目錄

The Swift Programming Language (Swift 4) 中文版

本文由 Kuntanury 进行翻译，允许保持署名及原文链接转载，但不得用于商业目的。

第一次做翻译，由于经验有限，难免存在诸多疏漏或者理解不到位的地方，主页中有我的联系方式，还请斧正～

英文原文地址

关于 Swift

> 提示
>
> 此文档仅包含在开发中遇到的关于 API 或者技术的基础内容，由于本文内容可能会有变动，所以根据本文实现的软件应在最新的操作系统上完成测试。

Swift 不管是在手机、桌面，服务器终端或者任何其他可以运行代码的终端上，都是编写软件的不二选择。它将苹果公司悠久的工程师文化和开源社区的贡献这两个现代编程语言中最精华的部分结合在一起，诞生了这么一门安全，快速、交互性强的编程语言，在优化编译保持高性能的同时兼顾了简化语言方便开发。

Swift 对新程序员非常友好，因为它虽然是工业级编程语言，但是却又像脚本语言那样既富有表现力又生动有趣，在 Playground 中编写 Swift 代码就能实时查看运行结果，而不必经常性的编译运行程序。

Swift 采用现代编程模式来避免各种基本性编程错误：

- 变量使用之前必须初始化
- 数组越界
- 整型溢出
- 可选类型确保nil值被明确处理
- 自动内存管理

错误处理能使程序从意外错误中恢复

Swift 为充分利用现代硬件进行了编译和优化，我们基于简单即是表现的极致的原则来设计语法和标准库，小到编写简单的 “Hello, world!” 程序，大到构建整个操作系统， Swift 集高效与高速与一身的特性都是完美的选择。

Swift 将强大的类型推导和模式匹配这些复杂的思想用现代的、轻语法这种清晰简洁的方式传达，这使得代码不仅书写起来更为容易，阅读和维护也变得更加轻松。

Swift 已经经过几年的打磨，目前仍然在继续增加着新的特性和功能。我们对于 Swift 寄予厚望，迫不及待地想一览你用 Swift 创建的天地～

版本兼容

本书描述的是 Swift 4.0 版本，在 Xcode 9 中为 Swift 默认版本。你可以用Xcode 9来编译 Swift 3 或者 4 编写的代码。

> 备注
>
> 当 Swfit 4 编译 Swift 3 代码时，会默认处理为3.2版本代码，这表示你可以用 `#if swift(>=3.2)` 这种判断条件来让代码兼容多个版本的 Swift 编译器

当你用 Xcode 9 编译 Swift 3 代码的时候， Swift 4 中的大部分功能也是可用的。只有如下功能是 Swift 4 代码独有的：

- 拆分字符串结果可以用 `Substring` 取代 `String` 类型
- `@objc` 修饰符可以在一些地方隐性调用
- 同一文件中类型的扩展可以访问当前类型中的私有成员

用 Swift 4 书写的 Target 可以依赖于 Swift 3 书写的 Target ，反之亦然。也就是说，如果你的工程分成了多个框架，你可以一个框架一个框架地从 Swift 3 迁移到 Swift 4 。

管中窥豹

依照传统，使用新语言写的第一个程序都应该是在屏幕上打印 “Hello, world!” ，用 Swift ，一行搞定：

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

如果你曾经写过 C 或者 Objective-C 代码，那么你对 Swift 的语法不会感觉到陌生，在 Swift 中，上面的一行代码就是一个完整的程序，不需要引入单独的库或者I/O模块或者字符串处理神马的，因为写在全局范围的代码被当作程序的入口，所以你连 main() 函数都省了，也不用在每句话结尾写分号了。

本节通过运用 Swfit 完成编程任务的方式，来让你 Swift 的有充足的了解。如果有什么地方不理解也不要担心——这个概览介绍的内容在本书的余下章节里都会进行详细解释的。

> 备注
>
> 为了获得最佳体验，推荐用 Xcode 创建 playground 来试验本章内容，Playgrounds 允许你编辑代码并立即看到代码的运算结果。
>
> 下载Playground

简单值

用 `let` 声明常量，用 `var` 声明变量。常量在定义时不需要初始值，但是后续只能对它赋一次值。也就是说，你可以用常量来定义一个在很多地方用到的统一的值：

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

常量或变量的类型必须和赋值类型一样，然而，你却不需要写明类型，因为编译器可以根据你赋值的类型来推断他们类型：比如在上面的例子中， `myConstant` 初始化的值为整型数字，所以编译器推断 `myConstant` 的类型为整型。

如果初始化值未能提供足够的推断信息（或者没有初始值），可以显式的将类型写在变量的后面，用冒号与变量隔开：

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

> 小试身手
>
> 创建一个显式类型为 **Float** 的值为 **4** 的常量。

值永远不会隐式转换为其他类型。如果你需要把一个值转换成不同类型，可以用显式类型转换来取得目标类型：

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

> 小试身手
>
> 尝试一下把最后一行的 String 类型转换去掉，看看编译器报什么错？

有一个更为简单的方法来将值转化为字符串——将值放在前面带有反斜线的括号 `\()` 里，如下所示：

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

> 小试身手
>
> 在一句欢迎语中用 `\()` 来把一个浮点运算结果转化为字符串并和某人的名字拼接起来
>
> ~~这个练习好奇怪，但是我还是做了：let greeting = "欢迎佳佳这个 \(3.0 - 1.0)货～"~~

用三双引号 `"""` 来定义多行的字符串，每行字符的缩进都和结束的三双引号的缩进相同，如下所示（官方文档此处举例可能有问题，括号内及注释为个人理解所加，如有偏差，敬请指教）：

```
let quotation = """
（空格)Even though there's whitespace to the left,//即使左边有空白字符
（空格)the actual lines aren't indented.//实际是不会包含在多行字符里面的
（空格)（空格)Except for this line.//除了这行
（空格)Double quotes (") can appear without being escaped.//双引号可以不用转义

（空格)I still have \(apples + oranges) pieces of fruit.//我还有几个苹果和橘子呢
（空格)"""
```

用方括号 `[]` 创建数组和字典，而访问的时候则是通过用方括号中写索引值或者键值的方式。最后一个元素后多加个逗号无碍：

```
var shoppingList = ["catfish", "water", "tulips", "blue paint"]
shoppingList[1] = "bottle of water"

var occupations = [
    "Malcolm": "Captain",