类 `Product` 的构造过程失败：

```
if let oneUnnamed = CartItem(name: "", quantity: 1) {  
    print("Item: \(oneUnnamed.name), quantity: \(oneUnnamed.quantity)")  
} else {  
    print("Unable to initialize one unnamed product")  
}  
// 打印“Unable to initialize one unnamed product”
```

## 重写一个可失败构造器

如同其它的构造器，你可以在子类中重写父类的可失败构造器。或者你也可以用子类的非可失败构造器重写一个父类的可失败构造器。这使你可以定义一个不会构造失败的子类，即使父类的构造器允许构造失败。

注意，当你用子类的非可失败构造器重写父类的可失败构造器时，向上代理到父类的可失败构造器的唯一方式是对父类的可失败构造器的返回值进行强制解包。

注意

你可以用非可失败构造器重写可失败构造器，但反过来却不行。

下例定义了一个名为 `Document` 的类。这个类模拟一个文档并可以用 `name` 属性来构造，属性的值必须为一个非空字符串或 `nil`，但不能是一个空字符串：

```
class Document {  
    var name: String?  
    // 该构造器创建了一个 name 属性的值为 nil 的 document 实例  
    init() {}  
    // 该构造器创建了一个 name 属性的值为非空字符串的 document 实例  
    init?(name: String) {  
        if name.isEmpty { return nil }  
        self.name = name  
    }  
}
```

下面这个例子，定义了一个 `Document` 类的子类 `AutomaticallyNamedDocument`。这个子类重写了所有父类引入的指定构造器。这些重写确保了无论是使用 `init()` 构造器，还是使用 `init(name:)` 构造器，在没有名字或者形参传入空字符串时，生成的实例中的 `name` 属性总有初始值 `"[Untitled]"`：

```
class AutomaticallyNamedDocument: Document {  
    override init() {  
        super.init()  
        self.name = "[Untitled]"  
    }  
    override init(name: String) {  
        super.init()  
        if name.isEmpty {  
            self.name = "[Untitled]"  
        } else {  
            self.name = name  
        }  
    }  
}
```

`AutomaticallyNamedDocument` 用一个不可失败构造器 `init(name:)` 重写了父类的可失败构造器 `init?(name:)`。因为子类用另一种方式处理了空字符串的情况，所以不再需要一个可失败构造器，因此子类用一个不可失败构造器代替了父类的可失败构造器。

你可以在子类的不可失败构造器中使用强制解包来调用父类的可失败构造器。比如，下面的 `UntitledDocument` 子类的 `name` 属性的值总是 `"[Untitled]"`，它在构造过程中使用了父类的可失败构造器 `init?(name:)`：

```
class UntitledDocument: Document {  
    override init() {  
        super.init(name: "[Untitled]")!  
    }  
}
```

在这个例子中，如果在调用父类的可失败构造器 `init?(name:)` 时传入的是空字符串，那么强制解包操作会引发运行时错误。不过，因为这里是通过字符串常量来调用它，构造器不会失败，所以并不会发生运行时错误。

## init! 可失败构造器

---

通常来说我们通过在 `init` 关键字后添加问号的方式（`init?`）来定义一个可失败构造器，但你也可以通过在 `init` 后面添加感叹号的方式来定义一个可失败构造器（`init!`），该可失败构造器将会构建一个对应类型的隐式解包可选类型的对象。

你可以在 `init?` 中代理到 `init!`，反之亦然。你也可以用 `init?` 重写 `init!`，反之亦然。你还可以用 `init` 代理到 `init!`，不过，一旦 `init!` 构造失败，则会触发一个断言。

## 必要构造器

在类的构造器前添加 `required` 修饰符表明所有该类的子类都必须实现该构造器：

```
class SomeClass {  
    required init() {  
        // 构造器的实现代码  
    }  
}
```

在子类重写父类的必要构造器时，必须在子类的构造器前也添加 `required` 修饰符，表明该构造器要求也应用于继承链后面的子类。在重写父类中必要的指定构造器时，不需要添加 `override` 修饰符：

```
class SomeSubclass: SomeClass {  
    required init() {  
        // 构造器的实现代码  
    }  
}
```

注意

如果子类继承的构造器能满足必要构造器的要求，则无须在子类中显式提供必要构造器的实现。

## 通过闭包或函数设置属性的默认值

如果某个存储型属性的默认值需要一些自定义或设置，你可以使用闭包或全局函数为其提供定制的默认值。每当某个属性所在类型的新实例被构造时，对应的闭包或函数会被调用，而它们的返回值会当做默认值赋值给这个属性。

这种类型的闭包或函数通常会创建一个跟属性类型相同的临时变量，然后修改它的值以满足预期的初始状态，最后返回这个临时变量，作为属性的默认值。

下面模板介绍了如何用闭包为属性提供默认值：

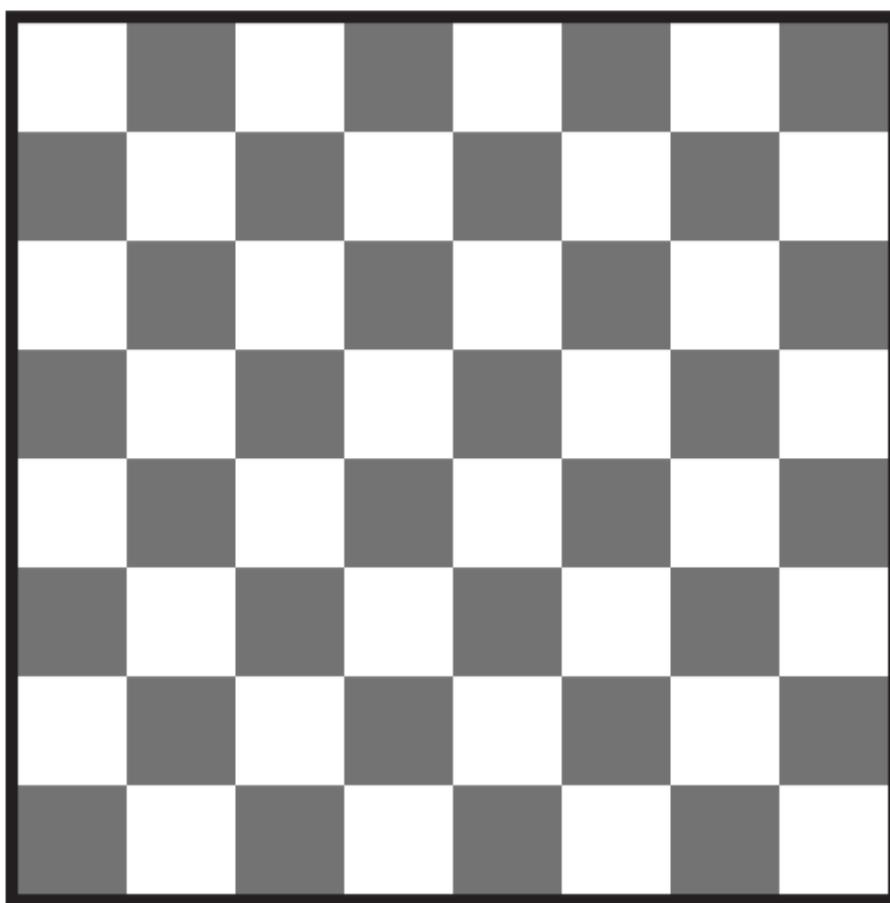
```
class SomeClass {  
    let someProperty: SomeType = {  
        // 在这个闭包中给 someProperty 创建一个默认值  
        // someValue 必须和 SomeType 类型相同  
        return someValue  
    }()  
}
```

注意闭包结尾的花括号后面接了一对空的小括号。这用来告诉 Swift 立即执行此闭包。如果你忽略了这对括号，相当于将闭包本身作为值赋值给了属性，而不是将闭包的返回值赋值给属性。

### 注意

如果你使用闭包来初始化属性，请记住在闭包执行时，实例的其它部分都还没有初始化。这意味着你不能在闭包里访问其它属性，即使这些属性有默认值。同样，你也不能使用隐式的 `self` 属性，或者调用任何实例方法。

下面例子中定义了一个结构体 `Chessboard`，它构建了西洋跳棋游戏的棋盘，西洋跳棋游戏在一副黑白格交替的 8 x 8 的棋盘中进行的：



为了呈现这副游戏棋盘，`Chessboard` 结构体定义了一个属性 `boardColors`，它是一个包含 64 个 `Bool` 值的数组。在数组中，值为 `true` 的元素表示一个黑格，值为 `false` 的元素表示一个白格。数组中第一个元素代表棋盘上左上角的格子，最后一个元素代表棋盘上右下角的格子。

`boardColors` 数组是通过一个闭包来初始化并设置颜色值的：

```

struct Chessboard {
    let boardColors: [Bool] = {
        var temporaryBoard = [Bool]()
        var isBlack = false
        for i in 1...8 {
            for j in 1...8 {
                temporaryBoard.append(isBlack)
                isBlack = !isBlack
            }
            isBlack = !isBlack
        }
        return temporaryBoard
    }()
    func squareIsBlackAt(row: Int, column: Int) -> Bool {
        return boardColors[(row * 8) + column]
    }
}

```

每当一个新的 `Chessboard` 实例被创建时，赋值闭包则会被执行，`boardColors` 的默认值会被计算出来并返回。上面例子中描述的闭包将计算出棋盘中每个格子对应的颜色，并将这些值保存到一个临时数组 `temporaryBoard` 中，最后在构建完成时将此数组作为闭包返回值返回。这个返回的数组会保存到 `boardColors` 中，并可以通过工具函数 `squareIsBlackAtRow` 来查询：

```

let board = Chessboard()
print(board.squareIsBlackAt(row: 0, column: 1))
// 打印“true”
print(board.squareIsBlackAt(row: 7, column: 7))
// 打印“false”

```

# 析构过程 · GitBook

---



[runoob.com/manual/gitbook/swift5/source/\\_book/chapter2/15\\_Deinitialization.html](https://runoob.com/manual/gitbook/swift5/source/_book/chapter2/15_Deinitialization.html)

## 析构过程

---

析构器只适用于类类型，当一个类的实例被释放之前，析构器会被立即调用。析构器用关键字 `deinit` 来标示，类似于构造器要用 `init` 来标示。

## 析构过程原理

---

Swift 会自动释放不再需要的实例以释放资源。如 [自动引用计数](#) 章节中所讲述，Swift 通过 [自动引用计数 \(ARC\)](#) 处理实例的内存管理。通常当你的实例被释放时不需要手动地去清理。但是，当使用自己的资源时，你可能需要进行一些额外的清理。例如，如果创建了一个自定义的类来打开一个文件，并写入一些数据，你可能需要在类实例被释放之前手动去关闭该文件。

在类的定义中，每个类最多只能有一个析构器，而且析构器不带任何参数和圆括号，如下所示：

```
deinit {  
    // 执行析构过程  
}
```

析构器是在实例释放发生前被自动调用的。你不能主动调用析构器。子类继承了父类的析构器，并且在子类析构器实现的最后，父类的析构器会被自动调用。即使子类没有提供自己的析构器，父类的析构器也同样会被调用。

因为直到实例的析构器被调用后，实例才会被释放，所以析构器可以访问实例的所有属性，并且可以根据那些属性可以修改它的行为（比如查找一个需要被关闭的文件）。

## 析构器实践

---

这是一个析构器实践的例子。这个例子描述了一个简单的游戏，这里定义了两种新类型，分别是 `Bank` 和 `Player`。`Bank` 类管理一种虚拟硬币，确保流通的硬币数量永远不可能超过 10,000。在游戏中有且只能有一个 `Bank` 存在，因此 `Bank` 用类来实现，并使用类型属性和类型方法来存储和管理其当前状态。

```

class Bank {
    static var coinsInBank = 10_000
    static func distribute(coins numberOfCoinsRequested: Int) -> Int {
        let numberOfCoinsToVend = min(numberOfCoinsRequested, coinsInBank)
        coinsInBank -= numberOfCoinsToVend
        return numberOfCoinsToVend
    }
    static func receive(coins: Int) {
        coinsInBank += coins
    }
}

```

`Bank` 使用 `coinsInBank` 属性来跟踪它当前拥有的硬币数量。`Bank` 还提供了两个方法，`distribute(coins:)` 和 `receive(coins:)`，分别用来处理硬币的分发和收集。

`distribute(coins:)` 方法在 `Bank` 对象分发硬币之前检查是否有足够的硬币。如果硬币不足，`Bank` 对象会返回一个比请求时小的数字（如果 `Bank` 对象中没有硬币了就返回 `0`）。此方法返回一个整型值，表示提供的硬币的实际数量。

`receive(coins:)` 方法只是将 `Bank` 实例接收到的硬币数目加回硬币存储中。

`Player` 类描述了游戏中的一个玩家。每一个玩家在任意时间都有一定数量的硬币存储在他们的钱包中。这通过玩家的 `coinsInPurse` 属性来表示：

```

class Player {
    var coinsInPurse: Int
    init(coins: Int) {
        coinsInPurse = Bank.distribute(coins: coins)
    }
    func win(coins: Int) {
        coinsInPurse += Bank.distribute(coins: coins)
    }
    deinit {
        Bank.receive(coins: coinsInPurse)
    }
}

```

每个 `Player` 实例在初始化的过程中，都从 `Bank` 对象获取指定数量的硬币。如果没有足够的硬币可用，`Player` 实例可能会收到比指定数量少的硬币。

`Player` 类定义了一个 `win(coins:)` 方法，该方法从 `Bank` 对象获取一定数量的硬币，并把它们添加到玩家的钱包。`Player` 类还实现了一个析构器，这个析构器在 `Player` 实例释放前被调用。在这里，析构器的作用只是将玩家的所有硬币都返还给 `Bank` 对象：

```

var playerOne: Player? = Player(coins: 100)
print("A new player has joined the game with \(playerOne!.coinsInPurse) coins")
// 打印“A new player has joined the game with 100 coins”
print("There are now \(Bank.coinsInBank) coins left in the bank")
// 打印“There are now 9900 coins left in the bank”

```

创建一个 `Player` 实例的时候，会向 `Bank` 对象申请得到 100 个硬币，前提是足够硬币可用。这个 `Player` 实例存储在一个名为 `playerOne` 的可选类型的变量中。这里使用了一个可选类型的变量，是因为玩家可以随时离开游戏，设置为可选使你可以追踪玩家当前是否在游戏中。

因为 `playerOne` 是可选的，所以在访问其 `coinsInPurse` 属性来打印钱包中的硬币数量和调用 `win(coins:)` 方法时，使用感叹号（`!`）强制解包：

```
playerOne!.win(coins: 2_000)
print("PlayerOne won 2000 coins & now has \(playerOne!.coinsInPurse) coins")
// 打印"PlayerOne won 2000 coins & now has 2100 coins"
print("The bank now only has \(Bank.coinsInBank) coins left")
// 打印"The bank now only has 7900 coins left"
```

在这里，玩家已经赢得了 2,000 枚硬币，所以玩家的钱包中现在有 2,100 枚硬币，而 `Bank` 对象只剩余 7,900 枚硬币。

```
playerOne = nil
print("PlayerOne has left the game")
// 打印"PlayerOne has left the game"
print("The bank now has \(Bank.coinsInBank) coins")
// 打印"The bank now has 10000 coins"
```

玩家现在已经离开了游戏。这通过将可选类型的 `playerOne` 变量设置为 `nil` 来表示，意味着“没有 `Player` 实例”。当这一切发生时，`playerOne` 变量对 `Player` 实例的引用被破坏了。没有其它属性或者变量引用 `Player` 实例，因此该实例会被释放，以便回收内存。在这之前，该实例的析构器被自动调用，玩家的硬币被返还给银行。

## 可选链式调用

可选链式调用是一种可以在当前值可能为 `nil` 的可选值上请求和调用属性、方法及下标的方法。如果可选值有值，那么调用就会成功；如果可选值是 `nil`，那么调用将返回 `nil`。多个调用可以连接在一起形成一个调用链，如果其中任何一个节点为 `nil`，整个调用链都会失败，即返回 `nil`。

注意

Swift 的可选链式调用和 Objective-C 中向 `nil` 发送消息有些相像，但是 Swift 的可选链式调用可以应用于任意类型，并且能检查调用是否成功。

## 使用可选链式调用代替强制展开

通过在想调用的属性、方法，或下标的可选值后面放一个问号（`?`），可以定义一个可选链。这一点很像在可选值后面放一个叹号（`!`）来强制展开它的值。它们的主要区别在于当可选值为空时可选链式调用只会调用失败，然而强制展开将会触发运行时错误。

为了反映可选链式调用可以在空值（`nil`）上调用的事实，不论这个调用的属性、方法及下标返回的值是不是可选值，它的返回结果都是一个可选值。你可以利用这个返回值来判断你的可选链式调用是否调用成功，如果调用有返回值则说明调用成功，返回 `nil` 则说明调用失败。

这里需要特别指出，可选链式调用的返回结果与原本的返回结果具有相同的类型，但是被包装成了一个可选值。例如，使用可选链式调用访问属性，当可选链式调用成功时，如果属性原本的返回结果是 `Int` 类型，则会变为 `Int?` 类型。

下面几段代码将解释可选链式调用和强制展开的不同。

首先定义两个类 `Person` 和 `Residence`：

```
class Person {  
    var residence: Residence?  
}
```

```
class Residence {  
    var numberOfRooms = 1  
}
```

`Residence` 有一个 `Int` 类型的属性 `numberOfRooms`，其默认值为 `1`。`Person` 具有一个可选的 `residence` 属性，其类型为 `Residence?`。

假如你创建了一个新的 `Person` 实例，它的 `residence` 属性由于是可选类型而将被初始化为 `nil`，在下面的代码中，`john` 有一个值为 `nil` 的 `residence` 属性：

```
let john = Person()
```

如果使用叹号（`!`）强制展开获得这个 `john` 的 `residence` 属性中的 `numberOfRooms` 值，会触发运行时错误，因为这时 `residence` 没有可以展开的值：

```
let roomCount = john.residence!.numberOfRooms  
// 这会引发运行时错误
```

`john.residence` 为非 `nil` 值的时候，上面的调用会成功，并且把 `roomCount` 设置为 `Int` 类型的房间数量。正如上面提到的，当 `residence` 为 `nil` 的时候，上面这段代码会触发运行时错误。

可选链式调用提供了另一种访问 `numberOfRooms` 的方式，使用问号（`?`）来替代原来的叹号（`!`）：

```
if let roomCount = john.residence?.numberOfRooms {  
    print("John's residence has \(roomCount) room(s).")  
} else {  
    print("Unable to retrieve the number of rooms.")  
}  
// 打印“Unable to retrieve the number of rooms.”
```

在 `residence` 后面添加问号之后，Swift 就会在 `residence` 不为 `nil` 的情况下访问 `numberOfRooms`。

因为访问 `numberOfRooms` 有可能失败，可选链式调用会返回 `Int?` 类型，或称为“可选的 `Int`”。如上例所示，当 `residence` 为 `nil` 的时候，可选的 `Int` 将会为 `nil`，表明无法访问 `numberOfRooms`。访问成功时，可选的 `Int` 值会通过可选绑定展开，并赋值给非可选类型的 `roomCount` 常量。

要注意的是，即使 `numberOfRooms` 是非可选的 `Int` 时，这一点也成立。只要使用可选链式调用就意味着 `numberOfRooms` 会返回一个 `Int?` 而不是 `Int`。

可以将一个 `Residence` 的实例赋给 `john.residence`，这样它就不再是 `nil` 了：

```
john.residence = Residence()
```

`john.residence` 现在包含一个实际的 `Residence` 实例，而不再是 `nil`。如果你试图使用先前的可选链式调用访问 `numberOfRooms`，它现在将返回值为 `1` 的 `Int?` 类型的值：

```
if let roomCount = john.residence?.numberOfRooms {  
    print("John's residence has \(roomCount) room(s).")  
} else {  
    print("Unable to retrieve the number of rooms.")  
}  
// 打印“John's residence has 1 room(s).”
```

## 为可选链式调用定义模型类

---

通过使用可选链式调用可以调用多层属性、方法和下标。这样可以在复杂的模型中向下访问各种子属性，并且判断能否访问子属性的属性、方法和下标。

下面这段代码定义了四个模型类，这些例子包括多层次可选链式调用。为了方便说明，在 `Person` 和 `Residence` 的基础上增加了 `Room` 类和 `Address` 类，以及相关的属性、方法以及下标。

`Person` 类的定义基本保持不变：

```
class Person {  
    var residence: Residence?  
}
```

`Residence` 类比之前复杂些，增加了一个名为 `rooms` 的变量属性，该属性被初始化为 `[Room]` 类型的空数组：

```
class Residence {  
    var rooms = [Room]()  
    var numberOfRooms: Int {  
        return rooms.count  
    }  
    subscript(i: Int) -> Room {  
        get {  
            return rooms[i]  
        }  
        set {  
            rooms[i] = newValue  
        }  
    }  
    func printNumberOfRooms() {  
        print("The number of rooms is \(numberOfRooms)")  
    }  
    var address: Address?  
}
```

现在 `Residence` 有了一个存储 `Room` 实例的数组，`numberOfRooms` 属性被实现为计算型属性，而不是存储型属性。`numberOfRooms` 属性简单地返回 `rooms` 数组的 `count` 属性的值。

`Residence` 还提供了访问 `rooms` 数组的快捷方式，即提供可读写的下标来访问 `rooms` 数组中指定位置的元素。

此外，`Residence` 还提供了 `printNumberOfRooms` 方法，这个方法的作用是打印 `numberOfRooms` 的值。

最后，`Residence` 还定义了一个可选属性 `address`，其类型为 `Address?`。`Address` 类的定义在下面会说明。

`Room` 类是一个简单类，其实例被存储在 `rooms` 数组中。该类只包含一个属性 `name`，以及一个用于将该属性设置为适当的房间名的初始化函数：

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

最后一个类是 `Address`，这个类有三个 `String?` 类型的可选属性。`buildingName` 以及 `buildingNumber` 属性分别表示大厦的名称和号码，第三个属性 `street` 表示大厦所在街道的名称：

```
class Address {  
    var buildingName: String?  
    var buildingNumber: String?  
    var street: String?  
    func buildingIdentifier() -> String? {  
        if buildingName != nil {  
            return buildingName  
        } else if let buildingNumber = buildingNumber, let street = street {  
            return "\(buildingNumber) \(street)"  
        } else {  
            return nil  
        }  
    }  
}
```

`Address` 类提供了 `buildingIdentifier()` 方法，返回值为 `String?`。如果 `buildingName` 有值则返回 `buildingName`。或者，如果 `buildingNumber` 和 `street` 均有值，则返回两者拼接得到的字符串。否则，返回 `nil`。

## 通过可选链式调用访问属性

正如 使用可选链式调用代替强制展开 中所述，可以通过可选链式调用在一个可选值上访问它的属性，并判断访问是否成功。

使用前面定义过的类，创建一个 `Person` 实例，然后像之前一样，尝试访问 `numberOfRooms` 属性：

```
let john = Person()  
if let roomCount = john.residence?.numberOfRooms {  
    print("John's residence has \(roomCount) room(s).")  
} else {  
    print("Unable to retrieve the number of rooms.")  
}  
// 打印“Unable to retrieve the number of rooms.”
```

因为 `john.residence` 为 `nil`，所以这个可选链式调用依旧会像先前一样失败。

还可以通过可选链式调用来自设置属性值：

```
let someAddress = Address()  
someAddress.buildingNumber = "29"  
someAddress.street = "Acacia Road"  
john.residence?.address = someAddress
```

在这个例子中，通过 `john.residence` 来设定 `address` 属性也会失败，因为 `john.residence` 当前为 `nil`。

上面代码中的赋值过程是可选链式调用的一部分，这意味着可选链式调用失败时，等号右侧的代码不会被执行。对于上面的代码来说，很难验证这一点，因为像这样赋值一个常量没有任何副作用。下面的代码完成了同样的事情，但是它使用一个函数来创建 `Address` 实例，然后将该实例返回用于赋值。该函数会在返回前打印“Function was called”，这使你能验证等号右侧的代码是否被执行。

```
func createAddress() -> Address {  
    print("Function was called."  
  
    let someAddress = Address()  
    someAddress.buildingNumber = "29"  
    someAddress.street = "Acacia Road"  
  
    return someAddress  
}  
john.residence?.address = createAddress()
```

没有任何打印消息，可以看出 `createAddress()` 函数并未被执行。

## 通过可选链式调用来调用方法

---

可以通过可选链式调用来调用方法，并判断是否调用成功，即使这个方法没有返回值。

`Residence` 类中的 `printNumberOfRooms()` 方法打印当前的 `numberOfRooms` 值，如下所示：

```
func printNumberOfRooms() {  
    print("The number of rooms is \(numberOfRooms)")  
}
```

这个方法没有返回值。然而，没有返回值的方法具有隐式的返回类型 `Void`，如 [无返回值函数](#) 中所述。这意味着没有返回值的方法也会返回 `()`，或者说空的元组。

如果在可选值上通过可选链式调用来调用这个方法，该方法的返回类型会是 `Void?`，而不是 `Void`，因为通过可选链式调用得到的返回值都是可选的。这样我们就可以使用 `if` 语句来判断能否成功调用 `printNumberOfRooms()` 方法，即使方法本身没有定义返回值。通过判断返回值是否为 `nil` 可以判断调用是否成功：

```
if john.residence?.printNumberOfRooms() != nil {  
    print("It was possible to print the number of rooms.")  
} else {  
    print("It was not possible to print the number of rooms.")  
}  
// 打印"It was not possible to print the number of rooms."
```

同样的，可以据此判断通过可选链式调用为属性赋值是否成功。在上面的 [通过可选链式调用访问属性](#) 的例子中，我们尝试给 `john.residence` 中的 `address` 属性赋值，即使 `residence` 为 `nil`。通过可选链式调用给属性赋值会返回 `Void?`，通过判断返回值是否为 `nil` 就可以知道赋值是否成功：

```
if (john.residence?.address = someAddress) != nil {  
    print("It was possible to set the address.")  
} else {  
    print("It was not possible to set the address.")  
}  
// 打印"It was not possible to set the address."
```

## 通过可选链式调用访问下标

---

通过可选链式调用，我们可以在一个可选值上访问下标，并且判断下标调用是否成功。

注意

通过可选链式调用访问可选值的下标时，应该将问号放在下标方括号的前面而不是后面。可选链式调用的问号一般直接跟在可选表达式的后面。

下面这个例子用下标访问 `john.residence` 属性存储的 `Residence` 实例的 `rooms` 数组中的第一个房间的名称，因为 `john.residence` 为 `nil`，所以下标调用失败了：

```
if let firstRoomName = john.residence?[0].name {  
    print("The first room name is \(firstRoomName).")  
} else {  
    print("Unable to retrieve the first room name.")  
}  
// 打印"Unable to retrieve the first room name."
```

在这个例子中，问号直接放在 `john.residence` 的后面，并且在方括号的前面，因为 `john.residence` 是可选值。

类似的，可以通过下标，用可选链式调用来赋值：

```
john.residence?[0] = Room(name: "Bathroom")
```

这次赋值同样会失败，因为 `residence` 目前是 `nil`。

如果你创建一个 `Residence` 实例，并为其 `rooms` 数组添加一些 `Room` 实例，然后将 `Residence` 实例赋值给 `john.residence`，那就可以通过可选链和下标来访问数组中的元素：

```
let johnsHouse = Residence()
johnsHouse.rooms.append(Room(name: "Living Room"))
johnsHouse.rooms.append(Room(name: "Kitchen"))
john.residence = johnsHouse

if let firstRoomName = john.residence?[0].name {
    print("The first room name is \(firstRoomName).")
} else {
    print("Unable to retrieve the first room name.")
}
// 打印"The first room name is Living Room."
```

## 访问可选类型的下标

---

如果下标返回可选类型值，比如 Swift 中 `Dictionary` 类型的键的下标，可以在下标的结尾括号后面放一个问号来在其可选返回值上进行可选链式调用：

```
var testScores = ["Dave": [86, 82, 84], "Bev": [79, 94, 81]]
testScores["Dave"]?[0] = 91
testScores["Bev"]?[0] += 1
testScores["Brian"]?[0] = 72
// "Dave" 数组现在是 [91, 82, 84], "Bev" 数组现在是 [80, 94, 81]
```

上面的例子中定义了一个 `testScores` 数组，包含了两个键值对，分别把 `String` 类型的键映射到一个 `Int` 值的数组。这个例子用可选链式调用把 `"Dave"` 数组中第一个元素设为 `91`，把 `"Bev"` 数组的第一个元素 `+1`，然后尝试把 `"Brian"` 数组中的第一个元素设为 `72`。前两个调用成功，因为 `testScores` 字典中包含 `"Dave"` 和 `"Bev"` 这两个键。但是 `testScores` 字典中没有 `"Brian"` 这个键，所以第三个调用失败。

## 连接多层可选链式调用

---

可以通过连接多个可选链式调用在更深的模型层级中访问属性、方法以及下标。然而，多层可选链式调用不会增加返回值的可选层级。

也就是说：

- 如果你访问的值不是可选的，可选链式调用将会返回可选值。
- 如果你访问的值就是可选的，可选链式调用不会让可选返回值变得“更可选”。

因此：

- 通过可选链式调用访问一个 `Int` 值，将会返回 `Int?`，无论使用了多少层可选链式调用。
- 类似的，通过可选链式调用访问 `Int?` 值，依旧会返回 `Int?` 值，并不会返回 `Int??`。

下面的例子尝试访问 `john` 中的 `residence` 属性中的 `address` 属性中的 `street` 属性。这里使用了两层可选链式调用，`residence` 以及 `address` 都是可选值：

```
if let johnsStreet = john.residence?.address?.street {  
    print("John's street name is \(johnsStreet).")  
} else {  
    print("Unable to retrieve the address.")  
}  
// 打印“Unable to retrieve the address.”
```

`john.residence` 现在包含一个有效的 `Residence` 实例。然而，`john.residence.address` 的值当前为 `nil`。因此，调用 `john.residence?.address?.street` 会失败。

需要注意的是，上面的例子中，`street` 的属性为 `String?`。`john.residence?.address?.street` 的返回值也依然是 `String?`，即使已经使用了两层可选链式调用。

如果为 `john.residence.address` 赋值一个 `Address` 实例，并且为 `address` 中的 `street` 属性设置一个有效值，我们就能通过可选链式调用来访问 `street` 属性：

```
let johnsAddress = Address()  
johnsAddress.buildingName = "The Larches"  
johnsAddress.street = "Laurel Street"  
john.residence?.address = johnsAddress  
  
if let johnsStreet = john.residence?.address?.street {  
    print("John's street name is \(johnsStreet).")  
} else {  
    print("Unable to retrieve the address.")  
}  
// 打印“John's street name is Laurel Street.”
```

在上面的例子中，因为 `john.residence` 包含一个有效的 `Address` 实例，所以对 `john.residence` 的 `address` 属性赋值将会成功。

## 在方法的可选返回值上进行可选链式调用

---

上面的例子展示了如何在一个可选值上通过可选链式调用来获取它的属性值。我们还可以在一个可选值上通过可选链式调用来调用方法，并且可以根据需要继续在方法的可选返回值上进行可选链式调用。

在下面的例子中，通过可选链式调用来调用 `Address` 的 `buildingIdentifier()` 方法。这个方法返回 `String?` 类型的值。如上所述，通过可选链式调用来调用该方法，最终的返回值依旧会是 `String?` 类型：

```
if let buildingIdentifier = john.residence?.address?.buildingIdentifier() {  
    print("John's building identifier is \(buildingIdentifier).")  
}  
// 打印“John's building identifier is The Larches.”
```

如果要在该方法的返回值上进行可选链式调用，在方法的圆括号后面加上问号即可：

```
if let beginsWithThe =  
    john.residence?.address?.buildingIdentifier()?.hasPrefix("The") {  
    if beginsWithThe {  
        print("John's building identifier begins with \"The\".")  
    } else {  
        print("John's building identifier does not begin with \"The\".")  
    }  
}  
// 打印“John's building identifier begins with "The".”
```

注意

在上面的例子中，在方法的圆括号后面加上问号是因为你要在 `buildingIdentifier()` 方法的可选返回值上进行可选链式调用，而不是 `buildingIdentifier()` 方法本身。

# 错误处理 · GitBook



[runoob.com/manual/gitbook/swift5/source/\\_book/chapter2/17\\_Error\\_Handling.html](http://runoob.com/manual/gitbook/swift5/source/_book/chapter2/17_Error_Handling.html)

## 错误处理

错误处理 (*Error handling*) 是响应错误以及从错误中恢复的过程。Swift 在运行时提供了抛出、捕获、传递和操作可恢复错误 (recoverable errors) 的一等支持 (first-class support)。

某些操作无法保证总是执行完所有代码或生成有用的结果。可选类型用来表示值缺失，但是当某个操作失败时，理解造成失败的原因有助于你的代码作出相应的应对。

举个例子，假如有个从磁盘上的某个文件读取数据并进行处理的任务，该任务会有多种可能失败的情况，包括指定路径下文件并不存在，文件不具有可读权限，或者文件编码格式不兼容。区分这些不同的失败情况可以让程序处理并解决某些错误，然后把它解决不了的错误报告给用户。

### 注意

Swift 中的错误处理涉及到错误处理模式，这会用到 Cocoa 和 Objective-C 中的 `NSError`。更多详情参见 [用 Swift 解决 Cocoa 错误](#)。

## 表示与抛出错误

在 Swift 中，错误用遵循 `Error` 协议的类型的值来表示。这个空协议表明该类型可以用于错误处理。

Swift 的枚举类型尤为适合构建一组相关的错误状态，枚举的关联值还可以提供错误状态的额外信息。例如，在游戏中操作自动贩卖机时，你可以这样表示可能出现的错误状态：

```
enum VendingMachineError: Error {
    case invalidSelection          //选择无效
    case insufficientFunds(coinsNeeded: Int) //金额不足
    case outOfStock                //缺货
}
```

抛出一个错误可以让你表明有意外情况发生，导致正常的执行流程无法继续执行。抛出错误使用 `throw` 语句。例如，下面的代码抛出一个错误，提示贩卖机还需要 5 个硬币：

```
throw VendingMachineError.insufficientFunds(coinsNeeded: 5)
```

## 处理错误

某个错误被抛出时，附近的某部分代码必须负责处理这个错误，例如纠正这个问题、尝试另外一种方式、或是向用户报告错误。

Swift 中有 4 种处理错误的方式。你可以把函数抛出的错误传递给调用此函数的代码、用 `do-catch` 语句处理错误、将错误作为可选类型处理、或者断言此错误根本不会发生。每种方式在下面的小节中都有描述。

当一个函数抛出一个错误时，你的程序流程会发生改变，所以重要的是你能迅速识别代码中会抛出错误的地方。为了标识出这些地方，在调用一个能抛出错误的函数、方法或者构造器之前，加上 `try` 关键字，或者 `try?` 或 `try!` 这种变体。这些关键字在下面的小节中有具体讲解。

### 注意

Swift 中的错误处理和其他语言中用 `try`，`catch` 和 `throw` 进行异常处理很像。和其他语言中（包括 Objective-C）的异常处理不同的是，Swift 中的错误处理并不涉及解除调用栈，这是一个计算代价高昂的过程。就此而言，`throw` 语句的性能特性是可以和 `return` 语句相媲美的。

## 用 `throwing` 函数传递错误

为了表示一个函数、方法或构造器可以抛出错误，在函数声明的参数之后加上 `throws` 关键字。一个标有 `throws` 关键字的函数被称作 *throwing 函数*。如果这个函数指明了返回值类型，`throws` 关键词需要写在返回箭头 (`->`) 的前面。

```
func canThrowErrors() throws -> String
```

```
func cannotThrowErrors() -> String
```

一个 `throwing` 函数可以在其内部抛出错误，并将错误传递到函数被调用时的作用域。

### 注意

只有 `throwing` 函数可以传递错误。任何在某个非 `throwing` 函数内部抛出的错误只能在函数内部处理。

下面的例子中，`VendingMachine` 类有一个 `vend(itemNamed:)` 方法，如果请求的物品不存在、缺货或者投入金额小于物品价格，该方法就会抛出一个相应的

`VendingMachineError`：

```

struct Item {
    var price: Int
    var count: Int
}

class VendingMachine {
    var inventory = [
        "Candy Bar": Item(price: 12, count: 7),
        "Chips": Item(price: 10, count: 4),
        "Pretzels": Item(price: 7, count: 11)
    ]
    var coinsDeposited = 0

    func vend(itemNamed name: String) throws {
        guard let item = inventory[name] else {
            throw VendingMachineError.invalidSelection
        }

        guard item.count > 0 else {
            throw VendingMachineError.outOfStock
        }

        guard item.price <= coinsDeposited else {
            throw VendingMachineError.insufficientFunds(coinsNeeded: item.price - coinsDeposited)
        }

        coinsDeposited -= item.price

        var newItem = item
        newItem.count -= 1
        inventory[name] = newItem

        print("Dispensing \(name)")
    }
}

```

在 `vend(itemNamed:)` 方法的实现中使用了 `guard` 语句来确保在购买某个物品所需的条件中有任一条件不满足时，能提前退出方法并抛出相应的错误。由于 `throw` 语句会立即退出方法，所以物品只有在所有条件都满足时才会被售出。

因为 `vend(itemNamed:)` 方法会传递出它抛出的任何错误，在你的代码中调用此方法的地方，必须要么直接处理这些错误——使用 `do-catch` 语句，`try?` 或 `try!`；要么继续将这些错误传递下去。例如下面例子中，`buyFavoriteSnack(person:vendingMachine:)` 同样是一个 `throwing` 函数，任何由 `vend(itemNamed:)` 方法抛出的错误会一直被传递到 `buyFavoriteSnack(person:vendingMachine:)` 函数被调用的地方。

```

let favoriteSnacks = [
    "Alice": "Chips",
    "Bob": "Licorice",
    "Eve": "Pretzels",
]
func buyFavoriteSnack(person: String, vendingMachine: VendingMachine) throws {
    let snackName = favoriteSnacks[person] ?? "Candy Bar"
    try vendingMachine.vend(itemNamed: snackName)
}

```

上例中，`buyFavoriteSnack(person:vendingMachine:)` 函数会查找某人最喜欢的零食，并通过调用 `vend(itemNamed:)` 方法来尝试为他们购买。因为 `vend(itemNamed:)` 方法能抛出错误，所以在调用的它时候在它前面加了 `try` 关键字。

`throwing` 构造器能像 `throwing` 函数一样传递错误。例如下面代码中的 `PurchasedSnack` 构造器在构造过程中调用了 `throwing` 函数，并且通过传递到它的调用者来处理这些错误。

```

struct PurchasedSnack {
    let name: String
    init(name: String, vendingMachine: VendingMachine) throws {
        try vendingMachine.vend(itemNamed: name)
        self.name = name
    }
}

```

## 用 Do-Catch 处理错误

---

你可以使用一个 `do-catch` 语句运行一段闭包代码来处理错误。如果在 `do` 子句中的代码抛出了一个错误，这个错误会与 `catch` 子句做匹配，从而决定哪条子句能处理它。

下面是 `do-catch` 语句的一般形式：

```

do {
    try expression
    statements
} catch pattern 1 {
    statements
} catch pattern 2 where condition {
    statements
} catch {
    statements
}

```

在 `catch` 后面写一个匹配模式来表明这个子句能处理什么样的错误。如果一条 `catch` 子句没有指定匹配模式，那么这条子句可以匹配任何错误，并且把错误绑定到一个名字为 `error` 的局部常量。关于模式匹配的更多信息请参考 [模式](#)。

举例来说，下面的代码处理了 `VendingMachineError` 枚举类型的全部三种情况：

```

var vendingMachine = VendingMachine()
vendingMachine.coinsDeposited = 8
do {
    try buyFavoriteSnack(person: "Alice", vendingMachine: vendingMachine)
    print("Success! Yum.")
} catch VendingMachineError.invalidSelection {
    print("Invalid Selection.")
} catch VendingMachineError.outOfStock {
    print("Out of Stock.")
} catch VendingMachineError.insufficientFunds(let coinsNeeded) {
    print("Insufficient funds. Please insert an additional \n(coinsNeeded) coins.")
} catch {
    print("Unexpected error: \n(error).")
}
// 打印“Insufficient funds. Please insert an additional 2 coins.”
```

上面的例子中，`buyFavoriteSnack(person:vendingMachine:)` 函数在一个 `try` 表达式中被调用，是因为它能抛出错误。如果错误被抛出，相应的执行会马上转移到 `catch` 子句中，并判断这个错误是否要被继续传递下去。如果错误没有被匹配，它会被最后一个 `catch` 语句捕获，并赋值给一个 `error` 常量。如果没有错误被抛出，`do` 子句中余下的语句就会被执行。

`catch` 子句不必将 `do` 子句中的代码所抛出的每一个可能的错误都作处理。如果所有 `catch` 子句都未处理错误，错误就会传递到周围的作用域。然而，错误还是必须要被某个周围的作用域处理的。在不会抛出错误的函数中，必须用 `do-catch` 语句处理错误。而能够抛出错误的函数既可以使用 `do-catch` 语句处理，也可以让调用方来处理错误。如果错误传递到了顶层作用域却依然没有被处理，你会得到一个运行时错误。

以下面的代码为例，不是 `VendingMachineError` 中申明的错误会在调用函数的地方被捕获：

```

func nourish(with item: String) throws {
    do {
        try vendingMachine.vend(itemNamed: item)
    } catch is VendingMachineError {
        print("Invalid selection, out of stock, or not enough money.")
    }
}

do {
    try nourish(with: "Beet-Flavored Chips")
} catch {
    print("Unexpected non-vending-machine-related error: \n(error)")
}
// 打印“Invalid selection, out of stock, or not enough money.”
```

如果 `vend(itemNamed:)` 抛出的是一个 `VendingMachineError` 类型的错误，`nourish(with:)` 会打印一条消息，否则 `nourish(with:)` 会将错误抛给它的调用方。这个错误之后会被通用的 `catch` 语句捕获。

## 将错误转换成可选值

可以使用 `try?` 通过将错误转换成一个可选值来处理错误。如果是在计算 `try?` 表达式时抛出错误，该表达式的结果就为 `nil`。例如，在下面的代码中，`x` 和 `y` 有着相同的数值和等价的含义：

```
func someThrowingFunction() throws -> Int {  
    // ...  
}
```

```
let x = try? someThrowingFunction()
```

```
let y: Int?  
do {  
    y = try someThrowingFunction()  
} catch {  
    y = nil  
}
```

如果 `someThrowingFunction()` 抛出一个错误，`x` 和 `y` 的值是 `nil`。否则 `x` 和 `y` 的值就是该函数的返回值。注意，无论 `someThrowingFunction()` 的返回值类型是什么类型，`x` 和 `y` 都是这个类型的可选类型。例子中此函数返回一个整型，所以 `x` 和 `y` 是可选整型。

如果你想对所有的错误都采用同样的方式来处理，用 `try?` 就可以让你写出简洁的错误处理代码。例如，下面的代码用几种方式来获取数据，如果所有方式都失败了则返回 `nil`。

```
func fetchData() -> Data? {  
    if let data = try? fetchDataFromDisk() { return data }  
    if let data = try? fetchDataFromServer() { return data }  
    return nil  
}
```

## 禁用错误传递

有时你知道某个 `throwing` 函数实际上在运行时是不会抛出错误的，在这种情况下，你可以在表达式前面写 `try!` 来禁用错误传递，这会把调用包装在一个不会有错误抛出的运行时断言中。如果真的抛出了错误，你会得到一个运行时错误。

例如，下面的代码使用了 `loadImage(atPath:)` 函数，该函数从给定的路径加载图片资源，如果图片无法载入则抛出一个错误。在这种情况下，因为图片是和应用绑定的，运行时不会有错误抛出，所以适合禁用错误传递。

```
let photo = try! loadImage(atPath: "./Resources/John Appleseed.jpg")
```

## 指定清理操作

你可以使用 `defer` 语句在即将离开当前代码块时执行一系列语句。该语句让你能执行一些必要的清理工作，不管是以何种方式离开当前代码块的——无论是由于抛出错误而离开，或是由于诸如 `return`、`break` 的语句。例如，你可以用 `defer` 语句来确保文件描述符得以关闭，以及手动分配的内存得以释放。

`defer` 语句将代码的执行延迟到当前的作用域退出之前。该语句由 `defer` 关键字和要被延迟执行的语句组成。延迟执行的语句不能包含任何控制转移语句，例如 `break`、`return` 语句，或是抛出一个错误。延迟执行的操作会按照它们声明的顺序从后往前执行——也就是说，第一条 `defer` 语句中的代码最后才执行，第二条 `defer` 语句中的代码倒数第二个执行，以此类推。最后一条语句会第一个执行。

```
func processFile(filename: String) throws {
    if exists(filename) {
        let file = open(filename)
        defer {
            close(file)
        }
        while let line = try file.readline() {
            // 处理文件。
        }
        // close(file) 会在这里被调用，即作用域的最后。
    }
}
```

上面的代码使用一条 `defer` 语句来确保 `open(_)` 函数有一个相应的对 `close(_)` 函数的调用。

注意

即使没有涉及到错误处理的代码，你也可以使用 `defer` 语句。

# 类型转换 · GitBook

---



[runoob.com/manual/gitbook/swift5/source/\\_book/chapter2/18\\_Type\\_Casting.html](https://runoob.com/manual/gitbook/swift5/source/_book/chapter2/18_Type_Casting.html)

## 类型转换

---

类型转换可以判断实例的类型，也可以将实例看做是其父类或者子类的实例。

类型转换在 Swift 中使用 `is` 和 `as` 操作符实现。这两个操作符分别提供了一种简单达意的方式去检查值的类型或者转换它的类型。

你也可以用它来检查一个类型是否遵循了某个协议，就像在 [检验协议遵循](#) 部分讲述的一样。

## 为类型转换定义类层次

---

你可以将类型转换用在类和子类的层次结构上，检查特定类实例的类型并且转换这个类实例的类型成为这个层次结构中的其他类型。下面的三个代码段定义了一个类层次和一个包含了这些类实例的数组，作为类型转换的例子。

第一个代码片段定义了一个新的基类 `MedialItem`。这个类为任何出现在数字媒体库的媒体项提供基础功能。特别的，它声明了一个 `String` 类型的 `name` 属性，和一个 `init(name:)` 初始化器。（假定所有的媒体项都有个名称。）

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

下一个代码段定义了 `MedialItem` 的两个子类。第一个子类 `Movie` 封装了与电影相关的额外信息，在父类（或者说基类）的基础上增加了一个 `director`（导演）属性，和相应的初始化器。第二个子类 `Song`，在父类的基础上增加了一个 `artist`（艺术家）属性，和相应的初始化器：

```

class Movie: MedialItem {
    var director: String
    init(name: String, director: String) {
        self.director = director
        super.init(name: name)
    }
}

class Song: MedialItem {
    var artist: String
    init(name: String, artist: String) {
        self.artist = artist
        super.init(name: name)
    }
}

```

最后一个代码段创建了一个数组常量 `library`，包含两个 `Movie` 实例和三个 `Song` 实例。`library` 的类型是在它被初始化时根据它数组中所包含的内容推断来的。Swift 的类型检测器能够推断出 `Movie` 和 `Song` 有共同的父类 `MedialItem`，所以它推断出 `[MedialItem]` 类作为 `library` 的类型：

```

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")
]
// 数组 library 的类型被推断为 [MedialItem]

```

在幕后 `library` 里存储的媒体项依然是 `Movie` 和 `Song` 类型的。但是，若你迭代它，依次取出的实例会是 `MedialItem` 类型的，而不是 `Movie` 和 `Song` 类型。为了让它们作为原本的类型工作，你需要检查它们的类型或者向下转换它们到其它类型，就像下面描述的一样。

## 检查类型

---

用 **类型检查操作符** (`is`) 来检查一个实例是否属于特定子类型。若实例属于那个子类型，类型检查操作符返回 `true`，否则返回 `false`。

下面的例子定义了两个变量，`movieCount` 和 `songCount`，用来计算数组 `library` 中 `Movie` 和 `Song` 类型的实例数量：

```

var movieCount = 0
var songCount = 0

for item in library {
    if item is Movie {
        movieCount += 1
    } else if item is Song {
        songCount += 1
    }
}

print("Media library contains \$(movieCount) movies and \$(songCount) songs")
// 打印“Media library contains 2 movies and 3 songs”

```

示例迭代了数组 `library` 中的所有项。每一次，`for-in` 循环设置 `item` 常量为数组中的下一个 `MediaItem` 实例。

若当前 `MediaItem` 是一个 `Movie` 类型的实例，`item is Movie` 返回 `true`，否则返回 `false`。同样的，`item is Song` 检查 `item` 是否为 `Song` 类型的实例。在循环结束后，`movieCount` 和 `songCount` 的值就是被找到的属于各自类型的实例的数量。

## 向下转型

---

某类型的一个常量或变量可能在幕后实际上属于一个子类。当确定是这种情况时，你可以尝试用类型转换操作符（`as?` 或 `as!`）向下转到它的子类型。

因为向下转型可能会失败，类型转换操作符带有两种不同形式。条件形式 `as?` 返回一个你试图向下转成的类型的可选值。强制形式 `as!` 把试图向下转型和强制解包转换结果结合为一个操作。

当你不确定向下转型可以成功时，用类型转换的条件形式（`as?`）。条件形式的类型转换总是返回一个可选值，并且若下转是不可能的，可选值将是 `nil`。这使你能够检查向下转型是否成功。

只有你可以确定向下转型一定会成功时，才使用强制形式（`as!`）。当你试图向下转型为一个不正确的类型时，强制形式的类型转换会触发一个运行时错误。

下面的例子，迭代了 `library` 里的每一个 `MediaItem`，并打印出适当的描述。要这样做，`item` 需要真正作为 `Movie` 或 `Song` 的类型来使用，而不仅仅是作为 `MediaItem`。为了能够在描述中使用 `Movie` 或 `Song` 的 `director` 或 `artist` 属性，这是必要的。

在这个示例中，数组中的每一个 `item` 可能是 `Movie` 或 `Song`。事前你不知道每个 `item` 的真实类型，所以这里使用条件形式的类型转换（`as?`）去检查循环里的每次下转：

```

for item in library {
    if let movie = item as? Movie {
        print("Movie: \(movie.name), dir. \(movie.director)")
    } else if let song = item as? Song {
        print("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

```

示例首先试图将 `item` 下转为 `Movie`。因为 `item` 是一个 `MediaItem` 类型的实例，它可能是一个 `Movie`；同样，它也可能是一个 `Song`，或者仅仅是基类 `MediaItem`。因为不确定，`as?` 形式在试图下转时将返回一个可选值。`item as? Movie` 的返回值是 `Movie?` 或者说“可选 `Movie`”。

当向下转型为 `Movie` 应用在两个 `Song` 实例时将会失败。为了处理这种情况，上面的例子使用了可选绑定（optional binding）来检查可选 `Movie` 真的包含一个值（这个是为了判断下转是否成功。）可选绑定是这样写的“`if let movie = item as? Movie`”，可以这样解读：

“尝试将 `item` 转为 `Movie` 类型。若成功，设置一个新的临时常量 `movie` 来存储返回的可选 `Movie` 中的值”

若向下转型成功，然后 `movie` 的属性将用于打印一个 `Movie` 实例的描述，包括它的导演的名字 `director`。相似的原理被用来检测 `Song` 实例，当 `Song` 被找到时则打印它的描述（包含 `artist` 的名字）。

### 注意

转换没有真的改变实例或它的值。根本的实例保持不变；只是简单地把它作为它被转换成的类型来使用。

## Any 和 AnyObject 的类型转换

Swift 为不确定类型提供了两种特殊的类型别名：

- `Any` 可以表示任何类型，包括函数类型。
- `AnyObject` 可以表示任何类类型的实例。

只有当你确实需要它们的行为和功能时才使用 `Any` 和 `AnyObject`。最好还是在代码中指明需要使用的类型。

这里有个示例，使用 `Any` 类型来和混合的不同类型一起工作，包括函数类型和非类类型。它创建了一个可以存储 `Any` 类型的数组 `things`：

```

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.append({ (name: String) -> String in "Hello, \(name)" })

```

`things` 数组包含两个 `Int` 值，两个 `Double` 值，一个 `String` 值，一个元组 `(Double, Double)`，一个 `Movie` 实例“`Ghostbusters`”，以及一个接受 `String` 值并返回另一个 `String` 值的闭包表达式。

你可以在 `switch` 表达式的 `case` 中使用 `is` 和 `as` 操作符来找出只知道是 `Any` 或 `AnyObject` 类型的常量或变量的具体类型。下面的示例迭代 `things` 数组中的每一项，并用 `switch` 语句查找每一项的类型。有几个 `switch` 语句的 `case` 绑定它们匹配到的值到一个指定类型的常量，从而可以打印这些值：

```

for thing in things {
    switch thing {
        case 0 as Int:
            print("zero as an Int")
        case 0 as Double:
            print("zero as a Double")
        case let someInt as Int:
            print("an integer value of \(someInt)")
        case let someDouble as Double where someDouble > 0:
            print("a positive double value of \(someDouble)")
        case is Double:
            print("some other double value that I don't want to print")
        case let someString as String:
            print("a string value of \"\(someString)\"")
        case let (x, y) as (Double, Double):
            print("an (x, y) point at \(x), \(y)")
        case let movie as Movie:
            print("a movie called \(movie.name), dir. \(movie.director)")
        case let stringConverter as (String) -> String:
            print(stringConverter("Michael"))
        default:
            print("something else")
    }
}

// zero as an Int
// zero as a Double
// an integer value of 42
// a positive double value of 3.14159
// a string value of "hello"
// an (x, y) point at 3.0, 5.0
// a movie called Ghostbusters, dir. Ivan Reitman
// Hello, Michael

```

## 注意

`Any` 类型可以表示所有类型的值，包括可选类型。Swift 会在你用 `Any` 类型来表示一个可选值的时候，给你一个警告。如果你确实想使用 `Any` 类型来承载可选值，你可以使用 `as` 操作符显式转换为 `Any`，如下所示：

```
let optionalNumber: Int? = 3
things.append(optionalNumber)      // 警告
things.append(optionalNumber as Any) // 没有警告
```



## 嵌套类型

---

枚举常被用于为特定类或结构体实现某些功能。类似地，枚举可以方便的定义工具类或结构体，从而为某个复杂的类型所使用。为了实现这种功能，Swift 允许你定义嵌套类型，可以在支持的类型中定义嵌套的枚举、类和结构体。

要在一个类型中嵌套另一个类型，将嵌套类型的定义写在其外部类型的 `{}` 内，而且可以根据需要定义多级嵌套。

## 嵌套类型实践

---

下面这个例子定义了一个结构体 `BlackjackCard`（二十一点），用来模拟 `BlackjackCard` 中的扑克牌点数。`BlackjackCard` 结构体包含两个嵌套定义的枚举类型 `Suit` 和 `Rank`。

在 `BlackjackCard` 中，`Ace` 牌可以表示 `1` 或者 `11`，`Ace` 牌的这一特征通过一个嵌套在 `Rank` 枚举中的结构体 `Values` 来表示：

```

struct BlackjackCard {

    // 嵌套的 Suit 枚举
    enum Suit: Character {
        case spades = "♠", hearts = "♥", diamonds = "◇", clubs = "♣"
    }

    // 嵌套的 Rank 枚举
    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.rawValue, second: nil)
            }
        }
    }
}

// BlackjackCard 的属性和方法
let rank: Rank, suit: Suit
var description: String {
    var output = "suit is \(suit.rawValue),"
    output += " value is \(rank.values.first)"
    if let second = rank.values.second {
        output += " or \(second)"
    }
    return output
}

```

`Suit` 枚举用来描述扑克牌的四种花色，并用一个 `Character` 类型的原始值表示花色符号。

`Rank` 枚举用来描述扑克牌从 `Ace ~ 10`，以及 `J`、`Q`、`K`，这 `13` 种牌，并用一个 `Int` 类型的原始值表示牌的面值。（这个 `Int` 类型的原始值未用于 `Ace`、`J`、`Q`、`K` 这 `4` 种牌。）

如上所述，`Rank` 枚举在内部定义了一个嵌套结构体 `Values`。结构体 `Values` 中定义了两个属性，用于反映只有 `Ace` 有两个数值，其余牌都只有一个数值：

- `first` 的类型为 `Int`
- `second` 的类型为 `Int?`，或者说“可选 `Int`”

`Rank` 还定义了一个计算型属性 `values`，它将会返回一个 `Values` 结构体的实例。这个计算型属性会根据牌的面值，用适当的数值去初始化 `Values` 实例。对于 `J`、`Q`、`K`、`Ace` 这四种牌，会使用特殊数值。对于数字面值的牌，使用枚举实例的 `Int` 类型的原始值。

`BlackjackCard` 结构体拥有两个属性——`rank` 与 `suit`。它也同样定义了一个计算型属性 `description`，`description` 属性用 `rank` 和 `suit` 中的内容来构建对扑克牌名字和数值的描述。该属性使用可选绑定来检查可选类型 `second` 是否有值，若有值，则在原有的描述中增加对 `second` 的描述。

因为 `BlackjackCard` 是一个没有自定义构造器的结构体，在 结构体的逐一成员构造器 中可知，结构体有默认的成员构造器，所以你可以用默认的构造器去初始化新常量 `theAceOfSpades`：

```
let theAceOfSpades = BlackjackCard(rank: .ace, suit: .spades)
print("theAceOfSpades: \(theAceOfSpades.description)")
// 打印“theAceOfSpades: suit is ♠, value is 1 or 11”
```

尽管 `Rank` 和 `Suit` 嵌套在 `BlackjackCard` 中，但它们的类型仍可从上下文中推断出来，所以在初始化实例时能够单独通过成员名称（`.ace` 和 `.spades`）引用枚举实例。在上面的例子中，`description` 属性正确地反映了黑桃 A 牌具有 `1` 和 `11` 两个值。

## 引用嵌套类型

---

在外部引用嵌套类型时，在嵌套类型的类型名前加上其外部类型的类型名作为前缀：

```
let heartsSymbol = BlackjackCard.Suit.hearts.rawValue
// 红心符号为“♥”
```

对于上面这个例子，这样可以使 `Suit`、`Rank` 和 `Values` 的名字尽可能的短，因为它们的名字可以由定义它们的上下文来限定。

# 扩展 · GitBook

---



[runoob.com/manual/gitbook/swift5/source/\\_book/chapter2/20\\_Extensions.html](http://runoob.com/manual/gitbook/swift5/source/_book/chapter2/20_Extensions.html)

## 扩展

---

扩展可以给一个现有的类，结构体，枚举，还有协议添加新的功能。它还拥有不需要访问被扩展类型源代码就能完成扩展的能力（即逆向建模）。扩展和 Objective-C 的分类很相似。（与 Objective-C 分类不同的是，Swift 扩展是没有名字的。）

Swift 中的扩展可以：

- 添加计算型实例属性和计算型类属性
- 定义实例方法和类方法
- 提供新的构造器
- 定义下标
- 定义和使用新的嵌套类型
- 使已经存在的类型遵循（conform）一个协议

在 Swift 中，你甚至可以扩展协议以提供其需要的实现，或者添加额外功能给遵循的类型所使用。你可以从 [协议扩展](#) 获取更多细节。

注意

扩展可以给一个类型添加新的功能，但是不能重写已经存在的功能。

## 扩展的语法

---

使用 `extension` 关键字声明扩展：

```
extension SomeType {  
    // 在这里给 SomeType 添加新的功能  
}
```

扩展可以扩充一个现有的类型，给它添加一个或多个协议。协议名称的写法和类或者结构体一样：

```
extension SomeType: SomeProtocol, AnotherProtocol {  
    // 协议所需要的实现写在这里  
}
```

这种遵循协议的方式在 [使用扩展遵循协议](#) 中有描述。

扩展可以使用在现有范型类型上，就像 [扩展范型类型](#) 中描述的一样。你还可以使用扩展给泛型类型有条件的添加功能，就像 [扩展一个带有 Where 字句的范型](#) 中描述的一样。

## 注意

对一个现有的类型，如果你定义了一个扩展来添加新的功能，那么这个类型的所有实例都可以使用这个新功能，包括那些在扩展定义之前就存在的实例。

## 计算型属性

扩展可以给现有类型添加计算型实例属性和计算型类属性。这个例子给 Swift 内建的 `Double` 类型添加了五个计算型实例属性，从而提供与距离单位相关工作的基本支持：

```
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  
print("One inch is \(oneInch) meters")  
// 打印“One inch is 0.0254 meters”  
let threeFeet = 3.ft  
print("Three feet is \(threeFeet) meters")  
// 打印“Three feet is 0.914399970739201 meters”
```

这些计算型属性表示的含义是把一个 `Double` 值看作是某单位下的长度值。即使它们被实现为计算型属性，但这些属性的名字仍可紧接一个浮点型字面值，从而通过点语法来使用，并以此实现距离转换。

在上述例子中，`Double` 类型的 `1.0` 代表的是“一米”。这就是为什么计算型属性 `m` 返回的是 `self` ——表达式 `1.m` 被认为是计算一个 `Double` 类型的 `1.0`。

其它单位则需要一些单位换算。一千米等于 1,000 米，所以计算型属性 `km` 要把值乘以 `1_000.00` 来实现千米到米的单位换算。类似地，一米有 3.28084 英尺，所以计算型属性 `ft` 要把对应的 `Double` 值除以 `3.28084`，来实现英尺到米的单位换算。

这些属性都是只读的计算型属性，所以为了简便，它们的表达式里面都不包含 `get` 关键字。它们使用 `Double` 作为返回值类型，并可用于所有接受 `Double` 类型的数学计算中：

```
let aMarathon = 42.km + 195.m  
print("A marathon is \(aMarathon) meters long")  
// 打印“A marathon is 42195.0 meters long”
```

## 注意

扩展可以添加新的计算属性，但是它们不能添加存储属性，或向现有的属性添加属性观察者。

## 构造器

扩展可以给现有的类型添加新的构造器。它使你可以把自定义类型作为参数来供其他类型的构造器使用，或者在类型的原始实现上添加额外的构造选项。

扩展可以给一个类添加新的便利构造器，但是它们不能给类添加新的指定构造器或者析构器。指定构造器和析构器必须始终由类的原始实现提供。

如果你使用扩展给一个值类型添加构造器只是用于给所有的存储属性提供默认值，并且没有定义任何自定义构造器，那么你可以在该值类型扩展的构造器中使用默认构造器和成员构造器。如果你把构造器写到了值类型的原始实现中，就像 [值类型的构造器委托](#) 中所描述的，那么就不属于在扩展中添加构造器。

如果你使用扩展给另一个模块中定义的结构体添加构造器，那么新的构造器直到定义模块中使用一个构造器之前，不能访问 `self`。

在下面的例子中，自定义了一个的 `Rect` 结构体用来表示一个几何矩形。这个例子中还定义了两个给予支持的结构体 `Size` 和 `Point`，它们都把属性的默认值设置为 `0.0`：

```
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()  
}
```

因为 `Rect` 结构体给所有的属性都提供了默认值，所以它自动获得了一个默认构造器和一个成员构造器，就像 [默认构造器](#) 中描述的一样。这些构造器可以用来创建新的 `Rect` 实例：

```
let defaultRect = Rect()  
let memberwiseRect = Rect(origin: Point(x: 2.0, y: 2.0),  
    size: Size(width: 5.0, height: 5.0))
```

你可以通过扩展 `Rect` 结构体来提供一个允许指定 `point` 和 `size` 的构造器：

```
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)  
    }  
}
```

这个新的构造器首先根据提供的 `center` 和 `size` 计算一个适当的原点。然后这个构造器调用结构体自带的成员构造器 `init(origin:size:)`，它会将新的 `origin` 和 `size` 值储存在适当的属性中：

```
let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
                      size: Size(width: 3.0, height: 3.0))
// centerRect 的 origin 是 (2.5, 2.5) 并且它的 size 是 (3.0, 3.0)
```

注意

如果你通过扩展提供一个新的构造器，你有责任确保每个通过该构造器创建的实例都是初始化完整的。

## 方法

扩展可以给现有类型添加新的实例方法和类方法。在下面的例子中，给 `Int` 类型添加了一个新的实例方法叫做 `repetitions`：

```
extension Int {
    func repetitions(task: () -> Void) {
        for _ in 0..
```

`repetitions(task:)` 方法仅接收一个 `() -> Void` 类型的参数，它表示一个没有参数没有返回值的方法。

定义了这个扩展之后，你可以对任意整形数值调用 `repetitions(task:)` 方法，来执行对应次数的任务：

```
3.repetitions {
    print("Hello!")
}
// Hello!
// Hello!
// Hello!
```

## 可变实例方法

通过扩展添加的实例方法同样也可以修改（或 *mutating*（改变））实例本身。结构体和枚举的方法，若是可以修改 `self` 或者它自己的属性，则必须将这个实例方法标记为 `mutating`，就像是改变了方法的原始实现。

在下面的例子中，对 Swift 的 `Int` 类型添加了一个新的 *mutating* 方法，叫做 `square`，它将原始值求平方：

```
extension Int {
    mutating func square() {
        self = self * self
    }
}
var someInt = 3
someInt.square()
// someInt 现在是 9
```

## 下标

---

扩展可以给现有的类型添加新的下标。下面的例子中，对 Swift 的 `Int` 类型添加了一个整数类型的下标。下标 `[n]` 从数字右侧开始，返回小数点后的第 `n` 位：

- `123456789[0]` 返回 `9`
- `123456789[1]` 返回 `8`

.....以此类推：

```
extension Int {  
    subscript(digitIndex: Int) -> Int {  
        var decimalBase = 1  
        for _ in 0..<digitIndex {  
            decimalBase *= 10  
        }  
        return (self / decimalBase) % 10  
    }  
}  
746381295[0]  
// 返回 5  
746381295[1]  
// 返回 9  
746381295[2]  
// 返回 2  
746381295[8]  
// 返回 7
```

如果操作的 `Int` 值没有足够的位数满足所请求的下标，那么下标的现实将返回 `0`，将好像在数字的左边补上了 `0`：

```
746381295[9]  
// 返回 0，就好像你进行了这个请求：  
0746381295[9]
```

## 嵌套类型

---

扩展可以给现有的类，结构体，还有枚举添加新的嵌套类型：

```

extension Int {
    enum Kind {
        case negative, zero, positive
    }
    var kind: Kind {
        switch self {
        case 0:
            return .zero
        case let x where x > 0:
            return .positive
        default:
            return .negative
        }
    }
}

```

这个例子给 `Int` 添加了一个新的嵌套枚举。这个枚举叫做 `Kind`，表示特定整数所代表的数字类型。具体来说，它表示数字是负的、零的还是正的。

这个例子同样给 `Int` 添加了一个新的计算型实例属性，叫做 `kind`，它返回被操作整数所对应的 `Kind` 枚举 case 分支。

现在，任意 `Int` 的值都可以使用这个嵌套类型：

```

func printIntegerKinds(_ numbers: [Int]) {
    for number in numbers {
        switch number.kind {
        case .negative:
            print("-", terminator: "")
        case .zero:
            print("0 ", terminator: "")
        case .positive:
            print("+ ", terminator: "")
        }
    }
    print("")
}

printIntegerKinds([3, 19, -27, 0, -6, 0, 7])
// 打印“+ + - 0 - 0 +”

```

方法 `printIntegerKinds(_)`，使用一个 `Int` 类型的数组作为输入，然后依次迭代这些值。对于数组中的每一个整数，方法会检查它的 `kind` 计算型属性，然后打印适当的描述。

### 注意

`number.kind` 已经被认为是 `Int.Kind` 类型。所以，在 `switch` 语句中所有的 `Int.Kind` case 分支可以被缩写，就像使用 `.negative` 替代 `Int.Kind.negative.`。

# 协议 · GitBook

---

 [runoob.com/manual/gitbook/swift5/source/\\_book/chapter2/21\\_Proocols.html](https://runoob.com/manual/gitbook/swift5/source/_book/chapter2/21_Proocols.html)

## 协议

---

协议定义了一个蓝图，规定了用来实现某一特定任务或者功能的方法、属性，以及其他需要的东西。类、结构体或枚举都可以遵循协议，并为协议定义的这些要求提供具体实现。某个类型能够满足某个协议的要求，就可以说该类型遵循这个协议。

除了遵循协议的类型必须实现的要求外，还可以对协议进行扩展，通过扩展来实现一部分要求或者实现一些附加功能，这样遵循协议的类型就能够使用这些功能。

## 协议语法

---

协议的定义方式与类、结构体和枚举的定义非常相似：

```
protocol SomeProtocol {  
    // 这里是协议的定义部分  
}
```

要让自定义类型遵循某个协议，在定义类型时，需要在类型名称后加上协议名称，中间以冒号（`:`）分隔。遵循多个协议时，各协议之间用逗号（`,`）分隔：

```
struct SomeStructure: FirstProtocol, AnotherProtocol {  
    // 这里是结构体的定义部分  
}
```

若一个拥有父类的类在遵循协议时，应该将父类名放在协议名之前，以逗号分隔：

```
class SomeClass: SomeSuperClass, FirstProtocol, AnotherProtocol {  
    // 这里是类的定义部分  
}
```

## 属性要求

---

协议可以要求遵循协议的类型提供特定名称和类型的实例属性或类型属性。协议不指定属性是存储属性还是计算属性，它只指定属性的名称和类型。此外，协议还指定属性是可读的还是可读可写的。

如果协议要求属性是可读可写的，那么该属性不能是常量属性或只读的计算型属性。如果协议只要求属性是可读的，那么该属性不仅可以是可读的，如果代码需要的话，还可以是可写的。

协议总是用 `var` 关键字来声明变量属性，在类型声明后加上 `{ set get }` 来表示属性是可读可写的，可读属性则用 `{ get }` 来表示：

```
protocol SomeProtocol {  
    var mustBeSettable: Int { get set }  
    var doesNotNeedToBeSettable: Int { get }  
}
```

在协议中定义类型属性时，总是使用 `static` 关键字作为前缀。当类类型遵循协议时，除了 `static` 关键字，还可以使用 `class` 关键字来声明类型属性：

```
protocol AnotherProtocol {  
    static var someTypeProperty: Int { get set }  
}
```

如下所示，这是一个只含有一个实例属性要求的协议：

```
protocol FullyNamed {  
    var fullName: String { get }  
}
```

`FullyNamed` 协议除了要求遵循协议的类型提供 `fullName` 属性外，并没有其他特别的要求。这个协议表示，任何遵循 `FullyNamed` 的类型，都必须有一个可读的 `String` 类型的实例属性 `fullName`。

下面是一个遵循 `FullyNamed` 协议的简单结构体：

```
struct Person: FullyNamed {  
    var fullName: String  
}  
let john = Person(fullName: "John Appleseed")  
// john.fullName 为 "John Appleseed"
```

这个例子中定义了一个叫做 `Person` 的结构体，用来表示一个具有名字的人。从第一行代码可以看出，它遵循了 `FullyNamed` 协议。

`Person` 结构体的每一个实例都有一个 `String` 类型的存储型属性 `fullName`。这正好满足了 `FullyNamed` 协议的要求，也就意味着 `Person` 结构体正确地符合了协议。（如果协议要求未被完全满足，在编译时会报错。）

下面是一个更为复杂的类，它采纳并遵循了 `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 != nil ? prefix! + " " : "") + name  
    }  
}  
var ncc1701 = Starship(name: "Enterprise", prefix: "USS")  
// ncc1701.fullName 为 "USS Enterprise"
```

`Starship` 类把 `fullName` 作为只读的计算属性来实现。每一个 `Starship` 类的实例都有一个名为 `name` 的非可选属性和一个名为 `prefix` 的可选属性。当 `prefix` 存在时，计算属性 `fullName` 会将 `prefix` 插入到 `name` 之前，从而得到一个带有 `prefix` 的 `fullName`。

## 方法要求

---

协议可以要求遵循协议的类型实现某些指定的实例方法或类方法。这些方法作为协议的一部分，像普通方法一样放在协议的定义中，但是不需要大括号和方法体。可以在协议中定义具有可变参数的方法，和普通方法的定义方式相同。但是，不支持为协议中的方法提供默认参数。

正如属性要求中所述，在协议中定义类方法的时候，总是使用 `static` 关键字作为前缀。即使在类实现时，类方法要求使用 `class` 或 `static` 作为关键字前缀，前面的规则仍然适用：

```
protocol SomeProtocol {  
    static func someTypeMethod()  
}
```

下面的例子定义了一个只含有一个实例方法的协议：

```
protocol RandomNumberGenerator {  
    func random() -> Double  
}
```

`RandomNumberGenerator` 协议要求遵循协议的类型必须拥有一个名为 `random`，返回值类型为 `Double` 的实例方法。尽管这里并未指明，但是我们假设返回值是从 `0.0` 到（但不包括） `1.0`。

`RandomNumberGenerator` 协议并不关心每一个随机数是怎样生成的，它只要求必须提供一个随机数生成器。

如下所示，下边是一个遵循并符合 `RandomNumberGenerator` 协议的类。该类实现了一个叫做 线性同余生成器 (*linear congruential generator*) 的伪随机数算法。

```
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).truncatingRemainder(dividingBy:m))  
        return lastRandom / m  
    }  
}  
let generator = LinearCongruentialGenerator()  
print("Here's a random number: \(generator.random())")  
// 打印 "Here's a random number: 0.37464991998171"  
print("And another one: \(generator.random())")  
// 打印 "And another one: 0.729023776863283"
```

## 异变方法要求

有时需要在方法中改变（或异变）方法所属的实例。例如，在值类型（即结构体和枚举）的实例方法中，将 `mutating` 关键字作为方法的前缀，写在 `func` 关键字之前，表示可以在该方法中修改它所属的实例以及实例的任意属性的值。这一过程在 [在实例方法中修改值类型](#) 章节中有详细描述。

如果你在协议中定义了一个实例方法，该方法会改变遵循该协议的类型的实例，那么在定义协议时需要在方法前加 `mutating` 关键字。这使得结构体和枚举能够遵循此协议并满足此方法要求。

### 注意

实现协议中的 `mutating` 方法时，若是类类型，则不用写 `mutating` 关键字。而对于结构体和枚举，则必须写 `mutating` 关键字。

如下所示，`Toggable` 协议只定义了一个名为 `toggle` 的实例方法。顾名思义，`toggle()` 方法将改变实例属性，从而切换遵循该协议类型的实例的状态。

`toggle()` 方法在定义的时候，使用 `mutating` 关键字标记，这表明当它被调用时，该方法将会改变遵循协议的类型的实例：

```
protocol Toggable {  
    mutating func toggle()  
}
```

当使用枚举或结构体来实现 `Toggable` 协议时，需要提供一个带有 `mutating` 前缀的 `toggle()` 方法。

下面定义了一个名为 `OnOffSwitch` 的枚举。这个枚举在两种状态之间进行切换，用枚举成员 `On` 和 `Off` 表示。枚举的 `toggle()` 方法被标记为 `mutating`，以满足 `Toggable` 协议的要求：

```
enum OnOffSwitch: Toggable {  
    case off, on  
    mutating func toggle() {  
        switch self {  
            case .off:  
                self = .on  
            case .on:  
                self = .off  
        }  
    }  
    var lightSwitch = OnOffSwitch.off  
    lightSwitch.toggle()  
    // lightSwitch 现在的值为 .on
```

## 构造器要求

协议可以要求遵循协议的类型实现指定的构造器。你可以像编写普通构造器那样，在协议的定义里写下构造器的声明，但不需要写花括号和构造器的实体：

```
protocol SomeProtocol {  
    init(someParameter: Int)  
}
```

## 协议构造器要求的类实现

---

你可以在遵循协议的类中实现构造器，无论是作为指定构造器，还是作为便利构造器。无论哪种情况，你都必须为构造器实现标上 `required` 修饰符：

```
class SomeClass: SomeProtocol {  
    required init(someParameter: Int) {  
        // 这里是构造器的实现部分  
    }  
}
```

使用 `required` 修饰符可以确保所有子类也必须提供此构造器实现，从而也能符合协议。

关于 `required` 构造器的更多内容，请参考 [必要构造器](#)。

### 注意

如果类已经被标记为 `final`，那么不需要在协议构造器的实现中使用 `required` 修饰符，因为 `final` 类不能有子类。关于 `final` 修饰符的更多内容，请参见 [防止重写](#)。

如果一个子类重写了父类的指定构造器，并且该构造器满足了某个协议的要求，那么该构造器的实现需要同时标注 `required` 和 `override` 修饰符：

```
protocol SomeProtocol {  
    init()  
}  
  
class SomeSuperClass {  
    init() {  
        // 这里是构造器的实现部分  
    }  
}  
  
class SomeSubClass: SomeSuperClass, SomeProtocol {  
    // 因为遵循协议，需要加上 required  
    // 因为继承自父类，需要加上 override  
    required override init() {  
        // 这里是构造器的实现部分  
    }  
}
```

## 可失败构造器要求

---

协议还可以为遵循协议的类型定义可失败构造器要求，详见 [可失败构造器](#)。

遵循协议的类型可以通过可失败构造器（`init?`）或非可失败构造器（`init`）来满足协议中定义的可失败构造器要求。协议中定义的非可失败构造器要求可以通过非可失败构造器（`init`）或隐式解包可失败构造器（`init!`）来满足。

## 协议作为类型

---

尽管协议本身并未实现任何功能，但是协议可以被当做一个功能完备的类型来使用。协议作为类型使用，有时被称作「存在类型」，这个名词来自「存在着一个类型 T，该类型遵循协议 T」。

协议可以像其他普通类型一样使用，使用场景如下：

- 作为函数、方法或构造器中的参数类型或返回值类型
- 作为常量、变量或属性的类型
- 作为数组、字典或其他容器中的元素类型

### 注意

协议是一种类型，因此协议类型的名称应与其他类型（例如 `Int`，`Double`，`String`）的写法相同，使用大写字母开头的驼峰式写法，例如（`FullyNamed` 和 `RandomNumberGenerator`）。

下面是将协议作为类型使用的例子：

```
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  
    }  
}
```

例子中定义了一个 `Dice` 类，用来代表桌游中拥有 N 个面的骰子。`Dice` 的实例含有 `sides` 和 `generator` 两个属性，前者是整型，用来表示骰子有几个面，后者为骰子提供一个随机数生成器，从而生成随机点数。

`generator` 属性的类型为 `RandomNumberGenerator`，因此任何遵循了 `RandomNumberGenerator` 协议的类型的实例都可以赋值给 `generator`，除此之外并无其他要求。并且由于其类型是 `RandomNumberGenerator`，在 `Dice` 类中与 `generator` 交互的代码，必须适用于所有 `generator` 实例都遵循的方法。这句话的意思是不能使用由 `generator` 底层类型提供的任何方法或属性。但是你可以通过向下转型，从协议类型转换成底层实现类型，比如从父类向下转型为子类。请参考 [向下转型](#)。

`Dice` 类还有一个构造器，用来设置初始状态。构造器有一个名为 `generator`，类型为 `RandomNumberGenerator` 的形参。在调用构造方法创建 `Dice` 的实例时，可以传入任何遵循 `RandomNumberGenerator` 协议的实例给 `generator`。

`Dice` 类提供了一个名为 `roll` 的实例方法，用来模拟骰子的面值。它先调用 `generator` 的 `random()` 方法来生成一个  $[0.0,1.0)$  区间内的随机数，然后使用这个随机数生成正确的骰子面值。因为 `generator` 遵循了 `RandomNumberGenerator` 协议，可以确保它有个 `random()` 方法可供调用。

下面的例子展示了如何使用 `LinearCongruentialGenerator` 的实例作为随机数生成器来创建一个六面骰子：

```
var d6 = Dice(sides: 6, generator: LinearCongruentialGenerator())
for _ in 1...5 {
    print("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
```

## 委托

---

委托是一种设计模式，它允许类或结构体将一些需要它们负责的功能委托给其他类型的实例。委托模式的实现很简单：定义协议来封装那些需要被委托的功能，这样就能确保遵循协议的类型能提供这些功能。委托模式可以用来响应特定的动作，或者接收外部数据源提供的数据，而无需关心外部数据源的类型。

下面的例子定义了两个基于骰子游戏的协议：

```
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` 协议可以被任意涉及骰子的游戏遵循。

`DiceGameDelegate` 协议可以被任意类型遵循，用来追踪 `DiceGame` 的游戏过程。为了防止强引用导致的循环引用问题，可以把协议声明为弱引用，更多相关的知识请看 [类实例之间的循环强引用](#)，当协议标记为类专属可以使 `SnakesAndLadders` 类在声明协议时强制要使用弱引用。若要声明类专属的协议就必须继承于 `AnyObject`，更多请看 [类专属的协议](#)。

如下所示，`SnakesAndLadders` 是控制流章节引入的蛇梯棋游戏的新版本。新版本使用 `Dice` 实例作为骰子，并且实现了 `DiceGame` 和 `DiceGameDelegate` 协议，后者用来记录游戏的过程：

```
class SnakesAndLadders: DiceGame {
    let finalSquare = 25
    let dice = Dice(sides: 6, generator: LinearCongruentialGenerator())
    var square = 0
    var board: [Int]
    init() {
        board = Array(repeating: 0, count: finalSquare + 1)
        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)
    }
}
```

关于这个蛇梯棋游戏的详细描述请参阅 [中断 \(Break\)](#)。

这个版本的游戏封装到了 `SnakesAndLadders` 类中，该类遵循了 `DiceGame` 协议，并且提供了相应的可读的 `dice` 属性和 `play()` 方法。（`dice` 属性在构造之后就不再改变，且协议只要求 `dice` 为可读的，因此将 `dice` 声明为常量属性。）

游戏使用 `SnakesAndLadders` 类的 `init()` 构造器来初始化游戏。所有的游戏逻辑被转移到了协议中的 `play()` 方法，`play()` 方法使用协议要求的 `dice` 属性提供骰子摇出的值。

注意，`delegate` 并不是游戏的必备条件，因此 `delegate` 被定义为 `DiceGameDelegate` 类型的可选属性。因为 `delegate` 是可选值，因此会被自动赋予初始值 `nil`。随后，可以在游戏中为 `delegate` 设置适当的值。

`DiceGameDelegate` 协议提供了三个方法用来追踪游戏过程。这三个方法被放置于游戏的逻辑中，即 `play()` 方法内。分别在游戏开始时，新一轮开始时，以及游戏结束时被调用。

因为 `delegate` 是一个 `DiceGameDelegate` 类型的可选属性，因此在 `play()` 方法中通过可选链式调用来调用它的方法。若 `delegate` 属性为 `nil`，则调用方法会优雅地失败，并不会产生错误。若 `delegate` 不为 `nil`，则方法能够被调用，并传递 `SnakesAndLadders` 实例作为参数。

如下示例定义了 `DiceGameTracker` 类，它遵循了 `DiceGameDelegate` 协议：

```
class DiceGameTracker: DiceGameDelegate {
    var numberOfTurns = 0
    func gameDidStart(_ game: DiceGame) {
        numberOfTurns = 0
        if game is SnakesAndLadders {
            print("Started a new game of Snakes and Ladders")
        }
        print("The game is using a \(game.dice.sides)-sided dice")
    }
    func game(_ game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int) {
        numberOfTurns += 1
        print("Rolled a \(diceRoll)")
    }
    func gameDidEnd(_ game: DiceGame) {
        print("The game lasted for \(numberOfTurns) turns")
    }
}
```

`DiceGameTracker` 实现了 `DiceGameDelegate` 协议要求的三个方法，用来记录游戏已经进行的轮数。当游戏开始时，`numberOfTurns` 属性被赋值为 `0`，然后在每新一轮中递增，游戏结束后，打印游戏的总轮数。

`gameDidStart(_)` 方法从 `game` 参数获取游戏信息并打印。`game` 参数是 `DiceGame` 类型而不是 `SnakeAndLadders` 类型，所以在 `gameDidStart(_)` 方法中只能访问 `DiceGame` 协议中的内容。当然了，`SnakeAndLadders` 的方法也可以在类型转换之后调用。在上例代码中，通过 `is` 操作符检查 `game` 是否为 `SnakesAndLadders` 类型的实例，如果是，则打印出相应的消息。

无论当前进行的是何种游戏，由于 `game` 符合 `DiceGame` 协议，可以确保 `game` 含有 `dice` 属性。因此在 `gameDidStart(_)` 方法中可以通过传入的 `game` 参数来访问 `dice` 属性，进而打印出 `dice` 的 `sides` 属性的值。

`DiceGameTracker` 的运行情况如下所示：

```
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
```

## 在扩展里添加协议遵循

即便无法修改源代码，依然可以通过扩展令已有类型遵循并符合协议。扩展可以为已有类型添加属性、方法、下标以及构造器，因此可以符合协议中的相应要求。详情请在[扩展](#)章节中查看。

注意

通过扩展令已有类型遵循并符合协议时，该类型的所有实例也会随之获得协议中定义的各项功能。

例如下面这个 `TextRepresentable` 协议，任何想要通过文本表示一些内容的类型都可以实现该协议。这些想要表示的内容可以是实例本身的描述，也可以是实例当前状态的文本描述：

```
protocol TextRepresentable {  
    var textualDescription: String { get }  
}
```

可以通过扩展，令先前提到的 `Dice` 类可以扩展来采纳和遵循 `TextRepresentable` 协议：

```
extension Dice: TextRepresentable {  
    var textualDescription: String {  
        return "A \(sides)-sided dice"  
    }  
}
```

通过扩展遵循并采纳协议，和在原始定义中遵循并符合协议的效果完全相同。协议名称写在类型名之后，以冒号隔开，然后在扩展的大括号内实现协议要求的内容。

现在所有 `Dice` 的实例都可以看做 `TextRepresentable` 类型：

```
let d12 = Dice(sides: 12, generator: LinearCongruentialGenerator())  
print(d12.textualDescription)  
// 打印 "A 12-sided dice"
```

同样，`SnakesAndLadders` 类也可以通过扩展来采纳和遵循 `TextRepresentable` 协议：

```
extension SnakesAndLadders: TextRepresentable {  
    var textualDescription: String {  
        return "A game of Snakes and Ladders with \(finalSquare) squares"  
    }  
}  
print(game.textualDescription)  
// 打印 “A game of Snakes and Ladders with 25 squares”
```

## 有条件地遵循协议

---

泛型类型可能只在某些情况下满足一个协议的要求，比如当类型的泛型形式参数遵循对应协议时。你可以通过在扩展类型时列出限制让泛型类型有条件地遵循某协议。在你采纳协议的名字后面写泛型 `where` 分句。更多关于泛型 `where` 分句，见 [泛型 Where 分句](#)。

下面的扩展让 `Array` 类型只要在存储遵循 `TextRepresentable` 协议的元素时就遵循 `TextRepresentable` 协议。

```
extension Array: TextRepresentable where Element: TextRepresentable {  
    var textualDescription: String {  
        let itemsAsText = self.map { $0.textualDescription }  
        return "[" + itemsAsText.joined(separator: ", ") + "]"  
    }  
}  
let myDice = [d6, d12]  
print(myDice.textualDescription)  
// 打印 “[A 6-sided dice, A 12-sided dice]”
```

## 在扩展里声明采纳协议

---

当一个类型已经符合了某个协议中的所有要求，却还没有声明采纳该协议时，可以通过空的扩展来让它采纳该协议：

```
struct Hamster {  
    var name: String  
    var textualDescription: String {  
        return "A hamster named \(name)"  
    }  
}  
extension Hamster: TextRepresentable {}
```

从现在起，`Hamster` 的实例可以作为 `TextRepresentable` 类型使用：

```
let simonTheHamster = Hamster(name: "Simon")  
let somethingTextRepresentable: TextRepresentable = simonTheHamster  
print(somethingTextRepresentable.textualDescription)  
// 打印 “A hamster named Simon”
```

注意

即使满足了协议的所有要求，类型也不会自动遵循协议，必须显式地遵循协议。

## 协议类型的集合

---

协议类型可以在数组或者字典这样的集合中使用，在 [协议类型](#) 提到了这样的用法。下面的例子创建了一个元素类型为 `TextRepresentable` 的数组：

```
let things: [TextRepresentable] = [game, d12, simonTheHamster]
```

如下所示，可以遍历 `things` 数组，并打印每个元素的文本表示：

```
for thing in things {  
    print(thing.textualDescription)  
}  
// A game of Snakes and Ladders with 25 squares  
// A 12-sided dice  
// A hamster named Simon
```

注意 `thing` 常量是 `TextRepresentable` 类型而不是 `Dice`，`DiceGame`，`Hamster` 等类型，即使实例在幕后确实是这些类型中的一种。由于 `thing` 是 `TextRepresentable` 类型，任何 `TextRepresentable` 的实例都有一个 `textualDescription` 属性，所以在每次循环中可以安全地访问 `thing.textualDescription`。

## 协议的继承

---

协议能够继承一个或多个其他协议，可以在继承的协议的基础上增加新的要求。协议的继承语法与类的继承相似，多个被继承的协议间用逗号分隔：

```
protocol InheritingProtocol: SomeProtocol, AnotherProtocol {  
    // 这里是协议的定义部分  
}
```

如下所示，`PrettyTextRepresentable` 协议继承了 `TextRepresentable` 协议：

```
protocol PrettyTextRepresentable: TextRepresentable {  
    var prettyTextualDescription: String { get }  
}
```

例子中定义了一个新的协议 `PrettyTextRepresentable`，它继承自 `TextRepresentable` 协议。任何遵循 `PrettyTextRepresentable` 协议的类型在满足该协议的要求时，也必须满足 `TextRepresentable` 协议的要求。在这个例子中，`PrettyTextRepresentable` 协议额外要求遵循协议的类型提供一个返回值为 `String` 类型的 `prettyTextualDescription` 属性。

如下所示，扩展 `SnakesAndLadders`，使其遵循并符合 `PrettyTextRepresentable` 协议：

```

extension SnakesAndLadders: PrettyTextRepresentable {
    var prettyTextualDescription: String {
        var output = textualDescription + ":\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
    }
}

```

上述扩展令 `SnakesAndLadders` 遵循了 `PrettyTextRepresentable` 协议，并提供了协议要求的 `prettyTextualDescription` 属性。每个 `PrettyTextRepresentable` 类型同时也是 `TextRepresentable` 类型，所以在 `prettyTextualDescription` 的实现中，可以访问 `textualDescription` 属性。然后，拼接上了冒号和换行符。接着，遍历数组中的元素，拼接一个几何图形来表示每个棋盘方格的内容：

- 当从数组中取出的元素的值大于 `0` 时，用 `▲` 表示。
- 当从数组中取出的元素的值小于 `0` 时，用 `▼` 表示。
- 当从数组中取出的元素的值等于 `0` 时，用 `○` 表示。

任意 `SnakesAndLadders` 的实例都可以使用 `prettyTextualDescription` 属性来打印一个漂亮的文本描述：

```

print(game.prettyTextualDescription)
// A game of Snakes and Ladders with 25 squares:
// ○○▲○○▲○○▲▲○○○▼○○○○▼○○○▼○▼○

```

## 类专属的协议

你通过添加  `AnyObject` 关键字到协议的继承列表，就可以限制协议只能被类类型采纳（以及非结构体或者非枚举的类型）。

```

protocol SomeClassOnlyProtocol: AnyObject, SomeInheritedProtocol {
    // 这里是类专属协议的定义部分
}

```

在以上例子中，协议 `SomeClassOnlyProtocol` 只能被类类型采纳。如果尝试让结构体或枚举类型采纳 `SomeClassOnlyProtocol`，则会导致编译时错误。

### 注意

当协议定义的要求需要遵循协议的类型必须是引用语义而非值语义时，应该采用类类型专属协议。关于引用语义和值语义的更多内容，请查看 [结构体和枚举是值类型](#) 和 [类是引用类型](#)。

## 协议合成

---

要求一个类型同时遵循多个协议是很有用的。你可以使用协议组合来复合多个协议到一个要求里。协议组合行为就和你定义的临时局部协议一样拥有构成中所有协议的需求。协议组合不定义任何新的协议类型。

协议组合使用 `SomeProtocol & AnotherProtocol` 的形式。你可以列举任意数量的协议，用和符号（`&`）分开。除了协议列表，协议组合也能包含类类型，这允许你标明一个需要的父类。

下面的例子中，将 `Named` 和 `Aged` 两个协议按照上述语法组合成一个协议，作为函数参数的类型：

```
protocol Named {  
    var name: String { get }  
}  
protocol Aged {  
    var age: Int { get }  
}  
struct Person: Named, Aged {  
    var name: String  
    var age: Int  
}  
func wishHappyBirthday(to celebrator: Named & Aged) {  
    print("Happy birthday, \(celebrator.name), you're \(celebrator.age)!")  
}  
let birthdayPerson = Person(name: "Malcolm", age: 21)  
wishHappyBirthday(to: birthdayPerson)  
// 打印“Happy birthday Malcolm - you're 21!”
```

`Named` 协议包含 `String` 类型的 `name` 属性。`Aged` 协议包含 `Int` 类型的 `age` 属性。`Person` 结构体采纳了这两个协议。

`wishHappyBirthday(to:)` 函数的参数 `celebrator` 的类型为 `Named & Aged`，这意味着“任何同时遵循 `Named` 和 `Aged` 的协议”。它不关心参数的具体类型，只要参数符合这两个协议即可。

上面的例子创建了一个名为 `birthdayPerson` 的 `Person` 的实例，作为参数传递给了 `wishHappyBirthday(to:)` 函数。因为 `Person` 同时符合这两个协议，所以这个参数合法，函数将打印生日问候语。

这里有一个例子：将 `Location` 类和前面的 `Named` 协议进行组合：

```

class Location {
    var latitude: Double
    var longitude: Double
    init(latitude: Double, longitude: Double) {
        self.latitude = latitude
        self.longitude = longitude
    }
}
class City: Location, Named {
    var name: String
    init(name: String, latitude: Double, longitude: Double) {
        self.name = name
        super.init(latitude: latitude, longitude: longitude)
    }
}
func beginConcert(in location: Location & Named) {
    print("Hello, \(location.name)!")
}

let seattle = City(name: "Seattle", latitude: 47.6, longitude: -122.3)
beginConcert(in: seattle)
// 打印 "Hello, Seattle!"

```

`beginConcert(in:)` 函数接受一个类型为 `Location & Named` 的参数，这意味着“任何 `Location` 的子类，并且遵循 `Named` 协议”。例如，`City` 就满足这样的条件。

将 `birthdayPerson` 传入 `beginConcert(in:)` 函数是不合法的，因为 `Person` 不是 `Location` 的子类。同理，如果你新建一个类继承于 `Location`，但是没有遵循 `Named` 协议，而用这个类的实例去调用 `beginConcert(in:)` 函数也是非法的。

## 检查协议一致性

---

你可以使用 [类型转换](#) 中描述的 `is` 和 `as` 操作符来检查协议一致性，即是否符合某协议，并且可以转换到指定的协议类型。检查和转换协议的语法与检查和转换类型是完全一样的：

- `is` 用来检查实例是否符合某个协议，若符合则返回 `true`，否则返回 `false`；
- `as?` 返回一个可选值，当实例符合某个协议时，返回类型为协议类型的可选值，否则返回 `nil`；
- `as!` 将实例强制向下转换到某个协议类型，如果强转失败，将触发运行时错误。

下面的例子定义了一个 `HasArea` 协议，该协议定义了一个 `Double` 类型的可读属性 `area`：

```

protocol HasArea {
    var area: Double { get }
}

```

如下所示，`Circle` 类和 `Country` 类都遵循了 `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 }
}

```

`Circle` 类把 `area` 属性实现为基于存储型属性 `radius` 的计算型属性。`Country` 类则把 `area` 属性实现为存储型属性。这两个类都正确地遵循了 `HasArea` 协议。

如下所示，`Animal` 是一个未遵循 `HasArea` 协议的类：

```

class Animal {
    var legs: Int
    init(legs: Int) { self.legs = legs }
}

```

`Circle`，`Country`，`Animal` 并没有一个共同的基类，尽管如此，它们都是类，它们的实例都可以作为 `AnyObject` 类型的值，存储在同一个数组中：

```

let objects: [AnyObject] = [
    Circle(radius: 2.0),
    Country(area: 243_610),
    Animal(legs: 4)
]

```

`objects` 数组使用字面量初始化，数组包含一个 `radius` 为 2 的 `Circle` 的实例，一个保存了英国国土面积的 `Country` 实例和一个 `legs` 为 4 的 `Animal` 实例。

如下所示，`objects` 数组可以被迭代，并对迭代出的每一个元素进行检查，看它是否符合 `HasArea` 协议：

```

for object in objects {
    if let objectWithArea = object as? HasArea {
        print("Area is \(objectWithArea.area)")
    } else {
        print("Something that doesn't have an area")
    }
}
// Area is 12.5663708
// Area is 243610.0
// Something that doesn't have an area

```

当迭代出的元素符合 `HasArea` 协议时，将 `as?` 操作符返回的可选值通过可选绑定，绑定到 `objectWithArea` 常量上。`objectWithArea` 是 `HasArea` 协议类型的实例，因此 `area` 属性可以被访问和打印。

`objects` 数组中的元素的类型并不会因为强转而丢失类型信息，它们仍然是 `Circle` , `Country` , `Animal` 类型。然而，当它们被赋值给 `objectWithArea` 常量时，只被视为 `HasArea` 类型，因此只有 `area` 属性能够被访问。

## 可选的协议要求

协议可以定义 可选要求，遵循协议的类型可以选择是否实现这些要求。在协议中使用 `optional` 关键字作为前缀来定义可选要求。可选要求用在你需要和 Objective-C 打交道的代码中。协议和可选要求都必须带上 `@objc` 属性。标记 `@objc` 特性的协议只能被继承自 Objective-C 类的类或者 `@objc` 类遵循，其他类以及结构体和枚举均不能遵循这种协议。

使用可选要求时（例如，可选的方法或者属性），它们的类型会自动变成可选的。比如，一个类型为 `(Int) -> String` 的方法会变成 `((Int) -> String)?`。需要注意的是整个函数类型是可选的，而不是函数的返回值。

协议中的可选要求可通过可选链式调用来使用，因为遵循协议的类型可能没有实现这些可选要求。类似 `someOptionalMethod?(someArgument)` 这样，你可以在可选方法名称后加上 `?` 来调用可选方法。详细内容可在 [可选链式调用](#) 章节中查看。

下面的例子定义了一个名为 `Counter` 的用于整数计数的类，它使用外部的数据源来提供每次的增量。数据源由 `CounterDataSource` 协议定义，它包含两个可选要求：

```
@objc protocol CounterDataSource {
    @objc optional func increment(forCount count: Int) -> Int
    @objc optional var fixedIncrement: Int { get }
}
```

`CounterDataSource` 协议定义了一个可选方法 `increment(forCount:)` 和一个可选属性 `fixedIncrement`，它们使用了不同的方法来从数据源中获取适当的增量值。

注意

严格来讲，`CounterDataSource` 协议中的方法和属性都是可选的，因此遵循协议的类可以不实现这些要求，尽管技术上允许这样做，不过最好不要这样写。

`Counter` 类含有 `CounterDataSource?` 类型的可选属性 `dataSource`，如下所示：

```
class Counter {
    var count = 0
    var dataSource: CounterDataSource?
    func increment() {
        if let amount = dataSource?.increment?(forCount: count) {
            count += amount
        } else if let amount = dataSource?.fixedIncrement {
            count += amount
        }
    }
}
```

`Counter` 类使用变量属性 `count` 来存储当前值。该类还定义了一个 `increment` 方法，每次调用该方法的时候，将会增加 `count` 的值。

`increment()` 方法首先试图使用 `increment(forCount:)` 方法来得到每次的增量。`increment()` 方法使用可选链式调用来尝试调用 `increment(forCount:)`，并将当前的 `count` 值作为参数传入。

这里使用了两层可选链式调用。首先，由于 `dataSource` 可能为 `nil`，因此在 `dataSource` 后边加上了 `?`，以此表明只在 `dataSource` 非空时才去调用 `increment(forCount:)` 方法。其次，即使 `dataSource` 存在，也无法保证其是否实现了 `increment(forCount:)` 方法，因为这个方法是可选的。因此，`increment(forCount:)` 方法同样使用可选链式调用进行调用，只有在该方法被实现的情况下才能调用它，所以在 `increment(forCount:)` 方法后边也加上了 `?`。

调用 `increment(forCount:)` 方法在上述两种情形下都有可能失败，所以返回值为 `Int?` 类型。虽然在 `CounterDataSource` 协议中，`increment(forCount:)` 的返回值类型是非可选 `Int`。另外，即使这里使用了两层可选链式调用，最后的返回结果依旧是单层的可选类型。关于这一点的更多信息，请查阅 [连接多层可选链式调用](#)。

在调用 `increment(forCount:)` 方法后，`Int?` 型的返回值通过可选绑定解包并赋值给常量 `amount`。如果可选值确实包含一个数值，也就是说，数据源和方法都存在，数据源方法返回了一个有效值。之后便将解包后的 `amount` 加到 `count` 上，增量操作完成。

如果没有从 `increment(forCount:)` 方法获取到值，可能由于 `dataSource` 为 `nil`，或者它并没有实现 `increment(forCount:)` 方法，那么 `increment()` 方法将试图从数据源的 `fixedIncrement` 属性中获取增量。`fixedIncrement` 是一个可选属性，因此属性值是一个 `Int?` 值，即使该属性在 `CounterDataSource` 协议中的类型是非可选的 `Int`。

下面的例子展示了 `CounterDataSource` 的简单实现。`ThreeSource` 类遵循了 `CounterDataSource` 协议，它实现了可选属性 `fixedIncrement`，每次会返回 `3`：

```
class ThreeSource: NSObject, CounterDataSource {  
    let fixedIncrement = 3  
}
```

可以使用 `ThreeSource` 的实例作为 `Counter` 实例的数据源：

```
var counter = Counter()  
counter.dataSource = ThreeSource()  
for _ in 1...4 {  
    counter.increment()  
    print(counter.count)  
}  
// 3  
// 6  
// 9  
// 12
```

上述代码新建了一个 `Counter` 实例，并将它的数据源设置为一个 `ThreeSource` 的实例，然后调用 `increment()` 方法 4 次。按照预期一样，每次调用都会将 `count` 的值增加 3。

下面是一个更为复杂的数据源 `TowardsZeroSource`，它将使得最后的值变为 0：

```
class TowardsZeroSource: NSObject, CounterDataSource {  
    func increment(forCount count: Int) -> Int {  
        if count == 0 {  
            return 0  
        } else if count < 0 {  
            return 1  
        } else {  
            return -1  
        }  
    }  
}
```

`TowardsZeroSource` 实现了 `CounterDataSource` 协议中的 `increment(forCount:)` 方法，以 `count` 参数为依据，计算出每次的增量。如果 `count` 已经为 0，此方法将返回 0，以此表明之后不应再有增量操作发生。

你可以使用 `TowardsZeroSource` 实例将 `Counter` 实例来从 -4 增加到 0。一旦增加到 0，数值便不会再有变动：

```
counter.count = -4  
counter.dataSource = TowardsZeroSource()  
for _ in 1...5 {  
    counter.increment()  
    print(counter.count)  
}  
// -3  
// -2  
// -1  
// 0  
// 0
```

## 协议扩展

---

协议可以通过扩展来为遵循协议的类型提供属性、方法以及下标的实现。通过这种方式，你可以基于协议本身来实现这些功能，而无需在每个遵循协议的类型中都重复同样的实现，也无需使用全局函数。

例如，可以扩展 `RandomNumberGenerator` 协议来提供 `randomBool()` 方法。该方法使用协议中定义的 `random()` 方法来返回一个随机的 `Bool` 值：

```
extension RandomNumberGenerator {  
    func randomBool() -> Bool {  
        return random() > 0.5  
    }  
}
```

通过协议扩展，所有遵循协议的类型，都能自动获得这个扩展所增加的方法实现而无需任何额外修改：

```
let generator = LinearCongruentialGenerator()
print("Here's a random number: \(generator.random())")
// 打印 "Here's a random number: 0.37464991998171"
print("And here's a random Boolean: \(generator.randomBool())")
// 打印 "And here's a random Boolean: true"
```

## 提供默认实现

可以通过协议扩展来为协议要求的属性、方法以及下标提供默认的实现。如果遵循协议的类型为这些要求提供了自己的实现，那么这些自定义实现将会替代扩展中的默认实现被使用。

注意

通过协议扩展为协议要求提供的默认实现和可选的协议要求不同。虽然在这两种情况下，遵循协议的类型都无需自己实现这些要求，但是通过扩展提供的默认实现可以直接调用，而无需使用可选链式调用。

例如，`PrettyTextRepresentable` 协议继承自 `TextRepresentable` 协议，可以为其提供一个默认的 `prettyTextualDescription` 属性来简单地返回 `textualDescription` 属性的值：

```
extension PrettyTextRepresentable {
    var prettyTextualDescription: String {
        return textualDescription
    }
}
```

## 为协议扩展添加限制条件

在扩展协议的时候，可以指定一些限制条件，只有遵循协议的类型满足这些限制条件时，才能获得协议扩展提供的默认实现。这些限制条件写在协议名之后，使用 `where` 子句来描述，正如 泛型 Where 子句 中所描述的。

例如，你可以扩展 `Collection` 协议，适用于集合中的元素遵循了 `Equatable` 协议的情况。通过限制集合元素遵 `Equatable` 协议，作为标准库的一部分，你可以使用 `==` 和 `!=` 操作符来检查两个元素的等价性和非等价性。

```
extension Collection where Element: Equatable {
    func allEqual() -> Bool {
        for element in self {
            if element != self.first {
                return false
            }
        }
        return true
    }
}
```

如果集合中的所有元素都一致，`allEqual()` 方法才返回 `true`。

看看两个整数数组，一个数组的所有元素都是一样的，另一个不一样：

```
let equalNumbers = [100, 100, 100, 100, 100]
let differentNumbers = [100, 100, 200, 100, 200]
```

由于数组遵循 `Collection` 而且整数遵循 `Equatable`，`equalNumbers` 和 `differentNumbers` 都可以使用 `allEqual()` 方法。

```
print(equalNumbers.allEqual())
// 打印 "true"
print(differentNumbers.allEqual())
// 打印 "false"
```

注意

如果一个遵循的类型满足了为同一方法或属性提供实现的多个限制型扩展的要求，Swift 会使用最匹配限制的实现。

# 泛型 · GitBook

 [runoob.com/manual/gitbook/swift5/source/\\_book/chapter2/22\\_Generics.html](https://runoob.com/manual/gitbook/swift5/source/_book/chapter2/22_Generics.html)

## 泛型

泛型代码让你能根据自定义的需求，编写出适用于任意类型的、灵活可复用的函数及类型。你可避免编写重复的代码，而是用一种清晰抽象的方式来表达代码的意图。

泛型是 Swift 最强大的特性之一，很多 Swift 标准库是基于泛型代码构建的。实际上，即使你没有意识到，你也一直在语言指南中使用泛型。例如，Swift 的 `Array` 和 `Dictionary` 都是泛型集合。你可以创建一个 `Int` 类型数组，也可创建一个 `String` 类型数组，甚至可以是任意其他 Swift 类型的数组。同样，你也可以创建一个存储任意指定类型的字典，并对该类型没有限制。

## 泛型解决的问题

下面是一个标准的非泛型函数 `swapTwoInts(_:_:)`，用来交换两个 `Int` 值：

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {  
    let temporaryA = a  
    a = b  
    b = temporaryA  
}
```

这个函数使用输入输出参数（`inout`）来交换 `a` 和 `b` 的值，具体请参考 [输入输出参数](#)。

`swapTwoInts(_:_:)` 函数将 `b` 的原始值换成了 `a`，将 `a` 的原始值换成了 `b`，你可以调用这个函数来交换两个 `Int` 类型变量：

```
var someInt = 3  
var anotherInt = 107  
swapTwoInts(&someInt, &anotherInt)  
print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")  
// 打印“someInt is now 107, and anotherInt is now 3”
```

`swapTwoInts(_:_:)` 函数很实用，但它只能作用于 `Int` 类型。如果你想交换两个 `String` 类型值，或者 `Double` 类型值，你必须编写对应的函数，类似下面 `swapTwoStrings(_:_:)` 和 `swapTwoDoubles(_:_:)` 函数：

```

func swapTwoStrings(_ a: inout String, _ b: inout String) {
    let temporaryA = a
    a = b
    b = temporaryA
}

func swapTwoDoubles(_ a: inout Double, _ b: inout Double) {
    let temporaryA = a
    a = b
    b = temporaryA
}

```

你可能注意到了，`swapTwoInts(_:_:)`、`swapTwoStrings(_:_:)` 和 `swapTwoDoubles(_:_:)` 函数体是一样的，唯一的区别是它们接受的参数类型（`Int`、`String` 和 `Double`）。

在实际应用中，通常需要一个更实用更灵活的函数来交换两个任意类型的值，幸运的是，泛型代码帮你解决了这种问题。（这些函数的泛型版本已经在下面定义好了。）

### 注意

在上面三个函数中，`a` 和 `b` 类型必须相同。如果 `a` 和 `b` 类型不同，那它们俩就不能互换值。Swift 是类型安全的语言，所以它不允许一个 `String` 类型的变量和一个 `Double` 类型的变量互换值。试图这样做将导致编译错误。

## 泛型函数

泛型函数可适用于任意类型，下面是函数 `swapTwoInts(_:_:)` 的泛型版本，命名为 `swapTwoValues(_:_:)`：

```

func swapTwoValues<T>(_ a: inout T, _ b: inout T) {
    let temporaryA = a
    a = b
    b = temporaryA
}

```

`swapTwoValues(_:_:)` 和 `swapTwoInts(_:_:)` 函数体内容相同，它们只在第一行不同，如下所示：

```

func swapTwoInts(_ a: inout Int, _ b: inout Int)
func swapTwoValues<T>(_ a: inout T, _ b: inout T)

```

泛型版本的函数使用 **占位符** 类型名（这里叫做 `T`），而不是 **实际类型名**（例如 `Int`、`String` 或 `Double`），**占位符** 类型名并不关心 `T` 具体的类型，但它要求 `a` 和 `b` 必须是相同的类型，`T` 的实际类型由每次调用 `swapTwoValues(_:_:)` 来决定。

泛型函数和非泛型函数的另外一个不同之处在于这个泛型函数名

（`swapTwoValues(_:_:)`）后面跟着占位类型名（`T`），并用尖括号括起来（`<T>`）。这个尖括号告诉 Swift 那个 `T` 是 `swapTwoValues(_:_:)` 函数定义内的一个占位类型名，因此 Swift 不会去查找名为 `T` 的实际类型。

`swapTwoValues(_:_)` 函数现在可以像 `swapTwoInts(_:_)` 那样调用，不同的是它能接受两个任意类型的值，条件是这两个值有着相同的类型。`swapTwoValues(_:_)` 函数被调用时，`T` 所代表的类型都会由传入的值的类型推断出来。

在下面的两个例子中，`T` 分别代表 `Int` 和 `String`：

```
var someInt = 3
var anotherInt = 107
swapTwoValues(&someInt, &anotherInt)
// someInt 现在是 107, anotherInt 现在是 3
```

```
var someString = "hello"
var anotherString = "world"
swapTwoValues(&someString, &anotherString)
// someString 现在是"world", anotherString 现在是"hello"
```

### 注意

上面定义的 `swapTwoValues(_:_)` 函数是受 `swap(_:_)` 函数启发而实现的。后者存在于 Swift 标准库，你可以在你的应用程序中使用它。如果你在代码中需要类似 `swapTwoValues(_:_)` 函数的功能，你可以使用已存在的 `swap(_:_)` 函数。

## 类型参数

上面 `swapTwoValues(_:_)` 例子中，占位类型 `T` 是一个类型参数的例子，类型参数指定并命名一个占位类型，并且紧随在函数名后面，使用一对尖括号括起来（例如 `<T>`）。

一旦一个类型参数被指定，你可以用它来定义一个函数的参数类型（例如 `swapTwoValues(_:_)` 函数中的参数 `a` 和 `b`），或者作为函数的返回类型，还可以用作函数主体中的注释类型。在这些情况下，类型参数会在函数调用时被实际类型所替换。（在上面的 `swapTwoValues(_:_)` 例子中，当函数第一次被调用时，`T` 被 `Int` 替换，第二次调用时，被 `String` 替换。）

你可提供多个类型参数，将它们都写在尖括号中，用逗号分开。

## 命名类型参数

大多情况下，类型参数具有描述性的名称，例如字典 `Dictionary<Key, Value>` 中的 `Key` 和 `Value` 及数组 `Array<Element>` 中的 `Element`，这能告诉阅读代码的人这些参数类型与泛型类型或函数之间的关系。然而，当它们之间没有有意义的关系时，通常使用单个字符来表示，例如 `T`、`U`、`V`，例如上面演示函数 `swapTwoValues(_:_)` 中的 `T`。

### 注意

请始终使用大写字母开头的驼峰命名法（例如 `T` 和 `MyTypeParameter`）来为类型参数命名，以表明它们是占位类型，而不是一个值。

## 泛型类型

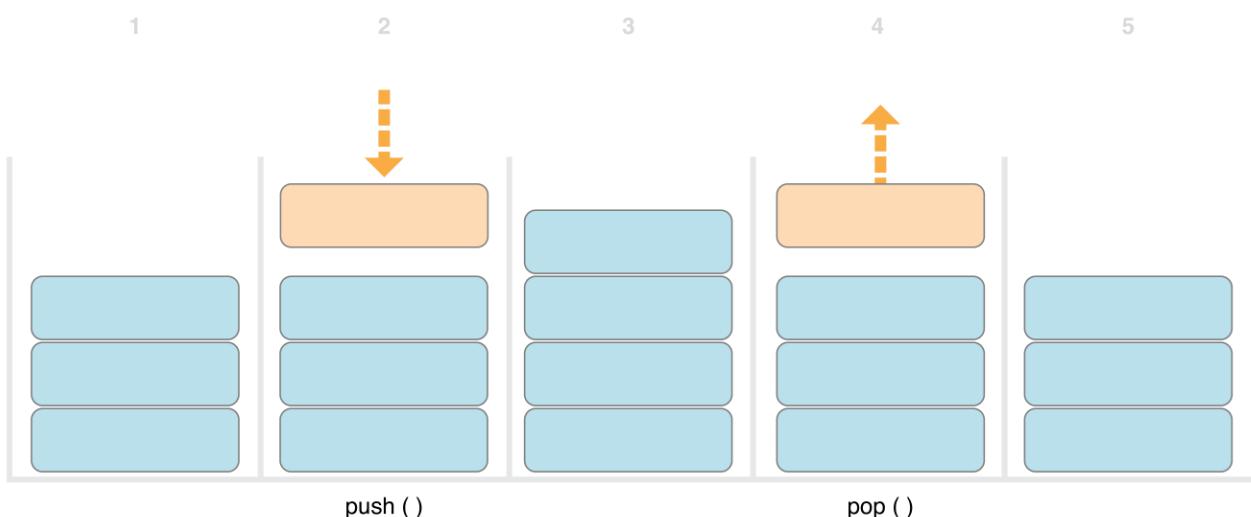
除了泛型函数，Swift 还允许自定义泛型类型。这些自定义类、结构体和枚举可以适用于任意类型，类似于 `Array` 和 `Dictionary`。

本节将向你展示如何编写一个名为 `Stack`（栈）的泛型集合类型。栈是值的有序集合，和数组类似，但比数组有更严格的操作限制。数组允许在其中任意位置插入或是删除元素。而栈只允许在集合的末端添加新的元素（称之为入栈）。类似的，栈也只能从末端移除元素（称之为出栈）。

### 注意

栈的概念已被 `UINavigationController` 类用来构造视图控制器的导航结构。你通过调用 `UINavigationController` 的 `pushViewController(_:animated:)` 方法来添加新的视图控制器到导航栈，通过 `popViewControllerAnimated(_:)` 方法来从导航栈中移除视图控制器。每当你需要一个严格的“后进先出”方式来管理集合，栈都是最实用的模型。

下图展示了入栈（push）和出栈（pop）的行为：



1. 现在有三个值在栈中。
2. 第四个值被压入到栈的顶部。
3. 现在栈中有四个值，最近入栈的那个值在顶部。
4. 栈中最顶部的那个值被移除出栈。
5. 一个值移除出栈后，现在栈又只有三个值了。

下面展示如何编写一个非泛型版本的栈，以 `Int` 型的栈为例：

```
struct IntStack {  
    var items = [Int]()  
    mutating func push(_ item: Int) {  
        items.append(item)  
    }  
    mutating func pop() -> Int {  
        return items.removeLast()  
    }  
}
```

这个结构体在栈中使用一个名为 `items` 的数组属性来存储值。栈提供了两个方法：`push(_)` 和 `pop()`，用来向栈中压入值以及从栈中移除值。这些方法被标记为 `mutating`，因为它们需要修改结构体的 `items` 数组。

上面的 `IntStack` 结构体只能用于 `Int` 类型。不过，可以定义一个泛型 `Stack` 结构体，从而能够处理任意类型的值。

下面是相同代码的泛型版本：

```
struct Stack<Element> {  
    var items = [Element]()  
    mutating func push(_ item: Element) {  
        items.append(item)  
    }  
    mutating func pop() -> Element {  
        return items.removeLast()  
    }  
}
```

注意，`Stack` 基本上和 `IntStack` 相同，只是用占位类型参数 `Element` 代替了实际的 `Int` 类型。这个类型参数包裹在紧随结构体名的一对尖括号里 (`< Element >`)。

`Element` 为待提供的类型定义了一个占位名。这种待提供的类型可以在结构体的定义中通过 `Element` 来引用。在这个例子中，`Element` 在如下三个地方被用作占位符：

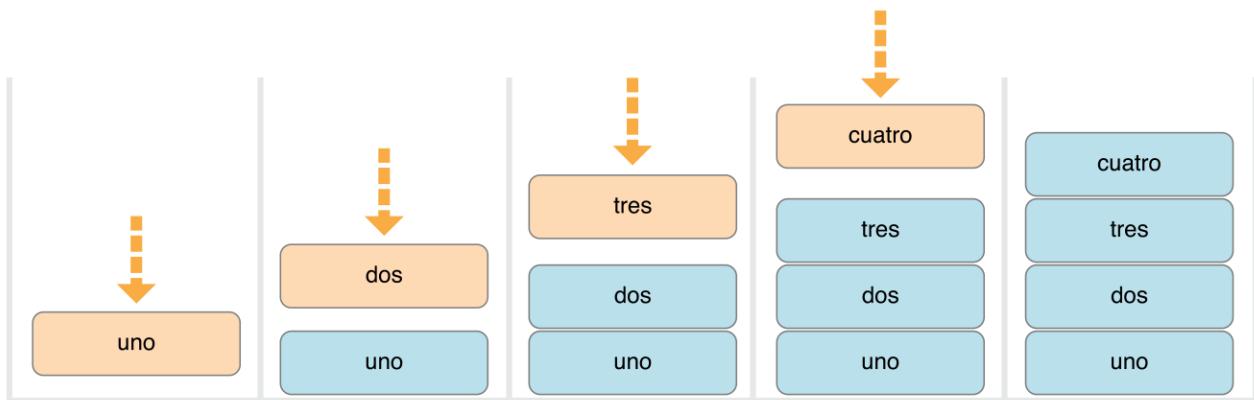
- 创建 `items` 属性，使用 `Element` 类型的空数组对其进行初始化。
- 指定 `push(_)` 方法的唯一参数 `item` 的类型必须是 `Element` 类型。
- 指定 `pop()` 方法的返回值类型必须是 `Element` 类型。

由于 `Stack` 是泛型类型，因此可以用来创建适用于 Swift 中任意有效类型的栈，就像 `Array` 和 `Dictionary` 那样。

你可以通过在尖括号中写出栈中需要存储的数据类型来创建并初始化一个 `Stack` 实例。例如，要创建一个 `String` 类型的栈，可以写成 `Stack<String>()`：

```
var stackOfStrings = Stack<String>()  
stackOfStrings.push("uno")  
stackOfStrings.push("dos")  
stackOfStrings.push("tres")  
stackOfStrings.push("cuatro")  
// 栈中现在有 4 个字符串
```

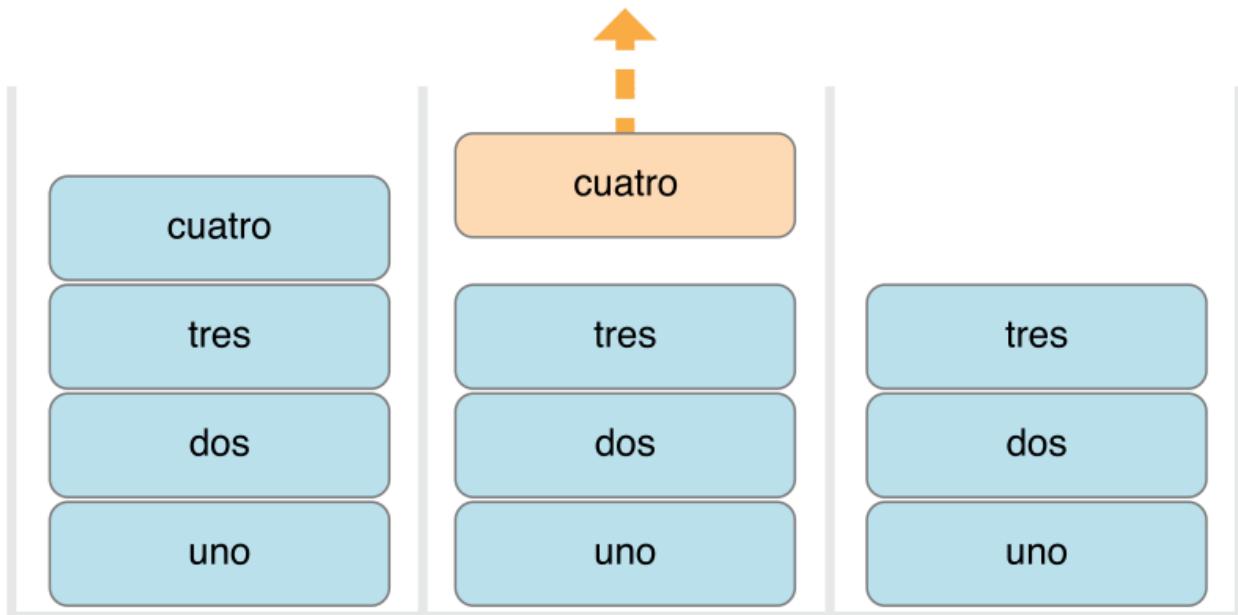
下图展示了 `stackOfStrings` 如何将这四个值压栈：



移除并返回栈顶部的值“cuatro”，即出栈：

```
let fromTheTop = stackOfStrings.pop()  
// fromTheTop 的值为“cuatro”，现在栈中还有 3 个字符串
```

下图展示了如何将顶部的值出栈：



## 泛型扩展

当对泛型类型进行扩展时，你并不需要提供类型参数列表作为定义的一部分。原始类型定义中声明的类型参数列表在扩展中可以直接使用，并且这些来自原始类型中的参数名称会被用作原始定义中类型参数的引用。

下面的例子扩展了泛型类型 `Stack`，为其添加了一个名为 `topItem` 的只读计算型属性，它将会返回当前栈顶元素且不会将其从栈中移除：

```
extension Stack {  
    var topItem: Element? {  
        return items.isEmpty ? nil : items[items.count - 1]  
    }  
}
```

`topItem` 属性会返回 `Element` 类型的可选值。当栈为空的时候，`topItem` 会返回 `nil`；当栈不为空的时候，`topItem` 会返回 `items` 数组中的最后一个元素。

注意：这个扩展并没有定义类型参数列表。相反的，`Stack` 类型已有的类型参数名称 `Element`，被用在扩展中来表示计算型属性 `topItem` 的可选类型。

计算型属性 `topItem` 现在可以用来访问任意 `Stack` 实例的顶端元素且不移除它：

```
if let topItem = stackOfStrings.topItem {  
    print("The top item on the stack is \(topItem).")  
}  
// 打印"The top item on the stack is tres."
```

泛型类型的扩展，还可以包括类型扩展需要额外满足的条件，从而对类型添加新功能，这一部分将在具有泛型 `Where` 子句的扩展中进行讨论。

## 类型约束

`swapTwoValues(_:_:)` 函数和 `Stack` 适用于任意类型。不过，如果能对泛型函数或泛型类型中添加特定的类型约束，这将在某些情况下非常有用。类型约束指定类型参数必须继承自指定类、遵循特定的协议或协议组合。

例如，Swift 的 `Dictionary` 类型对字典的键的类型做了些限制。在 [字典的描述](#) 中，字典键的类型必须是可哈希 (hashable) 的。也就是说，必须有一种方法能够唯一地表示它。字典键之所以要是可哈希的，是为了便于检查字典中是否已经包含某个特定键的值。若没有这个要求，字典将无法判断是否可以插入或替换某个指定键的值，也不能查找到已经存储在字典中的指定键的值。

这个要求通过 `Dictionary` 键类型上的类型约束实现，它指明了键必须遵循 Swift 标准库中定义的 `Hashable` 协议。所有 Swift 的基本类型（例如 `String`、`Int`、`Double` 和 `Bool`）默认都是可哈希的。

当自定义泛型类型时，你可以定义你自己的类型约束，这些约束将提供更为强大的泛型编程能力。像 `可哈希 (hashable)` 这种抽象概念根据它们的概念特征来描述类型，而不是它们的具体类型。

## 类型约束语法

在一个类型参数名后面放置一个类名或者协议名，并用冒号进行分隔，来定义类型约束。下面将展示泛型函数约束的基本语法（与泛型类型的语法相同）：

```
func someFunction<T: SomeClass, U: SomeProtocol>(someT: T, someU: U) {  
    // 这里是泛型函数的函数体部分  
}
```

上面这个函数有两个类型参数。第一个类型参数 `T` 必须是 `SomeClass` 子类；第二个类型参数 `U` 必须符合 `SomeProtocol` 协议。

## 类型约束实践

这里有个名为 `findIndex(ofString:in:)` 的非泛型函数，该函数的功能是在一个 `String` 数组中查找给定 `String` 值的索引。若查找到匹配的字符串，`findIndex(ofString:in:)` 函数返回该字符串在数组中的索引值，否则返回 `nil`：

```
func findIndex(ofString valueToFind: String, in array: [String]) -> Int? {
    for (index, value) in array.enumerated() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

`findIndex(ofString:in:)` 函数可以用于查找字符串数组中的某个字符串值：

```
let strings = ["cat", "dog", "llama", "parakeet", "terrapin"]
if let foundIndex = findIndex(ofString: "llama", in: strings) {
    print("The index of llama is \(foundIndex)")
}
// 打印"The index of llama is 2"
```

如果只能查找字符串在数组中的索引，用处不是很大。不过，你可以用占位类型 `T` 替换 `String` 类型来写出具有相同功能的泛型函数 `findIndex(_:_:)`。

下面展示了 `findIndex(ofString:in:)` 函数的泛型版本 `findIndex(of:in:)`。请注意这个函数返回值的类型仍然是 `Int?`，这是因为函数返回的是一个可选的索引数，而不是从数组中得到的一个可选值。需要提醒的是，这个函数无法通过编译，原因将在后面说明：

```
func findIndex<T>(of valueToFind: T, in array:[T]) -> Int? {
    for (index, value) in array.enumerated() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

上面所写的函数无法通过编译。问题出在相等性检查上，即 "`if value == valueToFind`"。不是所有的 Swift 类型都可以用等式符 (`==`) 进行比较。例如，如果你自定义类或结构体来描述复杂的数据模型，对于这个类或结构体而言，Swift 无法明确知道“相等”意味着什么。正因如此，这部分代码无法保证适用于任意类型 `T`，当你试图编译这部分代码时就会出现相应的错误。

不过，所有的这些并不会让我们无从下手。Swift 标准库中定义了一个 `Equatable` 协议，该协议要求任何遵循该协议的类型必须实现等式符 (`==`) 及不等符 (`!=`)，从而能对该类型的任意两个值进行比较。所有的 Swift 标准类型自动支持 `Equatable` 协

议。

遵循 `Equatable` 协议的类型都可以安全地用于 `findIndex(of:in:)` 函数，因为其保证支持等式操作符。为了说明这个事情，当定义一个函数时，你可以定义一个 `Equatable` 类型约束作为类型参数定义的一部分：

```
func findIndex<T: Equatable>(of valueToFind: T, in array:[T]) -> Int? {  
    for (index, value) in array.enumerated() {  
        if value == valueToFind {  
            return index  
        }  
    }  
    return nil  
}
```

`findIndex(of:in:)` 类型参数写做 `T: Equatable`，也就意味着“任何符合 `Equatable` 协议的类型 `T`”。

`findIndex(of:in:)` 函数现在可以成功编译了，并且适用于任何符合 `Equatable` 的类型，如 `Double` 或 `String`：

```
let doubleIndex = findIndex(of: 9.3, in: [3.14159, 0.1, 0.25])  
// doubleIndex 类型为 Int?, 其值为 nil, 因为 9.3 不在数组中  
let stringIndex = findIndex(of: "Andrea", in: ["Mike", "Malcolm", "Andrea"])  
// stringIndex 类型为 Int?, 其值为 2
```

## 关联类型

定义一个协议时，声明一个或多个关联类型作为协议定义的一部分将会非常有用。关联类型为协议中的某个类型提供了一个占位符名称，其代表的实际类型在协议被遵循时才会被指定。关联类型通过 `associatedtype` 关键字来指定。

## 关联类型实践

下面例子定义了一个 `Container` 协议，该协议定义了一个关联类型 `Item`：

```
protocol Container {  
    associatedtype Item  
    mutating func append(_ item: Item)  
    var count: Int { get }  
    subscript(i: Int) -> Item { get }  
}
```

`Container` 协议定义了三个任何遵循该协议的类型（即容器）必须提供的功能：

- 必须可以通过 `append(_:)` 方法添加一个新元素到容器里。
- 必须可以通过 `count` 属性获取容器中元素的数量，并返回一个 `Int` 值。
- 必须可以通过索引值类型为 `Int` 的下标检索到容器中的每一个元素。

该协议没有指定容器中元素该如何存储以及元素类型。该协议只指定了任何遵从 `Container` 协议的类型必须提供的三个功能。遵从协议的类型在满足这三个条件的情况下，也可以提供其他额外的功能。

任何遵从 `Container` 协议的类型必须能够指定其存储的元素的类型。具体来说，它必须确保添加到容器内的元素以及下标返回的元素类型是正确的。

为了定义这些条件，`Container` 协议需要在不知道容器中元素的具体类型的情况下引用这种类型。`Container` 协议需要指定任何通过 `append(_)` 方法添加到容器中的元素和容器内的元素是相同类型，并且通过容器下标返回的元素的类型也是这种类型。

为此，`Container` 协议声明了一个关联类型 `Item`，写作 `associatedtype Item`。协议没有定义 `Item` 是什么，这个信息留给遵从协议的类型来提供。尽管如此，`Item` 别名提供了一种方式来引用 `Container` 中元素的类型，并将之用于 `append(_)` 方法和下标，从而保证任何 `Container` 的行为都能如预期。

这是前面非泛型版本 `IntStack` 类型，使其遵循 `Container` 协议：

```
struct IntStack: Container {
    // IntStack 的原始实现部分
    var items = [Int]()
    mutating func push(_ item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
    // Container 协议的实现部分
    typealias Item = Int
    mutating func append(_ item: Int) {
        self.push(item)
    }
    var count: Int {
        return items.count
    }
    subscript(i: Int) -> Int {
        return items[i]
    }
}
```

`IntStack` 结构体实现了 `Container` 协议的三个要求，其原有功能也不会和这些要求相冲突。

此外，`IntStack` 在实现 `Container` 的要求时，指定 `Item` 为 `Int` 类型，即 `typealias Item = Int`，从而将 `Container` 协议中抽象的 `Item` 类型转换为具体的 `Int` 类型。

由于 Swift 的类型推断，实际上在 `IntStack` 的定义中不需要声明 `Item` 为 `Int`。因为 `IntStack` 符合 `Container` 协议的所有要求，Swift 只需通过 `append(_)` 方法的 `item` 参数类型和下标返回值的类型，就可以推断出 `Item` 的具体类型。事实上，如果你在上

面的代码中删除了 `typealias Item = Int` 这一行，一切也可正常工作，因为 Swift 清楚地知道 `Item` 应该是哪种类型。

你也可以让泛型 `Stack` 结构体遵循 `Container` 协议：

```
struct Stack<Element>: Container {  
    // Stack<Element> 的原始实现部分  
    var items = [Element]()  
    mutating func push(_ item: Element) {  
        items.append(item)  
    }  
    mutating func pop() -> Element {  
        return items.removeLast()  
    }  
    // Container 协议的实现部分  
    mutating func append(_ item: Element) {  
        self.push(item)  
    }  
    var count: Int {  
        return items.count  
    }  
    subscript(i: Int) -> Element {  
        return items[i]  
    }  
}
```

这一次，占位类型参数 `Element` 被用作 `append(_)` 方法的 `item` 参数和下标的返回类型。Swift 可以据此推断出 `Element` 的类型即是 `Item` 的类型。

## 扩展现有类型来指定关联类型

在扩展添加协议一致性 中描述了如何利用扩展让一个已存在的类型遵循一个协议，这包括使用了关联类型协议。

Swift 的 `Array` 类型已经提供 `append(_)` 方法，`count` 属性，以及带有 `Int` 索引的下标来检索其元素。这三个功能都符合 `Container` 协议的要求，也就意味着你只需声明 `Array` 遵循 `Container` 协议，就可以扩展 `Array`，使其遵从 `Container` 协议。你可以通过一个空扩展来实现这点，正如通过扩展采纳协议中的描述：

```
extension Array: Container {}
```

`Array` 的 `append(_)` 方法和下标确保了 Swift 可以推断出 `Item` 具体类型。定义了这个扩展后，你可以将任意 `Array` 当作 `Container` 来使用。

## 给关联类型添加约束

你可以在协议里给关联类型添加约束来要求遵循的类型满足约束。例如，下面的代码定义了 `Container` 协议，要求关联类型 `Item` 必须遵循 `Equatable` 协议：

```
protocol Container {  
    associatedtype Item: Equatable  
    mutating func append(_ item: Item)  
    var count: Int { get }  
    subscript(i: Int) -> Item { get }  
}
```

要遵守 `Container` 协议，`Item` 类型也必须遵守 `Equatable` 协议。

## 在关联类型约束里使用协议

协议可以作为它自身的要求出现。例如，有一个协议细化了 `Container` 协议，添加了一个 `suffix(_)` 方法。`suffix(_)` 方法返回容器中从后往前给定数量的元素，并把它们存储在一个 `Suffix` 类型的实例里。

```
protocol SuffixableContainer: Container {  
    associatedtype Suffix: SuffixableContainer where Suffix.Item == Item  
    func suffix(_ size: Int) -> Suffix  
}
```

在这个协议里，`Suffix` 是一个关联类型，就像上边例子中 `Container` 的 `Item` 类型一样。`Suffix` 拥有两个约束：它必须遵循 `SuffixableContainer` 协议（就是当前定义的协议），以及它的 `Item` 类型必须是和容器里的 `Item` 类型相同。`Item` 的约束是一个 `where` 分句，它在下面具有泛型 `Where` 子句的扩展中有讨论。

这是上面 泛型类型 中 `Stack` 类型的扩展，它遵循了 `SuffixableContainer` 协议：

```
extension Stack: SuffixableContainer {  
    func suffix(_ size: Int) -> Stack {  
        var result = Stack()  
        for index in (count-size)..<count {  
            result.append(self[index])  
        }  
        return result  
    }  
    // 推断 suffix 结果是Stack。  
}  
var stackOfInts = Stack<Int>()  
stackOfInts.append(10)  
stackOfInts.append(20)  
stackOfInts.append(30)  
let suffix = stackOfInts.suffix(2)  
// suffix 包含 20 和 30
```

在上面的例子中，`Suffix` 是 `Stack` 的关联类型，也是 `Stack`，所以 `Stack` 的后缀运算返回另一个 `Stack`。另外，遵循 `SuffixableContainer` 的类型可以拥有一个与它自己不同的 `Suffix` 类型——也就是说后缀运算可以返回不同的类型。比如说，这里有一个非泛型 `IntStack` 类型的扩展，它遵循了 `SuffixableContainer` 协议，使用 `Stack<Int>` 作为它的后缀类型而不是 `IntStack`：

```

extension IntStack: SuffixableContainer {
    func suffix(_ size: Int) -> Stack<Int> {
        var result = Stack<Int>()
        for index in (count-size)..<count {
            result.append(self[index])
        }
        return result
    }
    // 推断 suffix 结果是 Stack<Int>。
}

```

## 泛型 Where 语句

---

类型约束 让你能够为泛型函数、下标、类型的类型参数定义一些强制要求。

对关联类型添加约束通常是非常有用的。你可以通过定义一个泛型 `where` 子句来实现。通过泛型 `where` 子句让关联类型遵从某个特定的协议，以及某个特定的类型参数和关联类型必须类型相同。你可以通过将 `where` 关键字紧跟在类型参数列表后面来定义 `where` 子句，`where` 子句后跟一个或者多个针对关联类型的约束，以及一个或多个类型参数和关联类型间的相等关系。你可以在函数体或者类型的大括号之前添加 `where` 子句。

下面的例子定义了一个名为 `allItemsMatch` 的泛型函数，用来检查两个 `Container` 实例是否包含相同顺序的相同元素。如果所有的元素能够匹配，那么返回 `true`，否则返回 `false`。

被检查的两个 `Container` 可以不是相同类型的容器（虽然它们可以相同），但它们必须拥有相同类型的元素。这个要求通过一个类型约束以及一个 `where` 子句来表示：

```

func allItemsMatch<C1: Container, C2: Container>
    (_ someContainer: C1, _ anotherContainer: C2) -> Bool
    where C1.Item == C2.Item, C1.Item: Equatable {

    // 检查两个容器含有相同数量的元素
    if someContainer.count != anotherContainer.count {
        return false
    }

    // 检查每一对元素是否相等
    for i in 0..<someContainer.count {
        if someContainer[i] != anotherContainer[i] {
            return false
        }
    }

    // 所有元素都匹配，返回 true
    return true
}

```

这个函数接受 `someContainer` 和 `anotherContainer` 两个参数。参数 `someContainer` 的类型为 `C1`，参数 `anotherContainer` 的类型为 `C2`。`C1` 和 `C2` 是容器的两个占位类型参数，函数被调用时才能确定它们的具体类型。

这个函数的类型参数列表还定义了对两个类型参数的要求：

- `C1` 必须符合 `Container` 协议（写作 `C1: Container`）。
- `C2` 必须符合 `Container` 协议（写作 `C2: Container`）。
- `C1` 的 `Item` 必须和 `C2` 的 `Item` 类型相同（写作 `C1.Item == C2.Item`）。
- `C1` 的 `Item` 必须符合 `Equatable` 协议（写作 `C1.Item: Equatable`）。

前两个要求定义在函数的类型形式参数列表里，后两个要求定义在了函数的泛型 `where` 分句中。

这些要求意味着：

- `someContainer` 是一个 `C1` 类型的容器。
- `anotherContainer` 是一个 `C2` 类型的容器。
- `someContainer` 和 `anotherContainer` 包含相同类型的元素。
- `someContainer` 中的元素可以通过不等于操作符 (`!=`) 来检查它们是否相同。

第三个和第四个要求结合起来意味着 `anotherContainer` 中的元素也可以通过 `!=` 操作符来比较，因为它们和 `someContainer` 中的元素类型相同。

这些要求让 `allItemsMatch(_:_)` 函数能够比较两个容器，即使它们的容器类型不同。

`allItemsMatch(_:_)` 函数首先检查两个容器元素个数是否相同，如果元素个数不同，那么一定不匹配，函数就会返回 `false`。

进行这项检查之后，通过 `for-in` 循环和半闭区间操作符 (`..) 来迭代每个元素，检查 someContainer 中的元素是否不等于 anotherContainer 中的对应元素。如果两个元素不相等，那么两个容器不匹配，函数返回 false。`

如果循环体结束后未发现任何不匹配的情况，表明两个容器匹配，函数返回 `true`。

下面是 `allItemsMatch(_:_)` 函数的示例：

```
var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")

var arrayOfStrings = ["uno", "dos", "tres"]

if allItemsMatch(stackOfStrings, arrayOfStrings) {
    print("All items match.")
} else {
    print("Not all items match.")
}
// 打印"All items match."
```

上面的例子创建 `Stack` 实例来存储 `String` 值，然后将三个字符串压栈。这个例子还通过数组字面量创建了一个 `Array` 实例，数组中包含同栈中一样的三个字符串。即使栈和数组是不同的类型，但它们都遵从 `Container` 协议，而且它们都包含相同类型的值。因此你可以用这两个容器作为参数来调用 `allItemsMatch(_:_)` 函数。在上面的例子中，`allItemsMatch(_:_)` 函数正确地显示了这两个容器中的所有元素都是相互匹配的。

## 具有泛型 `where` 子句的扩展

---

你也可以使用泛型 `where` 子句作为扩展的一部分。基于以前的例子，下面的示例扩展了泛型 `Stack` 结构体，添加一个 `isTop(_:_)` 方法。

```
extension Stack where Element: Equatable {  
    func isTop(_ item: Element) -> Bool {  
        guard let topItem = items.last else {  
            return false  
        }  
        return topItem == item  
    }  
}
```

这个新的 `isTop(_:_)` 方法首先检查这个栈是不是空的，然后比较给定的元素与栈顶部的元素。如果你尝试不用泛型 `where` 子句，会有一个问题：在 `isTop(_:_)` 里面使用了 `==` 运算符，但是 `Stack` 的定义没有要求它的元素是符合 `Equatable` 协议的，所以使用 `==` 运算符导致编译时错误。使用泛型 `where` 子句可以为扩展添加新的条件，因此只有当栈中的元素符合 `Equatable` 协议时，扩展才会添加 `isTop(_:_)` 方法。

以下是 `isTop(_:_)` 方法的调用方式：

```
if stackOfStrings.isTop("tres") {  
    print("Top element is tres.")  
} else {  
    print("Top element is something else.")  
}  
// 打印“Top element is tres.”
```

如果尝试在其元素不符合 `Equatable` 协议的栈上调用 `isTop(_:_)` 方法，则会收到编译时错误。

```
struct NotEquatable { }  
var notEquatableStack = Stack<NotEquatable>()  
let notEquatableValue = NotEquatable()  
notEquatableStack.push(notEquatableValue)  
notEquatableStack.isTop(notEquatableValue) // 报错
```

你可以使用泛型 `where` 子句去扩展一个协议。基于以前的示例，下面的示例扩展了 `Container` 协议，添加一个 `startsWith(_:_)` 方法。

```
extension Container where Item: Equatable {  
    func startsWith(_ item: Item) -> Bool {  
        return count >= 1 && self[0] == item  
    }  
}
```

这个 `startsWith(_)` 方法首先确保容器至少有一个元素，然后检查容器中的第一个元素是否与给定的元素相等。任何符合 `Container` 协议的类型都可以使用这个新的 `startsWith(_)` 方法，包括上面使用的栈和数组，只要容器的元素是符合 `Equatable` 协议的。

```
if [9, 9, 9].startsWith(42) {  
    print("Starts with 42.")  
} else {  
    print("Starts with something else.")  
}  
// 打印“Starts with something else.”
```

上述示例中的泛型 `where` 子句要求 `Item` 遵循协议，但也可以编写一个泛型 `where` 子句去要求 `Item` 为特定类型。例如：

```
extension Container where Item == Double {  
    func average() -> Double {  
        var sum = 0.0  
        for index in 0..<count {  
            sum += self[index]  
        }  
        return sum / Double(count)  
    }  
}  
print([1260.0, 1200.0, 98.6, 37.0].average())  
// 打印“648.9”
```

此示例将一个 `average()` 方法添加到 `Item` 类型为 `Double` 的容器中。此方法遍历容器中的元素将其累加，并除以容器的数量计算平均值。它将数量从 `Int` 转换为 `Double` 确保能够进行浮点除法。

就像可以在其他地方写泛型 `where` 子句一样，你可以在一个泛型 `where` 子句中包含多个条件作为扩展的一部分。用逗号分隔列表中的每个条件。

## 具有泛型 Where 子句的关联类型

你可以在关联类型后面加上具有泛型 `where` 的字句。例如，建立一个包含迭代器（`Iterator`）的容器，就像是标准库中使用的 `Sequence` 协议那样。你应该这么写：

```

protocol Container {
    associatedtype Item
    mutating func append(_ item: Item)
    var count: Int { get }
    subscript(i: Int) -> Item { get }

    associatedtype Iterator: IteratorProtocol where Iterator.Element == Item
    func makeIterator() -> Iterator
}

```

迭代器（`Iterator`）的泛型 `where` 子句要求：无论迭代器是什么类型，迭代器中的元素类型，必须和容器项目的类型保持一致。`makeIterator()` 则提供了容器的迭代器的访问接口。

一个协议继承了另一个协议，你通过在协议声明的时候，包含泛型 `where` 子句，来添加了一个约束到被继承协议的关联类型。例如，下面的代码声明了一个 `ComparableContainer` 协议，它要求所有的 `Item` 必须是 `Comparable` 的。

```
protocol ComparableContainer: Container where Item: Comparable { }
```

## 泛型下标

---

下标可以是泛型，它们能够包含泛型 `where` 子句。你可以在 `subscript` 后用尖括号来写占位符类型，你还可以在下标代码块花括号前写 `where` 子句。例如：

```

extension Container {
    subscript<Indices: Sequence>(indices: Indices) -> [Item]
        where Indices.Iterator.Element == Int {
            var result = [Item]()
            for index in indices {
                result.append(self[index])
            }
            return result
        }
}

```

这个 `Container` 协议的扩展添加了一个下标方法，接收一个索引的集合，返回每一个索引所在的值的数组。这个泛型下标的约束如下：

- 在尖括号中的泛型参数 `Indices`，必须是符合标准库中的 `Sequence` 协议的类型。
- 下标使用的单一的参数，`indices`，必须是 `Indices` 的实例。
- 泛型 `where` 子句要求 `Sequence (Indices)` 的迭代器，其所有的元素都是 `Int` 类型。这样就能确保在序列（`Sequence`）中的索引和容器（`Container`）里面的索引类型是一致的。

综合一下，这些约束意味着，传入到 `indices` 下标，是一个整型的序列。

# 自动引用计数 · GitBook

 [runoob.com/manual/gitbook/swift5/source/\\_book/chapter2/23\\_Automatic\\_Reference\\_Counting.html](https://runoob.com/manual/gitbook/swift5/source/_book/chapter2/23_Automatic_Reference_Counting.html)

## 自动引用计数

Swift 使用自动引用计数 (ARC) 机制来跟踪和管理你的应用程序的内存。通常情况下，Swift 内存管理机制会一直起作用，你无须自己来考虑内存的管理。ARC 会在类的实例不再被使用时，自动释放其占用的内存。

然而在少数情况下，为了能帮助你管理内存，ARC 需要更多的，代码之间关系的信息。本章描述了这些情况，并且为你示范怎样才能使 ARC 来管理你的应用程序的所有内存。在 Swift 使用 ARC 与在 Objective-C 中使用 ARC 非常类似，具体请参考 [过渡到 ARC 的发布说明](#)。

### 注意

引用计数仅仅应用于类的实例。结构体和枚举类型是值类型，不是引用类型，也不是通过引用的方式存储和传递。

## 自动引用计数的工作机制

当你每次创建一个类的新的实例的时候，ARC 会分配一块内存来储存该实例信息。内存中会包含实例的类型信息，以及这个实例所有相关的存储型属性的值。

此外，当实例不再被使用时，ARC 释放实例所占用的内存，并让释放的内存能挪作他用。这确保了不再被使用的实例，不会一直占用内存空间。

然而，当 ARC 收回和释放了正在被使用中的实例，该实例的属性和方法将不能再被访问和调用。实际上，如果你试图访问这个实例，你的应用程序很可能会崩溃。

为了确保使用中的实例不会被销毁，ARC 会跟踪和计算每一个实例正在被多少属性，常量和变量所引用。哪怕实例的引用数为 1，ARC 都不会销毁这个实例。

为了使上述成为可能，无论你将实例赋值给属性、常量或变量，它们都会创建此实例的强引用。之所以称之为“强”引用，是因为它会将实例牢牢地保持住，只要强引用还在，实例是不允许被销毁的。

## 自动引用计数实践

下面的例子展示了自动引用计数的工作机制。例子以一个简单的 `Person` 类开始，并定义了一个叫 `name` 的常量属性：

```
class Person {  
    let name: String  
    init(name: String) {  
        self.name = name  
        print("\(name) is being initialized")  
    }  
    deinit {  
        print("\(name) is being deinitialized")  
    }  
}
```

`Person` 类有一个构造器，此构造器为实例的 `name` 属性赋值，并打印一条消息以表明初始化过程生效。`Person` 类也拥有一个析构器，这个析构器会在实例被销毁时打印一条消息。

接下来的代码片段定义了三个类型为 `Person?` 的变量，用来按照代码片段中的顺序，为新的 `Person` 实例建立多个引用。由于这些变量是被定义为可选类型 (`Person?`，而不是 `Person`)，它们的值会被自动初始化为 `nil`，目前还不会引用到 `Person` 类的实例。

```
var reference1: Person?  
var reference2: Person?  
var reference3: Person?
```

现在你可以创建 `Person` 类的新实例，并且将它赋值给三个变量中的一个：

```
reference1 = Person(name: "John Appleseed")  
// 打印“John Appleseed is being initialized”
```

应当注意到当你调用 `Person` 类的构造器的时候，“`John Appleseed is being initialized`”会被打印出来。由此可以确定构造器被执行。

由于 `Person` 类的新实例被赋值给了 `reference1` 变量，所以 `reference1` 到 `Person` 类的新实例之间建立了一个强引用。正是因为这一个强引用，ARC 会保证 `Person` 实例被保持在内存中不被销毁。

如果你将同一个 `Person` 实例也赋值给其他两个变量，该实例又会多出两个强引用：

```
reference2 = reference1  
reference3 = reference1
```

现在这一个 `Person` 实例已经有三个强引用了。

如果你通过给其中两个变量赋值 `nil` 的方式断开两个强引用（包括最先的那个强引用），只留下一个强引用，`Person` 实例不会被销毁：

```
reference1 = nil  
reference2 = nil
```

在你清楚地表明不再使用这个 `Person` 实例时，即第三个也就是最后一个强引用被断开时，ARC 会销毁它：

```
reference3 = nil
// 打印“John Appleseed is being deinitialized”
```

## 类实例之间的循环强引用

在上面的例子中，ARC 会跟踪你所新创建的 `Person` 实例的引用数量，并且会在 `Person` 实例不再被需要时销毁它。

然而，我们可能会写出一个类实例的强引用数永远不能变成 `0` 的代码。如果两个类实例互相持有对方的强引用，因而每个实例都让对方一直存在，就是这种情况。这就是所谓的 **循环强引用**。

你可以通过定义类之间的关系为弱引用或无主引用，以替代强引用，从而解决循环强引用的问题。具体的过程在 [解决类实例之间的循环强引用](#) 中有描述。不管怎样，在你学习怎样解决循环强引用之前，很有必要了解一下它是怎样产生的。

下面展示了一个不经意产生循环强引用的例子。例子定义了两个类：`Person` 和 `Apartment`，用来建模公寓和它其中的居民：

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("\(name) is being deinitialized") }
}

class Apartment {
    let unit: String
    init(unit: String) { self.unit = unit }
    var tenant: Person?
    deinit { print("Apartment \(unit) is being deinitialized") }
}
```

每一个 `Person` 实例有一个类型为 `String`，名字为 `name` 的属性，并有一个可选的初始化为 `nil` 的 `apartment` 属性。`apartment` 属性是可选的，因为一个人并不总是拥有公寓。

类似的，每个 `Apartment` 实例有一个叫 `unit`，类型为 `String` 的属性，并有一个可选的初始化为 `nil` 的 `tenant` 属性。`tenant` 属性是可选的，因为一栋公寓并不总是有居民。

这两个类都定义了析构器，用以在类实例被析构的时候输出信息。这让你能够知晓 `Person` 和 `Apartment` 的实例是否像预期的那样被销毁。

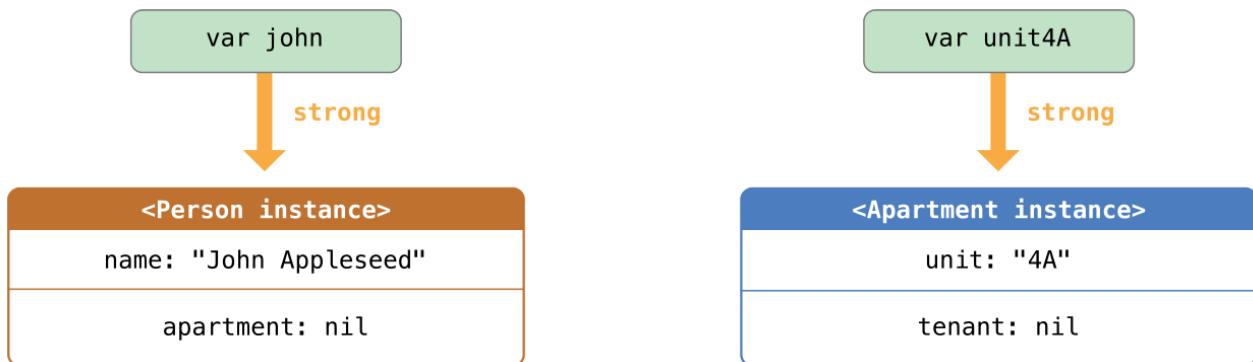
接下来的代码片段定义了两个可选类型的变量 `john` 和 `unit4A`，并分别被设定为下面的 `Apartment` 和 `Person` 的实例。这两个变量都被初始化为 `nil`，这正是可选类型的优点：

```
var john: Person?
var unit4A: Apartment?
```

现在你可以创建特定的 `Person` 和 `Apartment` 实例并将赋值给 `john` 和 `unit4A` 变量：

```
john = Person(name: "John Appleseed")
unit4A = Apartment(unit: "4A")
```

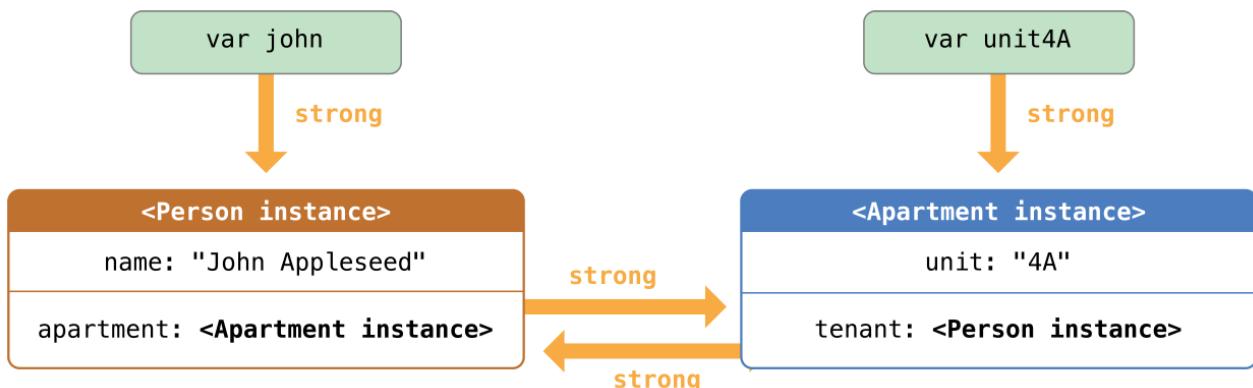
在两个实例被创建和赋值后，下图表现了强引用的关系。变量 `john` 现有一个指向 `Person` 实例的强引用，而变量 `unit4A` 有一个指向 `Apartment` 实例的强引用：



现在你能够将这两个实例关联在一起，这样人就能有公寓住了，而公寓也有了房客。注意感叹号是用来展开和访问可选变量 `john` 和 `unit4A` 中的实例，这样实例的属性才能被赋值：

```
john!.apartment = unit4A
unit4A!.tenant = john
```

在将两个实例联系在一起之后，强引用的关系如图所示：

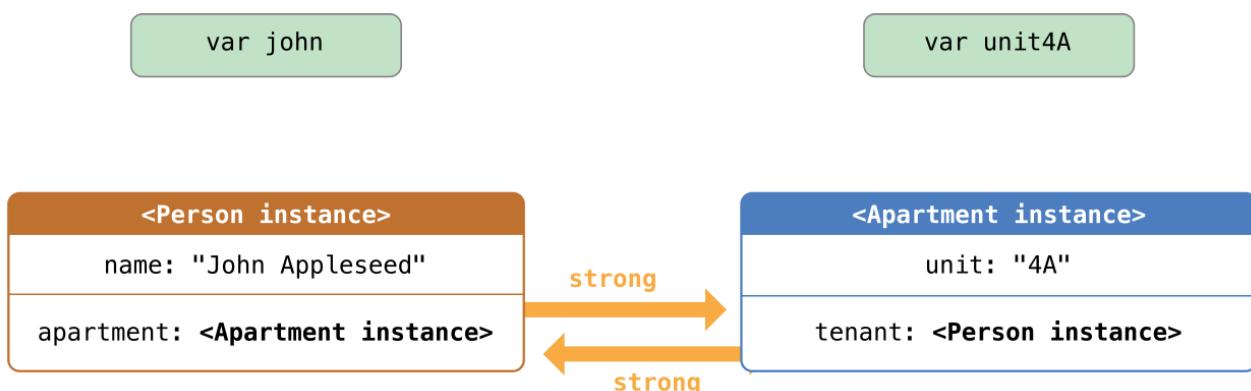


不幸的是，这两个实例关联后会产生一个循环强引用。`Person` 实例现在有了一个指向 `Apartment` 实例的强引用，而 `Apartment` 实例也有了一个指向 `Person` 实例的强引用。因此，当你断开 `john` 和 `unit4A` 变量所持有的强引用时，引用计数并不会降为 0，实例也不会被 ARC 销毁：

```
john = nil
unit4A = nil
```

注意，当你把这两个变量设为 `nil` 时，没有任何一个析构器被调用。循环强引用会一直阻止 `Person` 和 `Apartment` 类实例的销毁，这就在你的应用程序中造成了内存泄漏。

在你将 `john` 和 `unit4A` 赋值为 `nil` 后，强引用关系如下图：



`Person` 和 `Apartment` 实例之间的强引用关系保留了下来并且不会被断开。

## 解决实例之间的循环强引用

Swift 提供了两种办法用来解决你在使用类的属性时所遇到的循环强引用问题：弱引用（weak reference）和无主引用（unowned reference）。

弱引用和无主引用允许循环引用中的一个实例引用另一个实例而不保持强引用。这样实例能够互相引用而不产生循环强引用。

当其他的实例有更短的生命周期时，使用弱引用，也就是说，当其他实例析构在先时。在上面公寓的例子中，很显然一个公寓在它的生命周期内会在某个时间段没有它的主人，所以一个弱引用就加在公寓类里面，避免循环引用。相比之下，当其他实例有相同的或者更长生命周期时，请使用无主引用。

### 弱引用

弱引用不会对其引用的实例保持强引用，因而不会阻止 ARC 销毁被引用的实例。这个特性阻止了引用变为循环强引用。声明属性或者变量时，在前面加上 `weak` 关键字表明这是一个弱引用。

因为弱引用不会保持所引用的实例，即使引用存在，实例也有可能被销毁。因此，ARC 会在引用的实例被销毁后自动将其弱引用赋值为 `nil`。并且因为弱引用需要在运行时允许被赋值为 `nil`，所以它们会被定义为可选类型变量，而不是常量。

你可以像其他可选值一样，检查弱引用的值是否存在，你将永远不会访问已销毁的实例的引用。

注意

当 ARC 设置弱引用为 `nil` 时，属性观察不会被触发。

下面的例子跟上面 `Person` 和 `Apartment` 的例子一致，但是有一个重要的区别。这一次，`Apartment` 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 { print("\(name) is being deinitialized") }
}

```

```

class Apartment {
    let unit: String
    init(unit: String) { self.unit = unit }
    weak var tenant: Person?
    deinit { print("Apartment \(unit) is being deinitialized") }
}

```

然后跟之前一样，建立两个变量（`john` 和 `unit4A`）之间的强引用，并关联两个实例：

```

var john: Person?
var unit4A: Apartment?

```

```

john = Person(name: "John Appleseed")
unit4A = Apartment(unit: "4A")

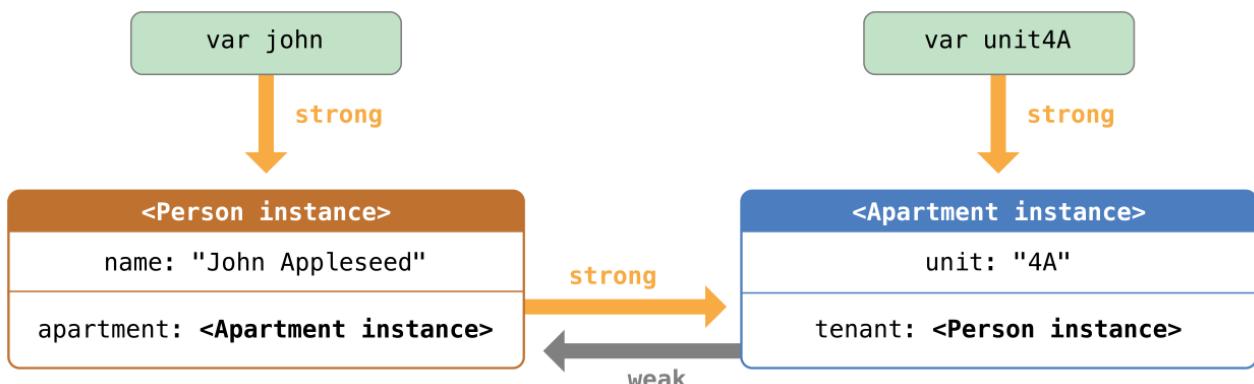
```

```

john!.apartment = unit4A
unit4A!.tenant = john

```

现在，两个关联在一起的实例的引用关系如下图所示：



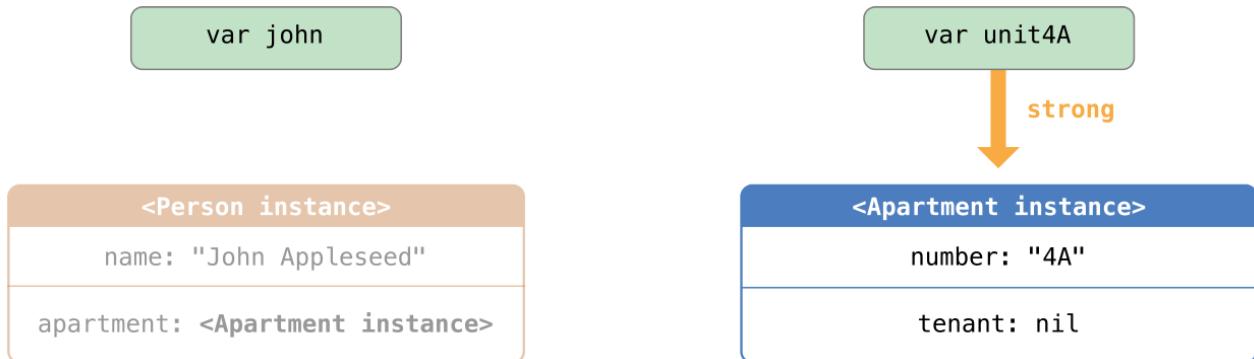
`Person` 实例依然保持对 `Apartment` 实例的强引用，但是 `Apartment` 实例只持有对 `Person` 实例的弱引用。这意味着当你通过把 `john` 变量赋值为 `nil` 而断开其所保持的强引用时，再也没有指向 `Person` 实例的强引用了：

```

john = nil
// 打印“John Appleseed is being deinitialized”

```

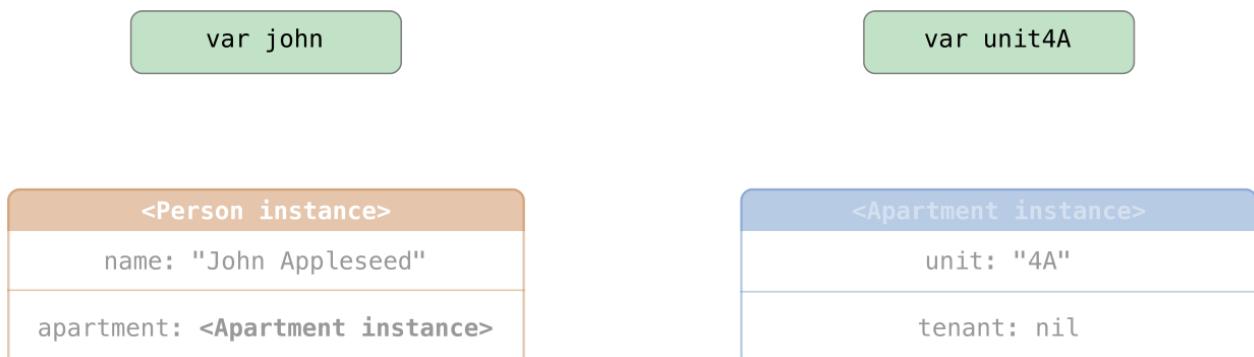
由于再也没有指向 `Person` 实例的强引用，该实例会被销毁，且 `tenant` 属性会被赋值为 `nil`：



唯一剩下的指向 `Apartment` 实例的强引用来自于变量 `unit4A`。如果你断开这个强引用，再也没有指向 `Apartment` 实例的强引用了：

```
unit4A = nil
// 打印“Apartment 4A is being deinitialized”
```

由于再也没有指向 `Person` 实例的强引用，该实例会被销毁：



### 注意

在使用垃圾收集的系统里，弱指针有时用来实现简单的缓冲机制，因为没有强引用的对象只会在内存压力触发垃圾收集时才被销毁。但是在 ARC 中，一旦值的最后一个强引用被移除，就会被立即销毁，这导致弱引用并不适合上面的用途。

## 无主引用

和弱引用类似，**无主引用**不会牢牢保持住引用的实例。和弱引用不同的是，无主引用在其他实例有相同或者更长的生命周期时使用。你可以在声明属性或者变量时，在前面加上关键字 `unowned` 表示这是一个无主引用。

无主引用通常都被期望拥有值。不过 ARC 无法在实例被销毁后将无主引用设为 `nil`，因为非可选类型的变量不允许被赋值为 `nil`。

### 重点

使用无主引用，你必须确保引用始终指向一个未销毁的实例。

如果你试图在实例被销毁后，访问该实例的无主引用，会触发运行时错误。

下面的例子定义了两个类，`Customer` 和 `CreditCard`，模拟了银行客户和客户的信用卡。这两个类中，每一个都将另外一个类的实例作为自身的属性。这种关系可能会造成循环强引用。

`Customer` 和 `CreditCard` 之间的关系与前面弱引用例子中 `Apartment` 和 `Person` 的关系略微不同。在这个数据模型中，一个客户可能有或者没有信用卡，但是是一张信用卡总是关联着一个客户。为了表示这种关系，`Customer` 类有一个可选类型的 `card` 属性，但是 `CreditCard` 类有一个非可选类型的 `customer` 属性。

此外，只能通过将一个 `number` 值和 `customer` 实例传递给 `CreditCard` 构造器的方式来创建 `CreditCard` 实例。这样可以确保当创建 `CreditCard` 实例时总是有一个 `customer` 实例与之关联。

由于信用卡总是关联着一个客户，因此将 `customer` 属性定义为无主引用，用以避免循环强引用：

```
class Customer {  
    let name: String  
    var card: CreditCard?  
    init(name: String) {  
        self.name = name  
    }  
    deinit { print("\(name) is being deinitialized") }  
}  
  
class CreditCard {  
    let number: UInt64  
    unowned let customer: Customer  
    init(number: UInt64, customer: Customer) {  
        self.number = number  
        self.customer = customer  
    }  
    deinit { print("Card #\(number) is being deinitialized") }  
}
```

### 注意

`CreditCard` 类的 `number` 属性被定义为 `UInt64` 类型而不是 `Int` 类型，以确保 `number` 属性的存储量在 32 位和 64 位系统上都能足够容纳 16 位的卡号。

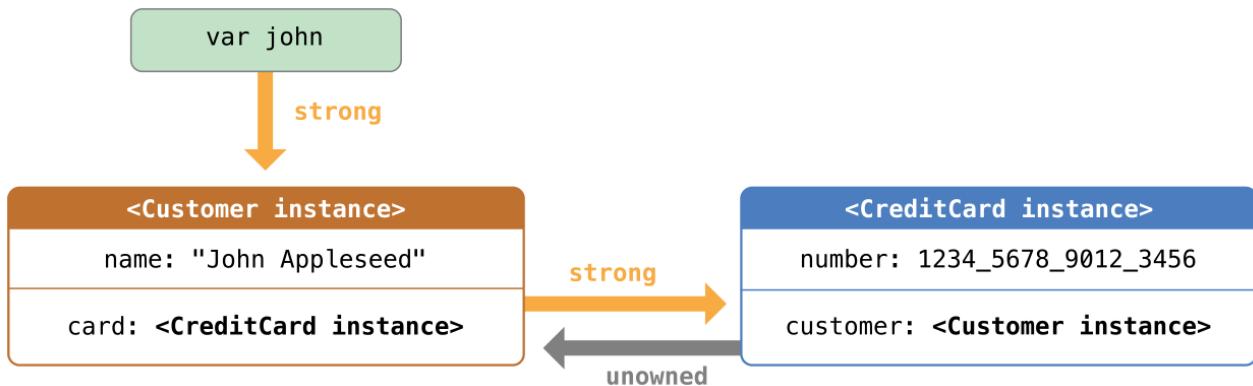
下面的代码片段定义了一个叫 `john` 的可选类型 `Customer` 变量，用来保存某个特定客户的引用。由于是可选类型，所以变量被初始化为 `nil`：

```
var john: Customer?
```

现在你可以创建 `Customer` 类的实例，用它初始化 `CreditCard` 实例，并将新创建的 `CreditCard` 实例赋值为客户的 `card` 属性：

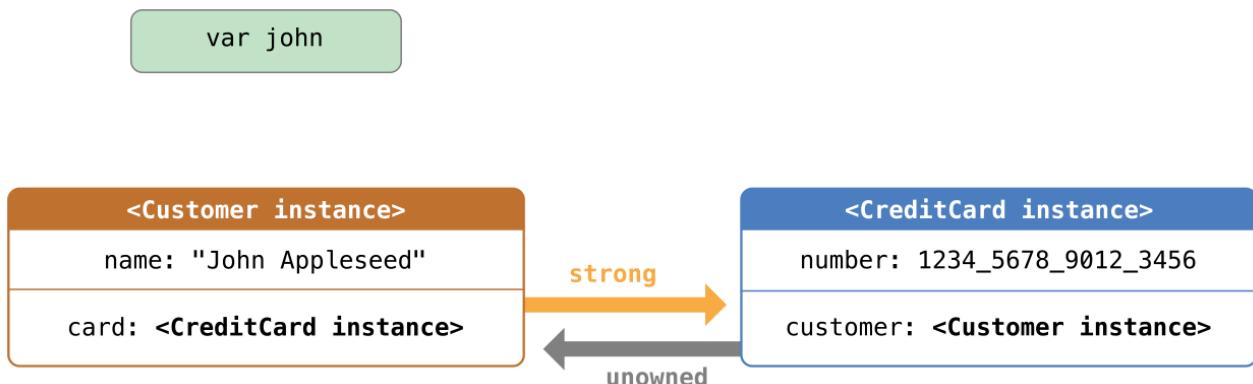
```
john = Customer(name: "John Appleseed")  
john!.card = CreditCard(number: 1234_5678_9012_3456, customer: john!)
```

在你关联两个实例后，它们的引用关系如下图所示：



`Customer` 实例持有对 `CreditCard` 实例的强引用，而 `CreditCard` 实例持有对 `Customer` 实例的无主引用。

由于 `customer` 的无主引用，当你断开 `john` 变量持有的强引用时，再也没有指向 `Customer` 实例的强引用了：



由于再也没有指向 `Customer` 实例的强引用，该实例被销毁了。其后，再也没有指向 `CreditCard` 实例的强引用，该实例也随之被销毁了：

```
john = nil
// 打印“John Appleseed is being deinitialized”
// 打印“Card #1234567890123456 is being deinitialized”
```

最后的代码展示了在 `john` 变量被设为 `nil` 后 `Customer` 实例和 `CreditCard` 实例的析构器都打印出了“销毁”的信息。

### 注意

上面的例子展示了如何使用安全的无主引用。对于需要禁用运行时的安全检查的情况（例如，出于性能方面的原因），Swift 还提供了不安全的无主引用。与所有不安全的操作一样，你需要负责检查代码以确保其安全性。你可以通过 `unowned(unsafe)` 来声明不安全无主引用。如果你试图在实例被销毁后，访问该实例的不安全无主引用，你的程序会尝试访问该实例之前所在的内存地址，这是一个不安全的操作。

## 无主引用和隐式解包可选值属性

---

上面弱引用和无主引用的例子涵盖了两种常用的需要打破循环强引用的场景。

`Person` 和 `Apartment` 的例子展示了两个属性的值都允许为 `nil`，并会潜在的产生循环强引用。这种场景最适合用弱引用来解决。

`Customer` 和 `CreditCard` 的例子展示了一个属性的值允许为 `nil`，而另一个属性的值不允许为 `nil`，这也可能会产生循环强引用。这种场景最适合通过无主引用来解决。

然而，存在着第三种场景，在这种场景中，两个属性都必须有值，并且初始化完成后永远不会为 `nil`。在这种场景中，需要一个类使用无主属性，而另外一个类使用隐式解包可选值属性。

这使两个属性在初始化完成后能被直接访问（不需要可选展开），同时避免了循环引用。这一节将为你展示如何建立这种关系。

下面的例子定义了两个类，`Country` 和 `City`，每个类将另外一个类的实例保存为属性。在这个模型中，每个国家必须有首都，每个城市必须属于一个国家。为了实现这种关系，`Country` 类拥有一个 `capitalCity` 属性，而 `City` 类有一个 `country` 属性：

```
class Country {
    let name: String
    var capitalCity: City!
    init(name: String, capitalName: String) {
        self.name = name
        self.capitalCity = City(name: capitalName, country: self)
    }
}

class City {
    let name: String
    unowned let country: Country
    init(name: String, country: Country) {
        self.name = name
        self.country = country
    }
}
```

为了建立两个类的依赖关系，`City` 的构造器接受一个 `Country` 实例作为参数，并且将实例保存到 `country` 属性。

`Country` 的构造器调用了 `City` 的构造器。然而，只有 `Country` 的实例完全初始化后，`Country` 的构造器才能把 `self` 传给 `City` 的构造器。在 两段式构造过程 中有具体描述。

为了满足这种需求，通过在类型结尾处加上感叹号（`City!`）的方式，将 `Country` 的 `capitalCity` 属性声明为隐式解包可选值类型的属性。这意味着像其他可选类型一样，`capitalCity` 属性的默认值为 `nil`，但是不需要展开它的值就能访问它。在 隐式解包可选值 中有描述。

由于 `capitalCity` 默认值为 `nil`，一旦 `Country` 的实例在构造器中给 `name` 属性赋值后，整个初始化过程就完成了。这意味着一旦 `name` 属性被赋值后，`Country` 的构造器就能引用并传递隐式的 `self`。`Country` 的构造器在赋值 `capitalCity` 时，就能将 `self` 作为参数传递给 `City` 的构造器。

以上的意义在于你可以通过一条语句同时创建 `Country` 和 `City` 的实例，而不产生循环强引用，并且 `capitalCity` 的属性能被直接访问，而不需要通过感叹号来展开它的可选值：

```
var country = Country(name: "Canada", capitalName: "Ottawa")
print("\(country.name)'s capital city is called \(country.capitalCity.name)")
// 打印"Canada's capital city is called Ottawa"
```

在上面的例子中，使用隐式解包可选值意味着满足了类的构造器的两个构造阶段的要求。`capitalCity` 属性在初始化完成后，能像非可选值一样使用和存取，同时还避免了循环强引用。

## 闭包的循环强引用

---

前面我们看到了循环强引用是在两个类实例属性互相保持对方的强引用时产生的，还知道了如何用弱引用和无主引用来打破这些循环强引用。

循环强引用还会发生在当你将一个闭包赋值给类实例的某个属性，并且这个闭包体中又使用了这个类实例时。这个闭包体中可能访问了实例的某个属性，例如 `self.someProperty`，或者闭包中调用了实例的某个方法，例如 `self.someMethod()`。这两种情况都导致了闭包“捕获”`self`，从而产生了循环强引用。

循环强引用的产生，是因为闭包和类相似，都是引用类型。当你把一个闭包赋值给某个属性时，你是将这个闭包的引用赋值给了属性。实质上，这跟之前的问题是一样的——两个强引用让彼此一直有效。但是，和两个类实例不同，这次一个是类实例，另一个是闭包。

Swift 提供了一种优雅的方法来解决这个问题，称之为 **闭包捕获列表**（closure capture list）。同样的，在学习如何用闭包捕获列表打破循环强引用之前，先来了解一下这里的循环强引用是如何产生的，这对我们很有帮助。

下面的例子为你展示了当一个闭包引用了 `self` 后是如何产生一个循环强引用的。例子中定义了一个叫 `HTMLElement` 的类，用一种简单的模型表示 HTML 文档中的一个单独的元素：

```

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) />"
        }
    }

    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }

    deinit {
        print("\(name) is being deinitialized")
    }
}

```

`HTMLElement` 类定义了一个 `name` 属性来表示这个元素的名称，例如代表头部元素的 `"h1"`，代表段落的 `"p"`，或者代表换行的 `"br"`。`HTMLElement` 还定义了一个可选属性 `text`，用来设置 HTML 元素呈现的文本。

除了上面的两个属性，`HTMLElement` 还定义了一个 `lazy` 属性 `asHTML`。这个属性引用了一个将 `name` 和 `text` 组合成 HTML 字符串片段的闭包。该属性是 `Void -> String` 类型，或者可以理解为“一个没有参数，返回 `String` 的函数”。

默认情况下，闭包赋值给了 `asHTML` 属性，这个闭包返回一个代表 HTML 标签的字符串。如果 `text` 值存在，该标签就包含可选值 `text`；如果 `text` 不存在，该标签就不包含文本。对于段落元素，根据 `text` 是 `"some text"` 还是 `nil`，闭包会返回 `"<p>some text</p>"` 或者 `"<p />"`。

可以像实例方法那样去命名、使用 `asHTML` 属性。然而，由于 `asHTML` 是闭包而不是实例方法，如果你想改变特定 HTML 元素的处理方式的话，可以用自定义的闭包来取代默认值。

例如，可以将一个闭包赋值给 `asHTML` 属性，这个闭包能在 `text` 属性是 `nil` 时使用默认文本，这是为了避免返回一个空的 HTML 标签：

```

let heading = HTMLElement(name: "h1")
let defaultText = "some default text"
heading.asHTML = {
    return "<(heading.name)>\\(heading.text ?? defaultText)</\\(heading.name)>"
}
print(heading.asHTML())
// 打印"<h1>some default text</h1>"

```

## 注意

`asHTML` 声明为 `lazy` 属性，因为只有当元素确实需要被处理为 HTML 输出的字符串时，才需要使用 `asHTML`。也就是说，在默认的闭包中可以使用 `self`，因为只有当初始化完成以及 `self` 确实存在后，才能访问 `lazy` 属性。

`HTMLElement` 类只提供了一个构造器，通过 `name` 和 `text`（如果有的话）参数来初始化一个新元素。该类也定义了一个析构器，当 `HTMLElement` 实例被销毁时，打印一条消息。

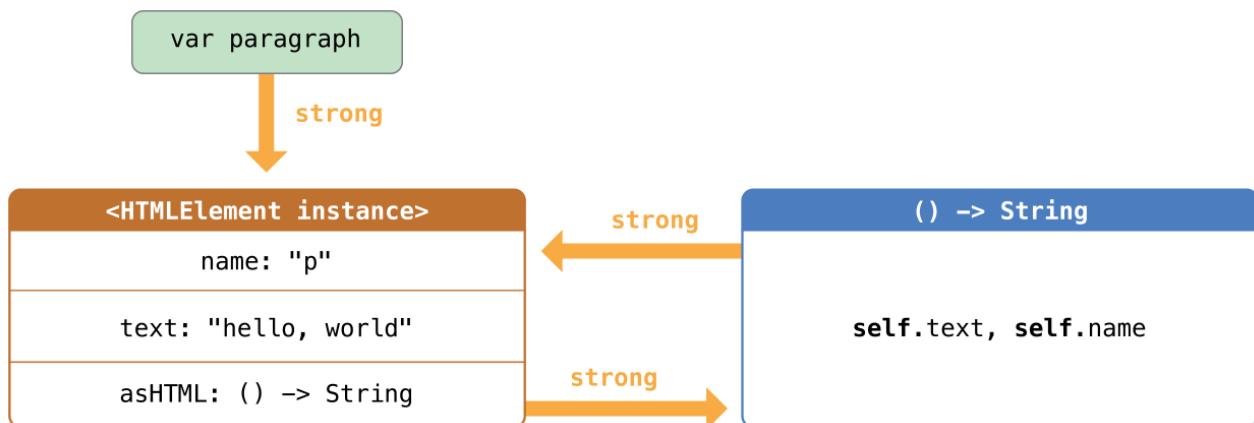
下面的代码展示了如何用 `HTMLElement` 类创建实例并打印消息：

```
var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
print(paragraph!.asHTML())
// 打印“<p>hello, world</p>”
```

## 注意

上面的 `paragraph` 变量定义为可选类型的 `HTMLElement`，因此我们可以赋值 `nil` 给它来演示循环强引用。

不幸的是，上面写的 `HTMLElement` 类产生了类实例和作为 `asHTML` 默认值的闭包之间的循环强引用。循环强引用如下图所示：



实例的 `asHTML` 属性持有闭包的强引用。但是，闭包在其闭包体内使用了 `self`（引用了 `self.name` 和 `self.text`），因此闭包捕获了 `self`，这意味着闭包又反过来持有了 `HTMLElement` 实例的强引用。这样两个对象就产生了循环强引用。（更多关于闭包捕获值的信息，请参考 [值捕获](#)）。

## 注意

虽然闭包多次使用了 `self`，它只捕获 `HTMLElement` 实例的一个强引用。

如果设置 `paragraph` 变量为 `nil`，打破它持有的 `HTMLElement` 实例的强引用，`HTMLElement` 实例和它的闭包都不会被销毁，也是因为循环强引用：

```
paragraph = nil
```

注意，`HTMLElement` 的析构器中的消息并没有被打印，证明了 `HTMLElement` 实例并没有被销毁。

## 解决闭包的循环强引用

在定义闭包时同时定义捕获列表作为闭包的一部分，通过这种方式可以解决闭包和类实例之间的循环强引用。捕获列表定义了闭包体内捕获一个或者多个引用类型的规则。跟解决两个类实例间的循环强引用一样，声明每个捕获的引用为弱引用或无主引用，而不是强引用。应当根据代码关系来决定使用弱引用还是无主引用。

### 注意

Swift 有如下要求：只要在闭包内使用 `self` 的成员，就要用 `self.someProperty` 或者 `self.someMethod()`（不只是 `someProperty` 或 `someMethod()`）。这提醒你可能会一不小心就捕获了 `self`。

## 定义捕获列表

捕获列表中的每一项都由一对元素组成，一个元素是 `weak` 或 `unowned` 关键字，另一个元素是类实例的引用（例如 `self`）或初始化过的变量（如 `delegate = self.delegate!`）。这些项在方括号中用逗号分开。

如果闭包有参数列表和返回类型，把捕获列表放在它们前面：

```
lazy var someClosure: (Int, String) -> String = {  
    [unowned self, weak delegate = self.delegate!] (index: Int, stringToProcess: String) -> String  
in  
    // 这里是闭包的函数体  
}
```

如果闭包没有指明参数列表或者返回类型，它们会通过上下文推断，那么可以把捕获列表和关键字 `in` 放在闭包最开始的地方：

```
lazy var someClosure: () -> String = {  
    [unowned self, weak delegate = self.delegate!] in  
    // 这里是闭包的函数体  
}
```

## 弱引用和无主引用

在闭包和捕获的实例总是互相引用并且总是同时销毁时，将闭包内的捕获定义为 **无主引用**。

相反的，在被捕获的引用可能会变为 `nil` 时，将闭包内的捕获定义为 **弱引用**。弱引用总是可选类型，并且当引用的实例被销毁后，弱引用的值会自动置为 `nil`。这使我们可以在闭包体内检查它们是否存在。

### 注意

如果被捕获的引用绝对不会变为 `nil`，应该用无主引用，而不是弱引用。

前面的 `HTMLElement` 例子中，无主引用是正确的解决循环强引用的方法。这样编写 `HTMLElement` 类来避免循环强引用：

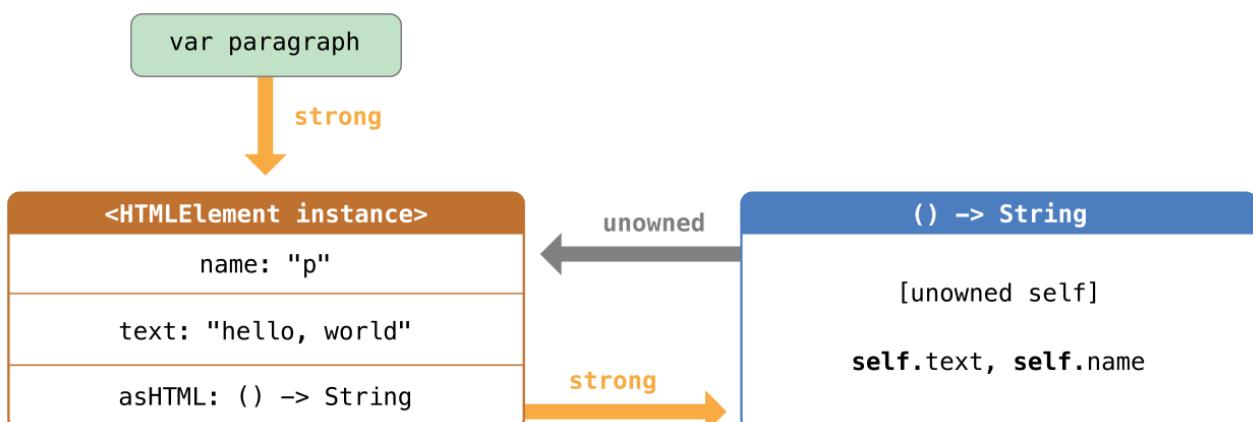
```
class HTMLElement {  
  
    let name: String  
    let text: String?  
  
    lazy var asHTML: () -> String = {  
        [unowned self] in  
        if let text = self.text {  
            return "<\(self.name)>\\(text)</\(self.name)>"  
        } else {  
            return "<\(self.name) />"  
        }  
    }  
  
    init(name: String, text: String? = nil) {  
        self.name = name  
        self.text = text  
    }  
  
    deinit {  
        print("\(name) is being deinitialized")  
    }  
  
}
```

上面的 `HTMLElement` 实现和之前的实现一致，除了在 `asHTML` 闭包中多了一个捕获列表。这里，捕获列表是 `[unowned self]`，表示“将 `self` 捕获为无主引用而不是强引用”。

和之前一样，我们可以创建并打印 `HTMLElement` 实例：

```
var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")  
print(paragraph!.asHTML())  
// 打印“<p>hello, world</p>”
```

使用捕获列表后引用关系如下图所示：



这一次，闭包以无主引用的形式捕获 `self`，并不会持有 `HTMLElement` 实例的强引用。如果将 `paragraph` 赋值为 `nil`，`HTMLElement` 实例将会被销毁，并能看到它的析构器打印出的消息：

```
paragraph = nil  
// 打印“p is being deinitialized”
```

你可以查看 [捕获列表](#) 章节，获取更多关于捕获列表的信息。

# 内存安全 · GitBook

 [runoob.com/manual/gitbook/swift5/source/\\_book/chapter2/24\\_Memory\\_Safety.html](https://runoob.com/manual/gitbook/swift5/source/_book/chapter2/24_Memory_Safety.html)

## 内存安全

默认情况下，Swift 会阻止你代码里不安全的行为。例如，Swift 会保证变量在使用之前就完成初始化，在内存被回收之后就无法被访问，并且数组的索引会做越界检查。

Swift 也保证同时访问同一块内存时不会冲突，通过约束代码里对于存储地址的写操作，去获取那一块内存的访问独占权。因为 Swift 自动管理内存，所以大部分时候你完全不需要考虑内存访问的事情。然而，理解潜在的冲突也是很重要的，可以避免你写出访问冲突的代码。而如果你的代码确实存在冲突，那在编译时或者运行时就会得到错误。

## 理解内存访问冲突

内存的访问，会发生在你给变量赋值，或者传递参数给函数时。例如，下面的代码就包含了读和写的访问：

```
// 向 one 所在的内存区域发起一次写操作  
var one = 1
```

```
// 向 one 所在的内存区域发起一次读操作  
print("We're number \(one)!")
```

内存访问的冲突会发生在你的代码尝试同时访问同一个存储地址的时候。同一个存储地址的多个访问同时发生会造成不可预计或不一致的行为。在 Swift 里，有很多修改值的行为都会持续好几行代码，在修改值的过程中进行访问是有可能发生的。

你可以思考一下预算表更新的过程，会看到同样的问题。更新预算表总共有两步：首先你把预算项的名字和费用加上，然后再更新总数来反映预算表的现况。在更新之前和之后，你都可以从预算表里读取任何信息并获得正确的答案，就像下面展示的那样。

Before	During	After																																
<table><tbody><tr><td>Eggs</td><td>\$3</td></tr><tr><td>Cookies</td><td>\$2</td></tr><tr><td><hr/></td><td></td></tr><tr><td>Total</td><td>\$5</td></tr></tbody></table>	Eggs	\$3	Cookies	\$2	<hr/>		Total	\$5	<table><tbody><tr><td>Eggs</td><td>\$3</td></tr><tr><td>Cookies</td><td>\$2</td></tr><tr><td>TV</td><td>\$298</td></tr><tr><td>T-shirt</td><td>\$17</td></tr><tr><td><hr/></td><td></td></tr><tr><td>Total</td><td>\$5</td></tr></tbody></table>	Eggs	\$3	Cookies	\$2	TV	\$298	T-shirt	\$17	<hr/>		Total	\$5	<table><tbody><tr><td>Eggs</td><td>\$3</td></tr><tr><td>Cookies</td><td>\$2</td></tr><tr><td>TV</td><td>\$298</td></tr><tr><td>T-shirt</td><td>\$17</td></tr><tr><td><hr/></td><td></td></tr><tr><td>Total</td><td>\$320</td></tr></tbody></table>	Eggs	\$3	Cookies	\$2	TV	\$298	T-shirt	\$17	<hr/>		Total	\$320
Eggs	\$3																																	
Cookies	\$2																																	
<hr/>																																		
Total	\$5																																	
Eggs	\$3																																	
Cookies	\$2																																	
TV	\$298																																	
T-shirt	\$17																																	
<hr/>																																		
Total	\$5																																	
Eggs	\$3																																	
Cookies	\$2																																	
TV	\$298																																	
T-shirt	\$17																																	
<hr/>																																		
Total	\$320																																	

而当你添加预算项进入表里的时候，它只是在一个临时的，错误的状态，因为总数还没有被更新。在添加数据的过程中读取总数就会读取到错误的信息。

这个例子也演示了你在修复内存访问冲突时会遇到的问题：有时修复的方式会有很多种，但哪一种是正确的就不总是那么明显了。在这个例子里，根据你是否需要更新后的总数，\$5 和 \$320 都可能是正确的值。在你修复访问冲突之前，你需要决定它的倾向。

### 注意

如果你写过并发和多线程的代码，内存访问冲突也许是同样的问题。然而，这里访问冲突的讨论是在单线程的情境下讨论的，并没有使用并发或者多线程。

如果你曾经在单线程代码里有访问冲突，Swift 可以保证你在编译或者运行时会得到错误。对于多线程的代码，可以使用 [Thread Sanitizer](#) 去帮助检测多线程的冲突。

## 内存访问性质

内存访问冲突时，要考虑内存访问上下文中的这三个性质：访问是读还是写，访问的时长，以及被访问的存储地址。特别是，冲突会发生在当你有两个访问符合下列的情况：

- 至少有一个是写访问
- 它们访问的是同一个存储地址
- 它们的访问在时间线上部分重叠

读和写访问的区别很明显：一个写访问会改变存储地址，而读操作不会。存储地址是指向正在访问的东西（例如一个变量，常量或者属性）的位置的值。内存访问的时长要么是瞬时的，要么是长期的。

如果一个访问不可能在其访问期间被其它代码访问，那么就是一个瞬时访问。正常来说，两个瞬时访问是不可能同时发生的。大多数内存访问都是瞬时的。例如，下面列举的所有读和写访问都是瞬时的：

```
func oneMore(than number: Int) -> Int {  
    return number + 1  
}  
  
var myNumber = 1  
myNumber = oneMore(than: myNumber)  
print(myNumber)  
// 打印“2”
```

然而，有几种被称为长期访问的内存访问方式，会在别的代码执行时持续进行。瞬时访问和长期访问的区别在于别的代码有没有可能在访问期间同时访问，也就是在时间线上的重叠。一个长期访问可以被别的长期访问或瞬时访问重叠。

重叠的访问主要出现在使用 `in-out` 参数的函数和方法或者结构体的 `mutating` 方法里。Swift 代码里典型的长期访问会在后面进行讨论。

## In-Out 参数的访问冲突

一个函数会对它所有的 in-out 参数进行长期写访问。in-out 参数的写访问会在所有非 in-out 参数处理完之后开始，直到函数执行完毕为止。如果有多个 in-out 参数，则写访问开始的顺序与参数的顺序一致。

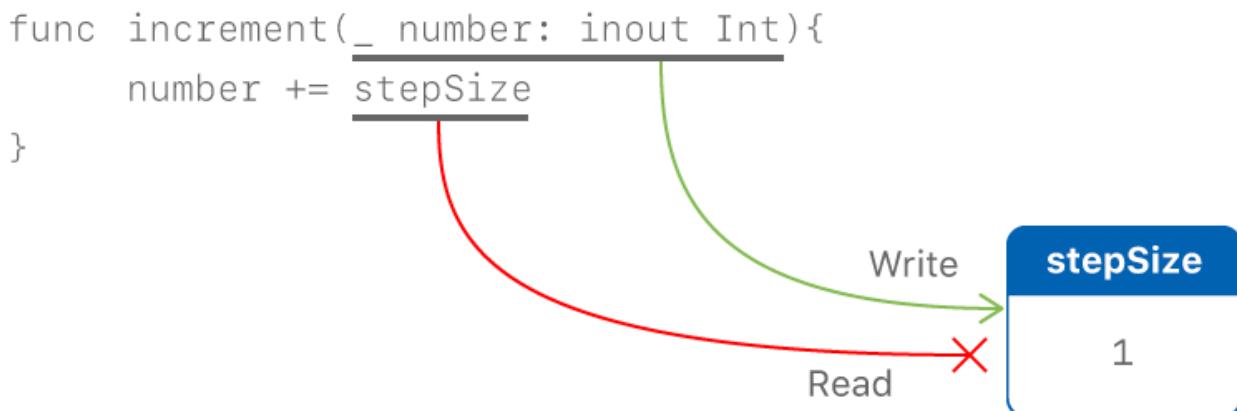
长期访问的存在会造成一个结果，你不能在访问以 in-out 形式传入后的原变量，即使作用域原则和访问权限允许——任何访问原变量的行为都会造成冲突。例如：

```
var stepSize = 1

func increment(_ number: inout Int) {
    number += stepSize
}

increment(&stepSize)
// 错误：stepSize 访问冲突
```

在上面的代码里，`stepSize` 是一个全局变量，并且它可以在 `increment(_)` 里正常访问。然而，对于 `stepSize` 的读访问与 `number` 的写访问重叠了。就像下面展示的那样，`number` 和 `stepSize` 都指向了同一个存储地址。同一块内存的读和写访问重叠了，就此产生了冲突。



解决这个冲突的一种方式，是显示拷贝一份 `stepSize`：

```
// 显式拷贝
var copyOfStepSize = stepSize
increment(&copyOfStepSize)

// 更新原来的值
stepSize = copyOfStepSize
// stepSize 现在的值是 2
```

当你在调用 `increment(_)` 之前做一份拷贝，显然 `copyOfStepSize` 就会根据当前的 `stepSize` 增加。读访问在写操作之前就已经结束了，所以不会有冲突。

长期写访问的存在还会造成另一种结果，往同一个函数的多个 in-out 参数里传入同一个变量也会产生冲突，例如：

```
func balance(_ x: inout Int, _ y: inout Int) {  
    let sum = x + y  
    x = sum / 2  
    y = sum - x  
}  
var playerOneScore = 42  
var playerTwoScore = 30  
balance(&playerOneScore, &playerTwoScore) // 正常  
balance(&playerOneScore, &playerOneScore)  
// 错误：playerOneScore 访问冲突
```

上面的 `balance(_:_)` 函数会将传入的两个参数平均化。将 `playerOneScore` 和 `playerTwoScore` 作为参数传入不会产生错误 —— 有两个访问重叠了，但它们访问的是不同的内存位置。相反，将 `playerOneScore` 作为参数同时传入就会产生冲突，因为它会发起两个写访问，同时访问同一个的存储地址。

注意

因为操作符也是函数，它们也会对 in-out 参数进行长期访问。例如，假设 `balance(_:_)` 是一个名为 `<^>` 的操作符函数，那么 `playerOneScore <^> playerOneScore` 也会造成像 `balance(&playerOneScore, &playerOneScore)` 一样的冲突。

## 方法里 self 的访问冲突

一个结构体的 `mutating` 方法会在调用期间对 `self` 进行写访问。例如，想象一下这么一个游戏，每一个玩家都有血量，受攻击时血量会下降，并且有敌人的数量，使用特殊技能时会减少敌人数量。

```
struct Player {  
    var name: String  
    var health: Int  
    var energy: Int  
  
    static let maxHealth = 10  
    mutating func restoreHealth() {  
        health = Player.maxHealth  
    }  
}
```

在上面的 `restoreHealth()` 方法里，一个对于 `self` 的写访问会从方法开始直到方法 `return`。在这种情况下，`restoreHealth()` 里的其它代码不可以对 `Player` 实例的属性发起重叠的访问。下面的 `shareHealth(with:)` 方法接受另一个 `Player` 的实例作为 `in-out` 参数，产生了访问重叠的可能性。

```

extension Player {
    mutating func shareHealth(with teammate: inout Player) {
        balance(&teammate.health, &health)
    }
}

```

```

var oscar = Player(name: "Oscar", health: 10, energy: 10)
var maria = Player(name: "Maria", health: 5, energy: 10)
oscar.shareHealth(with: &maria) // 正常

```

上面的例子里，调用 `shareHealth(with:)` 方法去把 `oscar` 玩家的血量分享给 `maria` 玩家并不会造成冲突。在方法调用期间会对 `oscar` 发起写访问，因为在 `mutating` 方法里 `self` 就是 `oscar`，同时对于 `maria` 也会发起写访问，因为 `maria` 作为 `in-out` 参数传入。过程如下，它们会访问内存的不同位置。即使两个写访问重叠了，它们也不会冲突。

```

mutating func shareHealth(with teammate: inout Player) {
    balance(&teammate.health, &health)
}

```

 `oscar.shareHealth(with: &maria)`



当然，如果你将 `oscar` 作为参数传入 `shareHealth(with:)` 里，就会产生冲突：

```

oscar.shareHealth(with: &oscar)
// 错误：oscar 访问冲突

```

`mutating` 方法在调用期间需要对 `self` 发起写访问，而同时 `in-out` 参数也需要写访问。在方法里，`self` 和 `teammate` 都指向了同一个存储地址——就像下面展示的那样。对于同一块内存同时进行两个写访问，并且它们重叠了，就此产生了冲突。

```

mutating func shareHealth(with teammate: inout Player) {
    balance(&teammate.health, &health)
}

```

 `oscar.shareHealth(with: &oscar)`



## 属性的访问冲突

如结构体，元组和枚举的类型都是由多个独立的值组成的，例如结构体的属性或元组的元素。因为它们都是值类型，修改值的任何一部分都是对于整个值的修改，意味着其中一个属性的读或写访问都需要访问整个值。例如，元组元素的写访问重叠会产生冲突：

```
var playerInformation = (health: 10, energy: 20)
balance(&playerInformation.health, &playerInformation.energy)
// 错误：playerInformation 的属性访问冲突
```

上面的例子里，传入同一元组的元素对 `balance(_:_)` 进行调用，产生了冲突，因为 `playerInformation` 的访问产生了写访问重叠。`playerInformation.health` 和 `playerInformation.energy` 都被作为 `in-out` 参数传入，意味着 `balance(_:_)` 需要在函数调用期间对它们发起写访问。任何情况下，对于元组元素的写访问都需要对整个元组发起写访问。这意味着对于 `playerInformation` 发起的两个写访问重叠了，造成冲突。

下面的代码展示了一样的错误，对于一个存储在全局变量里的结构体属性的写访问重叠了。

```
var holly = Player(name: "Holly", health: 10, energy: 10)
balance(&holly.health, &holly.energy) // 错误
```

在实践中，大多数对于结构体属性的访问都会安全的重叠。例如，将上面例子里的变量 `holly` 改为本地变量而非全局变量，编译器就会保证这个重叠访问是安全的：

```
func someFunction() {
    var oscar = Player(name: "Oscar", health: 10, energy: 10)
    balance(&oscar.health, &oscar.energy) // 正常
}
```

上面的例子里，`oscar` 的 `health` 和 `energy` 都作为 `in-out` 参数传入了 `balance(_:_)` 里。编译器可以保证内存安全，因为两个存储属性任何情况下都不会相互影响。

限制结构体属性的重叠访问对于保证内存安全不是必要的。保证内存安全是必要的，但因为访问独占权的要求比内存安全还要更严格——意味着即使有些代码违反了访问独占权的原则，也是内存安全的，所以如果编译器可以保证这种非专属的访问是安全的，那 Swift 就会允许这种行为的代码运行。特别是当你遵循下面的原则时，它可以保证结构体属性的重叠访问是安全的：

- 你访问的是实例的存储属性，而不是计算属性或类的属性
- 结构体是本地变量的值，而非全局变量
- 结构体要么没有被闭包捕获，要么只被非逃逸闭包捕获了

如果编译器无法保证访问的安全性，它就不会允许那次访问。

# 访问控制 · GitBook

 [runoob.com/manual/gitbook/swift5/source/\\_book/chapter2/25\\_Access\\_Control.html](https://runoob.com/manual/gitbook/swift5/source/_book/chapter2/25_Access_Control.html)

## 访问控制

访问控制可以限定其它源文件或模块中的代码对你的代码的访问级别。这个特性可以让我们隐藏代码的一些实现细节，并且可以为其他人可以访问和使用的代码提供接口。

你可以明确地给单个类型（类、结构体、枚举）设置访问级别，也可以给这些类型的属性、方法、构造器、下标等设置访问级别。协议也可以被限定在一定的范围内使用，包括协议里的全局常量、变量和函数。

Swift 不仅提供了多种不同的访问级别，还为某些典型场景提供了默认的访问级别，这样就不需要我们在每段代码中都申明显式访问级别。其实，如果只是开发一个单一 target 的应用程序，我们完全可以不用显式声明代码的访问级别。

注意

为了简单起见，对于代码中可以设置访问级别的特性（属性、基本类型、函数等），在下面的章节中我们会称之为“实体”。

## 模块和源文件

Swift 中的访问控制模型基于模块和源文件这两个概念。

模块指的是独立的代码单元，框架或应用程序会作为一个独立的模块来构建和发布。在 Swift 中，一个模块可以使用 `import` 关键字导入另外一个模块。

在 Swift 中，Xcode 的每个 target（例如框架或应用程序）都被当作独立的模块处理。如果你是为了实现某个通用的功能，或者是为了封装一些常用方法而将代码打包成独立的框架，这个框架就是 Swift 中的一个模块。当它被导入到某个应用程序或者其他框架时，框架内容都将属于这个独立的模块。

源文件就是 Swift 中的源代码文件，它通常属于一个模块，即一个应用程序或者框架。尽管我们一般会将不同的类型分别定义在不同的源文件中，但是同一个源文件也可以包含多个类型、函数之类的定义。

## 访问级别

Swift 为代码中的实体提供了五种不同的访问级别。这些访问级别不仅与源文件中定义的实体相关，同时也与源文件所属的模块相关。

- `Open` 和 `Public` 级别可以让实体被同一模块源文件中的所有实体访问，在模块外也可以通过导入该模块来访问源文件里的所有实体。通常情况下，你会使用 `Open` 或 `Public` 级别来指定框架的外部接口。`Open` 和 `Public` 的区别在后面会提到。

- *Internal* 级别让实体被同一模块源文件中的任何实体访问，但是不能被模块外的实体访问。通常情况下，如果某个接口只在应用程序或框架内部使用，就可以将其设置为 *Internal* 级别。
- *File-private* 限制实体只能在其定义的文件内部访问。如果功能的部分细节只需要在文件内使用时，可以使用 *File-private* 来将其隐藏。
- *Private* 限制实体只能在其定义的作用域，以及同一文件内的 extension 访问。如果功能的部分细节只需要在当前作用域内使用时，可以使用 *Private* 来将其隐藏。

`Open` 为最高访问级别（限制最少），`Private` 为最低访问级别（限制最多）。

`Open` 只能作用于类和类的成员，它和 `Public` 的区别如下：

- `Public` 或者其它更严访问级别的类，只能在其定义的模块内部被继承。
- `Public` 或者其它更严访问级别的类成员，只能在其定义的模块内部的子类中重写。
- `Open` 的类，可以在其定义的模块中被继承，也可以在引用它的模块中被继承。
- `Open` 的类成员，可以在其定义的模块中子类中重写，也可以在引用它的模块中的子类重写。

把一个类标记为 `open`，明确的表示你已经充分考虑过外部模块使用此类作为父类的影响，并且设计好了你的类的代码了。

## 访问级别基本原则

---

Swift 中的访问级别遵循一个基本原则：实体不能定义在具有更低访问级别（更严格）的实体中。

例如：

- 一个 `Public` 的变量，其类型的访问级别不能是 `Internal`，`File-private` 或是 `Private`。因为无法保证变量的类型在使用变量的地方也具有访问权限。
- 函数的访问级别不能高于它的参数类型和返回类型的访问级别。因为这样就会出现函数可以在任何地方被访问，但是它的参数类型和返回类型却不可以的情况。

关于此原则在各种情况下的具体表现，将在下文有所体现。

## 默认访问级别

---

如果你没有为代码中的实体显式指定访问级别，那么它们默认为 `internal` 级别（有一些例外情况，稍后会进行说明）。因此，在大多数情况下，我们不需要显式指定实体的访问级别。

## 单 target 应用程序的访问级别

---

当你编写一个单目标应用程序时，应用的所有功能都是为该应用服务，而不需要提供给其他应用或者模块使用，所以我们不需要明确设置访问级别，使用默认的访问级别 `Internal` 即可。但是，你也可以使用 `fileprivate` 访问或 `private` 访问级别，用于隐藏一些功能的实现细节。

## 框架的访问级别

当你开发框架时，就需要把一些对外的接口定义为 Open 或 Public，以便使用者导入该框架后可以正常使用其功能。这些被你定义为对外的接口，就是这个框架的 API。

### 注意

框架的内部实现仍然可以使用默认的访问级别 `internal`，当你需要对框架内部其它部分隐藏细节时可以使用 `private` 或 `fileprivate`。对于框架的对外 API 部分，你就需要将它们设置为 `open` 或 `public` 了。

## 单元测试 target 的访问级别

当你的应用程序包含单元测试 target 时，为了测试，测试模块需要访问应用程序模块中的代码。默认情况下只有 `open` 或 `public` 级别的实体才可以被其他模块访问。然而，如果在导入应用程序模块的语句前使用 `@testable` 特性，然后在允许测试的编译设置（`Build Options -> Enable Testability`）下编译这个应用程序模块，单元测试目标就可以访问应用程序模块中所有内部级别的实体。

## 访问控制语法

通过修饰符 `open`、`public`、`internal`、`fileprivate`、`private` 来声明实体的访问级别：

```
public class SomePublicClass {}  
internal class SomeInternalClass {}  
fileprivate class SomeFilePrivateClass {}  
private class SomePrivateClass {}  
  
public var somePublicVariable = 0  
internal let someInternalConstant = 0  
fileprivate func someFilePrivateFunction() {}  
private func somePrivateFunction() {}
```

除非专门指定，否则实体默认的访问级别为 `internal`，可以查阅 [默认访问级别](#) 这一节。这意味着在不使用修饰符显式声明访问级别的情况下，`SomeInternalClass` 和 `someInternalConstant` 仍然拥有隐式的 `internal`：

```
class SomeInternalClass {} // 隐式 internal  
var someInternalConstant = 0 // 隐式 internal
```

## 自定义类型

如果想为一个自定义类型指定访问级别，在定义类型时进行指定即可。新类型只能在它的访问级别限制范围内使用。例如，你定义了一个 `fileprivate` 级别的类，那这个类就只能在定义它的源文件中使用，可以作为属性类型、函数参数类型或者返回类型，等等。

一个类型的访问级别也会影响到类型成员（属性、方法、构造器、下标）的默认访问级别。如果你将类型指定为 `private` 或者 `fileprivate` 级别，那么该类型的所有成员的默认访问级别也会变成 `private` 或者 `fileprivate` 级别。如果你将类型指定为 `internal` 或 `public`（或者不明确指定访问级别，而使用默认的 `internal`），那么该类型的所有成员的默认访问级别将是 `internal`。

### 重点

上面提到，一个 `public` 类型的所有成员的访问级别默认为 `internal` 级别，而不是 `public` 级别。如果你想将某个成员指定为 `public` 级别，那么你必须显式指定。这样做的好处是，在你定义公共接口的时候，可以明确地选择哪些接口是需要公开的，哪些是内部使用的，避免不小心将内部使用的接口公开。

```
public class SomePublicClass {           // 显式 public 类
    public var somePublicProperty = 0     // 显式 public 类成员
    var someInternalProperty = 0         // 隐式 internal 类成员
    fileprivate func someFilePrivateMethod() {} // 显式 fileprivate 类成员
    private func somePrivateMethod() {}     // 显式 private 类成员
}

class SomeInternalClass {               // 隐式 internal 类
    var someInternalProperty = 0         // 隐式 internal 类成员
    fileprivate func someFilePrivateMethod() {} // 显式 fileprivate 类成员
    private func somePrivateMethod() {}     // 显式 private 类成员
}

fileprivate class SomeFilePrivateClass { // 显式 fileprivate 类
    func someFilePrivateMethod() {}       // 隐式 fileprivate 类成员
    private func somePrivateMethod() {}     // 显式 private 类成员
}

private class SomePrivateClass {        // 显式 private 类
    func somePrivateMethod() {}          // 隐式 private 类成员
}
```

## 元组类型

元组的访问级别将由元组中访问级别最严格的类型来决定。例如，如果你构建了一个包含两种不同类型的元组，其中一个类型为 `internal`，另一个类型为 `private`，那么这个元组的访问级别为 `private`。

### 注意

元组不同于类、结构体、枚举、函数那样有单独的定义。元组的访问级别是在它被使用时自动推断出的，而无法明确指定。

## 函数类型

函数的访问级别根据访问级别最严格的参数类型或返回类型的访问级别来决定。但是，如果这种访问级别不符合函数定义所在环境的默认访问级别，那么就需要明确地指定该函数的访问级别。

下面的例子定义了一个名为 `someFunction()` 的全局函数，并且没有明确地指定其访问级别。也许你会认为该函数应该拥有默认的访问级别 `internal`，但事实并非如此。事实上，如果按下面这种写法，代码将无法通过编译：

```
func someFunction() -> (SomeInternalClass, SomePrivateClass) {  
    // 此处是函数实现部分  
}
```

我们可以看到，这个函数的返回类型是一个元组，该元组中包含两个自定义的类（可查阅[自定义类型](#)）。其中一个类的访问级别是 `internal`，另一个的访问级别是 `private`，所以根据元组访问级别的原则，该元组的访问级别是 `private`（元组的访问级别与元组中访问级别最低的类型一致）。

因为该函数返回类型的访问级别是 `private`，所以你必须使用 `private` 修饰符，明确指定该函数的访问级别：

```
private func someFunction() -> (SomeInternalClass, SomePrivateClass) {  
    // 此处是函数实现部分  
}
```

将该函数指定为 `public` 或 `internal`，或者使用默认的访问级别 `internal` 都是错误的，因为如果把该函数当做 `public` 或 `internal` 级别来使用的话，可能会无法访问 `private` 级别的返回值。

## 枚举类型

---

枚举成员的访问级别和该枚举类型相同，你不能为枚举成员单独指定不同的访问级别。

比如下面的例子，枚举 `CompassPoint` 被明确指定为 `public`，那么它的成员 `North`、`South`、`East`、`West` 的访问级别同样也是 `public`：

```
public enum CompassPoint {  
    case north  
    case south  
    case east  
    case west  
}
```

## 原始值和关联值

---

枚举定义中的任何原始值或关联值的类型的访问级别至少不能低于枚举类型的访问级别。例如，你不能在一个 `internal` 的枚举中定义 `private` 的原始值类型。

## 嵌套类型

---

如果在 `private` 的类型中定义嵌套类型，那么该嵌套类型就自动拥有 `private` 访问级别。如果在 `public` 或者 `internal` 级别的类型中定义嵌套类型，那么该嵌套类型自动拥有 `internal` 访问级别。如果想让嵌套类型拥有 `public` 访问级别，那么需要明确指定该嵌套类型的访问级别。

## 子类

---

子类的访问级别不得高于父类的访问级别。例如，父类的访问级别是 `internal`，子类的访问级别就不能是 `public`。

此外，你可以在符合当前访问级别的条件下重写任意类成员（方法、属性、构造器、下标等）。

可以通过重写为继承来的类成员提供更高的访问级别。下面的例子中，类 `A` 的访问级别是 `public`，它包含一个方法 `someMethod()`，访问级别为 `private`。类 `B` 继承自类 `A`，访问级别为 `internal`，但是在类 `B` 中重写了类 `A` 中访问级别为 `private` 的方法 `someMethod()`，并重新指定为 `internal` 级别。通过这种方式，我们就可以将某类中 `private` 级别的类成员重新指定为更高的访问级别，以便其他人使用：

```
public class A {  
    fileprivate func someMethod() {}  
}  
  
internal class B: A {  
    override internal func someMethod() {}  
}
```

我们甚至可以在子类中，用子类成员去访问访问级别更低的父类成员，只要这一操作在相应访问级别的限制范围内（也就是说，在同一源文件中访问父类 `private` 级别的成员，在同一模块内访问父类 `internal` 级别的成员）：

```
public class A {  
    fileprivate func someMethod() {}  
}  
  
internal class B: A {  
    override internal func someMethod() {  
        super.someMethod()  
    }  
}
```

因为父类 `A` 和子类 `B` 定义在同一个源文件中，所以在子类 `B` 可以在重写的 `someMethod()` 方法中调用 `super.someMethod()`。

## 常量、变量、属性、下标

---

常量、变量、属性不能拥有比它们的类型更高的访问级别。例如，你不能定义一个 `public` 级别的属性，但是它的类型却是 `private` 级别的。同样，下标也不能拥有比索引类型或返回类型更高的访问级别。

如果常量、变量、属性、下标的类型是 `private` 级别的，那么它们必须明确指定访问级别为 `private`：

```
private var privateInstance = SomePrivateClass()
```

## Getter 和 Setter

常量、变量、属性、下标的 `Getters` 和 `Setters` 的访问级别和它们所属类型的访问级别相同。

`Setter` 的访问级别可以低于对应的 `Getter` 的访问级别，这样就可以控制变量、属性或下标的读写权限。在 `var` 或 `subscript` 关键字之前，你可以通过 `fileprivate(set)`，`private(set)` 或 `internal(set)` 为它们的写入权限指定更低的访问级别。

### 注意

这个规则同时适用于存储型属性和计算型属性。即使你不明确指定存储型属性的 `Getter` 和 `Setter`，Swift 也会隐式地为其创建 `Getter` 和 `Setter`，用于访问该属性的后备存储。使用 `fileprivate(set)`，`private(set)` 和 `internal(set)` 可以改变 `Setter` 的访问级别，这对计算型属性也同样适用。

下面的例子中定义了一个名为 `TrackedString` 的结构体，它记录了 `value` 属性被修改的次数：

```
struct TrackedString {  
    private(set) var numberOfEdits = 0  
    var value: String = "" {  
        didSet {  
            numberOfEdits += 1  
        }  
    }  
}
```

`TrackedString` 结构体定义了一个用于存储 `String` 值的属性 `value`，并将初始值设为 `""`（一个空字符串）。该结构体还定义了另一个用于存储 `Int` 值的属性 `numberOfEdits`，它用于记录属性 `value` 被修改的次数。这个功能通过属性 `value` 的 `didSet` 观察器实现，每当给 `value` 赋新值时就会调用 `didSet` 方法，然后将 `numberOfEdits` 的值加一。

结构体 `TrackedString` 和它的属性 `value` 都没有显式地指定访问级别，所以它们都是用默认的访问级别 `internal`。但是该结构体的 `numberOfEdits` 属性使用了 `private(set)` 修饰符，这意味着 `numberOfEdits` 属性只能在结构体的定义中进行赋值。`numberOfEdits` 属性的 `Getter` 依然是默认的访问级别 `internal`，但是 `Setter` 的访问级别是 `private`，这表示该属性只能在内部修改，而在结构体的外部则表现为一个只读属性。

如果你实例化 `TrackedString` 结构体，并多次对 `value` 属性的值进行修改，你就会看到 `numberOfEdits` 的值会随着修改次数而变化：

```
var stringToEdit = TrackedString()
stringToEdit.value = "This string will be tracked."
stringToEdit.value += " This edit will increment numberOfRows."
stringToEdit.value += " So will this one."
print("The number of edits is \(stringToEdit.numberOfEdits)")
// 打印"The number of edits is 3"
```

虽然你可以在其他的源文件中实例化该结构体并且获取到 `numberOfEdits` 属性的值，但是你不能对其进行赋值。这一限制保护了该记录功能的实现细节，同时还提供了方便的访问方式。

你可以在必要时为 `Getter` 和 `Setter` 显式指定访问级别。下面的例子将 `TrackedString` 结构体明确指定为了 `public` 访问级别。结构体的成员（包括 `numberOfEdits` 属性）拥有默认的访问级别 `internal`。你可以结合 `public` 和 `private(set)` 修饰符把结构体中的 `numberOfEdits` 属性的 `Getter` 的访问级别设置为 `public`，而 `Setter` 的访问级别设置为 `private`：

```
public struct TrackedString {
    public private(set) var numberOfRows = 0
    public var value: String = "" {
        didSet {
            numberOfRows += 1
        }
    }
    public init() {}
}
```

## 构造器

---

自定义构造器的访问级别可以低于或等于其所属类型的访问级别。唯一的例外是 必要构造器，它的访问级别必须和所属类型的访问级别相同。

如同函数或方法的参数，构造器参数的访问级别也不能低于构造器本身的访问级别。

## 默认构造器

---

如 默认构造器 所述，Swift 会为结构体和类提供一个默认的无参数的构造器，只要它们为所有存储型属性设置了默认初始值，并且未提供自定义的构造器。

默认构造器的访问级别与所属类型的访问级别相同，除非类型的访问级别是 `public`。如果一个类型被指定为 `public` 级别，那么默认构造器的访问级别将为 `internal`。如果你希望一个 `public` 级别的类型也能在其他模块中使用这种无参数的默认构造器，你只能自己提供一个 `public` 访问级别的无参数构造器。

## 结构体默认的成员逐一构造器

---

如果结构体中任意存储型属性的访问级别为 `private`，那么该结构体默认的成员逐一构造器的访问级别就是 `private`。否则，这种构造器的访问级别依然是 `internal`。

如同前面提到的默认构造器，如果你希望一个 `public` 级别的结构体也能在其他模块中使用其默认的成员逐一构造器，你依然只能自己提供一个 `public` 访问级别的成员逐一构造器。

## 协议

如果想为一个协议类型明确地指定访问级别，在定义协议时指定即可。这将限制该协议只能在适当的访问级别范围内被遵循。

协议中的每一个要求都具有和该协议相同的访问级别。你不能将协议中的要求设置为其他访问级别。这样才能确保该协议的所有要求对于任意遵循者都将可用。

### 注意

如果你定义了一个 `public` 访问级别的协议，那么该协议的所有实现也会是 `public` 访问级别。这一点不同于其他类型，例如，当类型是 `public` 访问级别时，其成员的访问级别却只是 `internal`。

## 协议继承

如果定义了一个继承自其他协议的新协议，那么新协议拥有的访问级别最高也只能和被继承协议的访问级别相同。例如，你不能将继承自 `internal` 协议的新协议定义为 `public` 协议。

## 协议遵循

一个类型可以遵循比它级别更低的协议。例如，你可以定义一个 `public` 级别类型，它能在别的模块中使用，但是如果它遵循一个 `internal` 协议，这个遵循的部分就只能在这个 `internal` 协议所在的模块中使用。

遵循协议时的上下文级别是类型和协议中级别最小的那个。如果一个类型是 `public` 级别，但它要遵循的协议是 `internal` 级别，那么这个类型对该协议的遵循上下文就是 `internal` 级别。

当你编写或扩展一个类型让它遵循一个协议时，你必须确保该类型对协议的每一个要求的实现，至少与遵循协议的上下文级别一致。例如，一个 `public` 类型遵循一个 `internal` 协议，这个类型对协议的所有实现至少都应是 `internal` 级别的。

### 注意

Swift 和 Objective-C 一样，协议遵循是全局的，也就是说，在同一程序中，一个类型不可能用两种不同的方式实现同一个协议。

## Extension

Extension 可以在访问级别允许的情况下对类、结构体、枚举进行扩展。Extension 的成员具有和原始类型成员一致的访问级别。例如，你使用 extension 扩展了一个 `public` 或者 `internal` 类型，extension 中的成员就默认使用 `internal` 访问级别，和原始类型中的

成员一致。如果你使用 extension 扩展了一个 `private` 类型，则 extension 的成员默认使用 `private` 访问级别。

或者，你可以明确指定 extension 的访问级别（例如，`private extension`），从而给该 extension 中的所有成员指定一个新的默认访问级别。这个新的默认访问级别仍然可以被单独指定的访问级别所覆盖。

如果你使用 extension 来遵循协议的话，就不能显式地声明 extension 的访问级别。extension 每个 protocol 要求的实现都默认使用 protocol 的访问级别。

## Extension 的私有成员

---

扩展同一文件内的类，结构体或者枚举，extension 里的代码会表现得跟声明在原类型里的一模一样。也就是说你可以这样：

- 在类型的声明里声明一个私有成员，在同一文件的 extension 里访问。
- 在 extension 里声明一个私有成员，在同一文件的另一个 extension 里访问。
- 在 extension 里声明一个私有成员，在同一文件的类型声明里访问。

这意味着你可以像组织的代码去使用 extension，而且不受私有成员的影响。例如，给定下面这样一个简单的协议：

```
protocol SomeProtocol {  
    func doSomething()  
}
```

你可以使用 extension 来遵循协议，就像这样：

```
struct SomeStruct {  
    private var privateVariable = 12  
}  
  
extension SomeStruct: SomeProtocol {  
    func doSomething() {  
        print(privateVariable)  
    }  
}
```

## 泛型

---

泛型类型或泛型函数的访问级别取决于泛型类型或泛型函数本身的访问级别，还需结合类型参数的类型约束的访问级别，根据这些访问级别中的最低访问级别来确定。

## 类型别名

---

你定义的任何类型别名都会被当作不同的类型，以便于进行访问控制。类型别名的访问级别不可高于其表示的类型的访问级别。例如，`private` 级别的类型别名可以作为 `private`、`file-private`、`internal`、`public` 或者 `open` 类型的别名，但是 `public` 级

别的类型别名只能作为 `public` 类型的别名，不能作为 `internal`、`file-private` 或 `private` 类型的别名。

注意

这条规则也适用于为满足协议遵循而将类型别名用于关联类型的情况。

# 高级运算符 · GitBook

 [runoob.com/manual/gitbook/swift5/source/\\_book/chapter2/26\\_Advanced\\_Operators.html](https://runoob.com/manual/gitbook/swift5/source/_book/chapter2/26_Advanced_Operators.html)

## 高级运算符

除了之前介绍过的 [基本运算符](#)，Swift 还提供了数种可以对数值进行复杂运算的高级运算符。它们包含了在 C 和 Objective-C 中已经被大家所熟知的位运算符和移位运算符。

与 C 语言中的算术运算符不同，Swift 中的算术运算符默认是不会溢出的。所有溢出行为都会被捕获并报告为错误。如果想让系统允许溢出行为，可以选择使用 Swift 中另一套默认支持溢出的运算符，比如溢出加法运算符（`&+`）。所有的这些溢出运算符都是以 `&` 开头的。

自定义结构体、类和枚举时，如果也为它们提供标准 Swift 运算符的实现，将会非常有用。在 Swift 中为这些运算符提供自定义的实现非常简单，运算符也会针对不同类型使用对应实现。

我们不用被预定义的运算符所限制。在 Swift 中可以自由地定义中缀、前缀、后缀和赋值运算符，它们具有自定义的优先级与关联值。这些运算符在代码中可以像预定义的运算符一样使用，你甚至可以扩展已有的类型以支持自定义运算符。

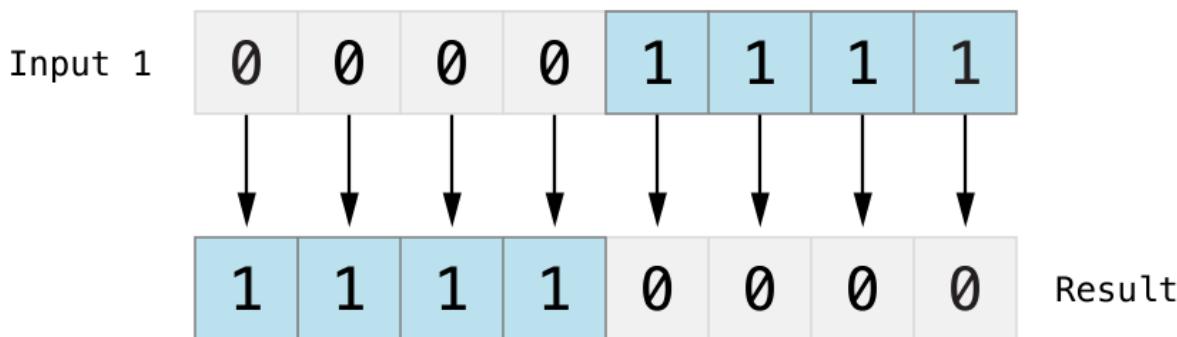
## 位运算符

位运算符可以操作数据结构中每个独立的比特位。它们通常被用在底层开发中，比如图形编程和创建设备驱动。位运算符在处理外部资源的原始数据时也十分有用，比如对自定义通信协议传输的数据进行编码和解码。

Swift 支持 C 语言中的全部位运算符，接下来会一一介绍。

### Bitwise NOT Operator (按位取反运算符)

按位取反运算符（`~`）对一个数值的全部比特位进行取反：



按位取反运算符是一个前缀运算符，直接放在运算数之前，并且它们之间不能添加任何空格：

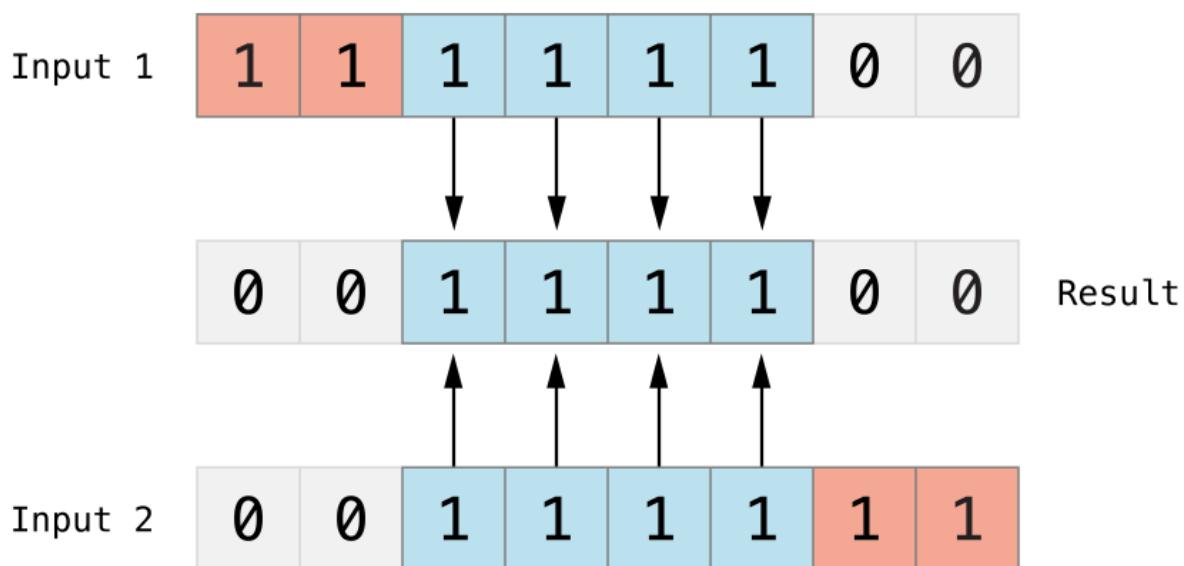
```
let initialBits: UInt8 = 0b00001111  
let invertedBits = ~initialBits // 等于 0b11110000
```

`UInt8` 类型的整数有 8 个比特位，可以存储 `0 ~ 255` 之间的任意整数。这个例子初始化了一个 `UInt8` 类型的整数，并赋值为二进制的 `00001111`，它的前 4 位为 `0`，后 4 位为 `1`。这个值等价于十进制的 `15`。

接着使用按位取反运算符创建了一个名为 `invertedBits` 的常量，这个常量的值与全部位取反后的 `initialBits` 相等。即所有的 `0` 都变成了 `1`，同时所有的 `1` 都变成 `0`。`invertedBits` 的二进制值为 `11110000`，等价于无符号十进制数的 `240`。

## Bitwise AND Operator (按位与运算符)

按位与运算符 (`&`) 对两个数的比特位进行合并。它返回一个新的数，只有当两个数的对应位都为 `1` 的时候，新数的对应位才为 `1`：

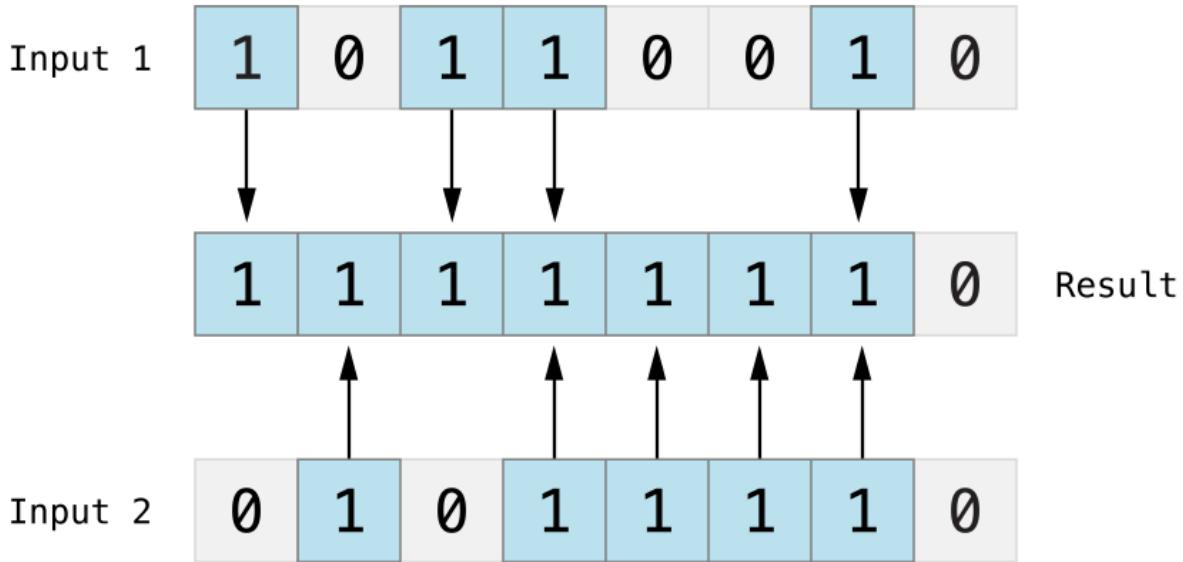


在下面的示例当中，`firstSixBits` 和 `lastSixBits` 中间 4 个位的值都为 `1`。使用按位与运算符之后，得到二进制数值 `00111100`，等价于无符号十进制数的 `60`：

```
let firstSixBits: UInt8 = 0b11111100  
let lastSixBits: UInt8 = 0b00111111  
let middleFourBits = firstSixBits & lastSixBits // 等于 00111100
```

## Bitwise OR Operator (按位或运算符)

按位或运算符 (`|`) 可以对两个数的比特位进行比较。它返回一个新的数，只要两个数的对应位中有任意一个为 `1` 时，新数的对应位就为 `1`：

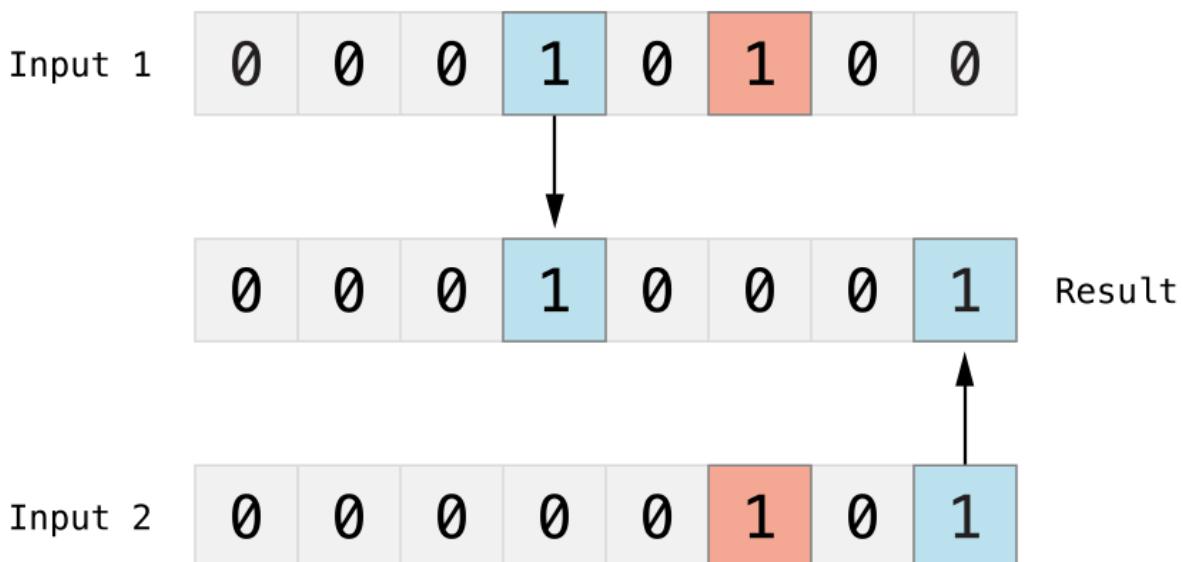


在下面的示例中，`someBits` 和 `moreBits` 存在不同的位被设置为 `1`。使用按位或运算符之后，得到二进制数值 `11111110`，等价于无符号十进制数的 `254`：

```
let someBits: UInt8 = 0b10110010
let moreBits: UInt8 = 0b01011110
let combinedbits = someBits | moreBits // 等于 11111110
```

## Bitwise XOR Operator (按位异或运算符)

按位异或运算符，或称“排外的或运算符”（`^`），可以对两个数的比特位进行比较。它返回一个新的数，当两个数的对应位不相同时，新数的对应位就为 `1`，并且对应位相同时则为 `0`：



在下面的示例当中，`firstBits` 和 `otherBits` 都有一个自己为 `1`，而对方为 `0` 的位。按位异或运算符将新数的这两个位都设置为 `1`。在其余的位上 `firstBits` 和 `otherBits` 是相同的，所以设置为 `0`：

```
let firstBits: UInt8 = 0b000010100  
let otherBits: UInt8 = 0b000000101  
let outputBits = firstBits ^ otherBits // 等于 00010001
```

## Bitwise Left and Right Shift Operators (按位左移、右移运算符)

按位左移运算符 (`<<`) 和 按位右移运算符 (`>>`) 可以对一个数的所有位进行指定位数的左移和右移，但是需要遵守下面定义的规则。

对一个数进行按位左移或按位右移，相当于对这个数进行乘以 2 或除以 2 的运算。将一个整数左移一位，等价于将这个数乘以 2，同样地，将一个整数右移一位，等价于将这个数除以 2。

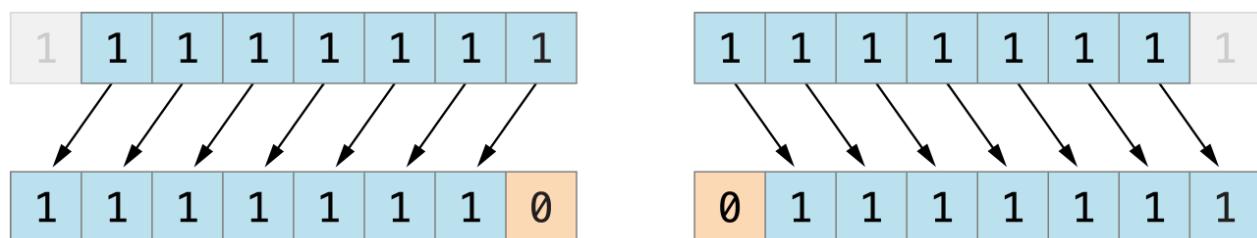
### 无符号整数的移位运算

对无符号整数进行移位的规则如下：

1. 已存在的位按指定的位数进行左移和右移。
2. 任何因移动而超出整型存储范围的位都会被丢弃。
3. 用 `0` 来填充移位后产生的空白位。

这种方法称为逻辑移位。

以下这张图展示了 `11111111 << 1` (即把 `11111111` 向左移动 1 位)，和 `11111111 >> 1` (即把 `11111111` 向右移动 1 位) 的结果。蓝色的数字是被移位的，灰色的数字是被抛弃的，橙色的 `0` 则是被填充进来的：



下面的代码演示了 Swift 中的移位运算：

```
let shiftBits: UInt8 = 4 // 即二进制的 00000100  
shiftBits << 1 // 00001000  
shiftBits << 2 // 00010000  
shiftBits << 5 // 10000000  
shiftBits << 6 // 00000000  
shiftBits >> 2 // 00000001
```

可以使用移位运算对其他的数据类型进行编码和解码：

```
let pink: UInt32 = 0xCC6699  
let redComponent = (pink & 0xFF0000) >> 16 // redComponent 是 0xCC，即 204  
let greenComponent = (pink & 0x00FF00) >> 8 // greenComponent 是 0x66，即 102  
let blueComponent = pink & 0x0000FF // blueComponent 是 0x99，即 153
```

这个示例使用了一个命名为 `pink` 的 `UInt32` 型常量来存储 Cascading Style Sheets (CSS) 中粉色的颜色值。该 CSS 的颜色值 `#CC6699`，在 Swift 中表示为十六进制的 `0xCC6699`。然后利用按位与运算符（`&`）和按位右移运算符（`>>`）从这个颜色值中分解出红（`CC`）、绿（`66`）以及蓝（`99`）三个部分。

红色部分是通过对 `0xCC6699` 和 `0xFF0000` 进行按位与运算后得到的。`0xFF0000` 中的 `0` 部分“掩盖”了 `0xCC6699` 中的第二、第三个字节，使得数值中的 `6699` 被忽略，只留下 `0xCC0000`。

然后，将这个数向右移动 16 位（`>> 16`）。十六进制中每两个字符占用 8 个比特位，所以移动 16 位后 `0xCC0000` 就变为 `0x0000CC`。这个数和 `0xCC` 是等同的，也就是十进制数值的 `204`。

同样的，绿色部分通过对 `0xCC6699` 和 `0x00FF00` 进行按位与运算得到 `0x006600`。然后将这个数向右移动 8 位，得到 `0x66`，也就是十进制数值的 `102`。

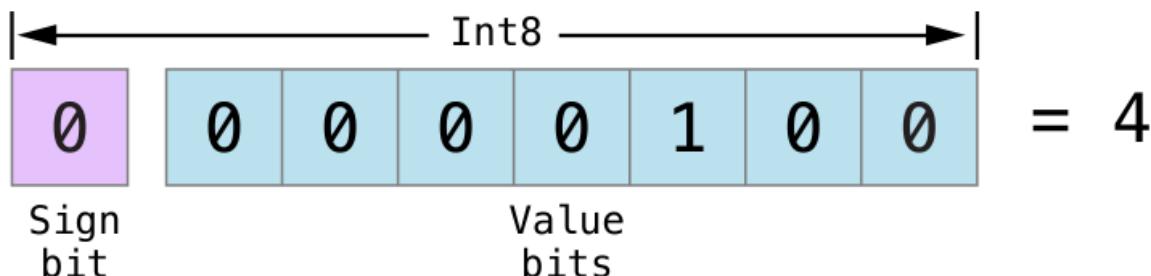
最后，蓝色部分通过对 `0xCC6699` 和 `0x0000FF` 进行按位与运算得到 `0x000099`。这里不需要再向右移位，而 `0x000099` 也就是 `0x99`，也就是十进制数值的 `153`。

## 有符号整数的移位运算

对比无符号整数，有符号整数的移位运算相对复杂得多，这种复杂性源于有符号整数的二进制表现形式。（为了简单起见，以下的示例都是基于 8 比特的有符号整数，但是其中的原理对任何位数的有符号整数都是通用的。）

有符号整数使用第 1 个比特位（通常被称为符号位）来表示这个数的正负。符号位为 `0` 代表正数，为 `1` 代表负数。

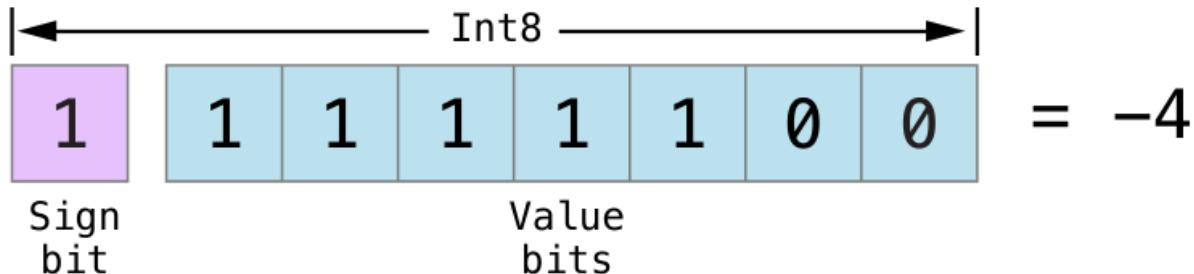
其余的比特位（通常被称为数值位）存储了实际的值。有符号正整数和无符号数的存储方式是一样的，都是从 `0` 开始算起。这是值为 `4` 的 `Int8` 型整数的二进制位表现形式：



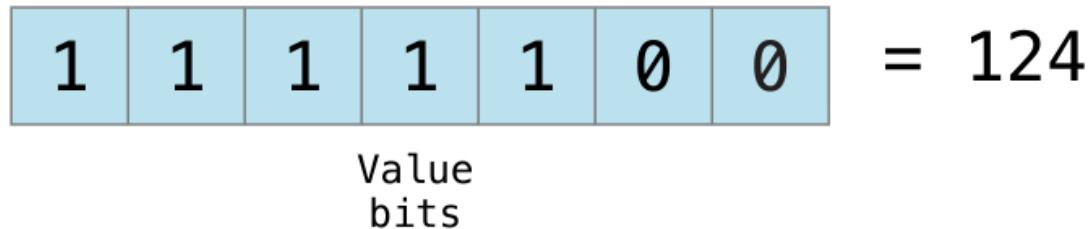
符号位为 `0`（代表这是一个“正数”），另外 7 位则代表了十进制数值 `4` 的二进制表示。

负数的存储方式略有不同。它存储 `2` 的 `n` 次方减去其实际值的绝对值，这里的 `n` 是数值位的位数。一个 8 比特位的数有 7 个比特位是数值位，所以是 `2` 的 `7` 次方，即 `128`。

这是值为 `-4` 的 `Int8` 型整数的二进制表现形式：

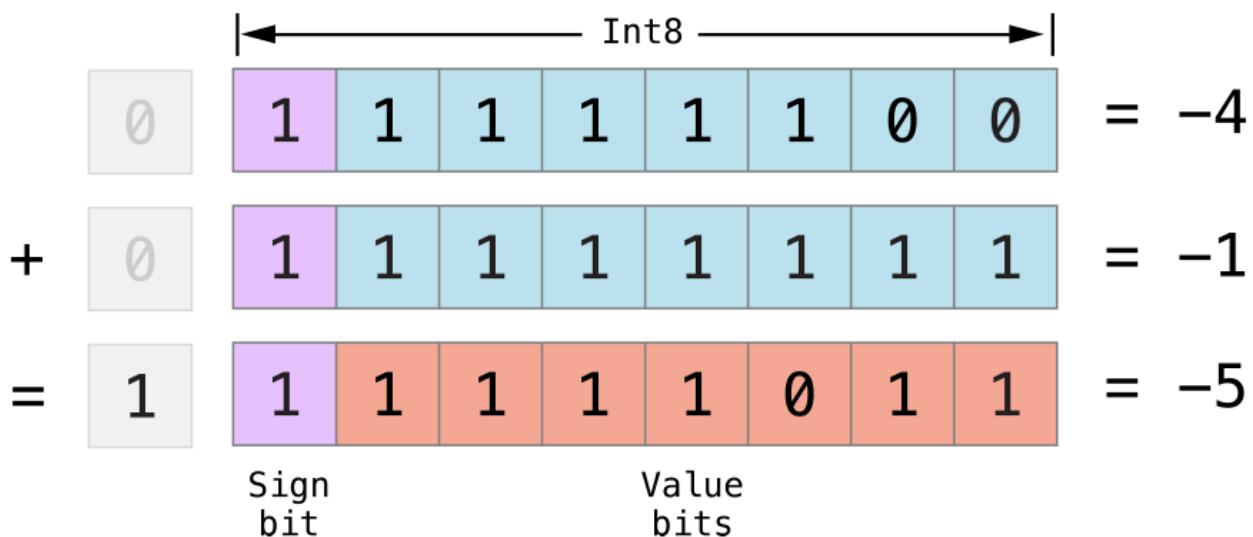


这次的符号位为 1，说明这是一个负数，另外 7 个位则代表了数值 124（即 128 - 4）的二进制表示：

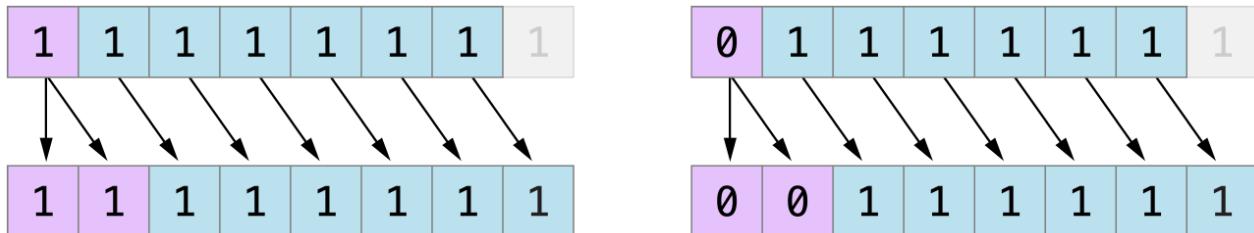


负数的表示通常被称为**二进制补码**。用这种方法来表示负数乍看起来有点奇怪，但它有几个优点。

首先，如果想对 -1 和 -4 进行加法运算，我们只需要对这两个数的全部 8 个比特位执行标准的二进制相加（包括符号位），并且将计算结果中超出 8 位的数值丢弃：



其次，使用二进制补码可以使负数的按位左移和右移运算得到跟正数同样的效果，即每向左移一位就将自身的数值乘以 2，每向右一位就将自身的数值除以 2。要达到此目的，对有符号整数的右移有一个额外的规则：当对有符号整数进行按位右移运算时，遵循与无符号整数相同的规则，但是对于移位产生的空白位使用符号位进行填充，而不是用 0。



这个行为可以确保有符号整数的符号位不会因为右移运算而改变，这通常被称为算术移位。

由于正数和负数的特殊存储方式，在对它们进行右移的时候，会使它们越来越接近 0。在移位的过程中保持符号位不变，意味着负整数在接近 0 的过程中会一直保持为负。

## 溢出运算符

当向一个整数类型的常量或者变量赋予超过它容量的值时，Swift 默认会报错，而不是允许生成一个无效的数。这个行为为我们在运算过大或者过小的数时提供了额外的安全性。

例如，`Int16` 型整数能容纳的有符号整数范围是 -32768 到 32767。当为一个 `Int16` 类型的变量或常量赋予的值超过这个范围时，系统就会报错：

```
var potentialOverflow = Int16.max
// potentialOverflow 的值是 32767，这是 Int16 能容纳的最大整数
potentialOverflow += 1
// 这里会报错
```

在赋值时为过大或者过小的情况提供错误处理，能让我们在处理边界值时更加灵活。

然而，当你希望的时候也可以选择让系统在数值溢出的时候采取截断处理，而非报错。Swift 提供的三个溢出运算符来让系统支持整数溢出运算。这些运算符都是以 & 开头的：

- 溢出加法 `&+`
- 溢出减法 `&-`
- 溢出乘法 `&*`

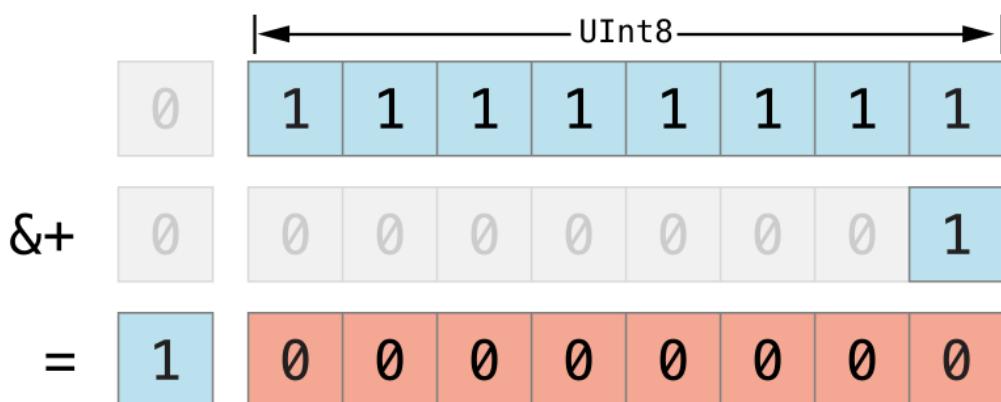
## 数值溢出

数值有可能出现上溢或者下溢。

这个示例演示了当我们对一个无符号整数使用溢出加法（`&+`）进行上溢运算时会发生什么：

```
var unsignedOverflow = UInt8.max
// unsignedOverflow 等于 UInt8 所能容纳的最大整数 255
unsignedOverflow = unsignedOverflow &+ 1
// 此时 unsignedOverflow 等于 0
```

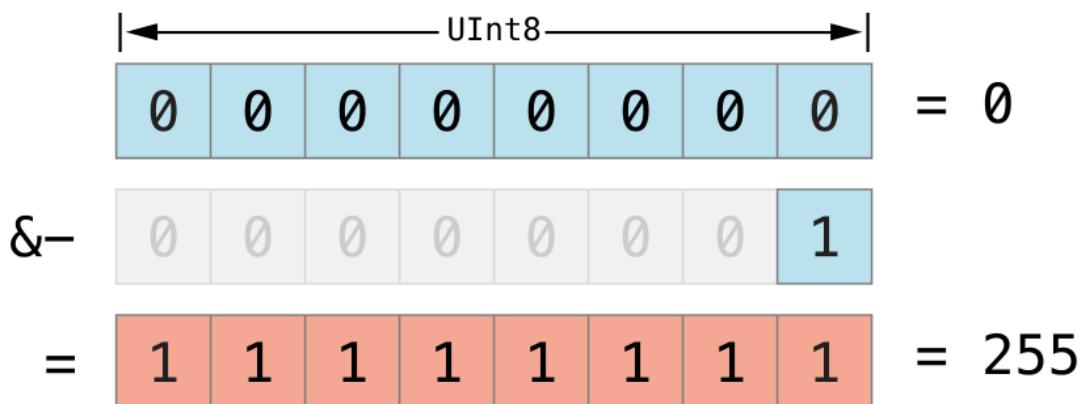
`unsignedOverflow` 被初始化为 `UInt8` 所能容纳的最大整数（`255`，以二进制表示即 `11111111`）。然后使用溢出加法运算符（`&+`）对其进行加 `1` 运算。这使得它的二进制表示正好超出 `UInt8` 所能容纳的位数，也就导致了数值的溢出，如下图所示。数值溢出后，仍然留在 `UInt8` 边界内的值是 `00000000`，也就是十进制数值的 `0`。



当允许对一个无符号整数进行下溢运算时也会产生类似的情况。这里有一个使用溢出减法运算符（`&-`）的例子：

```
var unsignedOverflow = UInt8.min  
// unsignedOverflow 等于 UInt8 所能容纳的最小整数 0  
unsignedOverflow = unsignedOverflow &- 1  
// 此时 unsignedOverflow 等于 255
```

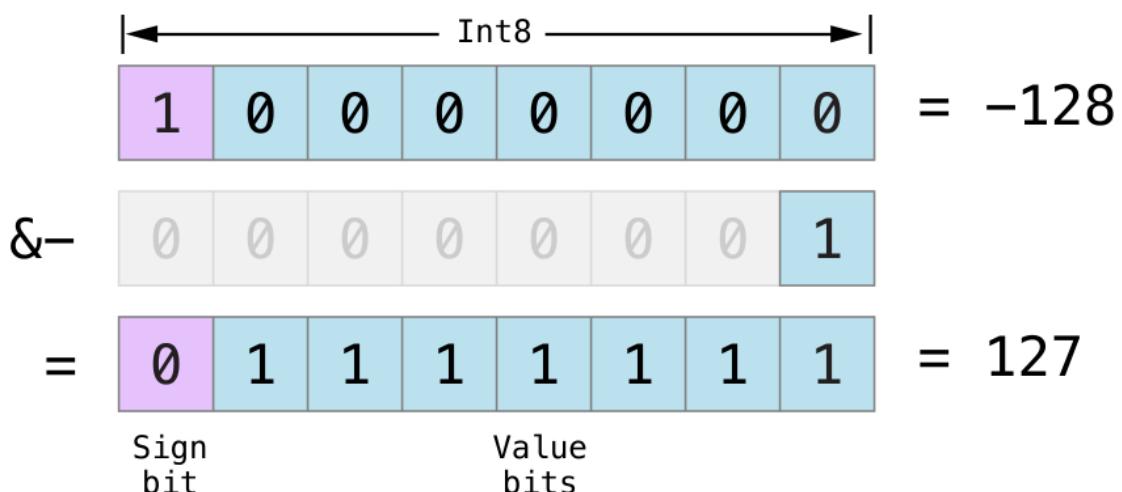
`UInt8` 型整数能容纳的最小值是 `0`，以二进制表示即 `00000000`。当使用溢出减法运算符对其进行减 `1` 运算时，数值会产生下溢并被截断为 `11111111`，也就是十进制数值的 `255`。



溢出也会发生在有符号整型上。针对有符号整型的所有溢出加法或者减法运算都是按位运算的方式执行的，符号位也需要参与计算，正如 按位左移、右移运算符 所描述的。

```
var signedOverflow = Int8.min  
// signedOverflow 等于 Int8 所能容纳的最小整数 -128  
signedOverflow = signedOverflow &- 1  
// 此时 signedOverflow 等于 127
```

`Int8` 型整数能容纳的最小值是 `-128`，以二进制表示即 `10000000`。当使用溢出减法运算符对其进行减 `1` 运算时，符号位被翻转，得到二进制数值 `01111111`，也就是十进制数值的 `127`，这个值也是 `Int8` 型整所能容纳的最大值。



对于无符号与有符号整型数值来说，当出现上溢时，它们会从数值所能容纳的最大数变成最小数。同样地，当发生下溢时，它们会从所能容纳的最小数变成最大数。

## 优先级和结合性

运算符的优先级使得一些运算符优先于其他运算符；它们会先被执行。

结合性定义了相同优先级的运算符是如何结合的，也就是说，是与左边结合为一组，还是与右边结合为一组。可以将其理解为“它们是与左边的表达式结合的”，或者“它们是与右边的表达式结合的”。

当考虑一个复合表达式的计算顺序时，运算符的优先级和结合性是非常重要的。举例来说，运算符优先级解释了为什么下面这个表达式的运算结果会是 `17`。

```
2 + 3 % 4 * 5  
// 结果是 17
```

如果你直接从左到右进行运算，你可能认为运算的过程是这样的：

- $2 + 3 = 5$
- $5 \% 4 = 1$
- $1 * 5 = 5$

但是正确答案是 `17` 而不是 `5`。优先级高的运算符要先于优先级低的运算符进行计算。与 C 语言类似，在 Swift 中，乘法运算符 (`*`) 与取余运算符 (`%`) 的优先级高于加法运算符 (`+`)。因此，它们的计算顺序要先于加法运算。

而乘法运算与取余运算的优先级相同。这时为了得到正确的运算顺序，还需要考虑结合性。乘法运算与取余运算都是左结合的。可以将这考虑成，从它们的左边开始为这两部分表达式都隐式地加上括号：

`2 + ((3 % 4) * 5)`

`(3 % 4)` 等于 `3`，所以表达式相当于：

`2 + (3 * 5)`

`3 * 5` 等于 `15`，所以表达式相当于：

`2 + 15`

因此计算结果为 `17`。

有关 Swift 标准库提供的操作符信息，包括操作符优先级组和结合性设置的完整列表，请参见 [操作符声明](#)。

### 注意

相对 C 语言和 Objective-C 来说，Swift 的运算符优先级和结合性规则更加简洁和可预测。但是，这也意味着它们相较于 C 语言及其衍生语言并不是完全一致。在对现有的代码进行移植的时候，要注意确保运算符的行为仍然符合你的预期。

## 运算符函数

类和结构体可以为现有的运算符提供自定义的实现。这通常被称为运算符重载。

下面的例子展示了如何让自定义的结构体支持加法运算符（`+`）。算术加法运算符是一个二元运算符，因为它是对两个值进行运算，同时它还可以称为中缀运算符，因为它出现在两个值中间。

例子中定义了一个名为 `Vector2D` 的结构体用来表示二维坐标向量 `(x, y)`，紧接着定义了一个可以将两个 `Vector2D` 结构体实例进行相加的运算符函数：

```
struct Vector2D {  
    var x = 0.0, y = 0.0  
}  
  
extension Vector2D {  
    static func + (left: Vector2D, right: Vector2D) -> Vector2D {  
        return Vector2D(x: left.x + right.x, y: left.y + right.y)  
    }  
}
```

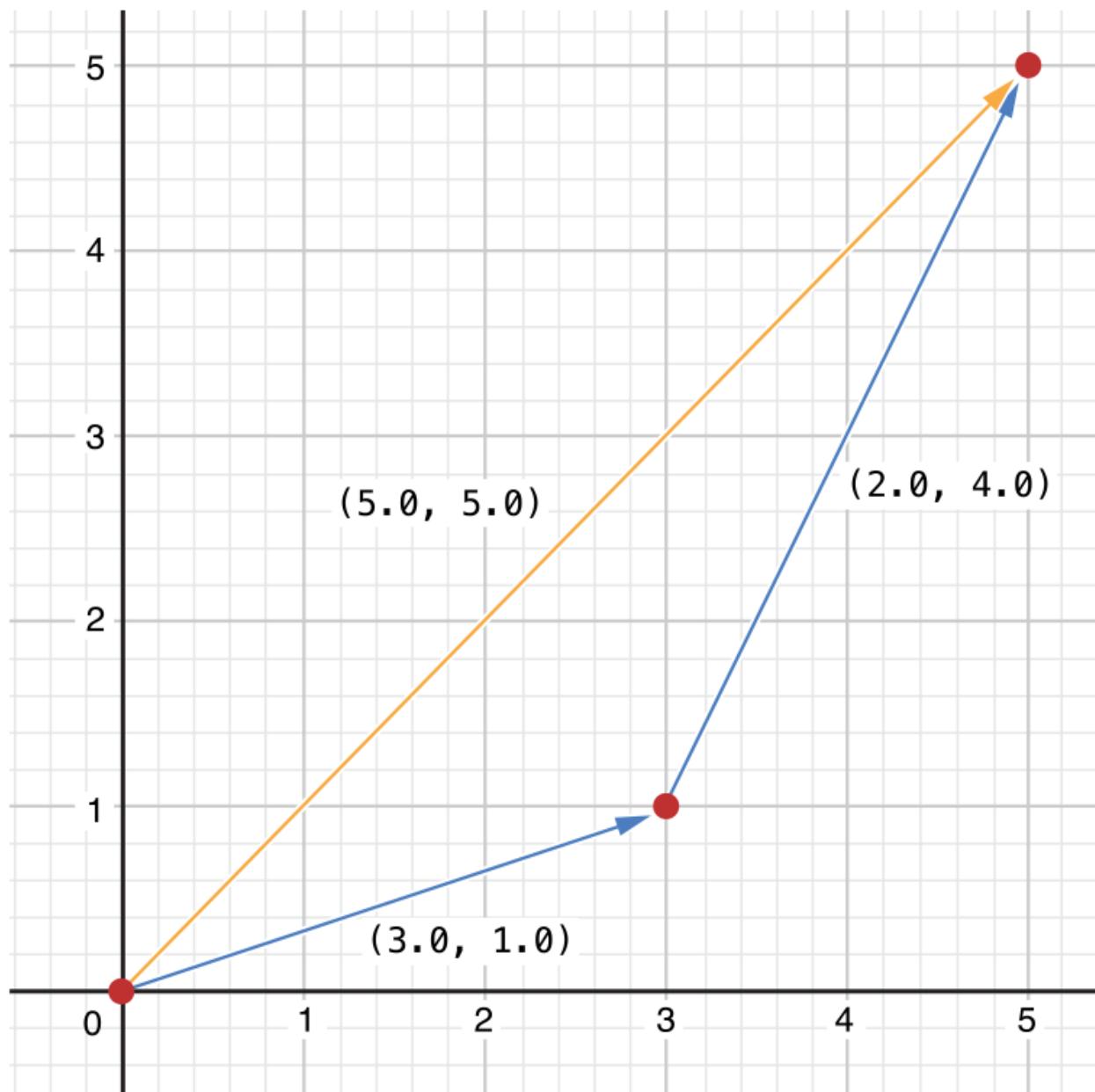
该运算符函数被定义为 `Vector2D` 上的一个类方法，并且函数的名字与它要进行重载的 `+` 名字一致。因为加法运算并不是一个向量必需的功能，所以这个类方法被定义在 `Vector2D` 的一个扩展中，而不是 `Vector2D` 结构体声明内。而算术加法运算符是二元运算符，所以这个运算符函数接收两个类型为 `Vector2D` 的参数，同时有一个 `Vector2D` 类型的返回值。

在这个实现中，输入参数分别被命名为 `left` 和 `right`，代表在 `+` 运算符左边和右边的两个 `Vector2D` 实例。函数返回了一个新的 `Vector2D` 实例，这个实例的 `x` 和 `y` 分别等于作为参数的两个实例的 `x` 和 `y` 的值之和。

这个类方法可以在任意两个 `Vector2D` 实例中间作为中缀运算符来使用：

```
let vector = Vector2D(x: 3.0, y: 1.0)
let anotherVector = Vector2D(x: 2.0, y: 4.0)
let combinedVector = vector + anotherVector
// combinedVector 是一个新的 Vector2D 实例，值为 (5.0, 5.0)
```

这个例子实现两个向量 `(3.0, 1.0)` 和 `(2.0, 4.0)` 的相加，并得到新的向量 `(5.0, 5.0)`。这个过程如下图示：



## 前缀和后缀运算符

上个例子演示了一个二元中缀运算符的自定义实现。类与结构体也能提供标准一元运算符的实现。一元运算符只运算一个值。当运算符出现在值之前时，它就是前缀的（例如 `-a`），而当它出现在值之后时，它就是后缀的（例如 `b!`）。

要实现前缀或者后缀运算符，需要在声明运算符函数的时候在 `func` 关键字之前指定 `prefix` 或者 `postfix` 修饰符：

```
extension Vector2D {  
    static prefix func - (vector: Vector2D) -> Vector2D {  
        return Vector2D(x: -vector.x, y: -vector.y)  
    }  
}
```

这段代码为 `Vector2D` 类型实现了一元运算符（`-a`）。由于该运算符是前缀运算符，所以这个函数需要加上 `prefix` 修饰符。

对于简单数值，一元负号运算符可以对它们的正负性进行改变。对于 `Vector2D` 来说，该运算将其 `x` 和 `y` 属性的正负性都进行了改变：

```
let positive = Vector2D(x: 3.0, y: 4.0)  
let negative = -positive  
// negative 是一个值为 (-3.0, -4.0) 的 Vector2D 实例  
let alsoPositive = -negative  
// alsoPositive 是一个值为 (3.0, 4.0) 的 Vector2D 实例
```

## 复合赋值运算符

复合赋值运算符将赋值运算符（`=`）与其它运算符进行结合。例如，将加法与赋值结合成加法赋值运算符（`+=`）。在实现的时候，需要把运算符的左参数设置成 `inout` 类型，因为这个参数的值会在运算符函数内直接被修改。

在下面的例子中，对 `Vector2D` 实例实现了一个加法赋值运算符函数：

```
extension Vector2D {  
    static func += (left: inout Vector2D, right: Vector2D) {  
        left = left + right  
    }  
}
```

因为加法运算在之前已经定义过了，所以在这里无需重新定义。在这里可以直接利用现有的加法运算符函数，用它来对左值和右值进行相加，并再次赋值给左值：

```
var original = Vector2D(x: 1.0, y: 2.0)  
let vectorToAdd = Vector2D(x: 3.0, y: 4.0)  
original += vectorToAdd  
// original 的值现在为 (4.0, 6.0)
```

注意

不能对默认的赋值运算符（`=`）进行重载。只有复合赋值运算符可以被重载。同样地，也无法对三元条件运算符（`a ? b : c`）进行重载。

## 等价运算符

通常情况下，自定义的类和结构体没有对等价运算符进行默认实现，等价运算符通常被称为相等运算符（`==`）与不等运算符（`!=`）。

为了使用等价运算符对自定义的类型进行判等运算，需要为“相等”运算符提供自定义实现，实现的方法与其它中缀运算符一样，并且增加对标准库 `Equatable` 协议的遵循：

```
extension Vector2D: Equatable {  
    static func == (left: Vector2D, right: Vector2D) -> Bool {  
        return (left.x == right.x) && (left.y == right.y)  
    }  
}
```

上述代码实现了“相等”运算符（`==`）来判断两个 `Vector2D` 实例是否相等。对于 `Vector2D` 来说，“相等”意味着“两个实例的 `x` 和 `y` 都相等”，这也是代码中用来进行判等的逻辑。如果你已经实现了“相等”运算符，通常情况下你并不需要自己再去实现“不等”运算符（`!=`）。标准库对于“不等”运算符提供了默认的实现，它简单地将“相等”运算符的结果进行取反后返回。

现在我们可以使用这两个运算符来判断两个 `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 {  
    print("These two vectors are equivalent.")  
}  
// 打印"These two vectors are equivalent."
```

多数简单情况下，您可以使用 Swift 为您提供的等价运算符默认实现。Swift 为以下数种自定义类型提供等价运算符的默认实现：

- 只拥有存储属性，并且它们全都遵循 `Equatable` 协议的结构体
- 只拥有关联类型，并且它们全都遵循 `Equatable` 协议的枚举
- 没有关联类型的枚举

在类型原始的声明中声明遵循 `Equatable` 来接收这些默认实现。

下面为三维位置向量 `(x, y, z)` 定义的 `Vector3D` 结构体，与 `Vector2D` 类似。由于 `x`，`y` 和 `z` 属性都是 `Equatable` 类型，`Vector3D` 获得了默认的等价运算符实现。

```
struct Vector3D: Equatable {  
    var x = 0.0, y = 0.0, z = 0.0  
}  
  
let twoThreeFour = Vector3D(x: 2.0, y: 3.0, z: 4.0)  
let anotherTwoThreeFour = Vector3D(x: 2.0, y: 3.0, z: 4.0)  
if twoThreeFour == anotherTwoThreeFour {  
    print("These two vectors are also equivalent.")  
}  
// 打印"These two vectors are also equivalent."
```

# 自定义运算符

---

除了实现标准运算符，在 Swift 中还可以声明和实现自定义运算符。可以来自定义运算符的字符列表请参考 [运算符](#)。

新的运算符要使用 `operator` 关键字在全局作用域内进行定义，同时还要指定 `prefix`、`infix` 或者 `postfix` 修饰符：

```
prefix operator +++
```

上面的代码定义了一个新的名为 `+++` 的前缀运算符。对于这个运算符，在 Swift 中并没有已知的意义，因此在针对 `Vector2D` 实例的特定上下文中，给予了它自定义的意义。对这个示例来讲，`+++` 被实现为“前缀双自增”运算符。它使用了前面定义的复合加法运算符来让矩阵与自身进行相加，从而让 `Vector2D` 实例的 `x` 属性和 `y` 属性值翻倍。你可以像下面这样通过对 `Vector2D` 添加一个 `+++` 类方法，来实现 `+++` 运算符：

```
extension Vector2D {  
    static prefix func +++(vector: inout Vector2D) -> Vector2D {  
        vector += vector  
        return vector  
    }  
}  
  
var toBeDoubled = Vector2D(x: 1.0, y: 4.0)  
let afterDoubling = +++toBeDoubled  
// toBeDoubled 现在的值为 (2.0, 8.0)  
// afterDoubling 现在的值也为 (2.0, 8.0)
```

## 自定义中缀运算符的优先级

---

每个自定义中缀运算符都属于某个优先级组。优先级组指定了这个运算符相对于其他中缀运算符的优先级和结合性。[优先级和结合性](#) 中详细阐述了这两个特性是如何对中缀运算符的运算产生影响的。

而没有明确放入某个优先级组的自定义中缀运算符将会被放到一个默认的优先级组内，其优先级高于三元运算符。

以下例子定义了一个新的自定义中缀运算符 `+-`，此运算符属于 `AdditionPrecedence` 优先组：

```
infix operator +-: AdditionPrecedence  
extension Vector2D {  
    static 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 是一个 Vector2D 实例，并且它的值为 (4.0, -2.0)
```

这个运算符把两个向量的 `x` 值相加，同时从第一个向量的 `y` 中减去第二个向量的 `y`。因为它本质上是属于“相加型”运算符，所以将它放置在 `+` 和 `-` 等默认中缀“相加型”运算符相同的优先级组中。关于 Swift 标准库提供的运算符，以及完整的运算符优先级组和结合性设置，请参考 [运算符声明](#)。而更多关于优先级组以及自定义操作符和优先级组的语法，请参考 [运算符声明](#)。

### 注意

当定义前缀与后缀运算符的时候，我们并没有指定优先级。然而，如果对同一个值同时使用前缀与后缀运算符，则后缀运算符会先参与运算。

# 关于语言参考 · GitBook

 [runoob.com/manual/gitbook/swift5/source/\\_book/chapter3/01\\_About\\_the\\_Language\\_Reference.html](https://runoob.com/manual/gitbook/swift5/source/_book/chapter3/01_About_the_Language_Reference.html)

## 关于语言参考 (About the Language Reference)

本书的这一节描述了 Swift 编程语言的形式语法。这里描述的语法是为了帮助您了解该语言的更多细节，而不是让您直接实现一个解析器或编译器。

Swift 语言相对较小，这是由于 Swift 代码中常用的类型、函数以及运算符都已经在 Swift 标准库中定义了。虽然这些类型、函数和运算符并不是 Swift 语言自身的一部分，但是它们被广泛应用于本书的讨论和代码范例中。

## 如何阅读语法

用来描述 Swift 编程语言形式语法的符号遵循下面几个约定：

- 箭头 ( $\rightarrow$ ) 用来标记语法产式，可以理解为“可由……构成”。
- 斜体文字用来表示句法类型，并出现在一个语法产式规则两侧。
- 标记语言和标点符号由固定宽度的粗体文本表示，只出现在一个语法产式规则的右侧。
- 可供选择的语法产式由竖线 ( $|$ ) 分隔。当可选用的语法产式太多时，为了阅读方便，它们将被拆分为多行语法产式规则。
- 少数情况下，标准字体文本被用来描述一个语法产生规则的右手侧内容。
- 可选的句法类型和文本标记用尾标 `opt` 来标记。

举个例子，getter-setter 方法块的语法定义如下：

getter-setter 方法块语法

getter-setter 方法块  $\rightarrow \{ \underline{\text{getter 子句}} \underline{\text{setter 子句}}_{\text{可选}} \} | \{ \underline{\text{setter 子句}} \underline{\text{getter 子句}} \}$

这个定义表明，一个 getter-setter 方法块可以由一个 getter 分句后跟一个可选的 setter 分句构成，然后用大括号括起来，或者由一个 setter 分句后跟一个 getter 分句构成，然后用大括号括起来。上述的语法产式等价于下面的两个语法产式，：

getter-setter 方法块语法

getter-setter 方法块  $\rightarrow \{ \underline{\text{getter 子句}} \underline{\text{setter 子句}}_{\text{可选}} \}$

getter-setter 方法块  $\rightarrow \{ \underline{\text{setter 子句}} \underline{\text{getter 子句}} \}$

# 词法结构 · GitBook

 [runoob.com/manual/gitbook/swift5/source/\\_book/chapter3/02\\_Lexical\\_Structure.html](http://runoob.com/manual/gitbook/swift5/source/_book/chapter3/02_Lexical_Structure.html)

## 词法结构 (Lexical Structure)

Swift 的“词法结构 (*lexical structure*)”描述了能构成该语言中有效符号 (token) 的字符序列。这些合法符号组成了语言中最底层的构建基块，并在之后的章节中用于描述语言的其他部分。一个合法符号由一个标识符 (identifier)、关键字 (keyword)、标点符号 (punctuation)、字面量 (literal) 或运算符 (operator) 组成。

通常情况下，通过考虑输入文本当中可能的最长子串，并且在随后将介绍的语法规则之下，根据随后将介绍的语法规则生成的，根据 Swift 源文件当中的字符来生成相应的“符号”。这种方法称为“最长匹配 (*longest match*)”，或者“最大适合 (*maximal munch*)”。

## 空白与注释

空白 (whitespace) 有两个用途：分隔源文件中的符号以及帮助区分运算符属于前缀还是后缀（参见 [运算符](#)），在其他情况下空白则会被忽略。以下的字符会被当作空白：空格 (U+0020)、换行符 (U+000A)、回车符 (U+000D)、水平制表符 (U+0009)、垂直制表符 (U+000B)、换页符 (U+000C) 以及空字符 (U+0000)。

注释被编译器当作空白处理。单行注释由 `//` 开始直至遇到换行符 (U+000A) 或者回车符 (U+000D)。多行注释由 `/*` 开始，以 `*/` 结束。注释允许嵌套，但注释标记必须匹配。

空白语法

空白 → 空白项 空白 可选

**whitespace-item**

空白项 → 断行符

空白项 → 注释

空白项 → 多行注释

空白项 → U+0000 , U+0009 , U+000B , U+000C 或者 U+0020

**line-break**

断行符 → U+000A

断行符 → U+000D

断行符 → U+000D 接着是 U+000A

**comment**

注释 → // 注释内容 断行符\*

**multiline-comment**

多行注释 → /\* 多行注释内容 \*/

**comment-text**

注释内容 → 注释内容项 注释内容 可选

**comment-text-item**

注释内容项 → 任何 Unicode 标量值，除了 U+000A 或者 U+000D

**multiline-commnet-text**

多行注释内容 → 多行注释内容项 多行注释内容 可选

多行注释内容项 → 多行注释

多行注释内容项 → 注释内容项

多行注释内容项 → 任何 Unicode 标量值，除了 /\* 或者 \*/

## 标识符

标识符 (*identifier*) 可以由以下的字符开始：大写或小写的字母 A 到 Z、下划线 (\_)、基本多文种平面 (Basic Multilingual Plane) 中非字符数字组合的 Unicode 字符以及基本多文种平面以外的非个人专用区字符。在首字符之后，允许使用数字和组合 Unicode 字符。

使用保留字作为标识符，需要在其前后增加反引号 (`)。例如，`class` 不是合法的标识符，但可以使用 ``class``。反引号不属于标识符的一部分，``x`` 和 `x` 表示同一标识符。

闭包中如果没有明确指定参数名称，参数将被隐式命名为 `$0`、`$1`、`$2` 等等。这些命名在闭包作用域范围内是合法的标识符。

## 标识符语法

标识符 → 头部标识符 标识符字符组 可选

标识符 → ` 头部标识符 标识符字符组 可选 `

标识符 → 隐式参数名

标识符列表 → 标识符 | 标识符, 标识符列表

### identifier-head

头部标识符 → 大写或小写字母 A - Z

头部标识符 → \_

头部标识符 → U+00A8, U+00AA, U+00AD, U+00AF, U+00B2-U+00B5, 或者 U+00B7-U+00BA

头部标识符 → U+00BC-U+00BE, U+00C0-U+00D6, U+00D8-U+00F6, 或者 U+00F8-U+00FF

头部标识符 → U+0100-U+02FF, U+0370-U+167F, U+1681-U+180D, 或者 U+180F-U+1DBF

头部标识符 → U+1E00-U+1FFF

头部标识符 → U+200B-U+200D, U+202A-U+202E, U+203F-U+2040, U+2054, 或者 U+2060-U+206F

头部标识符 → U+2070-U+20CF, U+2100-U+218F, U+2460-U+24FF, 或者 U+2776-U+2793

头部标识符 → U+2C00-U+2DFF 或者 U+2E80-U+2FFF

头部标识符 → U+3004-U+3007, U+3021-U+302F, U+3031-U+303F, 或者 U+3040-U+D7FF

头部标识符 → U+F900-U+FD3D, U+FD40-U+FDCF, U+FDF0-U+FE1F, 或者 U+FE30-U+FE44

头部标识符 → U+FE47-U+FFFD

头部标识符 → U+10000-U+1FFFD, U+20000-U+2FFFD, U+30000-U+3FFFD, 或者 U+40000-U+4FFFD

头部标识符 → U+50000-U+5FFFD, U+60000-U+6FFFD, U+70000-U+7FFFD, 或者 U+80000-U+8FFFD

头部标识符 → U+90000-U+9FFFD, U+A0000-U+AFFFD, U+B0000-U+BFFFD, 或者 U+C0000-U+CFFFD

头部标识符 → U+D0000-U+DFFFD 或者 U+E0000-U+EFFFD

标识符字符 → 数值 0 - 9

### identifier-character

标识符字符 → U+0300-U+036F, U+1DC0-U+1DFF, U+20D0-U+20FF, 或者 U+FE20-U+FE2F

标识符字符 → 头部标识符

### identifier-characters

标识符字符组 → 标识符字符 标识符字符组 可选

### implicit-parameter-name

隐式参数名 → \$ 十进制数字列表

## 关键字和标点符号

下面这些被保留的关键字不允许用作标识符，除非使用反引号转义，具体描述请参考 [标识符](#)。除了 `inout`、`var` 以及 `let` 之外的关键字可以用作某个函数声明或者函数调用当中的外部参数名，无需添加反引号转义。当一个成员与一个关键字具有相同的名称时，不需要使用反引号来转义对该成员的引用，除非在引用该成员和使用该关键字之间存在歧义 - 例如，`self`，`Type` 和 `Protocol` 在显式的成员表达式中具有特殊的含义，因此它们必须在该上下文中使用反引号进行转义。

- 用在声明中的关键字：  
  `associatedtype`、`class`、`deinit`、`enum`、`extension`、`fileprivate`、`func`、`import`、`init`、`inout`、`internal`、`let`、`open`、`oper` 以及 `var`。
- 用在语句中的关键  
  字：`break`、`case`、`continue`、`default`、`defer`、`do`、`else`、`fallthrough`、`for`、`guard`、`if`、`in`、`repeat`、`return`、`switch`、  
  以及 `while`。
- 用在表达式和类型中的关键  
  字：`as`、`Any`、`catch`、`false`、`is`、`nil`、`rethrows`、`super`、`self`、`Self`、`throw`、`throws`、`true` 以及 `try`。
- 用在模式中的关键字：`_`。
- 以井字号 (`#`) 开头的关键  
  字：`#available`、`#colorLiteral`、`#column`、`#else`、`#elseif`、`#endif`、`#error`、`#file`、`#fileLiteral`、`#function`、`#if`、`#imag` 及 `#warning`。

- 特定上下文中被保留的关键字：  
`associativity`、`convenience`、`dynamic`、`didSet`、`final`、`get`、`infix`、`indirect`、`lazy`、`left`、`mutating`、`none`、`nonmutating`以及`willSet`。这些关键字在特定上下文之外可以被用做标识符。

以下符号被当作保留符号，不能用于自定义运算符：(、)、{、}、[、]、.、,、:、;、=、@、#、&（作为前缀运算符）、->、`、?、!（作为后缀运算符）。

## 字面量

字面量 (*literal*) 用来表示源码中某种特定类型的值，比如一个数字或字符串。

下面是字面量的一些示例：

```
42          // 整数字面量  
3.14159    // 浮点数字面量  
"Hello, world!" // 字符串字面量  
true        // 布尔值字面量
```

字面量本身并不包含类型信息。事实上，一个字面量会被解析为拥有无限的精度，然后 Swift 的类型推导会尝试去推导出这个字面量的类型。比如，在`let x: Int8 = 42` 这个声明中，Swift 使用了显式类型注解 (`: Int8`) 来推导出 `42` 这个整数字面量的类型是 `Int8`。如果没有可用的类型信息，Swift 则会从标准库中定义的字面量类型中推导出一个默认的类型。整数字面量的默认类型是 `Int`，浮点数字面量的默认类型是 `Double`，字符串字面量的默认类型是 `String`，布尔值字面量的默认类型是 `Bool`。比如，在`let str = "Hello, world"` 这个声明中，字符串 `"Hello, world"` 的默认推导类型就是 `String`。

当为一个字面量值指定了类型注解的时候，这个标注的类型必须能通过这个字面量值实例化。也就是说，这个类型必须符合这些 Swift 标准库协议中的一个：整数字面量的 `IntegerLiteralConvertible` 协议、浮点数字面量的 `FloatingPointLiteralConvertible` 协议、字符串字面量的 `StringLiteralConvertible` 协议以及布尔值字面量的 `BooleanLiteralConvertible` 协议。比如，`Int8` 符合 `IntegerLiteralConvertible` 协议，因此它能在 `let x: Int8 = 42` 这个声明中作为整数字面量 `42` 的类型注解。

### 字面量语法

```
字面量 → 数值字面量 | 字符串字面量 | 布尔值字面量 | nil 字面量  
数值字面量 → -可选 整数字面量 | -可选 浮点数字面量  
布尔值字面量 → true | false  
nil 字面量 → nil
```

## 整数字面量

整数字面量 (*Integer Literals*) 表示未指定精度整数的值。整数字面量默认用十进制表示，可以加前缀来指定其他的进制。二进制字面量加 `0b`，八进制字面量加 `0o`，十六进制字面量加 `0x`。

十进制字面量包含数字 `0` 至 `9`。二进制字面量只包含 `0` 或 `1`，八进制字面量包含数字 `0` 至 `7`，十六进制字面量包含数字 `0` 至 `9` 以及字母 `A` 至 `F`（大小写均可）。

负整数的字面量在整数字面量前加负号 `-`，比如 `-42`。

整型字面量可以使用下划线 (`_`) 来增加数字的可读性，下划线会被系统忽略，因此不会影响字面量的值。同样地，也可以在数字前加 `0`，这同样也会被系统所忽略，并不会影响字面量的值。

除非特别指定，整数字面量的默认推导类型为 Swift 标准库类型中的 `Int`。Swift 标准库还定义了其他不同长度以及是否带符号的整数类型，请参考 [整数](#)。

整数字面量语法

### integer-literal

---

整数字面量 → 二进制字面量

整数字面量 → 八进制字面量

整数字面量 → 十进制字面量

整数字面量 → 十六进制字面量

### binary-literal

---

二进制字面量 → 0b 二进制数字 二进制字面量字符组 可选

### binary-digit

---

二进制数字 → 数值 0 到 1

二进制字面量字符 → 二进制数字 | \_

### binary-literal-characters

---

二进制字面量字符组 → 二进制字面量字符 二进制字面量字符组 可选

### octal-literal

---

八进制字面量 → 0o 八进字数字 八进制字符组 可选

### octal-digit

---

八进字数字 → 数值 0 到 7

八进制字符 → 八进字数字 | \_

### octal-literal-characters

---

八进制字符组 → 八进制字符 八进制字符组 可选

### decimal-literal

---

十进制字面量 → 十进制数字 十进制字符组 可选

### decimal-digit

---

十进制数字 → 数值 0 到 9

### decimal-literal-characters

---

十进制数字组 → 十进制数字 十进制数字组 可选

十进制字符 → 十进制数字 | \_

十进制字符组 → 十进制字符 十进制字符组 可选

### hexadecimal-literal

---

十六进制字面量 → 0x 十六进制数字 十六进制字面量字符组 可选

### hexadecimal-digit

---

十六进制数字 → 数值 0 到 9, 字母 a 到 f, 或 A 到 F

十六进制字符 → 十六进制数字 | \_

### hexadecimal-literal-characters

---

十六进制字面量字符组 → 十六进制字符 十六进制字面量字符组 可选

## 浮点数字面量

---

浮点数字面量 (*Floating-point literals*) 表示未指定精度浮点数的值。

浮点数字面量默认用十进制表示 (无前缀) , 也可以用十六进制表示 (加前缀 `0x` )。

十进制浮点数字面量由十进制数字串后跟小数部分或指数部分 (或两者皆有) 组成。十进制小数部分由小数点 (.) 后跟十进制数字串组成。指数部分由大写或小写字母 `e` 为前缀后跟十进制数字串组成, 这串数字表示 `e` 之前的数量乘以  $10$  的几次方。例如：`1.25e2` 表示  $1.25 \times 10^2$  , 也就是 `125.0` ; 同样, `1.25e-2` 表示  $1.25 \times 10^{-2}$  , 也就是 `0.0125` 。

十六进制浮点数字面量由前缀 `0x` 后跟可选的十六进制小数部分以及十六进制指数部分组成。十六进制小数部分由小数点后跟十六进制数字串组成。指数部分由大写或小写字母 `p` 为前缀后跟十进制数字串组成, 这串数字表示 `p` 之前的数量乘以  $2$  的几次方。例如：`0xFp2` 表示  $15 \times 2^2$  , 也就是 `60` ; 同样, `0xFp-2` 表示  $15 \times 2^{-2}$  , 也就是 `3.75` 。

负数的浮点数字面量由负号（`-`）和浮点数字面量组成，例如 `-42.5`。

浮点数字面量允许使用下划线（`_`）来增强数字的可读性，下划线会被系统忽略，因此不会影响字面量的值。同样地，也可以在数字前加 `0`，并不会影响字面量的值。

除非特别指定，浮点数字面量的默认推导类型为 Swift 标准库类型中的 `Double`，表示 64 位浮点数。Swift 标准库也定义了 `Float` 类型，表示 32 位浮点数。

浮点数字面量语法

#### **floating-point-literal**

浮点数字面量 → 十进制字面量 十进制分数 可选 十进制指数 可选

浮点数字面量 → 十六进制字面量 十六进制分数 可选 十六进制指数

#### **decimal-fraction**

十进制分数 → . 十进制字面量

#### **decimal-exponent**

十进制指数 → 十进制指数 e 正负号 可选 十进制字面量

#### **hexadecimal-fraction**

十六进制分数 → . 十六进制数字 十六进制字面量字符组 可选

#### **hexadecimal-exponent**

十六进制指数 → 十六进制指数 p 正负号 可选 十进制字面量

#### **floating-point-e**

十进制指数 e → `e` | `E`

#### **floating-point-p**

十六进制指数 p → `p` | `P`

#### **sign**

正负号 → `+` | `-`

## **字符串字面量**

字符串字面量是被引号包括的一串字符组成。单行字符串字面量被包在双引号中的一串字符组成，形式如下：

`"` 字符 `"`

字符串字面量中不能包含未转义的双引号（`"`）、未转义的反斜线（`\`）、回车符、换行符。

多行字符串字面量被包在三个双引号中的一串字符组成，形式如下：

`"""` 字符 `"""`

与单行字符串字面量不同的是，多行字符串字面量可以包含不转义的双引号（`"`），回车以及换行。它不能包含三个未转义的连续双引号。

`"""` 之后的回车或者换行开始多行字符串字面量，不是字符串的一部分。`"""` 之前回车或者换行结束字面量，也不是字符串的一部分。要让多行字符串字面量的开始或结束带有换行，就在第一行或者最后一行写一个空行。

多行字符串字面量可以使用任何空格或制表符组合进行缩进；这些缩进不会包含在字符串中。`"""` 的结束符号决定了缩进：字面量中的任何一个非空行必须起始于多行字符串字面量结束符号的前面；空格和制表符不会被转换。你可以包在缩进后含额外的空格和制表符；这些空格和制表符会在字符串中出现。

多行字符串字面量中的一行结束使用规范化的换行符号。尽管你的源代码混用了回车和换行符，字符串中所有的行结束都必须一样。

在多行字符串字面量里，在行末用反斜线（`\`）可以省略字符串行间中断。反斜线之间的空白和行间中断也可以省略。你可以在你的代码里用这种语法硬包裹多行字符串字面量，不需要改变产生的字符串的值。

可以在字符串字面量中使用的转义特殊符号如下：

- 空字符 `\0`
- 反斜线 `\\"`
- 水平制表符 `\t`
- 换行符 `\n`
- 回车符 `\r`
- 双引号 `\"`
- 单引号 `\'`
- Unicode 标量 `\u{n}`，n 为一到八位的十六进制数字

字符串字面量允许在反斜杠（\）后的括号（）中插入表达式的值。插入表达式可以包含字符串字面量，但不能包含未转义的反斜线（\）、回车符以及换行符。

例如，以下所有字符串字面量的值都是相同的：

```
"1 2 3"  
"1 2 \("3")"  
"1 2 \(3)"  
"1 2 \(1 + 2)"  
let x = 3; "1 2 \(\x)"
```

可以使用一对或多对扩展分隔符（#）包裹字符串进行分隔，被分隔的字符串的形式如下所示：

```
#" characters "#  
#####  
characters  
######
```

特殊字符在被分隔符分隔的结果字符串中会展示为普通字符，而不是特殊字符。你可以使用扩展分隔符来创建一些具有特殊效果的字符串。例如，生成字符串插值，启动或终止转义序列（字符串）。

以下所示，由字符串字面量和扩展分隔符所创建的字符串是等价的：

```
let string = #"\(x) \" \u{2603}"#  
let escaped = "\\\(x) \\ \" \u{2603}"  
print(string)  
// Prints "\u{2603}"  
print(string == escaped)  
// Prints "true"
```

如果在一个字符串中使用多对扩展分隔符，请不要在分隔符之间使用空格。

```
print(####"Line 1#\#\#\nLine 2"###) // OK  
print(# # #Line 1#\# # #nLine 2#\# # #) // Error
```

使用扩展分隔符创建的多行字符串字面量与普通多行字符串字面量具有相同的缩进要求。

字符串字面量的默认推导类型为 `String`。更多有关 `String` 类型的信息请参考 [字符串和字符](#) 以及 [字符串结构参考](#)。

用 `+` 操作符连接的字符型字面量是在编译时进行连接的。比如下面的 `textA` 和 `textB` 是完全一样的，`textA` 没有任何运行时的连接操作。

```
let textA = "Hello " + "world"  
let textB = "Hello world"
```

字符串字面量语法

字符串字面量 → 静态字符串字面量 | 插值字符串字面量

字符串开分隔定界符 → 字符串扩展分隔符"

字符串闭分隔定界符 → " 字符串扩展分隔符 可选

#### static-string-literal

静态字符串字面量 → 字符串开分隔定界符 引用文本 可选 字符串闭分隔定界符

静态字符串字面量 → 多行字符串开分隔定界符 多行引用文本 可选 多行字符串闭分隔定界符

多行字符串开分隔定界符 → 字符串扩展分隔符 """"

多行字符串闭分隔定界符 → """" 字符串扩展分隔符 """"

#### extended-string-literal-delimiter

字符串扩展分隔符 → # 字符串扩展分隔符 可选

#### quoted-text

引用文本 → 引用文本项 引用文本 可选

#### quoted-text-item

引用文本项 → 转义字符

引用文本项 → 除了 "、\、U+000A、U+000D 以外的所有 Unicode 字符

#### multiline-quoted-text

多行引用文本 → 多行引用文本项 多行引用文本 可选

#### multiline-quoted-text-item

多行引用文本项 转义字符 可选

#### multiline-quoted-text

多行引用文本 → 除了 \ 以外的任何 Unicode 标量值

多行引用文本 → 转义换行

#### interpolated-string-literal

插值字符串字面量 → 字符串开分隔定界符 插值文本 可选 字符串闭分隔定界符

插值字符串字面量 → 多行字符串开分隔定界符 插值文本 可选 多行字符串闭分隔定界符

#### interpolated-text

插值文本 → 插值文本项 插值文本 可选

#### interpolated-text-item

插值文本项 → \\*\\*(表达式)\* | /引用文本项\*](#quoted-text-item)

多行插值文本 → 多行插值文本项 多行插值文本 可选

多行插值文本项 → \(\) 表达式 | 多行引用文本项

#### escape-sequence

转义序列 → \ 字符串扩展分隔符

#### escaped-character

转义字符 → 转义序列0 | 转义序列\ | 转义序列t | 转义序列n | 转义序列r | 转义序列\" | 转义序列'

转义字符 → 转义序列u { unicode 标量数字 }

#### unicode-scalar-digits

unicode 标量数字 → 一到八位的十六进制数字

#### escaped-newline

转义换行符 → 转义序列 空白 可选 断行符

## 运算符

Swift 标准库定义了许多可供使用的运算符，其中大部分在 [基础运算符](#) 和 [高级运算符](#) 中进行了阐述。这一小节将描述哪些字符能用于自定义运算符。

自定义运算符可以由以下其中之一的 ASCII 字符 `/`、`=`、`-`、`+`、`!`、`*`、`%`、`<`、`>`、`&`、`|`、`^`、`?` 以及 `~`，或者后面语法中规定的任何一个 Unicode 字符（其中包含了数学运算符、零散符号 (*Miscellaneous Symbols*) 以及印刷符号 (Dingbats) 之类的 Unicode 块）开始。在第一个字符之后，允许使用组合型 Unicode 字符。

您也可以以点号 (.) 开头来定义自定义运算符。这些运算符可以包含额外的点，例如 `.+.`。如果某个运算符不是以点号开头的，那么它就无法再包含另外的点号了。例如，`+.+` 就会被看作为一个 `+` 运算符后面跟着一个 `.+` 运算符。

虽然您可以用问号 `?` 来自定义运算符，但是这个运算符不能只包含单独的一个问号。此外，虽然运算符可以包含一个惊叹号 `!`，但是前缀运算符不能够以问号或者惊叹号开头。

#### 注意

以下这些标记 `=`、`->`、`//`、`/*`、`*/`、`.`、`<` (前缀运算符)、`&`、`?`、`?` (中缀运算符)、`>` (后缀运算符)、`!`、`?` 是被系统保留的。这些符号不能被重载，也不能用于自定义运算符。

运算符两侧的空白被用来区分该运算符是否为前缀运算符、后缀运算符或二元运算符。规则总结如下：

- 如果运算符两侧都有空白或两侧都无空白，将被看作二元运算符。例如：`a+++b` 和 `a +++ b` 当中的 `+++` 运算符会被看作二元运算符。
- 如果运算符只有左侧空白，将被看作一元前缀运算符。例如 `a +++ b` 中的 `+++` 运算符会被看作是一元前缀运算符。
- 如果运算符只有右侧空白，将被看作一元后缀运算符。例如 `a +++ b` 中的 `+++` 运算符会被看作是一元后缀运算符。
- 如果运算符左侧没有空白并紧跟 `.`，将被看作一元后缀运算符。例如 `a +++.b` 中的 `+++` 运算符会被视为一元后缀运算符 (即上式被视为 `a +++ .b` 而不是 `a +++ .b` )。

鉴于这些规则，运算符前的字符 `(`、`[` 和 `{`，运算符后的字符 `)`、`]` 和 `}`，以及字符 `,`、`:` 和 `:` 都被视为空白。

以上规则需注意一点，如果预定义运算符 `!` 或 `?` 左侧没有空白，则不管右侧是否有空白都将被看作后缀运算符。如果将 `?` 用作可选链式调用运算符，左侧必须无空白。如果用于条件运算符 `?:`，必须两侧都有空白。

在某些特定的设计中，以 `<` 或 `>` 开头的运算符会被分离成两个或多个符号，剩余部分可能会以同样的方式被再次分离。因此，在 `Dictionary<String, Array<Int>>` 中没有必要添加空白来消除闭合字符 `>` 的歧义。在这个例子中，闭合字符 `>` 不会被视为单独的符号，因而不会被错误解析为 `>>` 运算符。

要学习如何自定义运算符，请参考 [自定义运算符](#) 和 [运算符声明](#)。要学习如何重载运算符，请参考 [运算符函数](#)。

## 运算符语法

运算符 → 头部运算符 运算符字符组 可选

运算符 → 头部点运算符 点运算符字符组

### **operator-head**

---

头部运算符 → / | = | - | + | ! | \* | % | < | > | & | || | ^ | ~ | ?

头部运算符 → U+00A1–U+00A7

头部运算符 → U+00A9 或 U+00AB

头部运算符 → U+00AC 或 U+00AE

头部运算符 → U+00B0–U+00B1 , U+00B6 , U+00BB , U+00BF , U+00D7 , 或 U+00F7

头部运算符 → U+2016–U+2017 或 U+2020–U+2027

头部运算符 → U+2030–U+203E

头部运算符 → U+2041–U+2053

头部运算符 → U+2055–U+205E

头部运算符 → U+2190–U+23FF

头部运算符 → U+2500–U+2775

头部运算符 → U+2794–U+2BFF

头部运算符 → U+2E00–U+2E7F

头部运算符 → U+3001–U+3003

头部运算符 → U+3008–U+3030

### **operator-character**

---

运算符字符 → 头部运算符

运算符字符 → U+0300–U+036F

运算符字符 → U+1DC0–U+1DFF

运算符字符 → U+20D0–U+20FF

运算符字符 → U+FE00–U+FE0F

运算符字符 → U+FE20–U+FE2F

运算符字符 → U+E0100–U+E01EF

### **operator-characters**

---

运算符字符组 → 运算符字符 运算符字符组 可选

### **dot-operator-head**

---

头部点运算符 → ..

### **dot-operator-character**

---

点运算符字符 → . | 运算符字符

### **dot-operator-characters**

---

点运算符字符组 → 点运算符字符 点运算符字符组 可选

二元运算符 → 运算符

前缀运算符 → 运算符

后缀运算符 → 运算符



## 类型 (Types)

---

Swift 语言存在两种类型：命名型类型和复合型类型。命名型类型是指定义时可以给定名字的类型。命名型类型包括类、结构体、枚举和协议。比如，一个用户定义类 `MyClass` 的实例拥有类型 `MyClass`。除了用户定义的命名型类型，Swift 标准库也定义了很多常用的命名型类型，包括那些表示数组、字典和可选值的类型。

那些通常被其它语言认为是基本或原始的数据型类型，比如表示数字、字符和字符串的类型，实际上就是命名型类型，这些类型在 Swift 标准库中是使用结构体来定义和实现的。因为它们是命名型类型，因此你可以按照 [扩展](#) 和 [扩展声明](#) 中讨论的那样，声明一个扩展来增加它们的行为以满足你程序的需求。

复合型类型是没有名字的类型，它由 Swift 本身定义。Swift 存在两种复合型类型：函数类型和元组类型。一个复合型类型可以包含命名型类型和其它复合型类型。例如，元组类型 `(Int, (Int, Int))` 包含两个元素：第一个是命名型类型 `Int`，第二个是另一个复合型类型 `(Int, Int)`。

你可以在命名型类型和复合型类型使用小括号。但是在类型旁加小括号没有任何作用。举个例子，`(Int)` 等同于 `Int`。

本节讨论 Swift 语言本身定义的类型，并描述 Swift 中的类型推断行为。

### type

---

类型语法

类型 → 函数类型

类型 → 数组类型

类型 → 字典类型

类型 → 类型标识

类型 → 元组类型

类型 → 可选类型

类型 → 隐式解析可选类型

类型 → 协议合成类型

类型 → 不透明类型

类型 → 元型类型

类型 → 自身类型

类型 → Any

类型 → ( 类型 )

## 类型注解

类型注解显式地指定一个变量或表达式的类型。类型注解始于冒号 : 终于类型，比如下面两个例子：

```
let someTuple: (Double, Double) = (3.14159, 2.71828)
func someFunction(a: Int) { /* ... */ }
```

在第一个例子中，表达式 `someTuple` 的类型被指定为 `(Double, Double)`。在第二个例子中，函数 `someFunction` 的参数 `a` 的类型被指定为 `Int`。

类型注解可以在类型之前包含一个类型特性的可选列表。

| 类型注解语法

## type-annotation

| 类型注解 → : 特性列表 可选 输入输出参数 可选 类型

## 类型标识符

类型标识符引用命名型类型，还可引用命名型或复合型类型的别名。

大多数情况下，类型标识符引用的是与之同名的命名型类型。例如类型标识符 `Int` 引用命名型类型 `Int`，同样，类型标识符 `Dictionary<String, Int>` 引用命名型类型 `Dictionary<String, Int>`。

在两种情况下类型标识符不引用同名的类型。情况一，类型标识符引用的是命名型或复合型类型的类型别名。比如，在下面的例子中，类型标识符使用 `Point` 来引用元组 `(Int, Int)`：

```
typealias Point = (Int, Int)
let origin: Point = (0, 0)
```

情况二，类型标识符使用点语法（`.`）来表示在其它模块或其它类型嵌套内声明的命名型类型。例如，下面例子中的类型标识符引用在 `ExampleModule` 模块中声明的命名型类型 `MyType`：

```
var someValue: ExampleModule.MyType
```

| 类型标识符语法

## type-identifier

---

| 类型标识符 → 类型名称泛型参数子句可选 | 类型名称泛型参数子句可选 · 类型标识符

## type-name

---

| 类型名称 → 标识符

## 元组类型

---

元组类型是使用括号括起来的零个或多个类型，类型间用逗号隔开。

你可以使用元组类型作为一个函数的返回类型，这样就可以使函数返回多个值。你也可以命名元组类型中的元素，然后用这些名字来引用每个元素的值。元素的名字由一个标识符紧跟一个冒号 `(:)` 组成。函数和多返回值 章节里有一个展示上述特性的例子。

当一个元组类型的元素有名字的时候，这个名字就是类型的一部分。

```
var someTuple = (top: 10, bottom: 12) // someTuple 的类型为 (top: Int, bottom: Int)
someTuple = (top: 4, bottom: 42) // 正确：命名类型匹配
someTuple = (9, 99)           // 正确：命名类型被自动推断
someTuple = (left: 5, right: 5) // 错误：命名类型不匹配
```

所有的元组类型都包含两个及以上元素，除了 `Void`。`Void` 是空元组类型 `()` 的别名。

| 元组类型语法

## tuple-type

---

| 元组类型 → () | (元组类型元素, 元组类型元素列表)

## tuple-type-element-list

---

| 元组类型元素列表 → 元组类型元素 | 元组类型元素, 元组类型元素列表

## tuple-type-element

---

| 元组类型元素 → 元素名 类型注解 | 类型

## element-name

---

| 元素名 → 标识符

# 函数类型

---

函数类型表示一个函数、方法或闭包的类型，它由参数类型和返回值类型组成，中间用箭头 (`->`) 隔开：

| ( 参数类型 ) -> ( 返回值类型 )

参数类型是由逗号间隔的类型列表。由于返回值类型可以是元组类型，所以函数类型支持多返回值的函数与方法。

你可以对参数类型为 `() -> T` (其中 `T` 是任何类型) 的函数使用 `autoclosure` 特性。这会自动将参数表达式转化为闭包，表达式的结果即闭包返回值。这从语法结构上提供了一种便捷：延迟对表达式的求值，直到其值在函数体中被调用。以自动闭包做为参数的函数类型的例子详见 [自动闭包](#)。

函数类型可以拥有一个可变长参数作为参数类型中的最后一个参数。从语法角度上讲，可变长参数由一个基础类型名字紧随三个点 (`...`) 组成，如 `Int...`。可变长参数被认为是一个包含了基础类型元素的数组。即 `Int...` 就是 `[Int]`。关于使用可变长参数的例子，请参阅 [可变参数](#)。

为了指定一个 `in-out` 参数，可以在参数类型前加 `inout` 前缀。但是你不可以对可变长参数或返回值类型使用 `inout`。关于这种参数的详细讲解请参阅 [输入输出参数](#)。

如果一个函数类型只有一个形式参数而且形式参数的类型是元组类型，那么元组类型在写函数类型的时候必须用圆括号括起来。比如说，`((Int, Int)) -> Void` 是接收一个元组 `(Int, Int)` 作为形式参数并且不返回任何值的函数类型。与此相对，不加括号的 `(Int, Int) -> Void` 是一个接收两个 `Int` 作为形式参数并且不返回任何值的函数类型。相似地，因为 `Void` 是空元组类型 `()` 的别名，函数类型 `(Void)-> Void` 与 `(())->()` 是一样的 - 一个将空元组作为唯一参数的函数。但这些类型和无变量的函数类型 `() -> ()` 是不一样的。

函数和方法中的变量名并不是函数类型的一部分。例如：

```
func someFunction(left: Int, right: Int) {}
func anotherFunction(left: Int, right: Int) {}
func functionWithDifferentLabels(top: Int, bottom: Int) {}

var f = someFunction // 函数 f 的类型为 (Int, Int) -> Void, 而不是 (left: Int, right: Int) -> Void.
```

```
f = anotherFunction      // 正确
f = functionWithDifferentLabels // 正确
```

```
func functionWithDifferentArgumentTypes(left: Int, right: String) {}
f = functionWithDifferentArgumentTypes // 错误
```

```
func functionWithDifferentNumberOfArguments(left: Int, right: Int, top: Int) {}
f = functionWithDifferentNumberOfArguments // 错误
```

由于变量标签不是函数类型的一部分，你可以在写函数类型的时候省略它们。

```
var operation: (lhs: Int, rhs: Int) -> Int    // 错误
var operation: (_ lhs: Int, _ rhs: Int) -> Int // 正确
var operation: (Int, Int) -> Int              // 正确
```

如果一个函数类型包涵多个箭头 ( $->$ )，那么函数类型将从右向左进行组合。例如，函数类型 `(Int) -> (Int) -> Int` 可以理解为 `(Int) -> ((Int) -> Int)`，也就是说，该函数类型的参数为 `Int` 类型，其返回类型是一个参数类型为 `Int`，返回类型为 `Int` 的函数。

函数类型若要抛出错误就必须使用 `throws` 关键字来标记，若要重抛错误则必须使用 `rethrows` 关键字来标记。`throws` 关键字是函数类型的一部分，非抛出函数是抛出函数函数的一个子类型。因此，在使用抛出函数的地方也可以使用不抛出函数。抛出和重抛函数的相关描述见章节 [抛出函数与方法](#) 和 [重抛函数与方法](#)。

## 对非逃逸闭包的限制

---

当非逃逸闭包函数是参数时，不能存储在属性、变量或任何 `Any` 类型的常量中，因为这可能导致值的逃逸。

当非逃逸闭包函数是参数时，不能作为参数传递到另一个非逃逸闭包函数中。这样的限制可以让 Swift 在编译时就完成更多的内存访问冲突检查，而不是在运行时。举个例子：

```
let external: (Any) -> Void = { _ in () }
func takesTwoFunctions(first: (Any) -> Void, second: (Any) -> Void) {
    first(first) // 错误
    second(second) // 错误

    first(second) // 错误
    second(first) // 错误

    first(external) // 正确
    external(first) // 正确
}
```

在上面代码里，`takesTwoFunctions(first:second:)` 的两个参数都是函数。它们都没有标记为 `@escaping`，因此它们都是非逃逸的。

上述例子里的被标记为“错误”的四个函数调用会产生编译错误。因为参数 `first` 和 `second` 是非逃逸函数，它们不能够作为参数被传递到另一个非闭包函数。相对的，标记“正确”的两个函数不会产生编译错误。这些函数调用不会违反限制，因为 `external` 不是 `takesTwoFunctions(first:second:)` 的参数之一。

如果你需要避免这个限制，标记其中之一的参数为逃逸，或者使用 `withoutActuallyEscaping(_:do:)` 函数临时地转换非逃逸函数的其中一个参数为逃逸函数。关于避免内存访问冲突，可以参阅 [内存安全](#)。

## 函数类型语法

### function-type

函数类型 → 特性列表 可选 函数类型子句 `throws` 可选 -> 类型  
函数类型 → 特性列表 可选 函数类型子句 `rethrows` -> 类型

### function-type-argument-clause

函数类型子句 → ()  
函数类型子句 → ( 函数类型参数列表 ... 可选 )

### function-type-argument-list

函数类型参数列表 → 函数类型参数 | 函数类型参数 , 函数类型参数列表

### function-type-argument

函数类型参数 → 特性列表 可选 输入输出参数 可选 类型 | 参数标签 类型注解

### argument-label

参数标签 → 标识符

## 数组类型

Swift 语言为标准库中定义的 `Array<Element>` 类型提供了如下语法糖：

[ 类型 ]

换句话说，下面两个声明是等价的：

```
let someArray: Array<String> = ["Alex", "Brian", "Dave"]
let someArray: [String] = ["Alex", "Brian", "Dave"]
```

上面两种情况下，常量 `someArray` 都被声明为字符串数组。数组的元素也可以通过下标访问：`someArray[0]` 是指第 0 个元素 `"Alex"`。

你也可以嵌套多对方括号来创建多维数组，最里面的方括号中指明数组元素的基本类型。比如，下面例子中使用三对方括号创建三维整数数组：

```
var array3D: [[[Int]]] = [[[1, 2], [3, 4]], [[5, 6], [7, 8]]]
```

访问一个多维数组的元素时，最左边的下标指向最外层数组的相应位置元素。接下来往右的下标指向第一层嵌入的相应位置元素，依次类推。这就意味着，在上面的例子中，`array3D[0]` 是 `[[1, 2], [3, 4]]`，`array3D[0][1]` 是 `[3, 4]`，`array3D[0][1][1]` 则是 `4`。

关于 Swift 标准库中 `Array` 类型的详细讨论，请参阅 [数组](#)。

| 数组类型语法

## array-type

---

| 数组类型 → [类型]

## 字典类型

---

Swift 语言为标准库中定义的 `Dictionary<Key, Value>` 类型提供了如下语法糖：

| [ 键类型 : 值类型 ]

换句话说，下面两个声明是等价的：

```
let someDictionary: [String: Int] = ["Alex": 31, "Paul": 39]
let someDictionary: Dictionary<String, Int> = ["Alex": 31, "Paul": 39]
```

上面两种情况，常量 `someDictionary` 被声明为一个字典，其中键为 `String` 类型，值为 `Int` 类型。

字典中的值可以通过下标来访问，这个下标在方括号中指明了具体的键：`someDictionary["Alex"]` 返回键 `Alex` 对应的值。通过下标访问会获取对应值的可选类型。如果键在字典中不存在的话，则这个下标返回 `nil`。

字典中键的类型必须符合 Swift 标准库中的 `Hashable` 协议。

关于 Swift 标准库中 `Dictionary` 类型的详细讨论，请参阅 [字典](#)。

| 字典类型语法

## dictionary-type

---

| 字典类型 → [类型: 类型]

## 可选类型

---

Swift 定义后缀 `?` 来作为标准库中定义的命名型类型 `Optional<Wrapped>` 的语法糖。换句话说，下面两个声明是等价的：

```
var optionalInteger: Int?  
var optionalInteger: Optional<Int>
```

在上述两种情况下，变量 `optionalInteger` 都被声明为可选整型类型。注意在类型和 `?` 之间没有空格。

类型 `Optional<Wrapped>` 是一个枚举，有两个成员，`none` 和 `some(Wrapped)`，用来表示可能有也可能没有的值。任意类型都可以被显式地声明（或隐式地转换）为可选类型。如果你在声明可选变量或属性的时候没有提供初始值，它的值则会自动赋为默认值 `nil`。

如果一个可选类型的实例包含一个值，那么你就可以使用后缀运算符 `!` 来获取该值，正如下面描述的：

```
optionalInteger = 42  
optionalInteger! // 42
```

使用 `!` 运算符解包值为 `nil` 的可选值会导致运行错误。

你也可以使用可选链式调用和可选绑定来选择性地在可选表达式上执行操作。如果值为 `nil`，不会执行任何操作，因此也就没有运行错误产生。

更多细节以及更多如何使用可选类型的例子，请参阅 [可选类型](#)。

| 可选类型语法

## optional-type

---

| 可选类型 → 类型?

## 隐式解析可选类型

---

当可以被访问时，Swift 语言定义后缀 `!` 作为标准库中命名类型 `Optional<Wrapped>` 的语法糖，来实现自动解包的功能。如果尝试对一个值为 `nil` 的可选类型进行隐式解包，将会产生运行时错误。因为隐式解包，下面两个声明等价：

```
var implicitlyUnwrappedString: String!  
var explicitlyUnwrappedString: Optional<String>
```

注意类型与 `!` 之间没有空格。

由于隐式解包会更改包含该类型的声明语义，嵌套在元组类型或泛型中可选类型（比如字典元素类型或数组元素类型），不能被标记为隐式解包。例如：

```
let tupleOfImplicitlyUnwrappedElements: (Int!, Int!) // 错误
let implicitlyUnwrappedTuple: (Int, Int)! // 正确
```

```
let arrayOfImplicitlyUnwrappedElements: [Int!] // 错误
let implicitlyUnwrappedArray: [Int!]! // 正确
```

由于隐式解析可选类型和可选类型有同样的类型 `Optional<Wrapped>`，你可以在所有使用可选类型的地方使用隐式解析可选类型。比如，你可以将隐式解析可选类型的值赋给变量、常量和可选属性，反之亦然。

正如可选类型一样，如果你在声明隐式解析可选类型的变量或属性的时候没有指定初始值，它的值则会自动赋为默认值 `nil`。

可以使用可选链式调用对隐式解析可选表达式选择性地执行操作。如果值为 `nil`，就不会执行任何操作，因此也不会产生运行错误。

关于隐式解析可选类型的更多细节，请参阅 [隐式解析可选类型](#)。

### | 隐式解析可选类型语法

## implicitly-unwrapped-optional-type

---

### | 隐式解析可选类型 → 类型!

## 协议合成类型

---

协议合成类型定义了一种遵循协议列表中每个指定协议的类型，或者一个现有类型的子类并遵循协议列表中每个指定协议。协议合成类型只能用在类型注解、泛型参数子句和泛型 `where` 子句中指定类型。

协议合成类型的形式如下：

### | `Protocol 1 & Protocol 2`

协议合成类型允许你指定一个值，其类型遵循多个协议的要求而不需要定义一个新的命名型协议来继承它想要符合的各个协议。比如，协议合成类型 `Protocol A & Protocol B & Protocol C` 等效于一个从 `Protocol A`，`Protocol B`，`Protocol C` 继承而来的新协议。同样的，你可以使用 `SuperClass & ProtocolA` 来取代申明一个新的协议作为 `SuperClass` 的子类并遵循 `ProtocolA`。

协议合成列表中的每一项都必须是下面所列情况之一，列表中最多只能包含一个类：

- 类名
- 协议名
- 一个类型别名，它的潜在类型是一个协议合成类型、一个协议或者一个类

当协议合成类型包含类型别名时，同一个协议可能多次出现在定义中 — 重复被忽略。例如，下面代码中定义的 `PQR` 等同于 `P & Q & R`。

```
typealias PQ = P & Q
typealias PQR = PQ & Q & R
```

| 协议合成类型语法

## protocol-composition-type

---

| 协议合成类型 → 协议标识符 & 协议合成延续

## protocol-composition-continuation

---

| 协议合成延续 → 协议标识符 | 协议合成类型

## 不透明类型

---

不透明类型定义了遵循某个协议或者合成协议的类型，但不需要指明底层的具体类型。

不透明类型可以作为函数或下标的返回值，亦或是属性的类型使用。

不透明类型不能作为元组类型的一部分或范型类型使用，比如数组元素类型或者可选值的包装类型。

不透明类型的形式如下：

| some `constraint`

`constraint` 可以是类类型，协议类型，协议组合类型或者 `Any`。值只有当它遵循该协议或者组合协议，或者从该类继承的时候，才能作为这个不透明类型的实例使用。和不透明值交互的代码只能使用该值定义在 `constraint` 上的接口。

协议声明里不能包括不透明类型。类不能使用不透明类型作为非 final 方法的返回值。

使用不透明类型作为返回值的函数必须返回单一公用底层类型。返回的类型可以包含函数范型类型参数的一部分。举个例子，函数 `someFunction<T>()` 可以返回类型 `T` 或者 `Dictionary<String,T>` 的值。

| 不透明类型语法

## opaque-type

---

| 不透明类型 → some type

## 元类型

---

元类型是指任意类型的类型，包括类类型、结构体类型、枚举类型和协议类型。

类、结构体或枚举类型的元类型是相应的类型名紧跟 `.Type`。协议类型的元类型——并不是运行时遵循该协议的具体类型——是该协议名字紧跟 `.Protocol`。比如，类 `SomeClass` 的元类型就是 `SomeClass.Type`，协议 `SomeProtocol` 的元类型就是

`SomeProtocol.Protocol`。

你可以使用后缀 `self` 表达式来获取类型。比如，`SomeClass.self` 返回 `SomeClass` 本身，而不是 `SomeClass` 的一个实例。同样，`SomeProtocol.self` 返回 `SomeProtocol` 本身，而不是运行时遵循 `SomeProtocol` 的某个类型的实例。还可以对类型的实例使用 `type(of:)` 表达式来获取该实例动态的、在运行阶段的类型，如下所示：

```
class SomeBaseClass {  
    class func printClassName() {  
        println("SomeBaseClass")  
    }  
}  
class SomeSubClass: SomeBaseClass {  
    override class func printClassName() {  
        println("SomeSubClass")  
    }  
}  
let someInstance: SomeBaseClass = SomeSubClass()  
// someInstance 在编译期是 SomeBaseClass 类型，  
// 但是在运行期则是 SomeSubClass 类型  
type(of: someInstance).printClassName()  
// 打印“SomeSubClass”
```

更多信息可以查看 Swift 标准库里的 `type(of:)`。

可以使用初始化表达式从某个类型的元类型构造出一个该类型的实例。对于类实例，被调用的构造器必须使用 `required` 关键字标记，或者整个类使用 `final` 关键字标记。

```
class AnotherSubClass: SomeBaseClass {  
    let string: String  
    required init(string: String) {  
        self.string = string  
    }  
    override class func printClassName() {  
        print("AnotherSubClass")  
    }  
}  
let metatype: AnotherSubClass.Type = AnotherSubClass.self  
let anotherInstance = metatype.init(string: "some string")
```

| 元类型语法

## metatype-type

---

| 元类型 → `类型`.`Type` | `类型`.`Protocol`

## 自身类型

---

`Self` 类型不是具体的类型，而是让你更方便的引用当前类型，不需要重复或者知道该类的名字。

在协议声明或者协议成员声明时，`Self` 类型引用的是最终遵循该协议的类型。

在结构体，类或者枚举值声明时，`Self` 类型引用的是声明的类型。在某个类型成员声明时，`Self` 类型引用的是该类型。在类成员声明时，`Self` 可以在方法的返回值和方法体中使用，但不能在其他上下文中使用。举个例子，下面的代码演示了返回值是 `Self` 的实例方法 `f`。

```
class Superclass {  
    func f() -> Self { return self }  
}  
let x = Superclass()  
print(type(of: x.f()))  
// 打印 "Superclass"  
  
class Subclass: Superclass { }  
let y = Subclass()  
print(type(of: y.f()))  
// 打印 "Subclass"  
  
let z: Superclass = Subclass()  
print(type(of: z.f()))  
// 打印 "Subclass"
```

上面例子的最后一部分表明 `Self` 引用的是值 `z` 的运行时类型 `Subclass`，而不是变量本身的编译时类型 `Superclass`。

在嵌套类型声明时，`Self` 类型引用的是最内层声明的类型。

`Self` 类型引用的类型和 Swift 标准库中 `type(of:)` 函数的结果一样。使用 `Self.someStaticMember` 访问当前类型中的成员和使用 `type(of: self).someStaticMember` 是一样的。

### 自身类型语法

## self-type

### 自身类型 → Self

## 类型继承子句

类型继承子句被用来指定一个命名型类型继承自哪个类、采纳哪些协议。类型继承子句开始于冒号 `:`，其后是类型标识符列表。

类可以继承自单个超类，并遵循任意数量的协议。当定义一个类时，超类的名字必须出现在类型标识符列表首位，然后跟上该类需要遵循的任意数量的协议。如果一个类不是从其它类继承而来，那么列表可以以协议开头。关于类继承更多的讨论和例子，请参阅 [继承](#)。

其它命名型类型只能继承自或采纳一系列协议。协议类型可以继承自任意数量的其他协议。当一个协议类型继承自其它协议时，其它协议中定义的要求会被整合在一起，然后从当前协议继承的任意类型必须符合所有这些条件。

枚举定义中的类型继承子句可以是一系列协议，或者是指定单一的命名类型，此时枚举为其用例分配原始值。在枚举定义中使用类型继承子句来指定原始值类型的例子，请参阅 [原始值](#)。

## 类型继承子句语法

### **type\_inheritance\_clause**

---

| 类型继承子句 → : [类型继承列表](#)

### **type-inheritance-list**

---

| 类型继承列表 → [类型标识符](#) | [类型标识符](#), [类型继承列表](#)

## 类型推断

---

Swift 广泛使用 [类型推断](#)，从而允许你省略代码中很多变量和表达式的类型或部分类型。比如，对于 `var x: Int = 0`，你可以完全省略类型而简写成 `var x = 0`，编译器会正确推断出 `x` 的类型 `Int`。类似的，当完整的类型可以从上下文推断出来时，你也可以省略类型的一部分。比如，如果你写了 `let dict: Dictionary = ["A": 1]`，编译器能推断出 `dict` 的类型是 `Dictionary<String, Int>`。

在上面的两个例子中，类型信息从表达式树的叶子节点传向根节点。也就是说，`var x: Int = 0` 中 `x` 的类型首先根据 `0` 的类型进行推断，然后将该类型信息传递到根节点（变量 `x`）。

在 Swift 中，类型信息也可以反方向流动——从根节点传向叶子节点。在下面的例子中，常量 `eFloat` 上的显式类型注解（`: Float`）将导致数字字面量 `2.71828` 的类型是 `Float` 而非 `Double`。

```
let e = 2.71828 // e 的类型会被推断为 Double  
let eFloat: Float = 2.71828 // eFloat 的类型为 Float
```

Swift 中的类型推断在单独的表达式或语句上进行。这意味着所有用于类型推断的信息必须可以从表达式或其某个子表达式的类型检查中获取到。

# 表达式 · GitBook

---

 [runoob.com/manual/gitbook/swift5/source/\\_book/chapter3/04\\_Expressions.html](https://runoob.com/manual/gitbook/swift5/source/_book/chapter3/04_Expressions.html)

## 表达式 (Expressions)

---

Swift 中存在四种表达式：前缀表达式，二元表达式，基本表达式和后缀表达式。表达式在返回一个值的同时还可以引发副作用。

通过前缀表达式和二元表达式可以对简单表达式使用各种运算符。基本表达式从概念上讲是最简单的一种表达式，它是一种访问值的方式。后缀表达式则允许你建立复杂的表达式，例如函数调用和成员访问。每种表达式都在下面有详细论述。

| 表达式语法

### expression

---

| 表达式 → *try 运算符* 可选 前缀表达式 二元表达式列表 可选

### expression-list

---

| 表达式列表 → 表达式 | 表达式, 表达式列表

## 前缀表达式

---

前缀表达式由可选的前缀运算符和表达式组成。前缀运算符只接收一个参数，表达式则紧随其后。

关于这些运算符的更多信息，请参阅 [基本运算符](#) 和 [高级运算符](#)。

关于 Swift 标准库提供的运算符的更多信息，请参阅 [\*Operators Declarations\*](#)。

除了标准库运算符，你也可以对某个变量使用 `&` 运算符，从而将其传递给函数的输入输出参数。更多信息，请参阅 [输入输出参数](#)。

| 前缀表达式语法

### prefix-expression

---

| 前缀表达式 → 前缀运算符 可选 后缀表达式

| 前缀表达式 → 输入输出表达式

### in-out-expression

---

| 输入输出表达式 → & 标识符

## Try 运算符

---

---

try 表达式由 `try` 运算符加上紧随其后的可抛出错误的表达式组成，形式如下：

| `try` 可抛出错误的表达式

可选的 try 表达式由 `try?` 运算符加上紧随其后的可抛出错误的表达式组成，形式如下：

| `try?` 可抛出错误的表达式

如果可抛出错误的表达式没有抛出错误，整个表达式返回的可选值将包含可抛出错误的表达式的返回值，否则，该可选值为 `nil`。

强制的 try 表达式由 `try!` 运算符加上紧随其后的可抛出错误的表达式组成，形式如下：

| `try!` 可抛出错误的表达式

如果可抛出错误的表达式抛出了错误，将会引发运行时错误。

在二元运算符左侧的表达式被标记上 `try`、`try?` 或者 `try!` 时，这个运算符对整个二元表达式都产生作用。也就是说，你可以使用括号来明确运算符的作用范围。

```
sum = try someThrowingFunction() + anotherThrowingFunction() // try 对两个函数调用都产生作用
sum = try (someThrowingFunction() + anotherThrowingFunction()) // try 对两个函数调用都产生作用
sum = (try someThrowingFunction()) + anotherThrowingFunction() // 错误：try 只对第一个函数调用产生作用
```

`try` 表达式不能出现在二元运算符的右侧，除非二元运算符是赋值运算符或者 `try` 表达式是被圆括号括起来的。

关于 `try`、`try?` 和 `try!` 的更多信息，以及该如何使用的例子，请参阅 [错误处理](#)。

| Try 表达式语法

## try-operator

---

| `try` 运算符 → `try` | `try?` | `try!`

## 二元表达式

---

二元表达式由中缀运算符和左右参数表达式组成。形式如下：

| 左侧参数 二元运算符 右侧参数

关于这些运算符的更多信息，请参阅 [基本运算符](#) 和 [高级运算符](#)。

关于 Swift 标准库提供的运算符的更多信息，请参阅 [Swift Standard Library Operators Reference](#)。

## 注意

在解析时，一个二元表达式将作为一个扁平列表表示，然后根据运算符的优先级，再进一步进行组合。例如，`2 + 3 * 5` 首先被看作具有五个元素的列表，即`2`、`+`、`3`、`*`、`5`，随后根据运算符优先级组合为`(2 + (3 * 5))`。

## binary-expression

---

二元表达式语法

二元表达式 → 二元运算符 前缀表达式

二元表达式 → 赋值运算符 try 运算符 可选 前缀表达式

二元表达式 → 条件运算符 try 运算符 可选 前缀表达式

二元表达式 → 类型转换运算符

## binary-expressions

---

二元表达式列表 → 二元表达式 二元表达式列表 可选

## 赋值表达式

---

赋值表达式会为某个给定的表达式赋值，形式如下；

表达式 = 值

右边的值会被赋值给左边的表达式。如果左边表达式是一个元组，那么右边必须是一个具有同样元素个数的元组。（嵌套元组也是允许的。）右边的值中的每一部分都会被赋值给左边的表达式中的相应部分。例如：

```
(a, _, (b, c)) = ("test", 9.45, (12, 3))
// a 为 "test", b 为 12, c 为 3, 9.45 会被忽略
```

赋值运算符不返回任何值。

赋值运算符语法

## assignment-operator

---

赋值运算符 → =

## 三元条件运算符

---

三元条件运算符会根据条件来对两个给定表达式中的一个进行求值，形式如下：

条件 ? 表达式 (条件为真则使用) : 表达式 (条件为假则使用)

如果条件为真，那么对第一个表达式进行求值并返回结果。否则，对第二个表达式进行求值并返回结果。未使用的表达式不会进行求值。

关于使用三元条件运算符的例子，请参阅 [三元条件运算符](#)。

### 三元条件运算符语法

## conditional-operator

---

三元条件运算符 → ? 表达式 :

## 类型转换运算符

---

有 4 种类型转换运算符：`is`、`as`、`as?` 和 `as!`。它们有如下的形式：

表达式 `is` 类型

表达式 `as` 类型

表达式 `as?` 类型

表达式 `as!` 类型

`is` 运算符在运行时检查表达式能否向下转化为指定的类型，如果可以则返回 `true`，否则返回 `false`。

`as` 运算符在编译时执行向上转换和桥接。向上转换可将表达式转换成父类的实例而无需使用任何中间变量。以下表达式是等价的：

```
func f(any: Any) { print("Function for Any") }
func f(int: Int) { print("Function for Int") }
let x = 10
f(x)
// 打印“Function for Int”

let y: Any = x
f(y)
// 打印“Function for Any”

f(x as Any)
// 打印“Function for Any”
```

桥接可将 Swift 标准库中的类型（例如 `String`）作为一个与之相关的 Foundation 类型（例如 `NSString`）来使用，而不需要新建一个实例。关于桥接的更多信息，请参阅 [Working with Foundation Types](#)。

`as?` 运算符有条件地执行类型转换，返回目标类型的可选值。在运行时，如果转换成功，返回的可选值将包含转换后的值，否则返回 `nil`。如果在编译时就能确定转换一定会成功或是失败，则会导致编译报错。

**as!** 运算符执行强制类型转换，返回目标类型的非可选值。如果转换失败，则会导致运行时错误。表达式 `x as! T` 效果等同于 `(x as? T)!`。

关于类型转换的更多内容和例子，请参阅 [类型转换](#)。

## type-casting-operator

---

类型转换运算符语法

类型转换运算符 → `is` 类型

类型转换运算符 → `as` 类型

类型转换运算符 → `as?` 类型

类型转换运算符 → `as!` 类型

## 基本表达式

---

基本表达式是最基本的表达式。它们可以单独使用，也可以跟前缀表达式、二元表达式、后缀表达式组合使用。

基本表达式语法

## primary-expression

---

基本表达式 → 标识符 泛型实参子句 可选

基本表达式 → 字面量表达式

基本表达式 → self 表达式

基本表达式 → 父类表达式

基本表达式 → 闭包表达式

基本表达式 → 圆括号表达式

基本表达式 → 隐式成员表达式

基本表达式 → 通配符表达式

基本表达式 → 选择器表达式

基本表达式 → key-path 字符串表达式

## 字面量表达式

---

字面量表达式可由普通字面量（例如字符串或者数字），字典或者数组字面量，或者下面列表中的特殊字面量组成：

字面量	类型	值
#file	String	所在的文件名
#line	Int	所在的行数
#column	Int	所在的列数
#function	String	所在的声明的名字

对于 `#function`，在函数中会返回当前函数的名字，在方法中会返回当前方法的名字，在属性的存取器中会返回属性的名字，在特殊的成员如 `init` 或 `subscript` 中会返回这个关键字的名字，在某个文件中会返回当前模块的名字。

当其作为函数或者方法的默认参数值时，该字面量的值取决于函数或方法的调用环境。

```
func logFunctionName(string: String = #function) {
    print(string)
}
func myFunction() {
    logFunctionName()
}
myFunction() // 打印“myFunction()”
```

数组字面量是值的有序集合，形式如下：

| [ 值 1 , 值 2 , ... ]

数组中的最后一个表达式可以紧跟一个逗号。数组字面量的类型是 `[T]`，这个 `T` 就是数组中元素的类型。如果数组中包含多种类型，`T` 则是跟这些类型最近的公共父类型。空数组字面量由一组方括号定义，可用来创建特定类型的空数组。

```
var emptyArray: [Double] = []
```

字典字面量是一个包含无序键值对的集合，形式如下：

| [ 键 1 : 值 1 , 键 2 : 值 2 , ... ]

字典中的最后一个表达式可以紧跟一个逗号。字典字面量的类型是 `[Key : Value]`，`Key` 表示键的类型，`Value` 表示值的类型。如果字典中包含多种类型，那么 `Key` 表示的类型则为所有键最接近的公共父类型，`Value` 与之相似。一个空的字典字面量由方括号中加一个冒号组成（`[:]`），从而与空数组字面量区分开，可以使用空字典字面量来创建特定类型的字典。

```
var emptyDictionary: [String : Double] = [:]
```

Xcode 使用 playground 字面量对程序编辑器中的颜色、文件或者图片创建可交互的展示。在 Xcode 之外的空白文本中，playground 字面量使用一种特殊的字面量语法来展示。

更多关于在 Xcode 中使用 playground 字面量的信息，请参阅 [添加颜色、文件或图片字面量](#)。

## 字面量表达式语法

### literal-expression

---

字面量表达式 → 字面量

字面量表达式 → 数组字面量 | 字典字面量 | 练习场字面量

字面量表达式 → #file | #line | #column | #function

### array-literal

---

数组字面量 → [ 数组字面量项列表 可选 ]

### array-literal-items

---

数组字面量项列表 → 数组字面量项, 可选 | 数组字面量项, 数组字面量项列表

### array-literal-item

---

数组字面量项 → 表达式

### dictionary-literal

---

字典字面量 → [ 字典字面量项列表 ] | [ : ]

### dictionary-literal-items

---

字典字面量项列表 → 字典字面量项, 可选 | 字典字面量项, 字典字面量项列表

### dictionary-literal-item

---

字典字面量项 → 表达式 : 表达式。

### playground-literal

---

playground 字面量 → #colorLiteral ( red : 表达式, green : 表达式 expression ), blue : 表达式, alpha : 表达式)

playground 字面量 → #fileLiteral ( resourceName : 表达式 )

playground 字面量 → \*\*#imageLiteral ( resourceName : 表达式 )  
(#expression) \*self\_expression

---

## Self 表达式

**self** 表达式是对当前类型或者当前实例的显式引用，它有如下形式：

```
self  
self. 成员名称  
self[ 下标索引 ]  
self( 构造器参数 )  
self.init( 构造器参数 )
```

如果在构造器、下标、实例方法中，`self` 引用的是当前类型的实例。在一个类型方法中，`self` 引用的是当前的类型。

当访问成员时，`self` 可用来区分重名变量，例如函数的参数：

```
class SomeClass {  
    var greeting: String  
    init(greeting: String) {  
        self.greeting = greeting  
    }  
}
```

在 `mutating` 方法中，你可以对 `self` 重新赋值：

```
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)  
    }  
}
```

| Self 表达式语法

## self-expression

---

| `self` 表达式 → `self` | `self` 方法表达式 | `self` 下标表达式 | `self` 构造器表达式

## self-method-expression

---

| `self` 方法表达式 → `self . 标识符`

## self-subscript-expression

---

| `self` 下标表达式 → `self [ 函数调用参数表 ]`

## self-initializer-expression

---

| `self` 构造器表达式 → `self . init`

## 父类表达式

---

父类表达式可以使我们在某个类中访问它的父类，它有如下形式：

```
super. 成员名称  
super[ 下标索引 ]  
super.init( 构造器参数 )
```

第一种形式用来访问父类的某个成员，第二种形式用来访问父类的下标，第三种形式用来访问父类的构造器。

子类可以通过父类表达式在它们的成员、下标和构造器中使用父类中的实现。

| 父类表达式语法

## superclass-expression

---

| 父类表达式 → 父类方法表达式 | 父类下标表达式 | 父类构造器表达式

### superclass-method-expression

---

| 父类方法表达式 → super . 标识符

### superclass-subscript-expression

---

| 父类下标表达式 → super [函数调用参数表]

### superclass-initializer-expression

---

| 父类构造器表达式 → super . init

## 闭包表达式

---

闭包表达式会创建一个闭包，在其他语言中也叫 *lambda* 或匿名函数。跟函数一样，闭包包含了待执行的代码，不同的是闭包还会捕获所在环境中的常量和变量。它的形式如下：

```
{ (parameters) -> return type in  
    statements  
}
```

闭包的参数声明形式跟函数一样，请参阅 [函数声明](#)。

闭包还有几种特殊的形式，能让闭包使用起来更加简洁：

- 闭包可以省略它的参数和返回值的类型。如果省略了参数名和所有的类型，也要省略 `in` 关键字。如果被省略的类型无法被编译器推断，那么就会导致编译错误。
- 闭包可以省略参数名，参数会被隐式命名为 `$` 加上其索引位置，例如 `$0`、`$1`、`$2` 分别表示第一个、第二个、第三个参数，以此类推。

- 如果闭包中只包含一个表达式，那么该表达式的结果就会被视为闭包的返回值。表达式结果的类型也会被推断为闭包的返回类型。

下面几个闭包表达式是等价的：

```
myFunction {
  (x: Int, y: Int) -> Int in
  return x + y
}

myFunction {
  (x, y) in
  return x + y
}

myFunction { return $0 + $1 }

myFunction { $0 + $1 }
```

关于如何将闭包作为参数来传递的内容，请参阅 [函数调用表达式](#)。

使用闭包表达式时，可以不必将其存储在一个变量或常量中，例如作为函数调用的一部分来立即使用一个闭包。在上面的例子中，传入 `myFunction` 的闭包表达式就是这种立即使用类型的闭包。因此，一个闭包是否逃逸与其使用时的上下文相关。一个会被立即调用或者作为函数的非逃逸参数传递的闭包表达式是非逃逸的，否则，这个闭包表达式是逃逸的。

关于逃逸闭包的内容，请参阅 [逃逸闭包](#)。

## 捕获列表

---

默认情况下，闭包会捕获附近作用域中的常量和变量，并使用强引用指向它们。你可以通过一个捕获列表来显式指定它的捕获行为。

捕获列表在参数列表之前，由中括号括起来，里面是由逗号分隔的一系列表达式。一旦使用了捕获列表，就必须使用 `in` 关键字，即使省略了参数名、参数类型和返回类型。

捕获列表中的项会在闭包创建时被初始化。每一项都会用闭包附近作用域中的同名常量或者变量的值初始化。例如下面的代码示例中，捕获列表包含 `a` 而不包含 `b`，这将导致这两个变量具有不同的行为。

```
var a = 0
var b = 0
let closure = { [a] in
  print(a, b)
}

a = 10
b = 10
closure()
// 打印“0 10”
```

在示例中，变量 `b` 只有一个，然而，变量 `a` 有两个，一个在闭包外，一个在闭包内。闭包内的变量 `a` 会在闭包创建时用闭包外的变量 `a` 的值来初始化，除此之外它们并无其他联系。这意味着在闭包创建后，改变某个 `a` 的值都不会对另一个 `a` 的值造成任何影响。与此相反，闭包内外都是同一个变量 `b`，因此在闭包外改变其值，闭包内的值也会受影响。

如果闭包捕获的值具有引用语义则有所不同。例如，下面示例中，有两个变量 `x`，一个在闭包外，一个在闭包内，由于它们的值是引用语义，虽然这是两个不同的变量，它们却都引用着同一实例。

```
class SimpleClass {  
    var value: Int = 0  
}  
var x = SimpleClass()  
var y = SimpleClass()  
let closure = { [x] in  
    print(x.value, y.value)  
}  
  
x.value = 10  
y.value = 10  
closure()  
// 打印“10 10”
```

如果捕获列表中的值是类类型，你可以使用 `weak` 或者 `unowned` 来修饰它，闭包会分别用弱引用和无主引用来捕获该值。

```
myFunction { print(self.title) }           // 隐式强引用捕获  
myFunction { [self] in print(self.title) }   // 显式强引用捕获  
myFunction { [weak self] in print(self!.title) } // 弱引用捕获  
myFunction { [unowned self] in print(self.title) } // 无主引用捕获
```

在捕获列表中，也可以将任意表达式的值绑定到一个常量上。该表达式会在闭包被创建时进行求值，闭包会按照指定的引用类型来捕获表达式的值。例如：

```
// 以弱引用捕获 self.parent 并赋值给 parent  
myFunction { [weak parent = self.parent] in print(parent!.title) }
```

关于闭包表达式的更多信息和例子，请参阅 [闭包表达式](#)。关于捕获列表的更多信息和例子，请参阅 [解决闭包引起的循环强引用](#)。

闭包表达式语法

### **closure-expression**

---

闭包表达式 → { 闭包签名 可选 语句 }

### **closure-signature**

---

闭包签名 → [参数子句] (#parameter-clause) [函数结果] (05\_Declarations.md#function-result) 可选 \*in

闭包签名 → 标识符列表 函数结果 可选 in

闭包签名 → 捕获列表 参数子句 函数结果 可选 in

闭包签名 → 捕获列表 标识符列表 函数结果 可选 in

闭包签名 → 捕获列表 in

### **capture-list**

---

捕获列表 → [ 捕获列表项列表 ] (#capture-list-items) \*

### **capture-list-items**

---

捕获列表项列表 → 捕获列表项 | 捕获列表项, 捕获列表项列表

### **capture-list-item**

---

捕获列表项 → 捕获说明符 可选 表达式

### **capture-specifier**

---

捕获说明符 → weak | unowned | unowned(safe) | unowned(unsafe)

## **隐式成员表达式**

---

若类型可被推断出来，可以使用隐式成员表达式来访问某个类型的成员（例如某个枚举成员或某个类型方法），形式如下：

| . 成员名称

例如：

```
var x = MyEnumeration.SomeValue  
x = .AnotherValue
```

| 隐式成员表达式语法

### **implicit-member-expression**

---

| 隐式成员表达式 → . 标识符

## 圆括号表达式

---

圆括号表达式是由圆括号包围的表达式。你可以用圆括号说明成组的表达式的先后操作。成组的圆括号不会改变表达式的类型 - 例如 `(1)` 的类型就是简单的 `Int`。

| 圆括号表达式语法

### **parenthesized-expression**

---

| 圆括号表达式 →  $(\text{表达式})$

## 元组表达式

---

元组表达式由圆括号和其中多个逗号分隔的子表达式组成。每个子表达式前面可以有一个标识符，用冒号隔开。元组表达式形式如下：

|  $(\text{标识符 } 1 : \text{表达式 } 1, \text{ 标识符 } 2 : \text{表达式 } 2, \dots)$

元组表达式可以一个表达式都没有，也可以包含两个或是更多的表达式。单个表达式用括号括起来就是括号表达式了。

| 注意

在 Swift 中，空的元组表达式和空的元组类型都写作 `()`。由于 `Void` 是 `()` 的类型别名，因此可以使用它来表示空的元组类型。虽然如此，`Void` 就像所有的类型别名一样，永远是一个类型——不能表示空的元组表达式。

| 元组表达式语法

### **tuple-expression**

---

| 元组表达式 →  $() | (\text{元组元素}, \text{元组元素列表})$

### **tuple-element-list**

---

| 元组元素列表 →  $\text{元组元素} | \text{元组元素}, \text{元组元素列表}$

### **tuple-element**

---

| 元组元素 →  $\text{表达式} | \text{标识符}: \text{表达式}$

## 通配符表达式

---

通配符表达式可以在赋值过程中显式忽略某个值。例如下面的代码中，`10` 被赋值给 `x`，而 `20` 则被忽略：

```
(x, _) = (10, 20)  
// x 为 10, 20 被忽略
```

| 通配符表达式语法

## wildcard-expression

---

| 通配符表达式 → \_

## Key-path 表达式

---

Key-path 表达式引用一个类型的属性或下标。在动态语言中使场景可以使用 Key-path 表达式，例如观察键值对。格式为：

| \类型名.路径

类型名是一个具体类型的名称，包含任何泛型参数，例如 `String`、`[Int]` 或 `Set<Int>`。

路径可由属性名称、下标、可选链表达式或者强制解包表达式组成。以上任意 key-path 组件可以以任何顺序重复多次。

在编译期，key-path 表达式会被一个 `KeyPath` 类的实例替换。

对于所有类型，都可以通过传递 key-path 参数到下标方法 `subscript(keyPath:)` 来访问它的值。例如：

```
struct SomeStructure {
    var someValue: Int
}

let s = SomeStructure(someValue: 12)
let pathToProperty = \SomeStructure.someValue

let value = s[keyPath: pathToProperty]
// 值为 12
```

在一些可以通过类型推断来确定所访问的具体类型的上下文中，可以省略 key-path 前的类型名字。下面的代码使用 `\.someProperty` 代替了 `SomeClass.someProperty`：

```
class SomeClass: NSObject {
    @objc var someProperty: Int
    init(someProperty: Int) {
        self.someProperty = someProperty
    }
}

let c = SomeClass(someProperty: 10)
c.observe(\.someProperty) { object, change in
    // ...
}
```

使用 `self` 作为路径可以创建一个恒等 key path (`\.self`)。恒等 key path 可以作为整个实例的引用，因此你仅需一步操作便可以利用它来访问以及修改其存储的所有数据。例如：

```
var compoundValue = (a: 1, b: 2)
// 等价于 compoundValue = (a: 10, b: 20)
compoundValue[keyPath: \.self] = (a: 10, b: 20)
```

通过点语法，可以让路径包含多个属性名称，以此来访问某实例的属性的属性。下面的代码使用 key-path 表达式 `\OuterStructure.outer.someValue` 来访问 `OuterStructure` 类型中 `outer` 属性的 `someValue` 属性。

```
struct OuterStructure {
    var outer: SomeStructure
    init(someValue: Int) {
        self.outer = SomeStructure(someValue: someValue)
    }
}
```

```
let nested = OuterStructure(someValue: 24)
let nestedKeyPath = \OuterStructure.outer.someValue
```

```
let nestedValue = nested[keyPath: nestedKeyPath]
// nestedValue 的值为 24
```

路径中也可以包含使用中括号的下标访问，只要下标访问的参数类型满足 `Hashable` 协议即可。下面的例子在 key path 中使用了下标来访问数组的第二个元素。

```
let greetings = ["hello", "hola", "bonjour", "안녕"]
let myGreeting = greetings[keyPath: \[String].[1]]
// myGreeting 的值为 'hola'
```

下标访问中使用的值可以是一个变量或者字面量，并且 key-path 表达式会使用值语义来捕获此值。下面的代码在 key-path 表达式和闭包中都使用了 `index` 变量来访问 `greetings` 数组的第三个元素。当 `index` 被修改时，key-path 表达式仍旧引用数组第三个元素，而闭包则使用了新的索引值。

```
var index = 2
let path = \[String].[index]
let fn: ([String]) -> String = { strings in strings[index] }

print(greetings[keyPath: path])
// 打印 "bonjour"
print(fn(greetings))
// 打印 "bonjour"

// 将 'index' 设置为一个新的值不会影响到 'path'
index += 1
print(greetings[keyPath: path])
// 打印 "bonjour"

// 'fn' 闭包使用了新值。
print(fn(greetings))
// 打印 "안녕"
```

路径可以使用可选链和强制解包。下面的代码在 key path 中使用了可选链来访问可选字符串的属性。

```

let firstGreeting: String? = greetings.first
print(firstGreeting?.count as Any)
// 打印 "Optional(5)"

// 使用 key path 实现同样的功能
let count = greetings[keyPath: \[String].first?.count]
print(count as Any)
// 打印 "Optional(5)"

```

可以混合使用各种 key-path 组件来访问一些深度嵌套类型的值。下面的代码通过组合不同的组件，使用 key-path 表达式访问了一个字典数组中不同的值和属性。

```

let interestingNumbers = ["prime": [2, 3, 5, 7, 11, 13, 17],
                         "triangular": [1, 3, 6, 10, 15, 21, 28],
                         "hexagonal": [1, 6, 15, 28, 45, 66, 91]]
print(interestingNumbers[keyPath: \[String: [Int]].["prime"]] as Any)
// 打印 "Optional([2, 3, 5, 7, 11, 13, 17])"
print(interestingNumbers[keyPath: \[String: [Int]].["prime"]![0]])
// 打印 "2"
print(interestingNumbers[keyPath: \[String: [Int]].["hexagonal"]!.count])
// 打印 "7"
print(interestingNumbers[keyPath: \[String: [Int]].["hexagonal"]!.count.bitWidth])
// 打印 "64"

```

关于更多如何使用 key path 与 Objective-C APIs 交互的信息，请参阅 [在 Swift 中使用 Objective-C 运行时特性](#)。关于更多 key-value 编程和 key-value 观察的信息，请参阅 [Key-Value 编程](#) 和 [Key-Value 观察编程](#)。

key-path 表达式语法

### **key-path-expression**

---

*key-path 表达式* → \类型可选 . 多个 key-path 组件

### **key-path-components**

---

多个 key-path 组件 → key-path 组件 | key-path 组件 . 多个 key-path 组件

### **key-path-component**

---

*key-path 组件* → 标识符 多个 key-path 后缀可选 | 多个 key-path 后缀

### **key-path-postfixes**

---

多个 key-path 后缀 → key-path 后缀 多个 key-path 后缀可选 key-path-postfixes

*key-path 后缀* → ? | ! | self | [ 函数调用参数表 ]

## **选择器表达式**

---

选择器表达式可以让你通过选择器来引用在 Objective-C 中方法（method）和属性（property）的 setter 和 getter 方法。

```
#selector(方法名)  
#selector(getter: 属性名) #selector(setter: 属性名)
```

方法名和属性名必须是存在于 Objective-C 运行时中的方法和属性的引用。选择器表达式的返回值是一个 Selector 类型的实例。例如：

```
class SomeClass: NSObject {  
    let property: String  
    @objc(doSomethingWithInt:)  
    func doSomething(_ x: Int) { }  
  
    init(property: String) {  
        self.property = property  
    }  
}  
let selectorForMethod = #selector(SomeClass.doSomething(_:))  
let selectorForPropertyGetter = #selector(getter: SomeClass.property)
```

当为属性的 getter 创建选择器时，属性名可以是变量属性或者常量属性的引用。但是当为属性的 setter 创建选择器时，属性名只可以是对变量属性的引用。

方法名称可以包含圆括号来进行分组，并使用 as 操作符来区分具有相同方法名但类型不同的方法，例如：

```
extension SomeClass {  
    @objc(doSomethingWithString:)  
    func doSomething(_ x: String) { }  
}  
let anotherSelector = #selector(SomeClass.doSomething(_:) as (SomeClass) -> (String) -> Void)
```

由于选择器是在编译时创建的，因此编译器可以检查方法或者属性是否存在，以及是否在运行时暴露给了 Objective-C。

注意

虽然方法名或者属性名是个表达式，但是它不会被求值。

更多关于如何在 Swift 代码中使用选择器来与 Objective-C API 进行交互的信息，请参阅 [在 Swift 中使用 Objective-C 运行时特性](#)。

选择器表达式语法

## selector-expression

---

选择器表达式 → #selector (表达式)

选择器表达式 → #selector (getter:表达式)

选择器表达式 → #selector (setter:表达式)

## Key-path 字符串表达式

---

key-path 字符串表达式可以访问一个引用 Objective-C 属性的字符串，通常在 key-value 编程和 key-value 观察 APIs 中使用。其格式如下：

```
| #keyPath ( 属性名 )
```

属性名必须是一个可以在 Objective-C 运行时使用的属性的引用。在编译期，key-path 字符串表达式会被一个字符串字面量替换。例如：

```
class SomeClass: NSObject {  
    @objc var someProperty: Int  
    init(someProperty: Int) {  
        self.someProperty = someProperty  
    }  
}  
  
let c = SomeClass(someProperty: 12)  
let keyPath = #keyPath(SomeClass.someProperty)  
  
if let value = c.value(forKey: keyPath) {  
    print(value)  
}  
// 打印 "12"
```

当在一个类中使用 key-path 字符串表达式时，可以省略类名，直接使用属性名来访问这个类的某个属性。

```
extension SomeClass {  
    func getSomeKeyPath() -> String {  
        >  
        return #keyPath(someProperty)  
    }  
}  
print(keyPath == c.getSomeKeyPath())  
// 打印 "true"
```

由于 key-path 字符串表达式在编译期才创建，编译期可以检查属性是否存在，以及属性是否暴露给 Objective-C 运行时。

关于更多如何使用 key path 与 Objective-C APIs 交互的信息，请参阅 [在 Swift 中使用 Objective-C 运行时特性](#)。关于更多 key-value 编程和 key-value 观察的信息，请参阅 [Key-Value 编程](#) 和 [Key-Value 观察编程](#)。

## 注意

尽管属性名是一个表达式，但它永远不会被求值

key-path 字符串表达式语法

### **key-path-string-expression**

---

*key-path* 字符串表达式 → #keyPath (表达式)

## 后缀表达式

---

后缀表达式就是在某个表达式的后面运用后缀运算符或其他后缀语法。从语法构成上来看，基本表达式也是后缀表达式。

关于这些运算符的更多信息，请参阅 [基本运算符](#) 和 [高级运算符](#)。

关于 Swift 标准库提供的运算符的更多信息，请参阅 [运算符定义](#)。

| 后缀表达式语法

### **postfix-expression**

---

后缀表达式 → 基本表达式

后缀表达式 → 后缀表达式 后缀运算符

后缀表达式 → 函数调用表达式

后缀表达式 → 构造器表达式

后缀表达式 → 显式成员表达式

后缀表达式 → 后缀 self 表达式

后缀表达式 → dynamicType 表达式

后缀表达式 → 下标表达式

后缀表达式 → 强制取值表达式

后缀表达式 → 可选链表达式

## 函数调用表达式

---

函数调用表达式由函数名和参数列表组成，形式如下：

| **函数名 ( 参数 1 , 参数 2 )**

函数名可以是值为函数类型的任意表达式。

如果函数声明中指定了参数的名字，那么在调用的时候也必须得写出来。这种函数调用表达式具有以下形式：

| 函数名 ( 参数名 1 : 参数 1 , 参数名 2 : 参数 2 )

如果函数的最后一个参数是函数类型，可以在函数调用表达式的尾部（右圆括号之后）加上一个闭包，该闭包会作为函数的最后一个参数。如下两种写法是等价的：

```
// someFunction 接受整数和闭包参数
someFunction(x, f: {$0 == 13})
someFunction(x) {$0 == 13}
```

如果闭包是该函数的唯一参数，那么圆括号可以省略。

```
// someFunction 只接受一个闭包参数
myData.someMethod() {$0 == 13}
myData.someMethod {$0 == 13}
```

函数调用表达式语法

### function-call-expression

---

函数调用表达式 → 后缀表达式 函数调用参数子句

函数调用表达式 → 后缀表达式 函数调用参数子句 可选 尾随闭包

### function-call-argument-clause

---

函数调用参数子句 → () | ( 函数调用参数表 )

### function-call-argument-list

---

函数调用参数表 → 函数调用参数 | 函数调用参数 , 函数调用参数表

### function-call-argument

---

函数调用参数 → 表达式 | 标识符 : 表达式

函数调用参数 → 运算符 | 标识符 : 运算符

### trailing-closure

---

尾随闭包 → 闭包表达式

## 构造器表达式

---

构造器表达式用于访问某个类型的构造器，形式如下：

| 表达式 .init( 构造器参数 )

你可以在函数调用表达式中使用构造器表达式来初始化某个类型的新实例。也可以使用构造器表达式来代理给父类构造器。

```
class SomeSubClass: SomeSuperClass {  
    override init() {  
        // 此处为子类构造过程  
        super.init()  
    }  
}
```

和函数类似，构造器表达式可以作为一个值。例如：

```
// 类型注解是必须的，因为 String 类型有多种构造器  
let initializer: Int -> String = String.init  
let oneTwoThree = [1, 2, 3].map(initializer).reduce("", combine: +)  
print(oneTwoThree)  
// 打印“123”
```

如果通过名字来指定某个类型，可以不用构造器表达式而直接使用类型的构造器。在其他情况下，你必须使用构造器表达式。

```
let s1 = SomeType.init(data: 3) // 有效  
let s2 = SomeType(data: 1) // 有效  
  
let s4 = someValue.dynamicType(data: 5) // 错误  
let s3 = someValue.dynamicType.init(data: 7) // 有效
```

## 构造器表达式语法

### initializer-expression

构造器表达式 → 后缀表达式.init

构造器表达式 → 后缀表达式.init(参数名称)

## 显式成员表达式

显式成员表达式允许我们访问命名类型、元组或者模块的成员，其形式如下：

表达式 . 成员名

命名类型的某个成员在原始实现或者扩展中定义，例如：

```
class SomeClass {  
    var someProperty = 42  
}  
let c = SomeClass()  
let y = c.someProperty // 访问成员
```

元组的成员会隐式地根据表示它们出现顺序的整数来命名，以 o 开始，例如：

```
var t = (10, 20, 30)  
t.0 = t.1  
// 现在元组 t 为 (20, 20, 30)
```

对于模块的成员来说，只能直接访问顶级声明中的成员。

使用 `dynamicMemberLookup` 属性声明的类型包含可以在运行时查找的成员，具体请参阅 [属性](#)。

为了区分只有参数名有所不同的方法或构造器，在圆括号中写出参数名，参数名后紧跟一个冒号，对于没有参数名的参数，使用下划线代替参数名。而对于重载方法，则需使用类型注解进行区分。例如：

```
class SomeClass {  
    func someMethod(x: Int, y: Int) {}  
    func someMethod(x: Int, z: Int) {}  
    func overloadedMethod(x: Int, y: Int) {}  
    func overloadedMethod(x: Int, y: Bool) {}  
}  
let instance = SomeClass()  
  
let a = instance.someMethod           // 有歧义  
let b = instance.someMethod(_:_:)  
// 无歧义  
  
let d = instance.overloadedMethod    // 有歧义  
let d = instance.overloadedMethod(_:_:) // 有歧义  
let d: (Int, Bool) -> Void = instance.overloadedMethod(_:_:) // 无歧义
```

如果点号（.）出现在行首，它会被视为显式成员表达式的一部分，而不是隐式成员表达式的一部分。例如如下代码所展示的被分为多行的链式方法调用：

```
let x = [10, 3, 20, 15, 4]  
.sort()  
.filter { $0 > 5 }  
.map { $0 * 100 }
```

| 显式成员表达式语法

## explicit-member-expression

---

显式成员表达式 → 后缀表达式.[十进制数字] (02\_Lexical\_Structure.md#decimal-digit)

显式成员表达式 → 后缀表达式.标识符 泛型实参子句 可选

显式成员表达式 → 后缀表达式.[标识符] (02\_Lexical\_Structure.md#identifier) (参数名称)

## argument-names

---

| 参数名称 → 参数名 参数名称 可选

## argument-name

---

| 参数名 → 标识符:

## 后缀 self 表达式

---

后缀 `self` 表达式由某个表达式或类型名紧跟 `.self` 组成，其形式如下：

表达式 `.self`

类型 `.self`

第一种形式返回表达式的值。例如：`x.self` 返回 `x`。

第二种形式返回相应的类型。我们可以用它来获取某个实例的类型作为一个值来使用。例如，`SomeClass.self` 会返回 `SomeClass` 类型本身，你可以将其传递给相应函数或者方法作为参数。

| 后缀 `self` 表达式语法

## postfix-self-expression

---

| 后缀 `self` 表达式 → 后缀表达式 `. self`

## 下标表达式

---

可通过下标表达式访问相应的下标，形式如下：

| 表达式 [ 索引表达式 ]

要获取下标表达式的值，可将索引表达式作为下标表达式的参数来调用下标 getter。下标 setter 的调用方式与之一样。

关于下标的声明，请参阅 [协议下标声明](#)。

| 下标表达式语法

## subscript-expression

---

| 下标表达式 → 后缀表达式 [ 表达式列表 ]

## 强制取值表达式

---

当你确定可选值不是 `nil` 时，可以使用强制取值表达式来强制解包，形式如下：

| 表达式 !

如果该表达式的值不是 `nil`，则返回解包后的值。否则，抛出运行时错误。

返回的值可以被修改，无论是修改值本身，还是修改值的成员。例如：

```
var x: Int? = 0
x!++
// x 现在是 1

var someDictionary = ["a": [1, 2, 3], "b": [10, 20]]
someDictionary["a"]![0] = 100
// someDictionary 现在是 [b: [10, 20], a: [100, 2, 3]]
```

| 强制取值语法

## forced-value-expression

---

| 强制取值表达式 → 后缀表达式!

## 可选链表达式

---

可选链表达式提供了一种使用可选值的便捷方法，形式如下：

| 表达式 ?

后缀 ? 运算符会根据表达式生成可选链表达式而不会改变表达式的值。

如果某个后缀表达式包含可选链表达式，那么它的执行过程会比较特殊。如果该可选链表达式的值是 nil，整个后缀表达式会直接返回 nil。如果该可选链表达式的值不是 nil，则返回可选链表达式解包后的值，并将该值用于后缀表达式中剩余的表达式。在这两种情况下，整个后缀表达式的值都会是可选类型。

如果某个后缀表达式中包含了可选链表达式，那么只有最外层的表达式会返回一个可选类型。例如，在下面的例子中，如果 c 不是 nil，那么它的值会被解包，然后通过 .property 访问它的属性，接着进一步通过 .performAction() 调用相应方法。整个 c?.property.performAction() 表达式返回一个可选类型的值，而不是多重可选类型。

```
var c: SomeClass?
var result: Bool? = c?.property.performAction()
```

上面的例子跟下面的不使用可选链表达式的例子等价：

```
var result: Bool? = nil
if let unwrappedC = c {
    result = unwrappedC.property.performAction()
}
```

可选链表达式解包后的值可以被修改，无论是修改值本身，还是修改值的成员。如果可选链表达式的值为 nil，则表达式右侧的赋值操作不会被执行。例如：

```
func someFunctionWithSideEffects() -> Int {  
    // 译者注：为了能看出此函数是否被执行，加上了一句打印  
    print("someFunctionWithSideEffects")  
    return 42  
}  
  
var someDictionary = ["a": [1, 2, 3], "b": [10, 20]]  
  
someDictionary["not here"]?[0] = someFunctionWithSideEffects()  
// someFunctionWithSideEffects 不会被执行  
// someDictionary 依然是 ["b": [10, 20], "a": [1, 2, 3]]  
  
someDictionary["a"]?[0] = someFunctionWithSideEffects()  
// someFunctionWithSideEffects 被执行并返回 42  
// someDictionary 现在是 ["b": [10, 20], "a": [42, 2, 3]]
```

| 可选链表达式语法

## optional-chaining-expression

---

| 可选链表达式 → 后缀表达式？

# 语句 · GitBook

 [runoob.com/manual/gitbook/swift5/source/\\_book/chapter3/05\\_Statements.html](https://runoob.com/manual/gitbook/swift5/source/_book/chapter3/05_Statements.html)

## 语句 (Statements)

在 Swift 中，有三种类型的语句：简单语句、编译器控制语句和控制流语句。简单语句是最常见的，用于构造表达式或者声明。编译器控制语句允许程序改变编译器的行为，包含编译配置语句和行控制语句。

控制流语句则用于控制程序执行的流程，Swift 中有多种类型的控制流语句：循环语句、分支语句和控制转移语句。循环语句用于重复执行代码块；分支语句用于执行满足特定条件的代码块；控制转移语句则用于改变代码的执行顺序。另外，Swift 提供了 `do` 语句，用于构建局部作用域，还用于错误的捕获和处理；还提供了 `defer` 语句，用于退出当前作用域之前执行清理操作。

是否将分号（`;`）添加到语句的末尾是可选的。但若要在同一行内写多条独立语句，则必须使用分号。

### 语句语法

语句 → 表达式 ; 可选

语句 → 声明 ; 可选

语句 → 循环语句 ; 可选

语句 → 分支语句 ; 可选

语句 → 带标签的语句 ; 可选

语句 → 控制转移语句 ; 可选

语句 → defer 语句 ; 可选

语句 → do 语句 ; 可选

语句 → 编译器控制语句

多条语句 → 语句 多条语句 可选

## 循环语句

循环语句会根据特定的循环条件来重复执行代码块。Swift 提供三种类型的循环语句：`for-in` 语句、`while` 语句和 `repeat-while` 语句。

通过 `break` 语句和 `continue` 语句可以改变循环语句的控制流。有关这两条语句，详情参见 [Break 语句](#) 和 [Continue 语句](#)。

## 循环语句语法

### loop-statement

---

循环语句 → for-in 语句

循环语句 → while 语句

循环语句 → repeat-while 语句

## For-In 语句

---

**for-in** 语句会为集合（或实现了 Sequence 协议的任意类型）中的每一项执行一次代码块。

**for-in** 语句的形式如下：

```
for item in collection {  
    statements  
}
```

**for-in** 语句在循环开始前会调用集合表达式（ collection expression ）的 makeliterator() 方法来获取一个实现了 IteratorProtocol 协议的迭代器类型。接下来循环开始，反复调用该迭代器的 next() 方法。如果其返回值不是 nil，它将会被赋给 item，然后执行循环体语句，执行完毕后回到循环开始处，继续重复这一过程；否则，既不会赋值也不会执行循环体语句， **for-in** 语句至此执行完毕。

| for-in 语句语法

### for-in-statement

---

| for-in 语句 → **for case** 可选 模式 in 表达式 where 子句 可选 代码块

## While 语句

---

只要循环条件为真， **while** 语句就会重复执行代码块。

**while** 语句的形式如下：

```
while condition {  
    statements  
}
```

**while** 语句的执行流程如下：

1. 判断条件（ condition ）的值。如果为 true，转到第 2 步；如果为 false， **while** 语句至此执行完毕。
2. 执行循环体中的语句，然后重复第 1 步。

由于会在执行循环体中的语句前判断条件的值，因此循环体中的语句可能会被执行若干次，也可能一次也不会被执行。

条件的结果必须是 Bool 类型或者 Bool 的桥接类型。另外，条件语句也可以使用可选绑定，请参阅 [可选绑定](#)。

| while 语句语法

## while-statement

---

| while 语句 → while 条件子句 代码块

### condition-clause

---

| 条件子句 → 表达式 | 表达式 , 条件列表

### condition

---

| 条件 → 表达式 | 可用性条件 | case 条件 | 可选绑定条件

### case-condition

---

| case 条件 → case 模式构造器

### optional-binding-condition

---

| 可选绑定条件 → let 模式构造器 | var 模式构造器

## Repeat-While 语句

---

`repeat-while` 语句至少执行一次代码块，之后只要循环条件为真，就会重复执行代码块。

`repeat-while` 语句的形式如下：

```
repeat {  
    statements  
} while condition
```

`repeat-while` 语句的执行流程如下：

1. 执行循环体中的语句，然后转到第 2 步。
2. 判断条件的值。如果为 `true`，重复第 1 步；如果为 `false`，`repeat-while` 语句至此执行完毕。

由于条件的值是在循环体中的语句执行后才进行判断，因此循环体中的语句至少会被执行一次。

条件的结果必须是 Bool 类型或者 Bool 的桥接类型。另外，条件语句也可以使用可选绑定，请参阅 [可选绑定](#)。

| repeat-while 语句语法

## repeat-while-statement

---

| *repeat-while* 语句 → repeat 代码块 while 表达式

# 分支语句

---

分支语句会根据一个或者多个条件来执行指定部分的代码。分支语句中的条件将会决定程序如何分支以及执行哪部分代码。Swift 提供三种类型的分支语句：`if` 语句、`guard` 语句和 `switch` 语句。

`if` 语句和 `switch` 语句中的控制流可以用 `break` 语句改变，请参阅 [Break 语句](#)。

| 分支语句语法

## branch-statement

---

| 分支语句 → if 语句

| 分支语句 → guard 语句

| 分支语句 → switch 语句

# If 语句

---

`if` 语句会根据一个或多个条件来决定执行哪一块代码。

`if` 语句有两种基本形式，无论哪种形式，都必须有花括号。

第一种形式是当且仅当条件为真时执行代码，像下面这样：

```
if condition {  
    statements  
}
```

第二种形式是在第一种形式的基础上添加 `else` 语句（通过引入 `else` 关键字），并且用于：当条件为真时执行一部分代码，当这同一个条件为假的时候执行另一部分代码。当只有一个 `else` 语句时，`if` 语句具有以下的形式：

```
if condition {  
    statements to execute if condition is true  
} else {  
    statements to execute if condition is false  
}
```

`if` 语句的 `else` 语句也可包含另一个 `if` 语句，从而形成一条链来测试更多的条件，像下面这样：

```
if condition 1 {  
    statements to execute if condition 1 is true  
} else if condition 2 {  
    statements to execute if condition 2 is true  
} else {  
    statements to execute if both conditions are false  
}
```

`if` 语句中条件的结果必须是 `Bool` 类型或者 `Bool` 的桥接类型。另外，条件语句也可以使用可选绑定，请参阅 [可选绑定](#)。

| `if` 语句语法

## if-statement

---

| `if` 语句 → `if` 条件子句 代码块 else 子句 可选

## else-clause

---

| `else` 子句 → `else` 代码块 | `else if` 语句

## Guard 语句

---

如果一个或者多个条件不成立，可用 `guard` 语句来退出当前作用域。

`guard` 语句的格式如下：

```
guard condition else {  
    statements  
}
```

`guard` 语句中条件的结果必须是 `Bool` 类型或者 `Bool` 的桥接类型。另外，条件也可以是一条可选绑定，请参阅 [可选绑定](#)。

在 `guard` 语句中进行可选绑定的任何常量或者变量，其可用范围从声明开始直到作用域结束。

`guard` 语句必须有 `else` 子句，而且必须在该子句中调用返回类型是 `Never` 的函数，或者使用下面的语句退出当前作用域：

- `return`
- `break`
- `continue`
- `throw`

关于控制转移语句，请参阅 [控制转移语句](#)。关于 `Never` 返回类型的函数，请参阅 [永不返回的函数](#)。

## guard-statement

| guard 语句 → guard 条件子句 else [代码块] (05\_Declarations.md#code-block)

## Switch 语句

`switch` 语句会根据控制表达式的值来决定执行哪部分代码。

`switch` 语句的形式如下：

```
switch control expression {
  case pattern 1:
    statements
  case pattern 2 where condition:
    statements
  case pattern 3 where condition,
    pattern 4 where condition:
    statements
  default:
    statements
}
```

`switch` 语句会先计算控制表达式的值，然后与每一个 `case` 的模式进行匹配。如果匹配成功，程序将会执行对应的 `case` 中的语句。另外，每一个 `case` 的作用域都不能为空，也就是说在每一个 `case` 的冒号（`:`）后面必须至少有一条语句。如果你不想在匹配到的 `case` 中执行代码，只需在该 `case` 中写一条 `break` 语句即可。

可以用作控制表达式的值是十分灵活的。除了标量类型外，如 `Int`、`Character`，你可以使用任何类型的值，包括浮点数、字符串、元组、自定义类型的实例和可选类型。控制表达式的值还可以用来匹配枚举类型中的成员值或是检查该值是否包含在指定的 `Range` 中。关于如何在 `switch` 语句中使用这些类型，请参阅 [控制流](#) 一章中的 [Switch](#)。

每个 `case` 的模式后面可以有一个 `where` 子句。`where` 子句由 `where` 关键字紧跟一个提供额外条件的表达式组成。因此，当且仅当控制表达式匹配一个 `case` 的模式且 `where` 子句的表达式为真时，`case` 中的语句才会被执行。在下面的例子中，控制表达式只会匹配包含两个相等元素的元组，例如 `(1, 1)`：

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

正如上面这个例子，也可以在模式中使用 `let`（或 `var`）语句来绑定常量（或变量）。这些常量（或变量）可以在对应的 `where` 子句以及 `case` 中的代码中使用。但是，如果一个 `case` 中含有多个模式，所有的模式必须包含相同的常量（或变量）绑定，并且每一个绑定的常量（或变量）必须在所有的条件模式中都有相同的类型。

`switch` 语句也可以包含默认分支，使用 `default` 关键字表示。只有所有 `case` 都无法匹配控制表达式时，默认分支中的代码才会被执行。一个 `switch` 语句只能有一个默认分支，而且必须在 `switch` 语句的最后面。

`switch` 语句中 `case` 的匹配顺序和源代码中的书写顺序保持一致。因此，当多个模式都能匹配控制表达式时，只有第一个匹配的 `case` 中的代码会被执行。

## Switch 语句必须是详尽的

---

在 Swift 中，`switch` 语句中控制表达式的每一个可能的值都必须至少有一个 `case` 与之对应。在某些无法面面俱到的情况下（例如，表达式的类型是 `Int`），你可以使用 `default` 分支满足该要求。

### 对未来枚举的 `case` 进行 `switch`

---

非冻结枚举（`nonfronzen enumeration`）是一种特殊的枚举类型，它可能在未来会增加新的枚举 `case`，即使这时候你已经编译并且发布了你的应用，所以在 `switch` 非冻结枚举前需要深思熟虑。当一个库的作者们把一个枚举标记为非冻结的，这意味着他们保留了增加新的枚举 `case` 的权利，并且任何和这个枚举交互的代码都要在不需要重新编译的条件下能够处理那些未来可能新加入的 `case`。只有那些标准库，比如用 Swift 实现的苹果的一些框架，C 以及 Objective-C 代码才能够声明非冻结枚举。你在 Swift 中声明的枚举不能是非冻结的。

当你对未来枚举进行 `switch` 时，你总是需要有一个 `default case`，即使每种枚举类型都已经有对应的 `case` 了。你可以在 `default` 前标注 `@unknown`，意思是这个 `case` 应该只匹配未来加入的枚举 `case`。如果你的 `default case` 中匹配了任何在编译时就能确定的枚举 `case`，Swift 会抛出一个警告。这可以很好地提醒你库的作者已经新增了一种 `case`，并且你还没有去处理。

以下就是一个例子，我们对标准库的 `Mirror.AncestorRepresentation` 枚举进行 `switch` 操作。每当有新的 `case` 加入，我们会得到一个警告，提示我们要去处理它。

```
let representation: Mirror.AncestorRepresentation = .generated
switch representation {
    case .customized:
        print("Use the nearest ancestor's implementation.")
    case .generated:
        print("Generate a default mirror for all ancestor classes.")
    case .suppressed:
        print("Suppress the representation of all ancestor classes.")
    @unknown default:
        print("Use a representation that was unknown when this code was compiled.")
}
// Prints "Generate a default mirror for all ancestor classes."
```

## 不存在隐式落入

---

当匹配到的 `case` 中的代码执行完毕后，`switch` 语句会直接退出，而不会继续执行下一个 `case`。这就意味着，如果你想执行下一个 `case`，需要显式地在当前 `case` 中使用 `fallthrough` 语句。关于 `fallthrough` 语句的更多信息，请参阅 [Fallthrough 语句](#)。

| switch 语句语法

## switch-statement

---

| switch 语句 → switch 表达式 { switch-case 列表 可选 }

## switch-cases

---

| switch case 列表 → switch-case switch-case 列表 可选

## switch-case

---

| switch case → case 标签 多条语句 | default 标签 多条语句 | conditional-switch-case

## case-label

---

| case 标签 → 属性 可选 case case 项列表 :

## case-item-list

---

| case 项列表 → 模式 where 子句 可选 | 模式 where 子句 可选 , case 项列表

## default-label

---

| default 标签 → 属性 可选 default :

## where-clause

---

| where-clause → where where 表达式

## where-expression

---

| where-expression → 表达式

## grammar\_conditional-switch-case

---

| conditional-switch-case → switch-if-directive-clause switch-elseif-directive-clauses 可选  
switch-else-directive-clause 可选 endif-directive

## grammar\_switch-if-directive-clause

---

| switch-if-directive 语句 → if-directive compilation-condition switch-cases 可选

## grammar\_switch-elseif-directive-clauses

---

| switch-elseif-directive 语句 (复数) → elseif-directive-clause switch-elseif-directive-clauses 可选

## grammar\_switch-elseif-directive-clause

---

| switch-elseif-directive 语句 → elseif-directive compilation-condition switch-cases 可选

## grammar\_switch-else-directive-clause

---

---

| *switch-else-directive* 语句 → [else-directive](#) [switch-cases](#) 可选

## 带标签的语句

---

你可以在循环语句或 `switch` 语句前面加上标签，它由标签名和紧随其后的冒号（`:`）组成。在 `break` 和 `continue` 后面跟上标签名可以显式地在循环语句或 `switch` 语句中改变相应的控制流。关于这两条语句用法，请参阅 [Break 语句](#) 和 [Continue 语句](#)。

标签的作用域在该标签所标记的语句内。可以嵌套使用带标签的语句，但标签名必须唯一。

关于使用带标签的语句的例子，请参阅 [控制流](#) 一章中的 [带标签的语句](#)。

| 带标签的语句语法

### **labeled-statement**

---

| 带标签的语句 → [语句标签](#) [循环语句](#)

| 带标签的语句 → [语句标签](#) [if 语句](#)

| 带标签的语句 → [语句标签](#) [switch 语句](#)

| 带标签的语句 → [语句标签](#) [do 语句](#)

### **statement-label**

---

| 语句标签 → [标签名称](#):

### **label-name**

---

| 标签名称 → [标识符](#)

## 控制转移语句

---

控制转移语句能够无条件地把控制权从一片代码转移到另一片代码，从而改变代码执行的顺序。Swift 提供五种类型的控制转移语句：`break` 语句、`continue` 语句、`fallthrough` 语句、`return` 语句和 `throw` 语句。

| 控制转移语句语法

### **control-transfer-statement**

---

控制转移语句 → `break` 语句

控制转移语句 → `continue` 语句

控制转移语句 → `fallthrough` 语句

控制转移语句 → `return` 语句

控制转移语句 → `throw` 语句

## Break 语句

---

`break` 语句用于终止循环语句、`if` 语句或 `switch` 语句的执行。使用 `break` 语句时，可以只写 `break` 这个关键词，也可以在 `break` 后面跟上标签名，像下面这样：

`break`

`break label name`

当 `break` 语句后面带标签名时，可用于终止由这个标签标记的循环语句、`if` 语句或 `switch` 语句的执行。

而只写 `break` 时，则会终止 `switch` 语句或 `break` 语句所属的最内层循环语句的执行。不能使用 `break` 语句来终止未使用标签的 `if` 语句。

无论哪种情况，控制权都会被转移给被终止的控制流语句后面的第一行语句。

关于使用 `break` 语句的例子，请参阅 [控制流一章的 Break 和 带标签的语句](#)。

`break` 语句语法

## **break-statement**

---

`break` 语句 → **break** 标签名称 可选

## Continue 语句

---

`continue` 语句用于终止循环中当前迭代的执行，但不会终止该循环的执行。使用 `continue` 语句时，可以只写 `continue` 这个关键词，也可以在 `continue` 后面跟上标签名，像下面这样：

`continue`

`continue label name`

当 `continue` 语句后面带标签名时，可用于终止由这个标签标记的循环中当前迭代的执行。

而当只写 `continue` 时，可用于终止 `continue` 语句所属的最内层循环中当前迭代的执行。

在这两种情况下，控制权都会被转移给循环语句的条件语句。

在 `for` 语句中，`continue` 语句执行后，增量表达式还是会被计算，这是因为每次循环体执行完毕后，增量表达式都会被计算。

关于使用 `continue` 语句的例子，请参阅 [控制流](#) 一章的 [Continue](#) 和 [带标签的语句](#)。

| `continue` 语句语法

## continue-statement

---

| `continue` 语句 → `continue` 标签名称 可选

## Fallthrough 语句

---

`fallthrough` 语句用于在 `switch` 语句中转移控制权。`fallthrough` 语句会把控制权从 `switch` 语句中的一个 `case` 转移到下一个 `case`。这种控制权转移是无条件的，即使下一个 `case` 的模式与 `switch` 语句的控制表达式的值不匹配。

`fallthrough` 语句可出现在 `switch` 语句中的任意 `case` 中，但不能出现在最后一个 `case` 中。同时，`fallthrough` 语句也不能把控制权转移到使用了值绑定的 `case`。

关于在 `switch` 语句中使用 `fallthrough` 语句的例子，请参阅 [控制流](#) 一章的 [控制转移语句](#)。

| `fallthrough` 语句语法

## fallthrough-statement

---

| `fallthrough` 语句 → `fallthrough`

## Return 语句

---

`return` 语句用于在函数或方法的实现中将控制权转移到调用函数或方法，接着程序将会从调用位置继续向下执行。

使用 `return` 语句时，可以只写 `return` 这个关键词，也可以在 `return` 后面跟上表达式，像下面这样：

| `return`

| `return expression`

当 `return` 语句后面带表达式时，表达式的值将会返回给调用函数或方法。如果表达式的值的类型与函数或者方法声明的返回类型不匹配，Swift 则会在返回表达式的值之前将表达式的值的类型转换为返回类型。

## 注意

正如 [可失败构造器](#) 中所描述的，`return nil` 在可失败构造器中用于表明构造失败。

而只写 `return` 时，仅仅是从该函数或方法中返回，而不返回任何值（也就是说，函数或方法的返回类型为 `Void` 或者说 `()`）。

## return 语句语法

### return-statement

`return` 语句 → `return 表达式` 可选

## Throw 语句

### Throw 语句

`throw` 语句出现在抛出函数或者抛出方法体内，或者类型被 `throws` 关键字标记的闭包表达式体内。

`throw` 语句使程序在当前作用域结束执行，并向外围作用域传播错误。抛出的错误会一直传递，直到被 `do` 语句的 `catch` 子句处理掉。

`throw` 语句由 `throw` 关键字紧跟一个表达式组成，如下所示：

`throw expression`

表达式的结果必须符合 `ErrorType` 协议。

关于如何使用 `throw` 语句的例子，请参阅 [错误处理](#) 一章的 [用 throwing 函数传递错误](#)。

## throw 语句语法

### throw-statement

`throw` 语句 → `throw 表达式`

## Defer 语句

`defer` 语句用于在退出当前作用域之前执行代码。

`defer` 语句形式如下：

```
defer {  
    statements  
}
```

在 `defer` 语句中的语句无论程序控制如何转移都会被执行。在某些情况下，例如，手动管理资源时，比如关闭文件描述符，或者即使抛出了错误也需要执行一些操作时，就可以使用 `defer` 语句。

如果多个 `defer` 语句出现在同一作用域内，那么它们执行的顺序与出现的顺序相反。给定作用域中的第一个 `defer` 语句，会在最后执行，这意味着代码中最靠后的 `defer` 语句中引用的资源可以被其他 `defer` 语句清理掉。

```
func f() {  
    defer { print("First") }  
    defer { print("Second") }  
    defer { print("Third") }  
}  
f()  
// 打印“Third”  
// 打印“Second”  
// 打印“First”
```

`defer` 语句中的语句无法将控制权转移到 `defer` 语句外部。

| `defer` 语句语法

## defer-statement

---

| 延迟语句 → `defer` 代码块

## Do 语句

---

`do` 语句用于引入一个新的作用域，该作用域中可以含有一个或多个 `catch` 子句，`catch` 子句中定义了一些匹配错误条件的模式。`do` 语句作用域内定义的常量和变量只能在 `do` 语句作用域内使用。

Swift 中的 `do` 语句与 C 中限定代码块界限的大括号（`{}`）很相似，也并不会降低程序运行时的性能。

`do` 语句的形式如下：

```
do {  
    try expression  
    statements  
} catch pattern 1 {  
    statements  
} catch pattern 2 where condition {  
    statements  
}
```

如同 `switch` 语句，编译器会判断 `catch` 子句是否有遗漏。如果 `catch` 子句没有遗漏，则认为错误已被处理。否则，错误会自动传递到外围作用域，被某个 `catch` 子句处理掉或者被用 `throws` 关键字声明的抛出函数继续向外抛出。

为了确保错误已经被处理，可以让 `catch` 子句使用匹配所有错误的模式，如通配符模式（`_`）。如果一个 `catch` 子句不指定一种具体模式，`catch` 子句会匹配任何错误，并绑定到名为 `error` 的局部常量。有关在 `catch` 子句中使用模式的更多信息，请参阅 [模式](#)。

关于如何在 `do` 语句中使用一系列 `catch` 子句的例子，请参阅 [错误处理](#)。

### | do 语句语法

## do-statement

---

| `do` 语句 → do 代码块 多条 catch 子句 可选

## catch-clauses

---

| 多条 catch 子句 → catch 子句 多条 catch 子句 可选

## catch-clause

---

| `catch` 子句 → catch 模式 可选 where 子句 可选 代码块

## 编译器控制语句

---

编译器控制语句允许程序改变编译器的行为。Swift 有三种编译器控制语句：条件编译语句、线路控制语句和编译时诊断语句。

### | 编译器控制语句语法

## compiler-control-statement

---

| 编译器控制语句 → 条件编译语句

| 编译器控制语句 → 线路控制语句

| 编译器控制语句 → 诊断语句

## 条件编译代码块

---

条件编译代码块可以根据一个或多个配置来有条件地编译代码。

每一个条件编译代码块都以 `#if` 开始，`#endif` 结束。如下：

```
#if compilation condition  
statements  
#endif
```

和 `if` 语句的条件不同，编译配置的条件是在编译时进行判断的。只有编译配置在编译时判断为 `true` 的情况下，相应的语句才会被编译和执行。

编译配置可以是 `true` 和 `false` 的字面量，也可以是使用 `-D` 命令行标志的标识符，或者是下列表格中的任意一个平台检测函数。

函数	可用参数
<code>os()</code>	<code>OSX</code> , <code>iOS</code> , <code>watchOS</code> , <code>tvOS</code> , <code>Linux</code>
<code>arch()</code>	<code>i386</code> , <code>x86_64</code> , <code>arm</code> , <code>arm64</code>
<code>swift()</code>	<code>&gt;=</code> 或 <code>&lt;</code> 后跟版本号
<code>compiler()</code>	<code>&gt;=</code> 或 <code>&lt;</code> 后跟版本号
<code>canImport()</code>	模块名
<code>targetEnvironment()</code>	模拟器

在 `swift()` 和 `compiler()` 之后的版本号包含有主版本号，可选副版本号，可选补丁版本号类似，并且用 `(.)` 来分隔。在比较符和版本号之间不能有空格，版本号与前面的函数相对应，比如 `compiler()` 对应的就是这个编译器的版本号，`swift()` 对应的就是你要编译的 `Swift` 语言的版本号。举个简单的例子，如果你在使用 `Swift 5` 的编译器，想编译 `Swift 4.2`，可以看下面的例子：

```
#if compiler(>=5)
print("Compiled with the Swift 5 compiler or later")
#endif
#if swift(>=4.2)
print("Compiled in Swift 4.2 mode or later")
#endif
#if compiler(>=5) && swift(<5)
print("Compiled with the Swift 5 compiler or later in a Swift mode earlier than 5")
#endif
// 打印 "Compiled with the Swift 5 compiler or later"
// 打印 "Compiled in Swift 4.2 mode or later"
// 打印 "Compiled with the Swift 5 compiler or later in a Swift mode earlier than 5"
```

`canImport()` 后面跟的变量是模块的名字，这里这个模块可能并不是每个平台上都存在的。使用它来检测是否可以导入这个模块，如果模块存在就返回 `true`，否则返回 `false`。

`targetEnvironment()` 当为模拟器编译时返回 `true`，否则返回 `false`。

### 注意

`arch(arm)` 平台检测函数在 ARM 64 位设备上不会返回 `true`。如果代码在 32 位的 iOS 模拟器上编译，`arch(i386)` 平台检测函数会返回 `true`。

你可以使用逻辑操作符 `&&`、`||` 和 `!` 来组合多个编译配置，还可以使用圆括号来进行分组。

就像 `if` 语句一样，你可以使用 `#elseif` 子句来添加任意多个条件分支来测试不同的编译配置。你也可以使用 `#else` 子句来添加最终的条件分支。包含多个分支的编译配置语句例子如下：

```
#if compilation condition 1
statements to compile if compilation condition 1 is true
#endif compilation condition 2
statements to compile if compilation condition 2 is true
#else
statements to compile if both compilation conditions are false
#endif
```

### 注意

即使没有被编译，编译配置中的语句仍然会被解析。然而，唯一的例外是编译配置语句中包含语言版本检测函数：仅当 Swift 编译器版本和语言版本检测函数中指定的版本号匹配时，语句才会被解析。这种设定能确保旧的编译器不会尝试去解析新 Swift 版本的语法。

## build-config-statement

---

条件编译代码块语法

### grammar\_conditional-compilation-block

---

条件编译代码块 → if-directive 语句 elseif-directive 语句 (复数) 可选 else-directive 语句 可选 endif-directive

### grammar\_if-directive-clause

---

if-directive 语句 → if-directive 编译条件语句 (复数) 可选

### grammar\_elseif-directive-clauses

---

elseif-directive 语句 (复数) → elseif-directive 语句 elseif-directive 语句 (复数)

### grammar\_elseif-directive-clauses

---

elseif-directive 语句 → elseif-directive 编译条件语句 (复数) 可选

### grammar\_else-directive-clause

---

*else-directive* 语句 → else-directive 语句 (复数) 可选

*if-directive* → #if

*elseif-directive* → #elseif

*else-directive* → #else

*endif-directive* → #endif

## compilation-condition

---

编译条件 → 平台条件

编译条件 → 标识符

编译条件 → 布尔值字面量

编译条件 → ( 编译条件 )

编译条件 → ! 编译条件

编译条件 → 编译条件 && 编译条件

编译条件 → 编译条件 || 编译条件

## grammar\_platform-condition

---

### grammar\_platform-condition-os

---

| 平台条件 → os ( 操作系统 )

### grammar\_platform-condition-arch

---

| 平台条件 → arch ( 架构 )

### grammar\_platform-condition-swift

---

| 平台条件 → swift (  $\geq$  swift 版本 ) | swift (  $<$  swift 版本 )

### grammar\_platform-condition-compiler

---

| 平台条件 → compiler (  $\geq$  swift 版本 ) | compiler (  $<$  swift 版本 )

### grammar\_platform-condition-canImport

---

| 平台条件 → canImport ( 模块名 )

### grammar\_platform-condition-targetEnvironment

---

| 平台条件 → targetEnvironment (环境)

## operating-system

---

| 操作系统 → macOS | iOS | watchOS | tvOS

## architecture

---

| 架构 → i386 | x86\_64 | arm | arm64

## swift-version

---

| swift 版本 → 十进制数字.swift 版本延续 可选

## grammar\_swift-version-continuation

---

| swift 版本延续 → .十进制数字 swift 版本延续 可选

## grammar\_module-name

---

| 模块名 → identifier

## grammar\_environment

---

| 环境 → 模拟器

# 行控制语句

---

行控制语句可以为被编译的源代码指定行号和文件名，从而改变源代码的定位信息，以便进行分析和调试。

行控制语句形式如下：

#sourceLocation(file: filename , line: line number )

#sourceLocation()

第一种的行控制语句会改变该语句之后的代码中的字面量表达式 #line 和 #file 所表示的值。行号 是一个大于 0 的整形字面量，会改变 #line 表达式的值。文件名 是一个字符串字面量，会改变 #file 表达式的值。

第二种的行控制语句，#sourceLocation()，会将源代码的定位信息重置回默认的行号和文件名。

## line-control-statement

---

行控制语句语法

行控制语句 → #sourceLocation(file: 文件名, line: 行号)

行控制语句 → #sourceLocation()

## line-number

---

行号 → 大于 0 的十进制整数

## file-name

---

文件名 → 静态字符串字面量

## 编译时诊断语句

---

编译时诊断语句允许编译器在编译的时候可以发出错误或者警告。语句形式如下：

```
#error("error message")
#warning("warning message")
```

第一句会抛出错误信息并终止编译，第二句会发出警告信息但是编译会继续进行。你可以通过静态字符串字面量来书写诊断信息，静态字符串字面量不能使用字符串 interpolation 或者 concatenation，但可以使用多行的形式。

编译时诊断语句语法

## grammar\_compile-time-diagnostic-statement

---

诊断语句 → #error (diagnostic-message)

诊断语句 → #warning (diagnostic-message)

诊断语句 → 静态字符串字面量

## 可用性条件

---

可用性条件可作为 if , while , guard 语句的条件，可以在运行时基于特定的平台参数来查询 API 的可用性。

可用性条件的形式如下：

```
if #available(platform name version, ...) {
    statements to execute if the APIs are available
} else {
    fallback statements to execute if the APIs are unavailable
}
```

使用可用性条件来执行一个代码块时，取决于使用的 API 在运行时是否可用，编译器会根据可用性条件提供的信息来决定是否执行相应的代码块。

可用性条件使用一系列逗号分隔的平台名称和版本。使用 `iOS` , `OSX` , 以及 `watchOS` 等作为平台名称，并写上相应的版本号。`*` 参数是必须写的，用于处理未来的潜在平台。可用性条件确保了运行时的平台不低于条件中指定的平台版本时才执行代码块。

与布尔类型的条件不同，不能用逻辑运算符 `&&` 和 `||` 组合可用性条件。

### 可用性条件语法

## **availability-condition**

---

可用性条件 → #available (可用性参数列表)

## **availability-arguments**

---

可用性参数列表 → 可用性参数 | 可用性参数, 可用性参数列表

## **availability-argument**

---

可用性参数 → 平台名称 平台版本

可用性条件 → \*

## **platform-name**

---

平台名称 → **iOS** | **iOSApplicationExtension**

平台名称 → **OSX** | **macOSApplicationExtension**

平台名称 → **watchOS**

平台名称 → **tvOS**

## **platform-version**

---

平台版本 → 十进制数字

平台版本 → 十进制数字.十进制数字

平台版本 → 十进制数字.十进制数字.十进制数字



## 声明 (Declarations)

---

*声明 (declaration)* 用以向程序里引入新的名字或者结构。举例来说，可以使用声明来引入函数和方法，变量和常量，或者定义新的具有命名的枚举、结构体、类和协议类型。还可以使用声明来扩展一个既有的具有命名的类型的行为，或者在程序里引入在其它地方声明的符号。

在 Swift 中，大多数声明在某种意义上讲也是定义，因为它们在声明时往往伴随着实现或初始化。由于协议并不提供实现，大多数协议成员仅仅只是声明而已。为了方便起见，也是因为这些区别在 Swift 中并不是很重要，“声明”这个术语同时包含了声明和定义两种含义。

## 声明语法

### **declaration**

---

声明 → 导入声明

声明 → 常量声明

声明 → 变量声明

声明 → 类型别名声明

声明 → 函数声明

声明 → 枚举声明

声明 → 结构体声明

声明 → 类声明

声明 → 协议声明

声明 → 构造器声明

声明 → 析构器声明

声明 → 扩展声明

声明 → 下标声明

声明 → 运算符声明

### **declarations**

---

多条声明 → 声明 多条声明 可选

## 顶级代码

---

Swift 的源文件中的顶级代码 (top-level code) 由零个或多个语句、声明和表达式组成。默认情况下，在一个源文件的顶层声明的变量，常量和其他具有命名的声明可以被同模块中的每一个源文件中的代码访问。可以使用一个访问级别修饰符来标记声明来覆盖这种默认行为，请参阅 [访问控制级别](#)。

### 顶级声明语法

顶级声明 → 多条语句 可选

## 代码块

---

代码块 (*code block*) 可以将一些声明和控制结构体组织在一起。它有如下的形式：

```
{  
    语句  
}
```

代码块中的“语句”包括声明、表达式和各种其他类型的语句，它们按照在源码中的出现顺序被依次执行。

代码块语法

### code-block

代码块 → { 多条语句 可选 }

## 导入声明

导入声明 (*import declaration*) 让你可以使用在其他文件中声明的内容。导入语句的基本形式是导入整个模块，它由 `import` 关键字和紧随其后的模块名组成：

`import` 模块

可以对导入操作提供更细致的控制，如指定一个特殊的子模块或者指定一个模块或子模块中的某个声明。提供了这些限制后，在当前作用域中，只有被导入的符号是可用的，而不是整个模块中的所有声明。

`import` 导入类型 模块.符号名  
`import` 模块.子模块

### grammar\_of\_an\_import\_declarati

导入声明语法

#### import-declaration

导入声明 → 特性列表 可选 `import` 导入类型 可选 导入路径

#### import-kind

导入类型 → typealias | struct | class | enum | protocol | let | var | func

#### import-path

导入路径 → 导入路径标识符 | 导入路径标识符. 导入路径

#### import-path-identifier

导入路径标识符 → 标识符 | 运算符

## 常量声明

常量声明 (*constant declaration*) 可以在程序中引入一个具有命名的常量。常量以关键字 `let` 来声明，遵循如下格式：

```
let 常量名称: 类型 = 表达式
```

常量声明在“常量名称”和用于初始化的“表达式”的值之间定义了一种不可变的绑定关系；当常量的值被设定之后，它就无法被更改。这意味着，如果常量以类对象来初始化，对象本身的内容是可以改变的，但是常量和该对象之间的绑定关系是不能改变的。

当一个常量被声明为全局常量时，它必须拥有一个初始值。在函数或者方法中声明一个常量时，它并不需要拥有一个初始值，只需要保证在第一次对其进行读操作之前为其设置一个值。在类或者结构体中声明一个常量时，它将作为常量属性 (*constant property*)。常量声明不能是计算型属性，因此也没有存取方法。

如果常量名称是元组形式，元组中每一项的名称都会和初始化表达式中对应的值进行绑定。

```
let (firstNumber, secondNumber) = (10, 42)
```

在上例中，`firstNumber` 是一个值为 `10` 的常量，`secondNumber` 是一个值为 `42` 的常量。所有常量都可以独立地使用：

```
print("The first number is \$(firstNumber).")
// 打印"The first number is 10."
print("The second number is \$(secondNumber).")
// 打印"The second number is 42."
```

当常量名称的类型（`:` 类型）可以被推断出时，类型注解在常量声明中是可选的，正如 [类型推断](#) 中所描述的。

声明一个常量类型属性要使用 `static` 声明修饰符。类的常量类型属性总是隐式地被标记为 `final`；你无法用 `class` 或 `final` 声明修饰符实现允许或禁止被子类重写的目的。类型属性在 [类型属性](#) 中有介绍。

如果还想获得更多关于常量的信息或者想在使用中获得帮助，请参阅 [常量和变量](#) 和 [存储属性](#)。

## grammar\_of\_a\_constant\_declaration

---

## 常量声明语法

### constant-declaration

常量声明 → 特性列表 可选 声明修饰符列表 可选 let 模式构造器列表

### pattern-initializer-list

模式构造器列表 → 模式构造器 | 模式构造器, 模式构造器列表

### pattern-initializer

模式构造器 → 模式构造器 可选

### initializer

构造器 → = 表达式

## 变量声明

变量声明 (*variable declaration*) 可以在程序中引入一个具有命名的变量，它以关键字 `var` 来声明。

变量声明有几种不同的形式，可以声明不同种类的命名值和可变值，如存储型和计算型变量和属性，属性观察器，以及静态变量属性。所使用的声明形式取决于变量声明的适用范围和打算声明的变量类型。

注意

也可以在协议声明中声明属性，详情请参阅 [协议属性声明](#)。

可以在子类中重写继承来的变量属性，使用 `override` 声明修饰符标记属性的声明即可，详情请参阅 [重写](#)。

## 存储型变量和存储型变量属性

使用如下形式声明一个存储型变量或存储型变量属性：

`var 变量名称: 类型 = 表达式`

可以在全局范围，函数内部，或者在类和结构体的声明中使用这种形式来声明一个变量。当变量以这种形式在全局范围或者函数内部被声明时，它代表一个存储型变量。当它在类或者结构体中被声明时，它代表一个 *存储型变量属性* (*stored variable property*)。

用于初始化的表达式不可以出现在协议的声明中出现，在其他情况下，该表达式是可选的。如果没有初始化表达式，那么变量声明必须包含类型注解 (`: type`)。

如同常量声明，如果变量名称是元组形式，元组中每一项的名称都会和初始化表达式中对应的值进行绑定。

正如名字所示，存储型变量和存储型变量属性的值会存储在内存中。

## 计算型变量和计算型属性

---

使用如下形式声明一个计算型变量或计算型属性：

```
var 变量名称: 类型 {  
    get {  
        语句  
    }  
    set(setter 名称) {  
        语句  
    }  
}
```

可以在全局范围、函数内部，以及类、结构体、枚举、扩展的声明中使用这种形式的声明。当变量以这种形式在全局范围或者函数内部被声明时，它表示一个计算型变量。当它在类、结构体、枚举、扩展声明的上下文中被声明时，它表示一个计算型属性 (*computed property*)。

getter 用来读取变量值，setter 用来写入变量值。setter 子句是可选的，getter 子句是必须的。不过也可以将这些子句都省略，直接返回请求的值，正如在 [只读计算型属性](#) 中描述的那样。但是如果提供了一个 setter 子句，就必须也提供一个 getter 子句。

setter 的圆括号以及 setter 名称是可选的。如果提供了 setter 名称，它就会作为 setter 的参数名称使用。如果不提供 setter 名称，setter 的参数的默认名称为 `newValue`，正如在 [便捷 setter 声明](#) 中描述的那样。

与存储型变量和存储型属性不同，计算型变量和计算型属性的值不存储在内存中。

要获得更多关于计算型属性的信息和例子，请参阅 [计算型属性](#)。

## 存储型变量和属性的观察器

---

可以在声明存储型变量或属性时提供 `willSet` 和 `didSet` 观察器。一个包含观察器的存储型变量或属性以如下形式声明：

```
var 变量名称: 类型 = 表达式 {  
    willSet(setter 名称) {  
        语句  
    }  
    didSet(setter 名称) {  
        语句  
    }  
}
```

可以在全局范围、函数内部，或者类、结构体的声明中使用这种形式的声明。当变量以这种形式在全局范围或者函数内部被声明时，观察器表示一个存储型变量观察器。当它在类和结构体的声明中被声明时，观察器表示一个属性观察器。

可以为任何存储型属性添加观察器。也可以通过重写父类属性的方式为任何继承的属性（无论是存储型还是计算型的）添加观察器，正如 [重写属性观察器](#) 中所描述的。

用于初始化的表达式在类或者结构的声明中是可选的，但是在其他声明中则是必须的。如果可以从初始化表达式中推断出类型信息，那么可以不提供类型注解。

当变量或属性的值被改变时，`willSet` 和 `didSet` 观察器提供了一种观察方法。观察器会在变量的值被改变时调用，但不会在初始化时被调用。

`willSet` 观察器只在变量或属性的值被改变之前调用。新的值作为一个常量传入 `willSet` 观察器，因此不可以在 `willSet` 中改变它。`didSet` 观察器在变量或属性的值被改变后立即调用。和 `willSet` 观察器相反，为了方便获取旧值，旧值会传入 `didSet` 观察器。这意味着，如果在变量或属性的 `didSet` 观察器中设置值，设置的新值会取代刚刚在 `willSet` 观察器中传入的那个值。

在 `willSet` 和 `didSet` 中，圆括号以及其中的 setter 名称是可选的。如果提供了一个 setter 名称，它就会作为 `willSet` 和 `didSet` 的参数被使用。如果不提供 setter 名称，`willSet` 观察器的默认参数名为 `newValue`，`didSet` 观察器的默认参数名为 `oldValue`。

提供了 `willSet` 时，`didSet` 是可选的。同样的，提供了 `didSet` 时，`willSet` 则是可选的。

要获得更多信息以及查看如何使用属性观察器的例子，请参阅 [属性观察器](#)。

## 类型变量属性

要声明一个类型变量属性，用 `static` 声明修饰符标记该声明。类可以改用 `class` 声明修饰符标记类的类型计算型属性从而允许子类重写超类的实现。类型属性在 [类型属性](#) 章节有详细讨论。

### `grammer_of_a_variable_declaration`

| 变量声明语法

#### `variable-declaration`

变量声明 → 变量声明头 模式构造器列表

变量声明 → 变量声明头 变量名称 类型注解 代码块

变量声明 → 变量声明头 变量名称 类型注解 getter-setter 代码块

变量声明 → 变量声明头 变量名称 类型注解 getter-setter 关键字代码块

变量声明 → 变量声明头 变量名称 构造器 willSet-didSet 代码块

变量声明 → 变量声明头 变量名称 类型注解 构造器可选 willSet-didSet 代码块

## variable-declaration-head

---

变量声明头 → 特性列表可选 声明修饰符列表可选 var

### variable-name

---

变量名称 → 标识符

## getter-setter-block

---

getter-setter 代码块 → 代码块

getter-setter 代码块 → { getter 子句 setter 子句 可选 }

getter-setter 代码块 → { setter 子句 getter 子句 }

### getter-clause

---

getter 子句 → 特性列表可选 get 代码块

### setter-clause

---

setter 子句 → 特性列表可选 set setter 名称可选 代码块

### setter-name

---

setter 名称 → ( 标识符 )

## getter-setter-keyword-block

---

getter-setter 关键字代码块 → { getter 关键字子句 setter 关键字子句 可选 }

getter-setter 关键字代码块 → { setter 关键字子句 getter 关键字子句 }

### getter-keyword-clause

---

getter 关键字子句 → 特性列表可选 get

### setter-keyword-clause

---

setter 关键字子句 → 特性列表可选 set

## willSet-didSet-block

---

*willSet-didSet* 代码块 → { willSet 子句 didSet 子句 可选 }

*willSet-didSet* 代码块 → { didSet 子句 willSet 子句 可选 }

### willSet-clause

---

*willSet 子句* → 特性列表 可选 willSet setter 名称 可选 代码块

### didSet-clause

---

*didSet 子句* → 特性列表 可选 didSet setter 名称 可选 代码块

## 类型别名声明

---

类型别名 (*type alias*) 声明可以在程序中为一个既有类型声明一个别名。类型别名声明语句使用关键字 `typealias` 声明，遵循如下的形式：

`typealias` 类型别名 = 现存类型

当声明一个类型的别名后，可以在程序的任何地方使用“别名”来代替现有类型。现有类型可以是具有命名的类型或者混合类型。类型别名不产生新的类型，它只是使用别名来引用现有类型。

类型别名声明可以通过泛型参数来给一个现有泛型类型提供名称。类型别名为现有类型的一部分或者全部泛型参数提供具体类型。例如：

```
typealias StringDictionary<Value> = Dictionary<String, Value>
```

// 下列两个字典拥有同样的类型

```
var dictionary1: StringDictionary<Int> = [:]  
var dictionary2: Dictionary<String, Int> = [:]
```

当一个类型别名带着泛型参数一起被声明时，这些参数的约束必须与现有参数的约束完全匹配。例如：

```
typealias DictionaryOfInts<Key: Hashable> = Dictionary<Key, Int>
```

因为类型别名可以和现有类型相互交换使用，类型别名不可以引入额外的类型约束。

如果在声明处省略所有泛型参数，一个类型别名可以传递已有类型的所有泛型参数。例如，此处声明的 `Diccionario` 类型别名拥有和 `Dictionary` 同样的约束和泛型参数。

```
typealias Diccionario = Dictionary
```

在协议声明中，类型别名可以为那些经常使用的类型提供一个更短更方便的名称，例如：

```
protocol Sequence {
    associatedtype Iterator: IteratorProtocol
    typealias Element = Iterator.Element
}

func sum<T: Sequence>(_ sequence: T) -> Int where T.Element == Int {
    // ...
}
```

假如没有类型别名，`sum` 函数将必须引用关联类型通过 `T.Iterator.Element` 的形式来替代 `T.Element`。

另请参阅 [协议关联类型声明](#)。

## [grammer\\_of\\_a\\_type\\_alias\\_declaration](#)

---

类型别名声明语法

### **typealias-declaration**

---

类型别名声明 → 特性列表 可选 访问级别修饰符 可选 `typealias` 类型别名名称 类型别子句 类型别名赋值

### **typealias-name**

---

类型别名名称 → 标识符

### **typealias-assignment**

---

类型别名赋值 → = 类型

## 函数声明

---

使用 **函数声明** (*function declaration*) 在程序中引入新的函数或者方法。在类、结构体、枚举，或者协议中声明的函数会作为方法。函数声明使用关键字 `func`，遵循如下的形式：

```
func 函数名称(参数列表) -> 返回类型 {
    语句
}
```

如果函数返回 `Void` 类型，返回类型可以省略，如下所示：

```
func 函数名称(参数列表) {
    语句
}
```

每个参数的类型都要标明，因为它们不能被推断出来。如果您在某个参数类型前面加上了 `inout`，那么这个参数就可以在这个函数作用域当中被修改。更多关于 `inout` 参数的讨论，请参阅 [输入输出参数](#)。

函数声明中语句只包含一个表达式，可以理解为返回该表达式的值。

函数可以使用元组类型作为返回类型来返回多个值。

函数定义可以出现在另一个函数声明内。这种函数被称作 **嵌套函数** (*nested function*)。

大多数时候，嵌套函数都是可逃逸的函数。仅当一个嵌套函数捕获了某个确保了永不逃逸的值——例如一个输入输出参数——或者传入一个非逃逸函数参数的时候，这个嵌套函数才是非逃逸的。

更多关于嵌套函数的讨论，请参阅 [嵌套函数](#)。

## 参数名

---

函数的参数列表由一个或多个函数参数组成，参数间以逗号分隔。函数调用时的参数顺序必须和函数声明时的参数顺序一致。最简单的参数列表有着如下的形式：

**参数名称** : **参数类型**

每个参数有一个参数名称，这个名称与实参标签一样都可以在函数体内被使用。默认情况下，参数名也会被作为实参标签来使用。例如：

```
func f(x: Int, y: Int) -> Int { return x + y }
f(x: 1, y: 2) // 参数 x 和 y 都有标签
```

可以按照如下两种形式之一，重写参数名称的默认行为：

**实参标签** **参数名称** : **参数类型** \_ **参数名称** : **参数类型**

在参数名称前的名称会作为这个参数的显式实参标签，它可以和参数名称不同。在函数或方法调用时，相对应的参数必须使用这个实参标签。

参数名称前的下划线（\_）可以去除参数的实参标签。在函数或方法调用时，相对应的参数必须去除标签。

```
func repeatGreeting(_ greeting: String, count n: Int) { /* Greet n times */ }
repeatGreeting("Hello, world!", count: 2) // count 有标签, greeting 没有
```

## 输入输出参数

---

输入输出参数被传递时遵循如下规则：

1. 函数调用时，参数的值被拷贝。
2. 函数体内部，拷贝后的值被修改。
3. 函数返回后，拷贝后的值被赋值给原参数。

这种行为被称为 **拷入拷出** (*copy-in copy-out*) 或 **值结果调用** (*call by value result*)。例如，当一个计算型属性或者一个具有属性观察器的属性被用作函数的输入输出参数时，其 getter 会在函数调用时被调用，而其 setter 会在函数返回时被调用。

作为一种优化手段，当参数值存储在内存中的物理地址时，在函数体内部和外部均会使用同一内存位置。这种优化行为被称为引用调用 (*call by reference*)，它满足了拷入拷出模式的所有要求，且消除了复制带来的开销。在代码中，要规范使用拷入拷出模式，不要依赖于引用调用。

不要使用传递给输入输出参数的值，即使原始值在当前作用域中依然可用。当函数返回时，你对原始值所做的更改会被拷贝的值所覆盖。不要依赖于引用调用的优化机制来试图避免这种覆盖。

不能将同一个值传递给多个输入输出参数，因为这种情况下的拷贝与覆盖行为的顺序是不确定的，因此原始值的最终值也将无法确定。

更多关于内存安全和内存独占权的讨论，请参阅 [内存安全](#)。

如果一个闭包或者嵌套函数捕获了一个输入输出参数，那么这个闭包或者嵌套函数必须是非逃逸的。如果你需要捕获一个输入输出参数，但并不对其进行修改或者在其他代码中观察其值变化，那么你可以使用捕获列表来显式地表明这是个不可变捕获。

```
func someFunction(a: inout Int) -> () -> Int {  
    return { [a] in return a + 1 }  
}
```

如果你需要捕获并修改一个输入输出参数，使用一个显式局部拷贝来进行修改操作，在一些例如多线程的场景中，这样做可以确保函数返回之前所有的修改都已完成。

如果嵌套函数在外层函数返回后才调用，嵌套函数对输入输出参数造成的任何改变将不会影响到原始值。例如：

```
func multithreadedFunction(queue: DispatchQueue, x: inout Int) {  
    // 创建一个局部拷贝并在适当时候手动拷贝回去  
    var localX = x  
    defer { x = localX }  
  
    // 并行地操作 localX，然后在函数返回前一直等待  
    queue.async { someMutatingOperation(&localX) }  
    queue.sync {}  
}
```

关于输入输出参数的详细讨论，请参阅 [输入输出参数](#)。

## 特殊参数

---

参数可以被忽略，数量可以不固定，还可以为其提供默认值，使用形式如下：

\_ : 参数类型  
参数名称: 参数类型...  
参数名称: 参数类型 = 默认参数值

以下划线（\_）命名的参数会被显式忽略，无法在函数内使用。

一个参数的基本类型名称如果紧跟着三个点 ( `...` ) , 会被视为可变参数。一个函数至少可以拥有一个可变参数 , 且必须是最后一个参数。可变参数会作为包含该参数类型元素的数组处理。举例来讲 , 可变参数 `Int...` 会作为 `[Int]` 来处理。关于使用可变参数的例子 , 请参阅 [可变参数](#)。

如果在参数类型后面有一个以等号 ( `=` ) 连接的表达式 , 该参数会拥有默认值 , 即给定表达式的值。当函数被调用时 , 给定的表达式会被求值。如果参数在函数调用时被省略了 , 就会使用其默认值。

```
func f(x: Int = 42) -> Int { return x }
```

```
f() // 有效, 使用默认值  
f(7) // 有效, 提供了值  
f(x: 7) // 无效, 该参数没有外部名称
```

## 特殊方法

---

枚举或结构体的方法如果会修改 `self` , 则必须以 `mutating` 声明修饰符标记。

子类重写超类中的方法必须以 `override` 声明修饰符标记。重写方法时不使用 `override` 修饰符 , 或者被 `override` 修饰符修饰的方法并未对超类方法构成重写 , 都会导致编译错误。

枚举或者结构体中的类型方法 , 要以 `static` 声明修饰符标记 , 而对于类中的类型方法 , 除了使用 `static` , 还可使用 `class` 声明修饰符标记。类中使用 `class` 声明修饰的方法可以被子类实现重写 ; 类中使用 `class final` 或 `static` 声明修饰的方法不可被重写。

## 抛出错误的函数和方法

---

可以抛出错误的函数或方法必须使用 `throws` 关键字标记。这类函数和方法被称为抛出函数和抛出方法。它们有着下面的形式 :

```
func 函数名称(参数列表) throws -> 返回类型 {  
    语句  
}
```

抛出函数或抛出方法的调用必须包裹在 `try` 或者 `try!` 表达式中 (也就是说 , 在作用域内使用 `try` 或者 `try!` 运算符) 。

`throws` 关键字是函数的类型的一部分 , 非抛出函数是抛出函数的子类型。所以 , 可以在使用抛出函数的地方使用非抛出函数。

不能仅基于函数能否抛出错误来进行函数重写。也就是说 , 可以基于函数的函数类型的参数能否抛出错误来进行函数重写。

抛出方法不能重写非抛出方法 , 而且抛出方法不能满足协议对于非抛出方法的要求。也就是说 , 非抛出方法可以重写抛出方法 , 而且非抛出方法可以满足协议对于抛出方法的要求。

## 重抛错误的函数和方法

---

函数或方法可以使用 `rethrows` 关键字来声明，从而表明仅当该函数或方法的一个函数类型的参数抛出错误时，该函数或方法才抛出错误。这类函数和方法被称为重抛函数和重抛方法。重新抛出错误的函数或方法必须至少有一个参数的类型为抛出函数。

```
func someFunction(callback: () throws -> Void) rethrows {
    try callback()
}
```

重抛函数或者方法不能够从自身直接抛出任何错误，这意味着它不能够包含 `throw` 语句。它只能够传递作为参数的抛出函数所抛出的错误。例如，在 `do-catch` 代码块中调用抛出函数，并在 `catch` 子句中抛出其它错误都是不允许的。

```
func alwaysThrows() throws {
    throw SomeError.error
}

func someFunction(callback: () throws -> Void) rethrows {
    do {
        try callback()
        try alwaysThrows() // 非法, alwaysThrows() 不是一个抛出函数类型的参数
    } catch {
        throw AnotherError.error
    }
}
```

抛出方法不能重写重抛方法，而且抛出方法不能满足协议对于重抛方法的要求。也就是说，重抛方法可以重写抛出方法，而且重抛方法可以满足协议对于抛出方法的要求。

## 永不返回的函数

---

Swift 定义了 `Never` 类型，它表示函数或者方法不会返回给它的调用者。`Never` 返回类型的函数或方法可以称为不归，不归函数、方法要么引发不可恢复的错误，要么永远不停地运作，这会使调用后本应执行得代码就不再执行了。但即使是不归函数、方法，抛错函数和重抛出函数也可以将程序控制转移到合适的 `catch` 代码块。

不归函数、方法可以在 `guard` 语句的 `else` 字句中调用，具体讨论在 [Guard 语句](#)。

你可以重写一个不归方法，但是新的方法必须保持原有的返回类型和没有返回的行为。

## grammar\_of\_a\_function\_declaration

---

| 函数声明语法

### function-declaration

---

| 函数声明 → 函数头 函数名 泛型形参子句<sub>可选</sub> 函数签名 泛型 where 子句 函数体<sub>可选</sub>

### function-head

---

函数头 → 特性列表<sub>可选</sub> 声明修饰符列表<sub>可选</sub> func

### function-name

---

函数名 → 标识符 | 运算符

### function-signature

---

函数签名 → 参数子句列表 throws<sub>可选</sub> 函数结果<sub>可选</sub>

函数签名 → 参数子句列表 rethrows 函数结果<sub>可选</sub>

### function-result

---

函数结果 → -> 特性列表<sub>可选</sub> 类型

### function-body

---

函数体 → 代码块

### parameter-clause

---

参数子句 → () | (参数列表)

### parameter-list

---

参数列表 → 参数 | 参数, 参数列表

### parameter

---

参数 → 外部参数名<sub>可选</sub> 内部参数名 类型注解 默认参数子句<sub>可选</sub>

参数 → 外部参数名<sub>可选</sub> 内部参数名 类型注解

参数 → 外部参数名<sub>可选</sub> 内部参数名 类型注解 ...

### external-parameter-name

---

外部参数名 → 标识符 | \_

### local-parameter-name

---

内部参数名 → 标识符 | \_

### default-argument-clause

---

默认参数子句 → = 表达式

## 枚举声明

---

在程序中使用枚举声明 (*enumeration declaration*) 来引入一个枚举类型。

枚举声明有两种基本形式，使用关键字 `enum` 来声明。枚举声明体包含零个或多个值，称为枚举用例，还可包含任意数量的声明，包括计算型属性、实例方法、类型方法、构造器、类型别名，甚至其他枚举、结构体和类。枚举声明不能包含析构器或者协议声明。

枚举类型可以采纳任意数量的协议，但是枚举不能从类、结构体和其他枚举继承。

不同于类或者结构体，枚举类型并不隐式提供默认构造器，所有构造器必须显式声明。一个构造器可以委托给枚举中的其他构造器，但是构造过程仅当构造器将一个枚举用例赋值给 `self` 后才算完成。

和结构体类似但是和类不同，枚举是值类型。枚举实例在被赋值到变量或常量时，或者传递给函数作为参数时会被复制。更多关于值类型的信息，请参阅 [结构体和枚举是值类型](#)。

可以扩展枚举类型，正如在 [扩展声明](#) 中讨论的一样。

## 任意类型的枚举用例

---

如下的形式声明了一个包含任意类型枚举用例的枚举变量：

```
enum 枚举名称: 采纳的协议 {  
    case 枚举用例1  
    case 枚举用例2(关联值类型)  
}
```

这种形式的枚举声明在其他语言中有时被叫做可识别联合。

在这种形式中，每个用例块由关键字 `case` 开始，后面紧接一个或多个以逗号分隔的枚举用例。每个用例名必须是独一无二的。每个用例也可以指定它所存储的指定类型的值，这些类型在关联值类型的元组中被指定，紧跟用例名之后。

具有关联值的枚举用例可以像函数一样使用，通过指定的关联值创建枚举实例。和真正的函数一样，你可以获取枚举用例的引用，然后在后续代码中调用它。

```
enum Number {  
    case integer(Int)  
    case real(Double)  
}  
  
// f 的类型为 (Int) -> Number  
let f = Number.integer  
  
// 利用 f 把一个整数数组转成 Number 数组  
let evenInts: [Number] = [0, 2, 4, 6].map(f)
```

要获得更多关于具有关联值的枚举用例的信息和例子，请参阅 [关联值](#)。

## 递归枚举

---

枚举类型可以具有递归结构，就是说，枚举用例的关联值类型可以是枚举类型自身。然而，枚举类型的实例具有值语义，这意味着它们在内存中有固定布局。为了支持递归，编译器必须插入一个间接层。

要让某个枚举用例支持递归，使用 `indirect` 声明修饰符标记该用例。

```
enum Tree<T> {  
    case empty  
    indirect case node(value: T, left: Tree, right:Tree)  
}
```

要让一个枚举类型的所有用例都支持递归，使用 `indirect` 修饰符标记整个枚举类型，当枚举有多个用例且每个用例都需要使用 `indirect` 修饰符标记的时候这将非常便利。

被 `indirect` 修饰符标记的枚举用例必须有一个关联值。使用 `indirect` 修饰符标记的枚举类型可以既包含有关联值的用例，同时还可包含没有关联值的用例。但是，它不能再单独使用 `indirect` 修饰符来标记某个用例。

## 拥有原始值的枚举用例

---

以下形式声明了一种枚举类型，其中各个枚举用例的类型均为同一种基本类型：

```
enum 枚举名称: 原始值类型, 采纳的协议 {  
    case 枚举用例1 = 原始值1  
    case 枚举用例2 = 原始值2  
}
```

在这种形式中，每一个用例块由 `case` 关键字开始，后面紧跟一个或多个以逗号分隔的枚举用例。和第一种形式的枚举用例不同，这种形式的枚举用例包含一个基础值，叫做原始值，各个枚举用例的原始值的类型必须相同。这些原始值的类型通过原始值类型指定，必须表示一个整数、浮点数、字符串或者字符。原始值类型必须符合 `Equatable` 协议和下列字面量转换协议中的一种：整型字面量需符合 `IntegerLiteralConvertible` 协议，浮点型字面量需符合 `FloatingPointLiteralConvertible` 协议，包含任意数量字符的字符串型字面量需符合 `StringLiteralConvertible` 协议，仅包含一个单一字符的字符串型字面量需符合 `ExtendedGraphemeClusterLiteralConvertible` 协议。每一个用例的名字和原始值必须唯一。

如果原始值类型被指定为 `Int`，则不必为用例显式地指定原始值，它们会隐式地被赋值 `0`、`1`、`2` 等。每个未被赋值的 `Int` 类型的用例会被隐式地赋值，其值为上一个用例的原始值加 `1`。

```
enum ExampleEnum: Int {  
    case a, b, c = 5, d  
}
```

在上面的例子中，`ExampleEnum.A` 的原始值是 `0`，`ExampleEnum.B` 的原始值是 `1`。因为 `ExampleEnum.C` 的原始值被显式地设定为 `5`，因此 `ExampleEnum.D` 的原始值会自动增长为 `6`。

如果原始值类型被指定为 `String` 类型，你不用明确地为用例指定原始值，每个没有指

定原始值的用例会隐式地将用例名字作为原始值。

```
enum GamePlayMode: String {  
    case cooperative, individual, competitive  
}
```

在上面这个例子中，`GamePlayMode.cooperative` 的原始值是 "cooperative"，`GamePlayMode.individual` 的原始值是 "individual"，`GamePlayMode.competitive` 的原始值是 "competitive"。

枚举用例具有原始值的枚举类型隐式地符合定义在 Swift 标准库中的 `RawRepresentable` 协议。所以，它们拥有一个 `rawValue` 属性和一个可失败构造器 `init?(rawValue: RawValue)`。可以使用 `rawValue` 属性去获取枚举用例的原始值，例如 `ExampleEnum.b.rawValue`。还可以根据原始值来创建一个相对应的枚举用例，只需调用枚举的可失败构造器，例如 `ExampleEnum(rawValue: 5)`，这个可失败构造器返回一个可选类型的用例。要获得更多关于具有原始值的枚举用例的信息和例子，请参阅 [原始值](#)。

## 访问枚举用例

使用点语法（`.`）来引用枚举类型的枚举用例，例如 `EnumerationType.enumerationCase`。当枚举类型可以由上下文推断而出时，可以省略它（但是 `.` 仍然需要），正如 [枚举语法](#) 和 [显式成员表达式](#) 所述。

可以使用 `switch` 语句来检验枚举用例的值，正如 [使用 switch 语句匹配枚举值](#) 所述。枚举类型是模式匹配的，依靠 `switch` 语句 `case` 块中的枚举用例模式，正如 [枚举用例模式](#) 所述。

## grammar\_of\_an\_enumeration\_declaration

枚举声明语法

### enum-declaration

枚举声明 → 特性列表 可选 访问级别修饰符 可选 联合风格枚举

枚举声明 → 特性列表 可选 访问级别修饰符 可选 原始值风格枚举

联合风格枚举 → indirect 可选 enum 枚举名称 泛型形参子句 可选 类型继承子句 可选 {  
 多个联合风格枚举成员 可选 }

### union-style-enum-members

多个联合风格枚举成员 → 联合风格枚举成员 多个联合风格枚举成员 可选

### union-style-enum-member

联合风格枚举成员 → 声明 | 联合风格枚举用例子句 | 编译控制流语句

### union-style-enum-case-clause

联合风格枚举用例子句 → 特性列表 可选 indirect 可选 case 联合风格枚举用例列表

### **union-style-enum-case-list**

---

联合风格枚举用例列表 → 联合风格枚举用例 | 联合风格枚举用例, 联合风格枚举用例列表

### **union-style-enum-case**

---

联合风格枚举用例 → 枚举用例名称 元组类型 可选

#### **enum-name**

---

枚举名称 → 标识符

#### **enum-case-name**

---

枚举用例名称 → 标识符

#### **raw-value-style-enum**

---

原始值风格枚举 → enum 枚举名称 泛型形参子句 可选 类型继承子句 泛型 where 子句 { 多个原始值风格枚举成员 }

#### **raw-value-style-enum-members**

---

多个原始值风格枚举成员 → 原始值风格枚举成员 多个原始值风格枚举成员 可选

#### **raw-value-style-enum-member**

---

原始值风格枚举成员 → 声明 | 原始值风格枚举用例子句 | 编译控制流语句

#### **raw-value-style-enum-case-clause**

---

原始值风格枚举用例子句 → 特性列表 可选 case 原始值风格枚举用例列表

#### **raw-value-style-enum-case-list**

---

原始值风格枚举用例列表 → 原始值风格枚举用例 | 原始值风格枚举用例, 原始值风格枚举用例列表

#### **raw-value-style-enum-case**

---

原始值风格枚举用例 → 枚举用例名称 原始值赋值 可选

#### **raw-value-assignment**

---

原始值赋值 → = 原始值字面量

#### **raw-value-literal**

---

原始值字面量 → 数字型字面量 | 字符串型字面量 | 布尔型字面量

## 结构体声明

---

使用 **结构体声明** (*structure declaration*) 可以在程序中引入一个结构体类型。结构体声明使用 **struct** 关键字，遵循如下的形式：

```
struct 结构体名称: 采纳的协议 {  
    多条声明  
}
```

结构体内可包含零个或多个声明。这些声明可以包括存储型和计算型属性、类型属性、实例方法、类型方法、构造器、下标、类型别名，甚至其他结构体、类、和枚举声明。结构体声明不能包含析构器或者协议声明。关于结构体的详细讨论和示例，请参阅 [类和结构体](#)。

结构体可以采纳任意数量的协议，但是不能继承自类、枚举或者其他结构体。

有三种方法可以创建一个已声明的结构体实例：

- 调用结构体内声明的构造器，正如 [构造器](#) 所述。
- 如果没有声明构造器，调用结构体的成员逐一构造器，正如 [结构体类型的成员逐一构造器](#) 所述。
- 如果没有声明构造器，而且结构体的所有属性都有初始值，调用结构体的默认构造器，正如 [默认构造器](#) 所述。

结构体的构造过程请参阅 [构造过程](#)。

结构体实例的属性可以用点语法（`.`）来访问，正如 [访问属性](#) 所述。

结构体是值类型。结构体的实例在被赋予变量或常量，或传递给函数作为参数时会被复制。关于值类型的更多信息，请参阅 [结构体和枚举是值类型](#)。

可以使用扩展声明来扩展结构体类型的行为，请参阅 [扩展声明](#)。

---

[grammar\\_of\\_a\\_structure\\_declaration](#)

## 结构体声明语法

### struct-declaration

结构体声明 → 特性列表 可选 访问级别修饰符 可选 struct 结构体名称 泛型形参子句 可选  
类型继承子句 可选 泛型 where 子句 可选 结构体主体

### struct-name

结构体名称 → 标识符

### struct-body

结构体主体 → { 多条声明 可选 }

### struct-name

结构体多个成员 → 结构体成员 结构体多个成员 可选

### struct-member

结构体成员 → 声明 | 编译控制流语句

## 类声明

可以在程序中使用类声明 (*class declaration*) 来引入一个类。类声明使用关键字 class，遵循如下的形式：

```
class 类名: 超类, 采纳的协议 {  
    多条声明  
}
```

类内可以包含零个或多个声明。这些声明可以包括存储型和计算型属性、实例方法、类型方法、构造器、唯一的析构器、下标、类型别名，甚至其他结构体、类和枚举声明。类声明不能包含协议声明。关于类的详细讨论和示例，请参阅 [类和结构体](#)。

一个类只能继承自一个超类，但是可以采纳任意数量的协议。超类紧跟在类名和冒号后面，其后跟着采纳的协议。泛型类可以继承自其它泛型类和非泛型类，但是非泛型类只能继承自其它非泛型类。当在冒号后面写泛型超类的名称时，必须写上泛型类的全名，包括它的泛型形参子句。

正如 [构造器声明](#) 所讨论的，类可以有指定构造器和便利构造器。类的指定构造器必须初始化类中声明的所有属性，并且必须在调用超类构造器之前。

类可以重写属性、方法、下标以及构造器。重写的属性、方法、下标和指定构造器必须以 `override` 声明修饰符标记。

为了要求子类去实现超类的构造器，使用 `required` 声明修饰符标记超类的构造器。子类实现超类构造器时也必须使用 `required` 声明修饰符。

虽然超类属性和方法声明可以被当前类继承，但是超类声明的指定构造器却不能。即便如此，如果当前类重写了超类的所有指定构造器，它就会继承超类的所有便利构造器。Swift 的类并不继承自一个通用基础类。

有两种方法来创建已声明的类的实例：

- 调用类中声明的构造器，请参阅 [构造器](#)。
- 如果没有声明构造器，而且类的所有属性都被赋予了初始值，调用类的默认构造器，请参阅 [默认构造器](#)。

类实例属性可以用点语法（`.`）来访问，请参阅 [访问属性](#)。

类是引用类型。当被赋予常量或变量，或者传递给函数作为参数时，类的实例会被引用，而不是被复制。关于引用类型的更多信息，请参阅 [结构体和枚举是值类型](#)。

可以使用扩展声明来扩展类的行为，请参阅 [扩展声明](#)。

## grammar\_of\_a\_class\_declaration

---

类声明语法

### class-declaration

---

类声明 → 特性列表 可选 访问级别修饰符 可选 final 可选 class 类名 泛型形参子句 可选 类型继承子句 可选 泛型 where 子句 可选 类主体

类声明 → 特性列表 可选 final 访问级别修饰符 可选 class 类名 泛型形参子句 可选 类型继承子句 可选 泛型 where 子句 可选 类主体

### class-name

---

类名 → 标识符

### class-body

---

类主体 → { 多条声明 可选 }

类多个成员 → 类成员 类多个成员 可选

### class-member

---

类成员 → 声明 | 编译控制流语句

## 协议声明

---

协议声明 (*protocol declaration*) 可以为程序引入一个命名的协议类型。协议声明只能在全局区域使用 `protocol` 关键字来进行声明，并遵循如下形式：

```
protocol 协议名称: 继承的协议 {  
    协议成员声明  
}
```

协议的主体包含零个或多个协议成员声明，这些成员描述了任何采纳该协议的类型必须满足的一致性要求。一个协议可以声明采纳者必须实现的某些属性、方法、构造器以及下标。协议也可以声明各种各样的类型别名，叫做关联类型，它可以指定协议的不同声明之间的关系。协议声明不能包括类、结构体、枚举或者其它协议的声明。协议成员声明会在后面进行讨论。

协议类型可以继承自任意数量的其它协议。当一个协议类型继承自其它协议的时候，来自其它协议的所有要求会聚合在一起，而且采纳当前协议的类型必须符合所有的这些要求。关于如何使用协议继承的例子，请参阅 [协议继承](#)。

### 注意

也可以使用协议合成类型来聚合多个协议的一致性要求，请参阅 [协议合成类型](#) 和 [协议合成](#)。

可以通过类型的扩展声明来采纳协议，从而为之前声明的类型添加协议一致性。在扩展中，必须实现所有采纳协议的要求。如果该类型已经实现了所有的要求，可以让这个扩展声明的主体留空。

默认地，符合某个协议的类型必须实现所有在协议中声明的属性、方法和下标。即便如此，可以用 `optional` 声明修饰符标注协议成员声明，以指定它们的实现是可选的。`optional` 修饰符仅仅可以用于使用 `objc` 特性标记过的协议。因此，仅仅类类型可以采用并符合包含可选成员要求的协议。更多关于如何使用 `optional` 声明修饰符的信息，以及如何访问可选协议成员的指导——例如不能确定采纳协议的类型是否实现了它们时——请参阅 [可选协议要求](#)。

为了限制协议只能被类类型采纳，需要使用 `AnyObject` 关键字来标记协议，将 `AnyObject` 关键字写在冒号后面的继承的协议列表的首位。例如，下面的协议只能被类类型采纳：

```
protocol SomeProtocol: AnyObject {  
    /* 这里是协议成员 */  
}
```

任何继承自标记有 `AnyObject` 关键字的协议也仅能被类类型采纳。

### 注意

如果协议已经用 `objc` 特性标记了，`AnyObject` 要求就隐式地应用于该协议，无需显式使用 `AnyObject` 关键字。

协议类型是命名的类型，因此它们可以像其他命名类型一样使用，正如 [协议作为类型](#) 所讨论的。然而，不能构造一个协议的实例，因为协议实际上不提供它们指定的要求的实现。

可以使用协议来声明作为代理的类或者结构体应该实现的方法，正如 [委托（代理）模式](#) 中所述。

## grammar\_of\_a\_protocol\_declaration

---

协议声明语法

### protocol-declaration

---

协议声明 → 特性列表 可选 访问级别修饰符 可选 protocol 协议名称 类型继承子句 可选 泛型 where 子句 可选 协议主体

### protocol-name

---

协议名称 → 标识符

### protocol-body

---

协议主体 → { 协议成员声明列表 可选 }

协议多个成员 → 协议成员 协议多个成员 可选

### protocol-member

---

协议成员 → 协议成员声明 | 编译控制流语句

### protocol-member-declaration

---

协议成员声明 → 协议属性声明

协议成员声明 → 协议方法声明

协议成员声明 → 协议构造器声明

协议成员声明 → 协议下标声明

协议成员声明 → 协议关联类型声明

### protocol-member-declarations

---

协议成员声明列表 → 协议成员声明 协议成员声明列表 可选

## 协议属性声明

---

协议可以通过在协议声明主体中引入一个协议属性声明，来声明符合的类型必须实现的属性。协议属性声明有一种特殊的变量声明形式：

```
var 属性名: 类型 { get set }
```

同其它协议成员声明一样，这些属性声明仅仅针对符合该协议的类型声明了 getter 和 setter 要求，你不能在协议中直接实现 getter 和 setter。

符合类型可以通过多种方式满足 getter 和 setter 要求。如果属性声明包含 `get` 和 `set` 关键字，符合类型就可以用存储型变量属性或可读可写的计算型属性来满足此要求，但是属性不能以常量属性或只读计算型属性实现。如果属性声明仅仅包含 `get` 关键字的话，它可以作为任意类型的属性被实现。关于如何实现协议中的属性要求的例子，请参阅 [属性要求](#)。

协议声明中声明一个类型属性，属性声明语句必须用 `static` 声明修饰符。当结构体和枚举遵循该协议时，使用 `static` 关键字修饰，而类遵循该协议时，使用 `static` 或 `class` 关键字皆可。当结构体，枚举或类添加扩展遵循协议时，和之前扩展用到的关键字保持一致。扩展为类属性提供默认实现时，必须使用 `static` 关键字修饰。

另请参阅 [变量声明](#)。

## grammar\_of\_an\_import\_declaration

---

协议属性声明语法

### protocol-property-declaration

---

协议属性声明 → 变量声明头 变量名称 类型注解 `getter-setter` 关键字代码块

## 协议方法声明

---

协议可以通过在协议声明主体中引入一个协议方法声明，来声明符合的类型必须实现的方法。协议方法声明和函数方法声明有着相同的形式，但有两项例外：它们不包括函数体，也不能包含默认参数。关于如何实现协议中的方法要求的例子，请参阅 [方法要求](#)。

协议声明中声明一个类型方法，方法声明语句必须用 `static` 声明修饰符。结构体和枚举遵循协议时，必须使用 `static` 关键字修饰，而类遵循协议时，使用 `static` 或 `class` 关键字皆可。当结构体，枚举或类添加扩展遵循协议时，和之前扩展用到的关键字保持一致。扩展为类方法提供默认实现时，必须使用 `static` 关键字修饰。

另请参阅 [函数声明](#)。

## grammar\_of\_a\_protocol\_declaration

---

协议方法声明语法

### protocol-method-declaration

---

协议方法声明 → 函数头 函数名 泛型形参子句 可选 函数签名 泛型 `where` 子句 可选

## 协议构造器声明

---

协议可以通过在协议声明主体中引入一个协议构造器声明，来声明符合的类型必须实现的构造器。协议构造器声明除了不包含实现主体外，和构造器声明有着相同的形式。

符合类型可以通过实现一个非可失败构造器或者 `init!` 可失败构造器来满足一个非可失败协议构造器的要求，可以通过实现任意类型的构造器来满足一个可失败协议构造器的要求。

类在实现一个构造器去满足一个协议的构造器要求时，如果这个类还没有用 `final` 声明修饰符标记，这个构造器必须用 `required` 声明修饰符标记。

另请参阅 [构造器声明](#)。

## grammar\_of\_a\_protocol\_initializer\_declaration

---

协议构造器声明语法

### protocol-initializer-declaration

---

协议构造器声明 → 构造器头 泛型形参子句 可选 参数子句 throws 可选 泛型 where 子句 可选

协议构造器声明 → 构造器头 泛型形参子句 可选 参数子句 rethrows 泛型 where 子句 可选

## 协议下标声明

---

协议可以通过在协议声明主体中引入一个协议下标声明，来声明符合的类型必须实现的下标。协议下标声明有一个特殊的下标声明形式：

`subscript (参数列表) -> 返回类型 { get set }`

下标声明只为符合类型声明了 `getter` 和 `setter` 要求。如果下标声明包含 `get` 和 `set` 关键字，符合类型也必须实现 `getter` 和 `setter` 子句。如果下标声明只包含 `get` 关键字，符合类型必须实现 `getter` 子句，可以选择是否实现 `setter` 子句。

协议声明中声明一个静态下标，下标声明语句必须用 `static` 声明修饰符。当结构体和枚举遵循该协议时，下标声明使用 `static` 关键字修饰，而类遵循该协议时，使用 `static` 或 `class` 关键字皆可。当结构体，枚举或类添加扩展遵循协议时，和之前扩展用到的关键字保持一致。扩展为下标声明提供默认实现时，必须使用 `static` 关键字修饰。

另请参阅 [下标声明](#)。

## grammar\_of\_a\_protocol\_subscript\_declaration

---

协议下标声明语法

### protocol-subscript-declaration

---

协议下标声明 → 下标头 下标结果 泛型 where 子句 可选 `getter-setter` 关键字代码块

## 协议关联类型声明

---

使用关键字 `associatedtype` 来声明协议关联类型。关联类型为作为协议声明的一部分

分，为某种类型提供了一个别名。关联类型和泛型参数子句中的类型参数很相似，但是它们和 `Self` 一样，用于协议中。`Self` 指代采纳协议的类型。要获得更多信息和例子，请参阅 [关联类型](#)。

在协议声明中使用泛型 `where` 子句来为继承的协议关联类型添加约束，且不需要重新声明关联类型。例如下面代码中的 `SubProtocol` 声明。

```
protocol SomeProtocol {  
    associatedtype SomeType  
}  
  
protocol SubProtocolA: SomeProtocol {  
    // 此类语法会引发警告。  
    associatedtype SomeType: Equatable  
}  
  
// 建议使用此语法。  
protocol SubProtocolB: SomeProtocol where SomeType: Equatable { }
```

另请参阅 [类型别名声明](#)。

## grammar\_of\_a\_protocol\_associated\_type\_declaration

---

协议关联类型声明语法

### protocol-associated-type-declaration

---

协议关联类型声明 → 特性列表 可选 访问级别修饰符 可选 `associatedtype` 类型别名头  
类型继承子句 可选 类型别名赋值 可选 泛型 `where` 子句 可选

## 构造器声明

---

构造器声明会为程序中的类、结构体或枚举引入构造器。构造器使用关键字 `init` 来声明，有两种基本形式。

结构体、枚举、类可以有任意数量的构造器，但是类的构造器具有不同的规则和行为。不同于结构体和枚举，类有两种构造器，即指定构造器和便利构造器，请参阅 [构造过程](#)。

采用如下形式声明结构体和枚举的构造器，以及类的指定构造器：

```
init(参数列表) {  
    构造语句  
}
```

类的指定构造器直接将类的所有属性初始化。它不能调用类中的其他构造器，如果类有超类，则必须调用超类的一个指定构造器。如果该类从它的超类继承了属性，必须在调用超类的指定构造器后才能修改这些属性。

指定构造器只能在类声明中声明，不能在扩展声明中声明。

结构体和枚举的构造器可以调用其他已声明的构造器，从而委托其他构造器来进行部分或者全部构造过程。

要为类声明一个便利构造器，用 `convenience` 声明修饰符来标记构造器声明：

```
convenience init(参数列表) {  
    构造语句  
}
```

便利构造器可以将构造过程委托给另一个便利构造器或一个指定构造器。但是，类的构造过程必须以一个将类中所有属性完全初始化的指定构造器的调用作为结束。便利构造器不能调用超类的构造器。

可以使用 `required` 声明修饰符，将便利构造器和指定构造器标记为每个子类都必须实现的构造器。这种构造器的子类实现也必须使用 `required` 声明修饰符标记。

默认情况下，超类中的构造器不会被子类继承。但是，如果子类的所有存储型属性都有默认值，而且子类自身没有定义任何构造器，它将继承超类的构造器。如果子类重写了超类的所有指定构造器，子类将继承超类的所有便利构造器。

和方法、属性和下标一样，需要使用 `override` 声明修饰符标记重写的指定构造器。

### 注意

如果使用 `required` 声明修饰符标记一个构造器，在子类中重写这种构造器时，无需使用 `override` 修饰符。

就像函数和方法，构造器也可以抛出或者重抛错误，你可以在构造器参数列表的圆括号之后使用 `throws` 或 `rethrows` 关键字来表明相应的抛出行为。

关于在不同类型中声明构造器的例子，请参阅 [构造过程](#)。

## 可失败构造器

可失败构造器可以生成所属类型的可选实例或者隐式解包可选实例，因此，这种构造器通过返回 `nil` 来指明构造过程失败。

声明生成可选实例的可失败构造器时，在构造器声明的 `init` 关键字后加追加一个问号（`init?`）。声明生成隐式解包可选实例的可失败构造器时，在构造器声明后追加一个叹号（`init!`）。使用 `init?` 可失败构造器生成结构体的一个可选实例的例子如下。

```
struct SomeStruct {  
    let string: String  
    //生成一个 SomeStruct 的可选实例  
    init?(input: String) {  
        if input.isEmpty {  
            // 丢弃 self，并返回 nil  
            return nil  
        }  
        string = input  
    }  
}
```

调用 `init?` 可失败构造器和调用非可失败构造器的方式相同，不过你需要处理可选类型的返回值。

```
if let actualInstance = SomeStruct(input: "Hello") {  
    // 利用 SomeStruct 实例做些事情  
} else {  
    // SomeStruct 实例的构造过程失败，构造器返回了 nil  
}
```

可失败构造器可以在构造器实现中的任意位置返回 `nil`。

可失败构造器可以委托任意种类的构造器。非可失败可以委托其它非可失败构造器或者 `init!` 可失败构造器。非可失败构造器可以委托超类的 `init?` 可失败指定构造器，但是需要使用强制解包，例如 `super.init()!`。

构造过程失败通过构造器委托来传递。具体来说，如果可失败构造器委托的可失败构造器构造过程失败并返回 `nil`，那么该可失败构造器也会构造失败并隐式地返回 `nil`。如果非可失败构造器委托的 `init!` 可失败构造器构造失败并返回了 `nil`，那么会发生运行时错误（如同使用 `!` 操作符去强制解包一个值为 `nil` 的可选值）。

子类可以用任意种类的指定构造器重写超类的可失败指定构造器，但是只能用非可失败指定构造器重写超类的非可失败指定构造器。

更多关于可失败构造器的信息和例子，请参阅 [可失败构造器](#)。

[grammar\\_of\\_an\\_initializer\\_declaration](#)

---

构造器声明语法

### initializer-declaration

---

构造器声明 → 构造器头 泛型形参子句可选 参数子句 throws 可选 泛型 where 子句可选  
构造器主体

构造器声明 → 构造器头 泛型形参子句可选 参数子句 rethrows 可选 泛型 where 子句可选  
构造器主体

### initializer-head

---

构造器头 → 特性列表可选 声明修饰符列表可选 init

构造器头 → 特性列表可选 声明修饰符列表可选 init ?

构造器头 → 特性列表可选 声明修饰符列表可选 init !

### initializer-body

---

构造器主体 → 代码块

## 析构器声明

---

析构器声明 (*deinitializer declaration*) 可以为类声明一个析构器。析构器没有参数，遵循如下格式：

```
deinit {  
    语句  
}
```

当没有任何强引用引用着类的对象，对象即将被释放时，析构器会被自动调用。析构器只能在类主体的声明中声明，不能在类的扩展声明中声明，并且每个类最多只能有一个析构器。

子类会继承超类的析构器，并会在子类对象将要被释放时隐式调用。继承链上的所有析构器全部调用完毕后子类对象才会被释放。

析构器不能直接调用。

关于如何在类声明中使用析构器的例子，请参阅 [析构过程](#)。

### grammar\_of\_a\_deinitializer\_declaration

---

析构器声明语法

### deinitializer-declaration

---

析构器声明 → 特性列表可选 deinit 代码块

## 扩展声明

---

扩展声明 (*extension declaration*) 可以扩展一个现存的类型的行为。扩展声明使用关键字 `extension`，遵循如下格式：

```
extension 类型名称 where 要求 {  
    声明语句  
}
```

扩展声明体可包含零个或多个声明语句。这些声明语句可以包括计算型属性、计算型类型属性、实例方法、类型方法、构造器、下标声明，甚至是类、结构体和枚举声明。扩展声明不能包含析构器、协议声明、存储型属性、属性观察器或其他扩展声明。关于扩展声明的详细讨论，以及各种扩展声明的例子，请参阅 [扩展](#)。

如果类型为类、结构体，或枚举类型，则扩展声明会扩展相应的类型。如果类型为协议类型，则扩展声明会扩展所有遵守这个协议的类型。在扩展的协议体中声明语句不能使用 `final` 标识符。

扩展声明可以为现存的类、结构体、枚举添加协议一致性，但是不能为类添加超类，因此在扩展声明的类型的冒号后面仅能指定一个协议列表。

扩展声明可以包含构造器声明。这意味着，如果被扩展的类型在其他模块中定义，构造器声明必须委托另一个在那个模块中声明的构造器，以确保该类型能被正确地初始化。

现存类型的属性、方法、构造器不能在扩展中被重写。

通过指定采纳的协议，扩展声明可以为一个现有的类、结构体或者枚举类型添加协议遵循：

```
extension 类型名称: 采纳的协议 where 约束条件 {  
    多条声明  
}
```

协议声明不能为现有的类添加类的继承关系，因此你只能在“类型名称”的冒号后面添加一系列协议。

## 条件遵循

---

你可以扩展一个泛型类型并使其有条件地遵循某协议，此后此类型的实例只有在特定的限制条件满足时才遵循此协议。在扩展声明中加入限制条件来为协议添加条件遵循。

## 已重写的限制条件会在某些泛型上下文中失效

---

对于一些通过条件遵循获得了特定行为的类型，在某些泛型上下文中，并不能够确保能够使用协议限制中的特定实现。为了说明这个行为，下面的例子中定义了两个协议以及一个有条件地遵循两个协议的泛型类型。

```

protocol Loggable {
    func log()
}

extension Loggable {
    func log() {
        print(self)
    }
}

protocol TitledLoggable: Loggable {
    static var logTitle: String { get }
}

extension TitledLoggable {
    func log() {
        print("\(Self.logTitle): \(self)")
    }
}

struct Pair<T>: CustomStringConvertible {
    let first: T
    let second: T
    var description: String {
        return "(\(first), \(second))"
    }
}

extension Pair: Loggable where T: Loggable { }

extension Pair: TitledLoggable where T: TitledLoggable {
    static var logTitle: String {
        return "Pair of '\(T.logTitle)'"
    }
}

extension String: TitledLoggable {
    static var logTitle: String {
        return "String"
    }
}

```

当其泛型类型遵循 `Loggable` 协议以及 `TitledLoggable` 协议时，结构体 `Pair` 遵循 `Loggable` 协议以及 `TitledLoggable` 协议。下面的例子中，`oneAndTwo` 是 `Pair<String>` 的一个实例。因为 `String` 遵循 `TitledLoggable`，因此 `oneAndTwo` 也遵循此协议。当 `log()` 方法被 `oneAndTwo` 直接调用时，此方法使用的是包含标题的特定版本。

```

let oneAndTwo = Pair(first: "one", second: "two")
oneAndTwo.log()
// Prints "Pair of 'String': (one, two)"

```

虽然如此，当 `oneAndTwo` 在泛型上下文中使用，或者它是 `Loggable` 类型的实例时，包含标题的特定版本 `log()` 方法不会被使用。Swift 只会根据这样的规则来选择 `log()` 的实现版本——`Pair` 遵循 `Loggable` 所需要的最少的限制条件。因此 `Loggable` 所提供的默认实现版本会被使用。

```
func doSomething<T: Loggable>(with x: T) {  
    x.log()  
}  
doSomething(with: oneAndTwo)  
// Prints "(one, two)"
```

当传入 `doSomething(_:)` 的实例调用 `log()` 时，打印结果省略了自定义标题。

## 协议遵循决不能冗余

一个具体的类型只能够遵循某特定协议一次。Swift 会把冗余的协议遵循标记为错误。你会在两种场景中遇到这种错误。第一种场景是，使用不同的限制条件来多次显式地遵循同一协议。第二种场景是，多次隐式地继承同一协议。以上两种场景会在下面章节中讨论。

## 解决显式冗余

对同一具体类型的多个扩展不能遵循同一协议，即便这些扩展有不同的显式限制条件。这个限制的具体示例在下面的例子中。两个扩展声明都试图添加对 `Serializable` 的条件遵循，一个为 `Int` 类型元素的数组，另一个为 `String` 类型元素的数组。

```
protocol Serializable {  
    func serialize() -> Any  
}  
  
extension Array: Serializable where Element == Int {  
    func serialize() -> Any {  
        // implementation  
    }  
}  
extension Array: Serializable where Element == String {  
    func serialize() -> Any {  
        // implementation  
    }  
}  
// 报错: redundant conformance of 'Array<Element>' to protocol 'Serializable'
```

如果你需要基于多个具体类型来添加条件遵循，那么创建一个新的协议，然后让每个类型都遵循此协议，最后在声明条件遵循时使用此协议作为条件限制。

```
protocol SerializableInArray { }  
extension Int: SerializableInArray { }  
extension String: SerializableInArray { }  
  
extension Array: Serializable where Element: SerializableInArray {  
    func serialize() -> Any {  
        // 具体实现  
    }  
}
```

## 解决隐式冗余

当一个具体类型有条件地遵循某协议，此类型会隐式地使用相同的条件遵循任一父协议。

如果你需要让一个类型有条件地遵循两个继承自同一父协议的协议，请显式地声明对父协议的遵循。这可以避免使用不同的限制条件隐式遵循同一父协议两次。

下面的例子中显式地声明了 `Array` 对 `Loggable` 的条件遵循，避免了在声明对 `TitledLoggable` 和 `TitledLoggable` 声明条件遵循时发生冲突。

```
protocol MarkedLoggable: Loggable {
    func markAndLog()
}

extension MarkedLoggable {
    func markAndLog() {
        print("-----")
        log()
    }
}

extension Array: Loggable where Element: Loggable { }
extension Array: TitledLoggable where Element: TitledLoggable {
    static var logTitle: String {
        return "Array of '\(Element.logTitle)'"
    }
}
extension Array: MarkedLoggable where Element: MarkedLoggable { }
```

如果不显式声明对 `Loggable` 的条件遵循，`Array` 其他的扩展会隐式地创建此声明，并引发错误：

```
extension Array: Loggable where Element: TitledLoggable { }
extension Array: Loggable where Element: MarkedLoggable { }
// 报错: redundant conformance of 'Array<Element>' to protocol 'Loggable'
```

## grammar\_of\_an\_extension\_declaration

扩展声明语法

### **extension-declaration**

扩展声明 → 特性 可选 访问级别修饰符 可选 extension 类型标识符 类型-继承-子句 可选 泛型 where 子句 可选 扩展主体

### **extension-body**

扩展主体 → { 多条声明 可选 }

多条声明 → 单条声明 多条声明 可选

单条声明 → 声明语句 | 编译控制流语句

## 下标声明

---

下标声明 (*subscript declaration*) 用于为特定类型的对象添加下标支持，通常也用于为访问集合、列表和序列中的元素提供语法便利。下标声明使用关键字 `subscript`，形式如下：

```
subscript (参数列表) -> 返回类型 {  
    get {  
        语句  
    }  
    set(setter 名称) {  
        语句  
    }  
}
```

下标声明只能出现在类、结构体、枚举、扩展和协议的声明中。

参数列表指定一个或多个用于在相关类型的下标表达式中访问元素的索引（例如，表达式 `object[i]` 中的 `i`）。索引可以是任意类型，但是必须包含类型注解。返回类型指定了被访问的元素的类型。

和计算型属性一样，下标声明支持对元素的读写操作。`getter` 用于读取值，`setter` 用于写入值。`setter` 子句是可选的，当仅需要一个 `getter` 子句时，可以将二者都忽略，直接返回请求的值即可。但是，如果提供了 `setter` 子句，就必须提供 `getter` 子句。

圆括号以及其中的 `setter` 名称是可选的。如果提供了 `setter` 名称，它会作为 `setter` 的参数名称。如果不提供 `setter` 名称，那么 `setter` 的参数名称默认是 `value`。`setter` 的参数类型必须与返回类型相同。

可以重写下标，只要参数列表或返回类型不同即可。还可以重写继承自超类的下标，此时必须使用 `override` 声明修饰符声明被重写的下标。

下标参数遵循与函数参数相同的规则，但有两个例外。默认情况下，下标中使用的参数不需要指定标签，这与函数，方法和构造器不同。但是你也可以同它们一样，显式地提供参数标签。此外，下标不能有 `In-out` 参数。

同样可以在协议声明中声明下标，正如 [协议下标声明](#) 中所述。

更多关于下标的信息和例子，请参阅 [下标](#)。

## 类型下标声明

---

声明一个由类型而不是类型实例公开的下标，请使用 `static` 声明修饰符标记下标声明。类可以使用 `class` 声明修饰符标记类型计算属性，以允许子类重写父类的实现。在类声明中，`static` 关键字具有与用 `class` 和 `final` 声明修饰符标记声明相同的效果。

[grammar\\_of\\_a\\_subscript\\_declaration](#)

---

## 下标声明语法

### subscript-declaration

---

下标声明 → 下标头 下标结果 泛型 where 子句 可选 代码块

下标声明 → 下标头 下标结果 泛型 where 子句 可选 getter-setter 代码块

下标声明 → 下标头 下标结果 泛型 where 子句 可选 getter-setter 关键字 代码块

### subscript-head

---

下标头 → 特性列表 可选 声明修饰符列表 可选 subscript 泛型参数子句 可选 参数子句

### subscript-result

---

下标结果 → -> 特性列表 可选 类型

## 运算符声明

运算符声明 (*operator declaration*) 会向程序中引入中缀、前缀或后缀运算符，使用关键字 `operator` 来声明。

可以声明三种不同的缀性：中缀、前缀和后缀。运算符的缀性指定了运算符与其运算对象的相对位置。

运算符声明有三种基本形式，每种缀性各一种。运算符的缀性通过在 `operator` 关键字之前添加声明修饰符 `infix`，`prefix` 或 `postfix` 来指定。每种形式中，运算符的名字只能包含 运算符 中定义的运算符字符。

下面的形式声明了一个新的中缀运算符：

```
infix operator 运算符名称: 优先级组
```

中缀运算符是二元运算符，置于两个运算对象之间，例如加法运算符（`+`）位于表达式 `1 + 2` 的中间。

中缀运算符可以选择指定优先级组。如果没有为运算符设置优先级组，Swift 会设置默认优先级组 `DefaultPrecedence`，它的优先级比三目优先级 `TernaryPrecedence` 要高，更多内容参考优先级组声明

下面的形式声明了一个新的前缀运算符：

```
prefix operator 运算符名称 { }
```

出现在运算对象前边的前缀运算符是一元运算符，例如表达式 `!a` 中的前缀非运算符（`!`）。

前缀运算符的声明中不指定优先级，而且前缀运算符是非结合的。

下面的形式声明了一个新的后缀运算符：

postfix operator 运算符名称 {}

紧跟在运算对象后边的后缀运算符是一元运算符，例如表达式 `a!` 中的后缀强制解包运算符（`!`）。

和前缀运算符一样，后缀运算符的声明中不指定优先级，而且后缀运算符是非结合的。

声明了一个新的运算符以后，需要实现一个跟这个运算符同名的函数来实现这个运算符。如果是实现一个前缀或者后缀运算符，也必须使用相符的 `prefix` 或者 `postfix` 声明修饰符标记函数声明。如果是实现中缀运算符，则不需要使用 `infix` 声明修饰符标记函数声明。关于如何实现一个新的运算符的例子，请参阅 [自定义运算符](#)。

## grammar\_of\_an\_operator\_declaration

---

| 运算符声明语法

### operator-declaration

---

| 运算符声明 → 前缀运算符声明 | 后缀运算符声明 | 中缀运算符声明

### prefix-operator-declaration

---

前缀运算符声明 → prefix 运算符 运算符 {}

### postfix-operator-declaration

---

后缀运算符声明 → postfix 运算符 运算符 {}

### infix-operator-declaration

---

中缀运算符声明 → infix 运算符 运算符 {} { 中缀运算符属性 可选 }

### infix-operator-group

---

| 中缀运算符组 → 优先级组名称

## 优先级组声明

---

优先级组声明 (*A precedence group declaration*) 会向程序的中缀运算符引入一个全新的优先级组。当没有用圆括号分组时，运算符优先级反应了运算符与它的操作数的关系的紧密程度。优先级组的声明如下所示：

```
precedencegroup 优先级组名称 {  
    higherThan: 较低优先级组的名称  
    lowerThan: 较高优先级组的名称  
    associativity: 结合性  
    assignment: 赋值性  
}
```

较低优先级组和较高优先级组的名称说明了新建的优先级组是依赖于现存的优先级组的。`lowerThan` 优先级组的属性只可以引用当前模块外的优先级组。当两个运算符为同一个操作数竞争时，比如表达式 `2 + 3 * 5`，优先级更高的运算符将优先参与运算。

### 注意

使用较低和较高优先级组相互联系的优先级组必须保持单一层次关系，但它们不必是线性关系。这意味着优先级组也许会有未定义的相关优先级。这些优先级组的运算符在没有用圆括号分组的情况下是不能紧邻着使用的。

Swift 定义了大量的优先级组来与标准库的运算符配合使用，例如相加（`+`）和相减（`-`）属于 `AdditionPrecedence` 组，相乘（`*`）和相除（`/`）属于 `MultiplicationPrecedence` 组，详细关于 Swift 标准库中一系列运算符和优先级组内容，参阅 [Swift 标准库操作符参考](#)。

运算符的结合性表示在没有圆括号分组的情况下，同样优先级的一系列运算符是如何被分组的。你可以指定运算符的结合性通过上下文关键字 `left`、`right` 或者 `none`，如果没有指定结合性，默认是 `none` 关键字。左关联性的运算符是从左至右分组的，例如，相减操作符（`-`）是左关联性的，所以表达式 `4 - 5 - 6` 被分组为 `(4 - 5) - 6`，得出结果-7。右关联性的运算符是从右往左分组的，指定为 `none` 结合性的运算符就没有结合性。同样优先级没有结合性的运算符不能相邻出现，例如 `<` 运算符是 `none` 结合性，那表示 `1 < 2 < 3` 就不是一个有效表达式。

优先级组的赋值性表示在包含可选链操作时的运算符优先级。当设为 `true` 时，与优先级组对应的运算符在可选链操作中使用和标准库中赋值运算符同样的分组规则，当设为 `false` 或者不设置，该优先级组的运算符与不赋值的运算符遵循同样的可选链规则。

## grammar\_of\_a\_precedence\_group\_declaration

---

### 优先级组声明语法

#### precedence-group-declaration

---

优先级组声明 → `precedence` 优先级组名称 { 多优先级组属性 可选 }

#### precedence-group-attributes

---

优先级组属性 → 优先级组属性 多优先级组属性 可选 { }

#### precedence-group-attribute

---

优先级组属性 → 优先级组关系

优先级组属性 → 优先级组赋值性

优先级组属性 → 优先级组相关性

### **precedence-group-relation**

---

优先级组关系 → higherThan:多优先级组名称

优先级组关系 → lowerThan:多优先级组名称

### **precedence-group-assignment**

---

优先级组赋值 → assignment:布尔字面值

## **precedence-group-associativity**

---

优先级组结合性 → associativity:left

优先级组结合性 → associativity:right

优先级组结合性 → associativity:none

## **precedence-group-names**

---

| 多优先级组名称 → 优先级组名称 | 优先级组名称 | 优先级组名称

## **precedence-group-name**

---

| 优先级组名称 → 标识符

# **声明修饰符**

---

声明修饰符都是关键字或上下文相关的关键字，可以修改一个声明的行为或者含义。可以在声明的特性（如果存在）和引入该声明的关键字之间，利用声明修饰符的关键字或上下文相关的关键字指定一个声明修饰符。

## **class**

该修饰符用于修饰任何类成员，表明是类自身的成员，而不是类实例的成员。父类中使用该修饰符标记或者未被 `final` 修饰符标记的成员，都允许被子类重写。

## **dynamic**

该修饰符用于修饰任何兼容 Objective-C 的类的成员。访问被 `dynamic` 修饰符标记的类成员将总是由 Objective-C 运行时系统进行动态派发，而不会由编译器进行内联或消虚拟化。

因为被标记 `dynamic` 修饰符的类成员会由 Objective-C 运行时系统进行动态派发，所以它们会被隐式标记 `objc` 特性。

### `final`

该修饰符用于修饰类或类中的属性、方法以及下标。如果用它修饰一个类，那么这个类不能被继承。如果用它修饰类中的属性、方法或下标，那么它们不能在子类中被重写。

### `lazy`

该修饰符用于修饰类或结构体中的存储型变量属性，表示该属性的初始值最多只被计算和存储一次，且发生在它被第一次访问时。关于如何使用 `lazy` 修饰符的例子，请参阅 [惰性存储型属性](#)。

### `optional`

该修饰符用于修饰协议中的属性、方法以及下标成员，表示符合类型可以不实现这些成员要求。

只能将 `optional` 修饰符用于被 `objc` 特性标记的协议。这样一来，就只有类类型可以采纳并符合拥有可选成员要求的协议。关于如何使用 `optional` 修饰符，以及如何访问可选协议成员（比如，不确定符合类型是否已经实现了这些可选成员）的信息，请参阅 [可选协议要求](#)。

### `required`

该修饰符用于修饰类的指定构造器或便利构造器，表示该类所有的子类都必须实现该构造器。在子类实现该构造器时，必须同样使用 `required` 修饰符修饰该构造器。

### `static`

该修饰符用于修饰结构体、类、枚举或协议的成员，表明是类型成员，而不是类型实例的成员。在类声明的作用范围内，使用 `static` 修饰符标记成员声明语句，同 `class` 和 `final` 修饰符具有相同的效果。但是类的常量类型属性是一个例外：`static` 没有问题，但是你无法为常量声明使用 `class` 或 `final` 修饰符。

### `unowned`

该修饰符用于修饰存储型变量、常量或者存储型变量属性，表示该变量或属性持有其存储对象的无主引用。如果在此存储对象释放后尝试访问该对象，会引发运行时错误。如同弱引用一样，该引用类型的变量或属性必须是类类型。与弱引用不同的是，这种类型的变量或属性是非可选的。关于 `unowned` 更多的信息和例子，请参阅 [无主引用](#)

### `unowned(safe)`

### `unowned` 的显式写法

### `unowned(unsafe)`

该修饰符用于修饰存储型变量、常量或者存储型变量属性，表示该变量或属性持有其存储对象的无主引用。如果在此存储对象释放后尝试访问该对象，会直接访问该对象释放前存储的内存地址，因此这是非内存安全的操作。如同弱引用一样，该引用类型的变量或属性必须是类类型。与弱引用不同的是，这种类型的变量或属性是非可选的。关于 [unowned](#) 更多的信息和例子，请参阅 [无主引用](#)。

### `weak`

该修饰符用于修饰变量或存储型变量属性，表示该变量或属性持有其存储的对象的弱引用。这种变量或属性的类型必须是可选的类类型。使用 `weak` 修饰符可避免强引用循环。关于 `weak` 修饰符的更多信息和例子，请参阅 [弱引用](#)。

## 访问控制级别

Swift 提供了三个级别的访问控制：`public`、`internal` 和 `private`。可以使用以下任意一种访问级别修饰符来指定声明的访问级别。访问控制在 [访问控制](#) 中有详细讨论。

### `public`

该修饰符表示声明可被同模块的代码访问，只要其他模块导入了声明所在的模块，也可以进行访问。

### `internal`

该修饰符表示声明只能被同模块的代码访问。默认情况下，绝大多数声明会被隐式标记 `internal` 访问级别修饰符。

### `private`

该修饰符表示声明只能被所在源文件的代码访问。

以上访问级别修饰符都可以选择带上一个参数，该参数由一对圆括号和其中的 `set` 关键字组成（例如，`private(set)`）。使用这种形式的访问级别修饰符来限制某个属性或下标的 setter 的访问级别低于其本身的访问级别，正如 [Getter 和 Setter](#) 中所讨论的。

## `grammar_of_a_declaration_modifier`

| 声明修饰符的语法

## `declaration-modifier`

声明修饰符 → `class` | `convenience` | `dynamic` | `final` | `infix` | `lazy` | `mutating` |  
`nonmutating` | `optional` | `override` | `postfix` | `prefix` | `required` | `static` | `unowned` |  
`unowned ( safe )` | `unowned ( unsafe )` | `weak`

声明修饰符 → [访问级别修饰符](#)

## `declaration-modifiers`

声明修饰符列表 → [声明修饰符](#) [声明修饰符列表](#) 可选

## access-level-modifier

---

访问级别修饰符 → **internal | internal ( set )**

访问级别修饰符 → **private | private ( set )**

访问级别修饰符 → **public | public ( set )**

# 特性 · GitBook

---

 [runoob.com/manual/gitbook/swift5/source/\\_book/chapter3/07\\_Attributes.html](https://runoob.com/manual/gitbook/swift5/source/_book/chapter3/07_Attributes.html)

## 特性 (Attributes)

---

在 Swift 中有两种特性，分别用于修饰声明和类型。特性提供了有关声明和类型的更多信息。例如，使用 `discardableResult` 特性声明的函数，表明该函数虽然有返回值，但如果沒有使用该返回值，编译器不会产生警告。

您可以通过以下方式指定一个特性，通过符号 `@` 后跟特性的名称和特性接收的任何参数：

`@ 特性名`

`@ 特性名 ( 特性参数 )`

有些声明特性通过接收参数来指定特性的更多信息以及它是如何修饰某个特定的声明的。这些特性的参数写在圆括号内，它们的格式由它们所属的特性来定义。

## 声明特性

---

声明特性只能应用于声明。

### available

将 `available` 特性用于声明时，表示该声明的生命周期是相对于特定的平台和操作系统版本。

`available` 特性经常与参数列表一同出现，该参数列表至少有两个特性参数，参数之间由逗号分隔。这些参数由以下这些平台名字中的一个起头：

- `iOS`
- `iOSApplicationExtension`
- `macOS`
- `macOSApplicationExtension`
- `watchOS`
- `watchOSApplicationExtension`
- `tvOS`
- `tvOSApplicationExtension`
- `swift`

当然，你也可以用一个星号 (`*`) 来表示上面提到的所有平台。指定 Swift 版本的 `available` 特性参数，不能使用星号表示。

其余的参数，可以按照任何顺序出现，并且可以添加关于声明生命周期的附加信息，包括重要事件。

- **unavailable** 参数表示该声明在指定的平台上是无效的。当指定 Swift 版本可用性时不可使用该参数。
- **introduced** 参数表示指定平台从哪一版本开始引入该声明。格式如下：

**introduced** : 版本号

版本号由一至三个正整数构成，由句点分隔的。

**deprecated** 参数表示指定平台从哪一版本开始弃用该声明。格式如下：

**deprecated** : 版本号

可选的版本号由一个或多个正整数构成，由句点分隔的。省略版本号表示该声明目前已弃用，当弃用出现时没有给出任何有关信息。如果你省略了版本号，冒号（:）也可省略。

**obsoleted** 参数表示指定平台或语言从哪一版本开始废弃该声明。当一个声明被废弃后，它就从平台或语言中移除，不能再被使用。格式如下：

**obsoleted** : 版本号

版本号由一至三个正整数构成，由句点分隔的。

**message** 参数用来提供文本信息。当使用被弃用或者被废弃的声明时，编译器会抛出警告或错误信息。格式如下：

**message** : 信息内容

信息内容由一个字符串构成。

**renamed** 参数用来提供文本信息，用以表示被重命名的声明的新名字。当使用声明的旧名字时，编译器会报错提示新名字。格式如下：

**renamed** : 新名字

新名字由一个字符串构成。

你可以将 **renamed** 参数和 **unavailable** 参数用于 **available** 特性，来表示声明在不同平台和 Swift 版本上的可用性。如下所示，表示声明的名字在一个框架或者库的不同发布版本间发生了变化。以此组合表示该声明被重命名的编译错误。

```
// 首发版本
protocol MyProtocol {
    // 这里是协议定义
}
```

```
// 后续版本重命名了 MyProtocol
protocol MyRenamedProtocol {
    // 这里是协议定义
}
@available(*, unavailable, renamed:"MyRenamedProtocol")
typealias MyProtocol = MyRenamedProtocol
```

你可以在某个声明上使用多个 `available` 特性，以指定该声明在不同平台和 Swift 版本上的可用性。编译器只有在与 `available` 特性中指定的平台或语言版本匹配时，才会使用 `available` 特性。

如果 `available` 特性除了平台名称参数外，只指定了一个 `introduced` 参数，那么可以使用以下简写语法代替：

```
@available( 平台名称  版本号 , *)  
@available(swift 版本号 )
```

`available` 特性的简写语法可以简明地表达出声明在多个平台上的可用性。尽管这两种形式在功能上是相同的，但请尽可能地使用简写语法形式。

```
@available(iOS 10.0, macOS 10.12, *)
class MyClass {
    // 这里是类定义
}
```

当 `available` 特性需要同时指定 Swift 版本和平台可用性，需要使用单独的 `available` 特性来声明。

```
@available(swift 3.0.2)
@available(macOS 10.12, *)
struct MyStruct {
    // 这里是结构体定义
}
```

## discardableResult

该特性用于的函数或方法声明，以抑制编译器中函数或方法的返回值被调而没有使用其结果的警告。

## dynamicCallable

该特性用于类、结构体、枚举或协议，以将该类型的实例视为可调用的函数。该类型必须实现 `dynamicallyCall(withArguments:)`、`dynamicallyCall(withKeywordArguments:)` 方法之一，或两者同时实现。

你可以调用 `dynamicCallable` 特性的实例，就像是调用一个任意数量参数的函数。

```

@dynamicCallable
struct TelephoneExchange {
    func dynamicallyCall(withArguments phoneNumber: [Int]) {
        if phoneNumber == [4, 1, 1] {
            print("Get Swift help on forums.swift.org")
        } else {
            print("Unrecognized number")
        }
    }
}

let dial = TelephoneExchange()

// 使用动态方法调用
dial(4, 1, 1)
// 打印“Get Swift help on forums.swift.org”

dial(8, 6, 7, 5, 3, 0, 9)
// 打印“Unrecognized number”

// 直接调用底层方法
dial.dynamicallyCall(withArguments: [4, 1, 1])

```

**dynamicallyCall(withArguments:)** 方法的声明必须至少有一个参数遵循 [ExpressibleByArrayLiteral](#) 协议，如 `[Int]`，而返回值类型可以是任意类型。

```

@dynamicCallable
struct Repeater {
    func dynamicallyCall(withKeywordArguments pairs: KeyValuePairs<String, Int>) -> String {
        return pairs
            .map { label, count in
                repeatElement(label, count: count).joined(separator: " ")
            }
            .joined(separator: "\n")
    }
}

let repeatLabels = Repeater()
print(repeatLabels(a: 1, b: 2, c: 3, b: 2, a: 1))
// a
// b b
// c c c
// b b
// a

```

**dynamicallyCall(withKeywordArguments:)** 方法声明必须至少有一个参数遵循 [ExpressibleByDictionaryLiteral](#) 协议，返回值可以任意类型。参数的 [Key](#) 必须遵循 [ExpressibleByStringLiteral](#) 协议。上述的示例使用 [KeyValuePairs](#) 作为参数类型，以便调用者可以传入重复的参数标签，`a` 和 `b` 在调用 `repeat` 中多次使用。

如果你同时实现两种 `dynamicallyCall` 方法，则当在方法调用中包含关键字参数时，会调用 `dynamicallyCall(withKeywordArguments:)` 方法，否则调用 `dynamicallyCall(withArguments:)` 方法。

你只能调用参数和返回值与 `dynamicallyCall` 方法实现匹配的动态调用实例。在下面示例的调用无法编译，因为其 `dynamicallyCall(withArguments:)` 实现不接受 `KeyValuePairs<String, String>` 参数。

```
repeatLabels(a: "four") // Error
```

## dynamicMemberLookup

---

该特性用于类、结构体、枚举或协议，让其能在运行时查找成员。该类型必须实现 `subscript(dynamicMemberLookup:)` 下标。

在显式成员表达式中，如果没有成名指定成员，则该表达式被理解为对该类型的 `subscript(dynamicMemberLookup:)` 下标的调用，传递包含成员名称字符串的参数。下标接收参数既可以是键路径，也可以是成员名称字符串；如果你同时实现这两种方式的下标调用，那么以键路径参数方式为准。

`subscript(dynamicMemberLookup:)` 实现允许接收 `KeyPath`，`WritableKeyPath` 或 `ReferenceWritableKeyPath` 类型的键路径参数。而遵循 `ExpressibleByStringLiteral` 协议，下标调用接收参数为成员名称字符串——在大多数情况下，下标的参数是一个 `String` 值。下标返回值类型可以为任意类型。

根据成员名称来动态地查找成员，可以帮助我们创建一个包裹数据的包装类型，但该类型无法在编译时进行类型检查，例如其他语言的数据桥接到 Swift 语言时。例如：

```
@dynamicMemberLookup
struct DynamicStruct {
    let dictionary = ["someDynamicMember": 325,
                     "someOtherMember": 787]
    subscript(dynamicMember member: String) -> Int {
        return dictionary[member] ?? 1054
    }
}
let s = DynamicStruct()

// 使用动态成员查找
let dynamic = s.someDynamicMember
print(dynamic)
// 打印“325”

// 直接调用底层下标
let equivalent = s[dynamicMember: "someDynamicMember"]
print(dynamic == equivalent)
// 打印“true”
```

根据键路径来动态地查找成员，可用于创建一个包裹数据的包装类型，该类型在编译时期进行类型检查。例如：

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

@dynamicMemberLookup
struct PassthroughWrapper<Value> {
    var value: Value
    subscript<T>(dynamicMember member: KeyPath<Value, T>) -> T {
        get { return value[keyPath: member] }
    }
}

let point = Point(x: 381, y: 431)
let wrapper = PassthroughWrapper(value: point)
print(wrapper.x)
```

## GKInspectable

---

应用此属性，暴露一个自定义 GameplayKit 组件属性给 SpriteKit 编辑器 UI。

## inlinable

---

该特性用于函数、方法、计算属性、下标、便利构造器或析构器的声明，以将该声明的实现公开为模块公开接口的一部分。编译器允许在调用处把 `inlinable` 标记的符号替换为符号实现的副本。

内联代码可以与任意模块中 `public` 访问级别的符号进行交互，同时可以与在相同模块中标记 `usableFromInline` 特性的 `internal` 访问级别的符号进行交互。内联代码不能与 `private` 或 `fileprivate` 级别的符号进行交互。

该特性不能用于嵌套在函数内的声明，也不能用于 `fileprivate` 或 `private` 访问级别的声明。在内联函数定义的函数和闭包是隐式非内联的，即使他们不能标记该特性。

## nonobjc

---

该特性用于方法、属性、下标、或构造器的声明，这些声明本可以在 Objective-C 代码中使用，而使用 `nonobjc` 特性则告诉编译器这个声明不能在 Objective-C 代码中使用。

该特性用在扩展中，与在没有明确标记为 `objc` 特性的扩展中给每个成员添加该特性具有相同效果。

可以使用 `nonobjc` 特性解决标有 `objc` 的类中桥接方法的循环问题，该特性还允许对标有 `objc` 的类中的构造器和方法进行重载。

标有 `nonobjc` 特性的方法不能重写标有 `objc` 特性的方法。然而，标有 `objc` 特性的方法可以重写标有 `nonobjc` 特性的方法。同样，标有 `nonobjc` 特性的方法不能满足标有 `@objc` 特性的协议中的方法要求。

## NSApplicationMain

---

在类上使用该特性表示该类是应用程序委托类，使用该特性与调用 `NSApplicationMain(_:_:)` 函数并且把该类的名字作为委托类的名字传递给函数的效果相同。

如果你不想使用这个特性，可以提供一个 `main.swift` 文件，并在代码顶层调用 `NSApplicationMain(_:_:)` 函数，如下所示：

```
import AppKit  
NSApplicationMain(CommandLine argc, CommandLine.unsafeArgv)
```

## NSCopying

---

该特性用于修饰一个类的存储型变量属性。该特性将使属性的设值方法使用传入值的副本进行赋值，这个值由传入值的 `copyWithZone(_:)` 方法返回。该属性的类型必需符合 `NSCopying` 协议。

`NSCopying` 特性的行为与 Objective-C 中的 `copy` 特性相似。

## NSManaged

---

该特性用于修饰 `NSManagedObject` 子类中的实例方法或存储型变量属性，表明它们的实现由 `Core Data` 在运行时基于相关实体描述动态提供。对于标记了 `NSManaged` 特性的属性，`Core Data` 也会在运行时为其提供存储。应用这个特性也意味着 `objc` 特性。

## objc

---

该特性用于修饰任何可以在 Objective-C 中表示的声明。比如，非嵌套类、协议、非泛型枚举（仅限原始值为整型的枚举）、类和协议中的属性和方法（包括存取方法）、构造器、析构器以及下标运算符。`objc` 特性告诉编译器这个声明可以在 Objective-C 代码中使用。

该特性用在扩展中，与在没有明确标记为 `nonobjc` 特性的扩展中给每个成员添加该特性具有相同效果。

编译器隐式地将 `objc` 特性添加到 Objective-C 中定义的任何类的子类。但是，子类不能是泛型的，并且不能继承于任何泛型类。你可以将 `objc` 特性显式添加到满足这些条件的子类，来指定其 Objective-C 名称，如下所述。添加了 `objc` 的协议不能继承于没有此特性的协议。

在以下情况中隐式添加了 `objc` 特性。

- 父类有 `objc` 特性，且重写为子类的声明。
- 遵循带有 `objc` 特性协议的声明。
- 带有 `IBAction`、`IBSegueAction`、`IBOutlet`、`IBDesignable`、`IBInspectable`、`NSManaged` 或 `GKInspectable` 特性的声明。

如果你将 `objc` 特性应用于枚举，每一个枚举用例都会以枚举名称和用例名称组合的方式暴露在 Objective-C 代码中。例如，在 `Planet` 枚举中有一个名为 `Venus` 的用例，该用例暴露在 Objective-C 代码中时叫做 `PlanetVenus`。

`objc` 特性有一个可选的参数，由标识符构成。当你想把 `objc` 所修饰的实体以一个不同的名字暴露给 Objective-C 时，你就可以使用这个特性参数。你可以使用这个参数来命名类、枚举类型、枚举用例、协议、方法、存取方法以及构造器。如果你要指定类、协议或枚举在 Objective-C 下的名称，在名称上包含三个字母的前缀，如 Objective-C 编程中的约定。下面的例子把 `ExampleClass` 中的 `enabled` 属性的取值方法暴露给 Objective-C，名字是 `isEnabled`，而不是它原来的属性名。

```
class ExampleClass: NSObject {  
    @objc var enabled: Bool {  
        @objc(isEnabled) get {  
            // 返回适当的值  
        }  
    }  
}
```

## objcMembers

该特性用于类声明，以将 `objc` 特性应用于该类、扩展、子类以及子类的扩展的所有 Objective-C 兼容成员。

大多数代码应该使用 `objc` 特性，以暴露所需的声明。如果需要暴露多个声明，可以将其分组到添加 `objc` 特性的扩展中。`objcMembers` 特性为大量使用 Objective-C 运行时的内省工具的库提供了便利。添加不必要的 `objc` 特性会增加二进制体积并影响性能。

## requires\_stored\_property\_inits

该特性用于类声明，以要求类中所有存储属性提供默认值作为其定义的一部分。对于从中继承的任何类都推断出 `NSManagedObject` 特性。

## testable

在导入允许测试的编译模块时，该特性用于修饰 `import` 声明，这样就能访问被导入模块中的任何标有 `internal` 访问级别修饰符的实体，犹如它们被标记了 `public` 访问级别修饰符。测试也可以访问使用 `internal` 或者 `public` 访问级别修饰符标记的类和类成员，就像它们是 `open` 访问修饰符声明的。

## UIApplicationMain

在类上使用该特性表示该类是应用程序委托类，使用该特性与调用 `UIApplicationMain` 函数并且把该类的名字作为委托类的名字传递给函数的效果相同。

如果你不想使用这个特性，可以提供一个 `main.swift` 文件，并在代码顶层调用 `UIApplicationMain(_:_:_:_)` 函数。比如，如果你的应用程序使用一个继承于 `UIApplication` 的自定义子类作为主要类，你可以调用 `UIApplicationMain(_:_:_:_)` 函数而不是使用该特性。

## usableFromInline

---

该特性用于函数、方法、计算属性、下标、构造器或析构器的声明，以在同一模块中允许该符号用于内联代码的声明。声明必须具有 `internal` 访问级别修饰符。

与 `public` 访问修饰符相同的是，该特性将声明公开为模块公共接口的一部分。区别于 `public`，编译器不允许在模块外部的代码通过名称引用 `usableFromInline` 标记的声明，即使导出了声明符号也是无法引用。但是，模块外的代码仍然可以通过运行时交换声明符号。

标记为 `inlinable` 特性的声明，在内联代码中可以隐式使用。虽然 `inlinable` 或 `usableFromInline` 可以用于 `internal` 声明，但这两者不能同时使用。

## warn\_unqualified\_access

---

该特性应用于顶级函数、实例方法、类方法或静态方法，以在没有前置限定符（例如模块名称、类型名称、实例变量或常量）的情况下使用该函数或方法时触发警告。使用该特性可以帮助减少在同一作用于访问同名函数之间的歧义。

例如，Swift 标准库包含 `min(_:_:)` 顶级函数和用于序列比较元素的 `min()` 方法。序列方法声明使用了 `warn_unqualified_access`，以减少在 `Sequence` 扩展中使用它们的歧义。

## Interface Builder 使用的声明特性

---

Interface Builder 特性是 Interface Builder 用来与 Xcode 同步的声明特性。Swift 提供了以下的 Interface Builder 特性：`IBAction`，`IBSegueAction`，`IBOutlet`，`IBDesignable`，以及 `IBInspectable`。这些特性与 Objective-C 中对应的特性在概念上是相同的。

`IBOutlet` 和 `IBInspectable` 用于修饰一个类的属性声明，`IBAction` 特性用于修饰一个类的方法声明，`IBDesignable` 用于修饰类的声明。

应用 `IBAction`、`IBSegueAction`、`IBOutlet`、`IBDesignable` 或者 `IBInspectable` 特性都意味着同时应用 `objc` 特性。

## 类型特性

---

类型特性只能用于修饰类型。

## autoclosure

---

这个特性通过把表达式自动封装成无参数的闭包来延迟表达式的计算。它可以修饰类型为返回表达式结果类型的无参数函数类型的函数参数。关于如何使用 `autoclosure` 特性的例子，请参阅 [自动闭包](#) 和 [函数类型](#)。

## convention

---

该特性用于修饰函数类型，它指出了函数调用的约定。

convention 特性总是与下面的参数之一一起出现。

- **swift** 参数用于表示一个 Swift 函数引用。这是 Swift 中函数值的标准调用约定。
- **block** 参数用于表示一个 Objective-C 兼容的块引用。函数值会作为一个块对象的引用，块是一种 **id** 兼容的 Objective-C 对象，其中嵌入了调用函数。调用函数使用 C 的调用约定。
- **c** 参数用于表示一个 C 函数引用。函数值没有上下文，不具备捕获功能，同样使用 C 的调用约定。

使用 C 函数调用约定的函数也可用作使用 Objective-C 块调用约定的函数，同时使用 Objective-C 块调用约定的函数也可用作使用 Swift 函数调用约定的函数。然而，只有非泛型的全局函数、局部函数以及未捕获任何局部变量的闭包，才可以被用作使用 C 函数调用约定的函数。

## escaping

---

在函数或者方法声明上使用该特性，它表示参数将不会被存储以供延迟执行，这将确保参数不会超出函数调用的生命周期。在使用 **escaping** 声明特性的函数类型中访问属性和方法时不需要显式地使用 **self.**。关于如何使用 **escaping** 特性的例子，请参阅 [逃逸闭包](#)。

## Switch Case 特性

---

你只能在 switch cases 中使用 switch case 特性。

## unknown

---

次特性用于 switch case，表示在编译时该地方不会匹配枚举的任何情况。有关如何使用 **unknown** 特性的示例，可参阅 [对未来枚举的 case 进行 switch](#)。

## 特性语法

### **attribute**

---

特性 → 特性名 特性参数子句 可选

### **attribute\_name**

---

特性名 → 标识符

### **attribute\_argument\_clause**

---

特性参数子句 → ( 均衡令牌列表 可选 )

### **attributes**

---

特性列表 → 特性 特性列表 可选

### **balanced\_tokens**

---

均衡令牌列表 → 均衡令牌 均衡令牌列表 可选

### **balanced\_token**

---

均衡令牌 → ( 均衡令牌列表 可选 )

均衡令牌 → [ 均衡令牌列表 可选 ]

均衡令牌 → { 均衡令牌列表 可选 }

均衡令牌 → 任意标识符，关键字，字面量或运算符

均衡令牌 → 任意标点除了 (, ), [ , ], { , 或 }

# 模式 · GitBook

 [runoob.com/manual/gitbook/swift5/source/\\_book/chapter3/08\\_Patterns.html](https://runoob.com/manual/gitbook/swift5/source/_book/chapter3/08_Patterns.html)

## 模式 (Patterns)

模式代表单个值或者复合值的结构。例如，元组 `(1, 2)` 的结构是由逗号分隔的，包含两个元素的列表。因为模式代表一种值的结构，而不是特定的某个值，你可以利用模式来匹配各种各样的值。比如，`(x, y)` 可以匹配元组 `(1, 2)`，以及任何含两个元素的元组。除了利用模式匹配一个值以外，你可以从复合值中提取出部分或全部值，然后分别把各个部分的值和一个常量或变量绑定起来。

Swift 中的模式分为两类：一种能成功匹配任何类型的值，另一种在运行时匹配某个特定值时可能会失败。

第一类模式用于解构简单变量、常量和可选绑定中的值。此类模式包括通配符模式、标识符模式，以及包含前两种模式的值绑定模式和元组模式。你可以为这类模式指定一个类型注解，从而限制它们只能匹配某种特定类型的值。

第二类模式用于全模式匹配，这种情况下你试图匹配的值在运行时可能不存在。此类模式包括枚举用例模式、可选模式、表达式模式和类型转换模式。你在 `switch` 语句的 `case` 标签中，`do` 语句的 `catch` 子句中，或者在 `if`、`while`、`guard` 和 `for-in` 语句的 `case` 条件句中使用这类模式。

| 模式语法

### pattern

模式 → 通配符模式 类型注解 可选

模式 → 标识符模式 类型注解 可选

模式 → 值绑定模式

模式 → 元组模式 类型注解 可选

模式 → 枚举用例模式

模式 → 可选模式

模式 → 类型转换模式

模式 → 表达式模式

## 通配符模式 (Wildcard Pattern)

通配符模式由一个下划线（`_`）构成，用于匹配并忽略任何值。当你想忽略被匹配的值时可以使用该模式。例如，下面这段代码在闭区间 `1...3` 中迭代，每次迭代都忽略该区间的当前值：

```
for _ in 1...3 {  
    // ...  
}
```

| 通配符模式语法

## wildcard-pattern

---

| 通配符模式 → `_`

## 标识符模式 (Identifier Pattern)

---

标识符模式匹配任何值，并将匹配的值和一个变量或常量绑定起来。例如，在下面的常量声明中，`someValue` 是一个标识符模式，匹配了 `Int` 类型的 `42`：

```
let someValue = 42
```

当匹配成功时，`42` 被绑定（赋值）给常量 `someValue`。

如果一个变量或常量声明的左边是一个标识符模式，那么这个标识符模式是值绑定模式的子模式。

| 标识符模式语法

## identifier-pattern

---

| 标识符模式 → 标识符

## 值绑定模式 (Value-Binding Pattern)

---

值绑定模式把匹配到的值绑定给一个变量或常量。把匹配到的值绑定给常量时，用关键字 `let`，绑定给变量时，用关键字 `var`。

在值绑定模式中的标识符模式会把新命名的变量或常量与匹配到的值做绑定。例如，你可以拆开一个元组，然后把每个元素绑定到相应的标识符模式中。

```
let point = (3, 2)  
switch point {  
    // 将 point 中的元素绑定到 x 和 y  
    case let (x, y):  
        print("The point is at (\(x), \(y)).")  
    }  
    // 打印"The point is at (3, 2)."
```

在上面这个例子中，`let` 会分配到元组模式 `(x, y)` 中的各个标识符模式。因此，`switch` 语句中 `case let (x, y):` 和 `case (let x, let y):` 的匹配效果是一样的。

| 值绑定模式语法

## value-binding-pattern

---

| 值绑定模式 → var 模式 | let 模式

## 元组模式

---

元组模式是由逗号分隔的，具有零个或多个模式的列表，并由一对圆括号括起来。元组模式匹配相应元组类型的值。

你可以使用类型注解去限制一个元组模式能匹配哪种元组类型。例如，在常量声明 `let (x, y): (Int, Int) = (1, 2)` 中的元组模式 `(x, y): (Int, Int)` 只匹配两个元素都是 `Int` 类型的元组。

当元组模式被用于 `for-in` 语句或者变量和常量声明时，它仅可以包含通配符模式、标识符模式、可选模式或者其他包含这些模式的元组模式。比如下面这段代码就不正确，因为 `(x, 0)` 中的元素 `0` 是一个表达式模式：

```
let points = [(0, 0), (1, 0), (1, 1), (2, 0), (2, 1)]  
// 下面的代码是错误的  
for (x, 0) in points {  
    /* ... */  
}
```

只包含一个元素的元组模式的圆括号没有效果，模式只匹配这个单个元素的类型。举例来说，下面的语句是等效的：

```
let a = 2      // a: Int = 2  
let (a) = 2    // a: Int = 2  
let (a): Int = 2 // a: Int = 2
```

| 元组模式语法

## tuple-pattern

---

| 元组模式 → (元组模式元素列表 可选)

### tuple-pattern-element-list

---

| 元组模式元素列表 → 元组模式元素 | 元组模式元素, 元组模式元素列表

### tuple-pattern-element

---

| 元组模式元素 → 模式

## 枚举用例模式 (Enumeration Case Pattern)

---

枚举用例模式匹配现有的某个枚举类型的某个用例。枚举用例模式出现在 `switch` 语句中的 `case` 标签中，以及 `if`、`while`、`guard` 和 `for-in` 语句的 `case` 条件中。

如果你准备匹配的枚举用例有任何关联的值，则相应的枚举用例模式必须指定一个包含每个关联值元素的元组模式。关于使用 `switch` 语句来匹配包含关联值的枚举用例的例子，请参阅 [关联值](#)。

## 枚举用例模式语法

### enum-case-pattern

枚举用例模式 → 类型标识 可选 · 枚举用例名 元组模式 可选

## 可选模式 (Optional Pattern)

可选模式匹配包装在一个 `Optional(Wrapped)` 或者 `ExplicitlyUnwrappedOptional(Wrapped)` 枚举中的 `Some(Wrapped)` 用例中的值。可选模式由一个标识符模式和紧随其后的一个问号组成，可以像枚举用例模式一样使用。

由于可选模式是 `Optional` 和 `ImplicitlyUnwrappedOptional` 枚举用例模式的语法糖，下面两种写法是等效的：

```
let someOptional: Int? = 42
// 使用枚举用例模式匹配
if case .Some(let x) = someOptional {
    print(x)
}

// 使用可选模式匹配
if case let x? = someOptional {
    print(x)
}
```

可选模式为 `for-in` 语句提供了一种迭代数组的简便方式，只为数组中非 `nil` 的元素执行循环体。

```
let arrayOfOptionalInts: [Int?] = [nil, 2, 3, nil, 5]
// 只匹配非 nil 的元素
for case let number? in arrayOfOptionalInts {
    print("Found a \(number)")
}
// Found a 2
// Found a 3
// Found a 5
```

## 可选模式语法

### optional-pattern

可选模式 → 标识符模式 ?

## 类型转换模式 (Type-Casting Patterns)

---

有两种类型转换模式，`is` 模式和 `as` 模式。`is` 模式只出现在 `switch` 语句中的 `case` 标签中。`is` 模式和 `as` 模式形式如下：

| `is` 类型  
|    模式 `as` 类型

`is` 模式仅当一个值的类型在运行时和 `is` 模式右边的指定类型一致，或者是其子类的情况下，才会匹配这个值。`is` 模式和 `is` 运算符有相似表现，它们都进行类型转换，但是 `is` 模式没有返回类型。

`as` 模式仅当一个值的类型在运行时和 `as` 模式右边的指定类型一致，或者是其子类的情况下，才会匹配这个值。如果匹配成功，被匹配的值的类型被转换成 `as` 模式右边指定的类型。

关于使用 `switch` 语句配合 `is` 模式和 `as` 模式来匹配值的例子，请参阅 [Any](#) 和 [AnyObject](#) 的类型转换。

| 类型转换模式语法

### type-casting-pattern

---

| 类型转换模式 → is 模式 | as 模式

#### is-pattern

---

| is 模式 → is 类型

#### as-pattern

---

| as 模式 → 模式 as 类型

## 表达式模式 (Expression Pattern)

---

表达式模式代表表达式的值。表达式模式只出现在 `switch` 语句中的 `case` 标签中。

表达式模式代表的表达式会使用 Swift 标准库中的 `~=` 运算符与输入表达式的值进行比较。如果 `~=` 运算符返回 `true`，则匹配成功。默认情况下，`~=` 运算符使用 `==` 运算符来比较两个相同类型的值。它也可以将一个整型数值与一个 `Range` 实例中的一段整数区间做匹配，正如下面这个例子所示：

```
let point = (1, 2)
switch point {
case (0, 0):
    print("(0, 0) is at the origin.")
case (-2...2, -2...2):
    print("(\\(point.0), \\(point.1)) is near the origin.")
default:
    print("The point is at (\\(point.0), \\(point.1)).")
}
// 打印“(1, 2) is near the origin.”
```

你可以重载 `~=` 运算符来提供自定义的表达式匹配行为。比如你可以重写上面的例子，将 `point` 表达式与字符串形式表示的点进行比较。

```
// 重载 ~= 运算符对字符串和整数进行比较
func ~=(pattern: String, value: Int) -> Bool {
    return pattern == "\\(value)"
}

switch point {
case ("0", "0"):
    print("(0, 0) is at the origin.")
default:
    print("The point is at (\\(point.0), \\(point.1)).")
}
// 打印“The point is at (1, 2).”
```

| 表达式模式语法

## expression-pattern

---

| 表达式模式 → 表达式

## 泛型参数 (Generic Parameters and Arguments)

本节涉及泛型类型、泛型函数以及泛型构造器的参数，包括形参和实参。声明泛型类型、函数或构造器时，须指定相应的类型参数。类型参数相当于一个占位符，当实例化泛型类型、调用泛型函数或泛型构造器时，就用具体的类型实参替代之。

关于 Swift 语言的泛型概述，请参阅 [泛型](#)。

### 泛型形参子句

泛型形参子句指定泛型类型或函数的类型形参，以及这些参数相关的约束和要求。泛型形参子句用尖括号 (`<>`) 包住，形式如下：

| < 泛型形参列表 >

泛型形参列表中泛型形参用逗号分开，其中每一个采用以下形式：

| 类型形参 : 约束

泛型形参由两部分组成：类型形参及其后的可选约束。类型形参只是占位符类型（如 `T`，`U`，`V`，`Key`，`Value` 等）的名字而已。你可以在泛型类型、函数的其余部分或者构造器声明，包括函数或构造器的签名中使用它（以及它的关联类型）。

约束用于指明该类型形参继承自某个类或者符合某个协议或协议组合。例如，在下面的泛型函数中，泛型形参 `T: Comparable` 表示任何用于替代类型形参 `T` 的类型实参必须满足 `Comparable` 协议。

```
func simpleMax<T: Comparable>(_ x: T, _ y: T) -> T {  
    if x < y {  
        return y  
    }  
    return x  
}
```

例如，因为 `Int` 和 `Double` 均满足 `Comparable` 协议，所以该函数可以接受这两种类型。与泛型类型相反，调用泛型函数或构造器时不需要指定泛型实参子句。类型实参由传递给函数或构造器的实参推断而出。

```
simpleMax(17, 42) // T 被推断为 Int 类型  
simpleMax(3.14159, 2.71828) // T 被推断为 Double 类型
```

### Where 子句

要想对类型形参及其关联类型指定额外要求，可以在函数体或者类型的大括号之前添加 `where` 子句。`where` 子句由关键字 `where` 及其后的用逗号分隔的一个或多个要求组成。

### where : 类型要求

`where` 子句中的要求用于指明该类型形参继承自某个类或符合某个协议或协议组合。尽管 `where` 子句提供了语法糖使其有助于表达类型形参上的简单约束（如 `<T: Comparable>` 等同于 `<T> where T: Comparable`，等等），但是依然可以用来对类型形参及其关联类型提供更复杂的约束，例如你可以强制形参的关联类型遵守协议，如，`<S: Sequence> where S.Iterator.Element: Equatable` 表示泛型类型 `S` 遵守 `Sequence` 协议并且关联类型 `S.Iterator.Element` 遵守 `Equatable` 协议，这个约束确保队列的每一个元素都是符合 `Equatable` 协议的。

也可以用操作符 `==` 来指定两个类型必须相同。例如，泛型形参子句 `<S1: Sequence, S2: Sequence> where S1.Iterator.Element == S2.Iterator.Element` 表示 `S1` 和 `S2` 必须都符合 `SequenceType` 协议，而且两个序列中的元素类型必须相同。

当然，替代类型形参的类型实参必须满足所有的约束和要求。

泛型函数或构造器可以重载，但在泛型形参子句中的类型形参必须有不同的约束或要求，抑或二者皆不同。当调用重载的泛型函数或构造器时，编译器会根据这些约束来决定调用哪个重载函数或构造器。

更多关于泛型 `where` 从句的信息和关于泛型函数声明的例子，可以看一看 [泛型 where 子句](#)。

### 泛型形参子句语法

## generic-parameter-clause

泛型形参子句 → <泛型形参列表 约束子句<sub>可选</sub>>

## generic-parameter-list

泛型形参列表 → 泛形形参 | 泛形形参, 泛型形参列表

## generic-parameter

泛形形参 → 类型名称

泛形形参 → 类型名称: 类型标识符

泛形形参 → 类型名称: 协议合成类型

## requirement-clause

约束子句 → where 约束列表

## requirement-list

---

| 约束列表 → 约束 | 约束, 约束列表

## requirement

---

约束 → 一致性约束 | 同类型约束

conformance-requirement

---

一致性约束 → 类型标识符 : 类型标识符

一致性约束 → 类型标识符 : 协议合成类型

## same-type-requirement

---

| 同类型约束 → 类型标识符 == 类型

## 泛型实参子句

---

泛型实参子句指定泛型类型的类型实参。泛型实参子句用尖括号 (`<>`) 包住，形式如下：

| < 泛型实参列表 >

泛型实参列表中类型实参用逗号分开。类型实参是实际具体类型的名字，用来替代泛型类型的泛型形参子句中的相应的类型形参。从而得到泛型类型的一个特化版本。例如，Swift 标准库中的泛型字典类型的简化定义如下：

```
struct Dictionary<Key: Hashable, Value>: CollectionType, DictionaryLiteralConvertible {  
    /* ... */  
}
```

泛型 `Dictionary` 类型的特化版本，`Dictionary<String, Int>` 就是用具体的 `String` 和 `Int` 类型替代泛型类型 `Key: Hashable` 和 `Value` 产生的。每一个类型实参必须满足它所替代的泛型形参的所有约束，包括任何 `where` 子句所指定的额外的关联类型要求。上面的例子中，类型形参 `Key` 的类型必须符合 `Hashable` 协议，因此 `String` 也必须满足 `Hashable` 协议。

可以用本身就是泛型类型的特化版本的类型实参替代类型形参（假设已满足合适的约束和关联类型要求）。例如，为了生成一个元素类型是整型数组的数组，可以用数组的特化版本 `Array<Int>` 替代泛型类型 `Array<T>` 的类型形参 `T` 来实现。

```
let arrayOfArrays: Array<Array<Int>> = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

如 泛型形参子句 所述，不能用泛型实参子句来指定泛型函数或构造器的类型实参。

| 泛型实参子句语法

## generic-argument-clause

---

| 泛型实参子句 → < 泛型实参列表 >

## generic-argument-list

---

| 泛型实参列表 → 泛型实参 | 泛型实参, 泛型实参列表

## generic-argument

---

| 泛型实参 → 类型

# 语法总结 · GitBook



[runoob.com/manual/gitbook/swift5/source/\\_book/chapter3/10\\_Summary\\_of\\_the\\_Grammar.html](https://runoob.com/manual/gitbook/swift5/source/_book/chapter3/10_Summary_of_the_Grammar.html)

## 语法总结 (Summary of the Grammar)

### 词法结构

空白字符语法

空白字符 → 空白字符项 空白字符 可选

空白字符项 → 换行符

空白字符项 → 注释

空白字符项 → 多行注释

空白字符项 → U+0000 , U+0009 , U+000B , U+000C 或 U+0020

换行符\* → U+000A

换行符 → U+000D

换行符 → U+000D 后面是 U+000A

注释 → // 单行内容注释 换行符

注释 → /\* 多行内容注释 \*/

注释内容 → 注释内容项 注释内容 可选

注释内容项 → 除 U+000A 或 U+000D 外的任何 Unicode 标量值

多行注释内容 → 多行注释内容项 多行注释内容 可选

多行注释内容项 → 多行内容

多行注释内容项 → 注释内容项

多行注释内容项 → 除 /\* 或 \*/ 外的任何 Unicode 标量值

标识符语法

标识符 → 标识符头 (Head) 标识符字符集 可选

标识符 → 标识符头 (Head) 标识符字符集 可选

标识符 → 隐式参数名

标识符集 → 标识符 | 标识符 , 标识符集

标识符头 (Head) → 大写或者小写字母 A 到 Z

标识符头 (Head) → \_

标识符头 (Head) → U+00A8, U+00AA, U+00AD, U+00AF, U+00B2–U+00B5, or  
U+00B7–U+00BA

标识符头 (Head) → U+00BC–U+00BE, U+00C0–U+00D6, U+00D8–U+00F6, or  
U+00F8–U+00FF

标识符头 (Head) → U+0100–U+02FF, U+0370–U+167F, U+1681–U+180D, or  
U+180F–U+1DBF

标识符头 (Head) → U+1E00–U+1FFF

标识符头 (Head) → U+200B–U+200D, U+202A–U+202E, U+203F–U+2040,  
U+2054, or U+2060–U+206F

标识符头 (Head) → U+2070–U+20CF, U+2100–U+218F, U+2460–U+24FF, or  
U+2776–U+2793

标识符头 (Head) → U+2C00–U+2DFF or U+2E80–U+2FFF

标识符头 (Head) → U+3004–U+3007, U+3021–U+302F, U+3031–U+303F, or  
U+3040–U+D7FF

标识符头 (Head) → U+F900–U+FD3D, U+FD40–U+FDCF, U+FDF0–U+FE1F, or  
U+FE30–U+FE44

标识符头 (Head) → U+FE47–U+FFFD

标识符头 (Head) → U+10000–U+1FFFD, U+20000–U+2FFFD, U+30000–  
U+3FFFD, or U+40000–U+4FFFD

标识符头 (Head) → U+50000–U+5FFFD, U+60000–U+6FFFD, U+70000–  
U+7FFFD, or U+80000–U+8FFFD

标识符头 (Head) → U+90000–U+9FFFD, U+A0000–U+AFFFD, U+B0000–  
U+BFFFD, or U+C0000–U+CFFFD

标识符头 (Head) → U+D0000–U+DFFFD or U+E0000–U+EFFFD

标识符字符 → 数值 0 到 9

标识符字符 → U+0300–U+036F, U+1DC0–U+1DFF, U+20D0–U+20FF, or U+FE20–  
U+FE2F

标识符字符 → 标识符头 (Head)

标识符字符集 → 标识符字符 标识符字符集<sub>可选</sub>

隐式参数名 → \$ 十进制数字集

字面量语法

字面量 → 数值型字面量 | 字符串字面量 | 布尔字面量 | 空字面量

数值型字面量 → -<sub>可选</sub> 整形字面量 | -<sub>可选</sub> 浮点型字面量

布尔字面量 → true | false

空字面量 → nil

## 整型字面量语法

整型字面量 → 二进制字面量

整型字面量 → 八进制字面量

整型字面量 → 十进制字面量

整型字面量 → 十六进制字面量

二进制字面量 → 0b 二进制数字 二进制字面量字符集<sub>可选</sub>

二进制数字 → 数值 0 到 1

二进制字面量字符 → 二进制数字 | \_

二进制字面量字符集 → 二进制字面量字符 二进制字面量字符集<sub>可选</sub>

八进制字面量 → 0o 八进制数字 八进制字符集<sub>可选</sub>

八进制数字 → 数值 0 到 7

八进制字符 → 八进制数字 | \_

八进制字符集 → 八进制字符 八进制字符集<sub>可选</sub>

十进制字面量 → 十进制数字 十进制字符集<sub>可选</sub>

十进制数字 → 数值 0 到 9

十进制数字集 → 十进制数字 十进制数字集<sub>可选</sub>

十进制字面量字符 → 十进制数字 | \_

十进制字面量字符集 → 十进制字面量字符 十进制字面量字符集<sub>可选</sub>

十六进制字面量 → 0x 十六进制数字 十六进制字面量字符集<sub>可选</sub>

十六进制数字 → 数值 0 到 9 , a 到 f , 或者 A 到 F

十六进制字符 → 十六进制数字 | \_

十六进制字面量字符集 → 十六进制字符 十六进制字面量字符集<sub>可选</sub>

## 浮点型字面量语法

浮点数字面量 → 十进制字面量 十进制分数<sub>可选</sub> 十进制指数<sub>可选</sub>

浮点数字面量 → 十六进制字面量 十六进制分数<sub>可选</sub> 十六进制指数

十进制分数 → .十进制字面量

十进制指数 → 浮点数 e 正负号<sub>可选</sub> 十进制字面量

十六进制分数 → .十六进制数

十六进制指数 → 浮点数 p 正负号<sub>可选</sub> 十六进制字面量

浮点数 e → e | E

浮点数 p → p | P

正负号 → + | -

## 字符串型字面量语法

字符串字面量 → 静态字符串字面量 | 插值字符串字面量

字符串开分隔定界符 → 字符串扩展分隔符 " "

字符串闭分隔定界符 → " 字符串扩展分隔符<sub>可选</sub>

静态字符串字面量 → 字符串开分隔定界符 引用文本<sub>可选</sub> 字符串闭分隔定界符

静态字符串字面量 → 多行字符串开分隔定界符 多行引用文本<sub>可选</sub> 多行字符串闭分隔定界符

多行字符串开分隔定界符 → 字符串扩展分隔符 " " " "

多行字符串闭分隔定界符 → " " " 字符串扩展分隔符

字符串扩展分隔符 → # 字符串扩展分隔符<sub>可选</sub>

引用文本 → 引用文本项 引用文本<sub>可选</sub>

引用文本项 → 转义字符

引用文本项 → 除了 "、\\"、U+000A、U+000D 以外的所有 Unicode 字符

多行引用文本 → 多行引用文本项 多行引用文本<sub>可选</sub>

多行引用文本项 → 转义字符<sub>可选</sub>

多行引用文本 → 除了 \ 以外的任何 Unicode 标量值

多行引用文本 → 转义换行

插值字符串字面量 → 字符串开分隔定界符 插值文本<sub>可选</sub> 字符串闭分隔定界符

插值字符串字面量 → 多行字符串开分隔定界符 插值文本<sub>可选</sub> 多行字符串闭分隔定界符

插值文本 → 插值文本项 插值文本<sub>可选</sub>

插值文本项 → \(表达式) | 引用文本项

多行插值文本 → 多行插值文本项 多行插值文本<sub>可选</sub>

多行插值文本项 → \(表达式) | 多行引用文本项

转义序列 → \字符串扩展分隔符

转义字符 → 转义序列 0 | 转义序列 \ | 转义序列 t | 转义序列 n | 转义序列 r | 转义序列 \" | 转义序列 '

转义字符 → 转义序列 u { unicode 标量数字 }

unicode 标量数字 → 一到八位的十六进制数字

转义换行符 → 转义序列 空白<sub>可选</sub> 换行符

运算符语法语法

运算符 → 运算符头 运算符字符集<sub>可选</sub>

运算符 → 点运算符头 点运算符字符集<sub>可选</sub>

运算符字符 → / | = | - | + | ! | \* | % | < | > | & | || | ^ | ~ | ?

运算符头 → U+00A1–U+00A7

运算符头 → U+00A9 or U+00AB

运算符头 → U+00AC or U+00AE

运算符头 → U+00B0–U+00B1, U+00B6, U+00BB, U+00BF, U+00D7, or U+00F7

运算符头 → U+2016–U+2017 or U+2020–U+2027

运算符头 → U+2030–U+203E

运算符头 → U+2041–U+2053

运算符头 → U+2055–U+205E

运算符头 → U+2190–U+23FF

运算符头 → U+2500–U+2775

运算符头 → U+2794–U+2BFF

运算符头 → U+2E00–U+2E7F

运算符头 → U+3001–U+3003

运算符头 → U+3008–U+3030

运算符字符 → 运算符头

运算符字符 → U+0300–U+036F

运算符字符 → U+1DC0–U+1DFF

运算符字符 → U+20D0–U+20FF

运算符字符 → U+FE00–U+FE0F

运算符字符 → U+FE20–U+FE2F

运算符字符 → U+E0100–U+E01EF

运算符字符集 → 运算符字符 运算符字符集<sub>可选</sub>

点运算符头 → ..

点运算符字符 → . | 运算符字符

点运算符字符集 → 点运算符字符 点运算符字符集<sub>可选</sub>

二元运算符 → 运算符

前置运算符 → 运算符

后置运算符 → 运算符

## 类型

---

类型语法

类型 → 数组类型

类型 → 字典类型

类型 → 函数类型

类型 → 类型标识符

类型 → 元组类型

类型 → 可选类型

类型 → 隐式解析可选类型

类型 → 协议合成类型

类型 → Any

类型 → Self

类型 → ( type )

类型注解语法

类型注解 → : 属性 (Attributes) 集 可选类型

类型标识语法

类型标识 → 类型名称 泛型参数从句 可选 | 类型名称 泛型参数从句 可选 · 类型标识符

类型名 → 标识符

元组类型语法

元组类型 → () | ( 元组类型元素 , 元组类型元素列表 )

元组类型元素列表 → 元组类型元素 | 元组类型元素 , 元组类型元素列表

元组类型元素 → 元素名 类型注解 | 类型

元素名 → 标识符

## 函数类型语法

函数类型 → 类型 throws 可选 -> 类型

函数类型 → 类型 rethrows -> 类型

函数类型子句 → ()

函数类型子句 → ( 函数类型参数列表 ... 可选 )

函数类型参数列表 → 函数类型参数 | 函数类型参数 , 函数类型参数列表

函数类型参数 → 特性列表 可选 输入输出参数 可选 类型 | 参数标签 类型注解

参数标签 → 标识符

## 数组类型语法

数组类型 → [ 类型 ]

## 字典类型语法

字典类型 → [ 类型 : 类型 ]

## 可选类型语法

可选类型 → 类型 ?

隐式解析可选类型 (Implicitly Unwrapped Optional Type) 语法

隐式解析可选类型 → 类型 !

## 协议合成类型语法

协议合成类型 → 类型标识符 | 协议合成延续

协议持续延续 → 类型标识符 | 协议合成类型

## 元 (Metatype) 类型语法

元类型 → 类型 . Type | 类型 . Protocol

## 类型继承从句语法

类型继承从句 → : 类型继承集

类型继承集 → 类型标识符 | 类型标识符 , 类型继承集

类条件 → class

# 表达式

## 表达式语法

表达式 → try 运算符 可选 前缀表达式 二元表达式列表

表达式列表 → 表达式 | 表达式, 表达式列表

## 前缀表达式语法

前缀表达式 → 前缀操作符 可选 前缀表达式

前缀表达式 → 输入输出表达式

输入输出表达式 → & 标识符

## try 表达式语法

try 操作符 → try | try ? | try !

## 二元表达式语法

二元表达式 → 二元运算符 前缀表达式

二元表达式 → 赋值操作符 try 运算符 可选 前缀表达式

二元表达式 → 条件运算符 try 运算符 可选 前缀表达式

二元表达式 → 类型转换运算符

二元表达式 → 二元表达式 二元表达式列表 可选

## 赋值操作符语法

赋值运算符 → [=]

## 条件运算符

条件运算符 → [?] 表达式:

## 类型转换运算符语法

类型转换运算符 → [is] 类型

类型转换运算符 → [as] 类型

类型转换运算符 → [as ?] 类型

类型转换运算符 → [as !] 类型

## 基础表达式语法

基础表达式 → 标识符 泛型实参子句 可选

基础表达式 → 字面量表达式

基础表达式 → self 表达式

基础级表达式 → 父类表达式

基础表达式 → 闭包表达式

基础表达式 → 圆括号表达式

基础表达式 → 元组表达式

基础表达式 → 隐式成员表达式

基础表达式 → 通配符表达式

基础表达式 → key-path 表达式

基础表达式 → 选择器表达式

基础表达式 → key-path 字符串表达式

## 字面量表达式语法

字面量表达式 → 字面量

字面量表达式 → 数组字面量 | 字典字面量 | 练习场字面量

字面量表达式 → #file | #line | #column | #function | dsohandle

数组字面量 → [ 数组字面量项列表 可选 ] 数组字面量项列表 → 数组字面量项 可选 | 数组字面量项, 数组字面量项列表 数组字面量项 → 表达式

字典字面量 → [ 字典字面量项列表 ] | [ : ]

字典字面量项列表 → 字典字面量项 , \*\* 可选 | 字典字面量项 , 字典字面量项列表

字典字面量项 → 表达式 : 表达式

palyground 字面量 → #colorLiteral ( red : 表达式 , green : 表达式 , blue : 表达式 , alpha : 表达式 )

playground 字面量 → #fileLiteral ( resourceName : 表达式 )

playground 字面量 → \*\*#imageLiteral ( resourceName : 表达式 )

## self 表达式语法

self 表达式 → self | self 方法表达式 | self 下标表达式 | self 构造器表达式

self 方法表达式 → self . 标识符

self 下标表达式 → self [ 函数调用参数表 ]

self 构造器表达式 → self . init

## 父类表达式语法

父类表达式 → 父类方法表达式 | 父类下标表达式 | 父类构造器表达式

父类方法表达式 → super . 标识符

父类下标表达式 → super [ 函数调用参数表 ]

父类构造器表达式 → super . init

## 闭包表达式语法

闭包表达式 → { 闭包签名 可选 语句 }

闭包签名 → 参数子句 函数结果 可选 \*in

闭包签名 → 标识符列表 函数结果 可选 in

闭包参数子句 () | ( 闭包参数列表 | 标识符列表 )

闭包参数列表 闭包参数 | 闭包参数, 闭包参数列表

闭包参数 闭包参数名 类型声明 可选

闭包参数 闭包参数名 类型声明 ...

闭包参数名 标识符

捕获列表 → 捕获列表 [ 捕获列表项列表 ]

捕获列表项列表 → 捕获列表项 | 捕获列表项 , 捕获列表项列表

捕获列表项 → 捕获说明符 可选 表达式

捕获说明符 → weak | unowned | unowned(safe) | unowned(unsafe)

## 隐式成员表达式语法

隐式成员表达式 → . 标识符

## 圆括号表达式语法

圆括号表达式 → ( 表达式 )

## 元组表达式语法

元组表达式 → () | ( 元组元素 , 元组元素列表 )

元组元素列表 → 元组元素 | 元组元素 , 元组元素列表 元组元素 → 表达式 | 标识符 : 表达式

## 通配符表达式语法

通配符表达式 → \_

## key-path 表达式语法

key-path 表达式 → \ 类型 可选 . [ 多个 key-path 组件 ] 多个 key-path 组件 → key-path 组件 | key-path 组件 . 多个 key-path 组件 key-path 组件 → 标识符 多个 key-path 后缀 可选 | 多个 key-path 后缀 多个 key-path 后缀 → key-path 后缀 多个 key-path 后缀 可选 key-path-postfixes ./04\_Expressions.md#key-path-postfixes }

key-path 后缀 → ? | ! | self | [ 函数调用参数表 ]

## 选择器表达式语法

选择器表达式 → #selector (表达式)

选择器表达式 → #selector (getter: 表达式)

选择器表达式 → #selector (setter: 表达式)

key-path 字符串表达式语法 key-path 字符串表达式 → #keyPath (表达式)

## 后缀表达式表达式语法

后缀表达式 → 基本表达式

后缀表达式 → 后缀表达式 后缀运算符

后缀表达式 → 函数调用表达式

后缀表达式 → 构造器表达式

后缀表达式 → 显式成员表达式

后缀表达式 → 后缀 self 表达式

后缀表达式 → 下标表达式

后缀表达式 → 强制取值表达式

后缀表达式 → 可选链表达式

## 函数调用表达式语法

函数调用表达式 → 后缀表达式 函数调用参数子句

函数调用表达式 → 后缀表达式 函数调用参数子句 可选 尾随闭包

函数调用参数子句 → () | (函数调用参数表)

函数调用参数表 → 函数调用参数 | 函数调用参数, 函数调用参数表

函数调用参数 → 表达式 | 标识符 : 表达式

函数调用参数 → 运算符 | 标识符 : 运算符

尾随闭包 → 闭包表达式

## 初始化表达式语法

构造器表达式 → 后缀表达式 . init

构造器表达式 → 后缀表达式 . init ( 参数名称 )

## 显式成员表达式语法

显式成员表达式 → 后缀表达式 . [十进制数字] (02\_Lexical\_Structure.md#decimal-digit)

显式成员表达式 → 后缀表达式 . 标识符 泛型实参子句 可选

显式成员表达式 → 后缀表达式 . [标识符] (02\_Lexical\_Structure.md#identifier) ( 参数名称 )

参数名称 → 参数名 参数名称 可选

参数名 → 标识符 :

## 后缀 self 表达式语法

后缀 self 表达式 → 后缀表达式 . self

## 下标表达式语法

下标表达式 → 后缀表达式 [ 表达式列表 ]

强制取值表达式语法 强制取值表达式 → 后缀表达式 !

可选链式表达式语法 可选链表达式 → 后缀表达式 ?

# 语句

---

## 语句语法

语句 → 表达式 ; 可选

语句 → 声明 ; 可选

语句 → 循环语句 ; 可选

语句 → 分支语句 ; 可选

语句 → 标签语句 ; 可选

语句 → 控制转移语句 ; 可选

语句 → 延迟语句 ; 可选

语句 → 执行语句 ; 可选

语句 → 编译控制语句

语句集 → 语句 语句集 可选

## 循环语句语法

循环语句 → for-in 语句

循环语句 → while 语句

循环语句 → repeat-while 语句

### For-In 循环语法

*for-in 语句* → **for** case 可选 模式 in 表达式 where 子句 可选 代码块

### While 循环语法

*while 语句* → **while** 条件集 代码块

条件集 → 条件 | 条件 , 条件集 条件 → 表达式 | 可用性条件 | case 条件 | 可选绑定条件

*case 条件* → **case** 模式 构造器

可选绑定条件 → **let** 模式 构造器 | **var** 模式 构造器

### Repeat-While 语句语法

*repeat-while-statement* → **repeat** 代码块 while 表达式

## 分支语句语法

分支语句 → if 语句

分支语句 → guard 语句

分支语句 → switch 语句

### If 语句语法

*if 语句* → **if** 条件集 代码块 else 子句 可选

*else 子句* → **else** 代码块 | else if 语句

### Guard 语句语法

*guard 语句* → **guard** 条件集 else 代码块

## Switch 语句语法

*switch 语句* → **switch 表达式 { switch-case 集 可选 }**

*switch-case 集* → **switch-case switch-case 集 可选**

*switch-case* → **case 标签 语句 集**

*switch-case* → **default 标签 语句 集**

*switch-case* → **条件 switch-case**

*case 标签* → **特性 可选 case case 项集 :**

*case 项集* → **模式 where 子句 可选 | 模式 where 子句 可选 , case 项集**

*default 标签* → **特性 可选 default :**

*where 子句* → **where where 表达式**

*where 表达式* → **表达式**

*条件 switch-case* → **switch if 指令子句 switch elseif 指令子句 集 可选 switch else 指令子句 可选 endif 指令**

*switch if 指令子句* → **if 指令 编译条件 switch-case 集 可选**

*switch elseif 指令子句 集* → **elseif 指令子句 switch elseif 指令子句 集 可选**

*switch elseif 指令子句* → **elseif 指令 编译条件 switch-case 集 可选**

*switch else 指令子句* → **else 指令 switch-case 集 可选**

## 标签语句语法

*标签语句* → **语句 标签 循环语句**

*标签语句* → **语句 标签 if 语句**

*标签语句* → **语句 标签 switch 语句**

*标签语句* → **语句 标签 do 语句**

*语句 标签* → **标签名称 :**

*标签名称* → **标识符**

## 控制转移语句语法

控制转移语句 → break 语句

控制转移语句 → continue 语句

控制转移语句 → fallthrough 语句

控制转移语句 → return 语句

控制转移语句 → throw 语句

## Break 语句语法

*break* 语句 → **break** 标签名称 可选

## Continue 语句语法

*continue* 语句 → **continue** 标签名称 可选

## Fallthrough 语句语法

*fallthrough* 语句 → **fallthrough**

## Return 语句语法

*return* 语句 → **return** 表达式 可选

## Throw 语句语法

*throw* 语句 → **throw** 表达式

## Defer 语句语法

*defer* 语句 → **defer** 代码块

## Do 语句语法

*do* 语句 → **do** 代码块 catch 子句集 可选

*catch 子句集* → catch 子句 catch 子句集 可选

*catch 子句* → **catch 模式** 可选 where 子句 可选 代码块 可选

## 编译控制语句

编译控制语句 → 条件编译块

编译控制语句 → 行控制语句

编译控制语句 → 诊断语句

## 条件编译块语法

条件编译块 → if 指令子句 elseif 指令子句集 可选 else 指令子句 可选 endif 指令

if 指令子句 → if 指令 编译条件 语句集 可选

elseif 指令子句集 → elseif 指令子句 elseif 指令子句集 可选

elseif 指令子句 → elseif 指令 编译条件 语句集 可选

else 指令子句 → else 指令 语句集 可选

if 指令 → `#if`

elseif 指令 → `#elseif`

else 指令 → `#else`

endif 指令 → `#endif`

编译条件 → 平台条件

编译条件 → 标识符

编译条件 → 布尔字面量

编译条件 → (编译条件)

编译条件 → !编译条件

编译条件 → 编译条件 && 编译条件

编译条件 → 编译条件 || 编译条件

平台条件 → os (操作系统)

平台条件 → arch (架构)

平台条件 → swift (  $\geq$  swift 版本 ) | swift (  $<$  swift 版本 )

平台条件 → compiler (  $\geq$  swift 版本 ) | compiler (  $<$  swift 版本 )

平台条件 → canImport ( 模块名 )

平台条件 → targetEnvironment ( 环境 )

操作系统 → macOS | iOS | watchOS | tvOS

架构 → i386 | x86\_64 | arm | arm64

swift 版本 → 十进制数字集 swift 版本后缀 可选

`swift` 版本后缀 → .十进制数字集 `swift` 版本集可选

模块名 → 标识符

环境 → **simulator**

行控制语句语法

行控制语句 → `#sourceLocation ( file: 文件名 , line: 行号 )`

行控制语句 → `#sourceLocation ()`

行号 → 一个大于 0 的十进制数字

文件名 → 静态字符串字面量

编译期诊断语句语法

诊断语句 → `#error ( 诊断信息 )`

诊断语句 → `#warning ( 诊断信息 )`

诊断信息 → 静态字符串字面量

可用性条件语法

可用性条件 → `#available ( 可用性参数集 )`

可用性参数集 → 可用性参数 | 可用性参数 , 可用性参数集

可用性参数 → 平台名 平台版本

可用性参数 → \*

平台名 → **iOS | iOSApplicationExtension**

平台名 → **macOS | macOSApplicationExtension**

平台名 → **watchOS**

平台名 → **tvOS**

平台版本 → 十进制数字集

平台版本 → 十进制数字集 . 十进制数字集

平台版本 → 十进制数字集 . 十进制数字集 . 十进制数字集

## 声明

---

声明语法

声明→导入声明

声明→常量声明

声明→变量声明

声明→类型别名声明

声明→函数声明

声明→枚举声明

声明→结构体声明

声明→类声明

声明→协议声明

声明→构造器声明

声明→析构器声明

声明→扩展声明

声明→下标声明

声明→运算符声明

声明→优先级组声明

声明集→声明 声明集<sub>可选</sub>

顶级声明语法

顶级声明→多条语句<sub>可选</sub>

代码块语法

代码块→{多条语句<sub>可选</sub>}

导入声明语法

导入声明→特性<sub>可选</sub> import 导入类型<sub>可选</sub> 导入路径

导入类型→typealias | struct | class | enum | protocol | let | var | func

导入路径→导入路径标识符 | 导入路径标识符.导入路径

导入路径标识符→标识符 | 运算符

## 常数声明语法

常量声明 → 特性 可选 声明修饰符集 可选 let 模式构造器集

模式构造器集 → 模式构造器 | 模式构造器 , 模式构造器集

模式构造器 → 模式 构造器 可选

构造器 → = 表达式

## 变量声明语法

变量声明 → 变量声明头 模式构造器集

变量声明 → 变量声明头 变量名 类型注解 代码块

变量声明 → 变量声明头 变量名 类型注解 getter-setter 块

变量声明 → 变量声明头 变量名 类型注解 getter-setter 关键字块

变量声明 → 变量声明头 变量名 构造器可选 willSet-didSet 代码块

变量声明 → 变量声明头 变量名 类型注解 构造器可选 willSet-didSet 代码块

变量声明头 → 特性可选 声明修饰符集可选 var

变量名称 → 标识符

getter-setter 块 → 代码块

getter-setter 块 → { getter 子句 setter 子句 可选 }

getter-setter 块 → { setter 子句 getter 子句 }

getter 子句 → 特性可选 可变性修饰符可选 get 代码块

setter 子句 → 特性可选 可变性修饰符可选 set setter 名称可选 代码块

setter 名称 → ( 标识符 )

getter-setter 关键字 (Keyword) 块 → { getter 关键字子句 setter 关键字子句 可选 }

getter-setter 关键字 (Keyword) 块 → { setter 关键字子句 getter 关键字子句 }

getter 关键字 (Keyword) 子句 → 特性可选 可变性修饰符可选 get

setter 关键字 (Keyword) 子句 → 特性可选 可变性修饰符可选 set

willSet-didSet 代码块 → { willSet 子句 didSet 子句 可选 }

willSet-didSet 代码块 → { didSet 子句 willSet 子句 可选 }

willSet 子句 → 特性可选 willSet setter 名称可选 代码块

didSet 子句 → 特性可选

didSet setter 名称可选 代码块

## 类型别名声明语法

类型别名声明 → 特性可选 访问级别修饰符 typealias 类型别名名称 泛型参数子句可选  
选 类型别名赋值

类型别名名称 → 标识符

类型别名赋值 → = 类型

## 函数声明语法

函数声明 → 函数头 函数名 泛型参数子句可选 函数签名 泛型 where 子句可选 函数  
体可选

函数头 → 特性可选 声明描述符集可选 func

函数名 → 标识符 | 运算符

函数签名 → 参数子句 throws 可选 函数结果 可选

函数签名 → 参数子句 rethrows 函数结果 可选

函数结果 → -> 特性可选 类型

函数体 → 代码块

参数子句 → () | (参数集)

参数集 → 参数 | 参数, 参数集

参数 → 外部参数名 可选 本地参数名 类型注解 默认参数子句 可选

参数 → 外部参数名 可选 本地参数名 类型注解

参数 → 外部参数名 可选 本地参数名 类型注解 ...

外部参数名 → 标识符

本地参数名 → 标识符

默认参数子句 → = 表达式

## 枚举声明语法

枚举声明 → 特性<sub>可选</sub> 访问级别修饰符<sub>可选</sub> 联合式枚举

枚举声明 → 特性<sub>可选</sub> 访问级别修饰符<sub>可选</sub> 原始值式枚举

联合式枚举 → indirect<sub>可选</sub> enum 枚举名 泛型参数子句<sub>可选</sub> 类型继承子句<sub>可选</sub> 泛型 where 子句<sub>可选</sub> { 联合式枚举成员<sub>可选</sub> }

联合式枚举成员集 → 联合式枚举成员 联合样式枚举成员集<sub>可选</sub>

联合样式枚举成员 → 声明 | 联合式枚举 case 子句 | 编译控制语句

联合式枚举 case 子句 → 特性<sub>可选</sub> indirect<sub>可选</sub> case 联合式枚举 case 集

联合式枚举 case 集 → 联合式枚举 case | 联合式枚举 case , 联合式枚举 case 集

联合式枚举 case → 枚举的 case 名 元组类型<sub>可选</sub>

枚举名 → 标识符

枚举的 case 名 → 标识符

原始值式枚举 → enum 枚举名 泛型参数子句<sub>可选</sub> 类型继承子句 泛型 where 子句<sub>可选</sub> { 原始值式枚举成员集 }

原始值式枚举成员集 → 原始值式枚举成员 原始值式枚举成员集<sub>可选</sub>

原始值式枚举成员 → 声明 | 原始值式枚举 case 子句 | 编译控制语句

原始值式枚举 case 子句 → 特性<sub>可选</sub> case 原始值式枚举 case 集

原始值式枚举 case 集 → 原始值式枚举 case | 原始值式枚举 case , 原始值式枚举 case 集

原始值式枚举 case → 枚举的 case 名 原始值赋值<sub>可选</sub>

原始值赋值 → = 原始值字面量

原始值字面量 (raw-value-literal) → 数值字面量 | 静态字符串字面量 | 布尔字面量

## 结构体声明语法

结构体声明 → 特性 可选 访问级别修饰符 可选 **struct** 结构体名称 泛型参数子句 可选 类型继承子句 可选 泛型 where 子句 可选 结构体主体

结构体名称 → 标识符

结构体主体 → { 结构体成员集 可选 }

结构体成员集 → 结构体成员 结构体成员集 可选

结构体成员 → 声明集 | 编译控制语句

## 类声明语法

类声明 → 特性 可选 访问级别修饰符 可选 **final** 可选 **class** 类名 泛型参数子句 可选 类型继承子句 泛型 where 子句 可选 类主体

类声明 → 特性 可选 **final** 访问级别修饰符 可选 **class** 类名 泛型参数子句 可选 类型继承子句 泛型 where 子句 可选 类主体

类名 → 标识符

类主体 → { 类成员集 可选 }

类成员集 → 类成员 类成员集 可选

类成员 → 声明集 | 编译控制语句

## 协议声明语法

协议声明 → 特性<sub>可选</sub> 访问级别修饰符<sub>可选</sub> protocol 协议名 类型继承子句<sub>可选</sub> 泛型 where 子句<sub>可选</sub> 协议主体

协议名 → 标识符

协议主体 → { 协议成员集<sub>可选</sub> }

协议成员集 → 协议成员 协议成员集<sub>可选</sub>

协议成员 → 协议成员声明 | 编译控制语句

协议成员声明 → 协议属性声明

协议成员声明 → 协议方法声明

协议成员声明 → 协议构造器声明

协议成员声明 → 协议下标声明

协议成员声明 → 协议关联类型声明

协议成员声明 → 类型别名声明

## 协议属性声明语法

协议属性声明 → 变量声明头 变量名 类型注解 getter-setter 关键字块

## 协议方法声明语法

协议方法声明 → 函数头 函数名 泛型参数子句<sub>可选</sub> 函数签名 泛型 where 子句<sub>可选</sub>

## 协议构造器声明语法

协议构造器声明 → 构造器头 泛型参数子句<sub>可选</sub> 参数子句 throws<sub>可选</sub> 泛型 where 子句<sub>可选</sub>

协议构造器声明 → 构造器头 泛型参数子句<sub>可选</sub> 参数子句 rethrows 泛型 where 子句<sub>可选</sub>

## 协议下标声明语法

协议下标声明 → 下标头 下标结果 泛型 where 子句<sub>可选</sub> getter-setter 关键字块

## 协议关联类型声明语法

协议关联类型声明 → 特性<sub>可选</sub> 访问级别修饰符<sub>可选</sub> associatedtype 类型别名 类型继承子句<sub>可选</sub> 类型别名赋值<sub>可选</sub> 泛型 where 子句<sub>可选</sub>

## 构造器声明语法

构造器声明 → 构造器头 泛型参数子句 可选 参数子句 throws 可选 泛型 where 子句 可选  
构造器主体

构造器声明 → 构造器头 泛型参数子句 可选 参数子句 rethrows 泛型 where 子句 可选  
构造器主体

构造器头 (Head) → 特性 可选 声明修饰符集 可选 init

构造器头 (Head) → 特性 可选 声明修饰符集 可选 init ?

构造器头 (Head) → 特性 可选 声明修饰符集 可选 init !

构造器主体 → 代码块

## 析构器声明语法

析构器声明 → 特性 可选 deinit 代码块

## 扩展声明语法

扩展声明 → 特性 可选 访问级别修饰符 可选 extension 类型标识 类型继承子句 可选 泛型 where 子句 可选 扩展主体

扩展主体 → { 扩展成员集 可选 }

扩展成员集 → 扩展成员 扩展成员集 可选

扩展成员 → 声明集 | 编译控制语句

## 下标声明语法

下标声明 → 下标头 下标结果 泛型 where 子句 可选 代码块

下标声明 → 下标头 下标结果 泛型 where 子句 可选 getter-setter 块

下标声明 → 下标头 下标结果 泛型 where 子句 可选 getter-setter 关键字块

下标头 (Head) → 特性 可选 声明修饰符集 可选 subscript 泛型参数子句 可选 参数子句

下标结果 (Result) → -> 特性 可选 类型

## 运算符声明语法

运算符声明 → 前置运算符声明 | 后置运算符声明 | 中置运算符声明

前置运算符声明 → **prefix operator** 运算符

后置运算符声明 → **postfix operator** 运算符

中置运算符声明 → **infix operator** 运算符 中置运算符特性<sub>可选</sub>

中置运算符特性 → 优先级组名

## 优先级组声明语法

优先级组声明 → **precedencegroup** 优先级组名 { 优先级组特性<sub>可选</sub> }

优先级组特性 → 优先级组属性 优先级组特性<sub>可选</sub>

优先级组属性 → 优先级组关系

优先级组属性 → 优先级组赋值

优先级组属性 → 优先级组结合

优先级组关系 → **higherThan** : 优先级组名集

优先级组关系 → **lowerThan** : 优先级组名集

优先级组赋值 → **assignment** : 布尔字面量

优先级组结合 → **associativity** : **left**

优先级组结合 → **associativity** : **right**

优先级组结合 → **associativity** : **none**

优先级组名集 → 优先级组名 | 优先级组名, 优先级组名集

优先级组名 → 标识符

## 声明修饰符语法

声明修饰符 → class | convenience | dynamic | final | infix | lazy | optional | override | postfix | prefix | required | static | unowned | unowned(safe) | unowned(unsafe) | weak

声明修饰符 → 访问级别修饰符

声明修饰符 → 可变性修饰符

声明修饰符集 → 声明修饰符 声明修饰符集 可选

访问级别修饰符 → private | private(set)

访问级别修饰符 → fileprivate | fileprivate(set)

访问级别修饰符 → internal | internal(set)

访问级别修饰符 → public | public(set)

访问级别修饰符 → open | open(set)

可变性修饰符 → mutating | nonmutating

## 属性

---

### 属性语法

属性 → @ 属性名 属性参数子句 可选

属性名 → 标识符

属性参数子句 → { 平衡令牌集 可选 }

属性 (Attributes) 集 → 属性 特性 可选

平衡令牌集 → 平衡令牌 平衡令牌集 可选

平衡令牌 → ( 平衡令牌集 可选 )

平衡令牌 → [ 平衡令牌集 可选 ]

平衡令牌 → { 平衡令牌集 可选 }

平衡令牌 → 任意标识符、关键字、字面量或运算符

平衡令牌 → 除 ( 、 )、[ 、 ]、{ 、 } 外的任意标点符号

## 模式

---

模式语法

模式 → 通配符模式 类型注解 可选

模式 → 标识符模式 类型注解 可选

模式 → 值绑定模式

模式 → 元组模式 类型注解 可选

模式 → 枚举 case 模式

模式 → 可选模式

模式 → 类型转换模式

模式 → 表达式模式

通配符模式语法

通配符模式 → \_

标识符模式语法

标识符模式 → 标识符

值绑定模式语法

值绑定模式 → var 模式 | let 模式

元组模式语法

元组模式 → ( 元组模式元素集 可选 )

元组模式元素集 → 元组模式元素 | 元组模式元素 , 元组模式元素集

元组模式元素 → 模式 | 标识符 : 模式

枚举 case 模式语法

enum-case-pattern → 类型标识 可选 . 枚举 case 名 元组模式 可选

可选模式语法

可选模式 → 标识符模式 ?

类型转换模式语法

类型转换模式 → is 模式 | as 模式

is 模式 → is 类型

as 模式 → 模式 as 类型

表达式模式语法

表达式模式 → 表达式

## 泛型参数

---

泛型形参子句语法

泛型参数子句 → <泛型参数集>

泛型参数集 → 泛型参数 | 泛形参数, 泛型参数集

泛形参数 → 类型名称

泛形参数 → 类型名称 : 类型标识

泛形参数 → 类型名称 : 协议合成类型

泛型 where 子句 → where 约束集

约束集 → 约束 | 约束, 约束集

约束 → 一致性约束 | 同类型约束

一致性约束 → 类型标识 : 类型标识

一致性约束 → 类型标识 : 协议合成类型

同类型约束 → 类型标识 == 类型

泛型实参子句语法

泛型实参子句 → <泛型实参集>

泛型实参集 → 泛型实参 | 泛形实参, 泛型实参集

泛形实参 → 类型

# 翻译贡献者 · GitBook

---

 [runoob.com/manual/gitbook/swift5/source/\\_book/contributors.html](https://runoob.com/manual/gitbook/swift5/source/_book/contributors.html)

## 文档翻译 & 校对工作记录

---

Swift 官方文档中文翻译由 [numbbbb](#) 发起并主导，本项目已经得到了苹果官方的认可（Translations 部分）。下面是各个版本官方文档翻译和校对工作的主要贡献者，排名不分先后。

### Swift 5.x 主要贡献者

---

### Swift 4.x 主要贡献者

---

### Swift 3.x 主要贡献者

---

- [bqlin](#)
- [chenmingjia](#)
- [CMB](#)
- [crayygy](#)
- [kemchenji](#)
- [Lanford](#)
- [mmoaay](#)
- [mylittleswift](#)
- [qhd](#)
- [shanks](#)

### Swift 2.x 主要贡献者

---

### Swift 1.x 主要贡献者

---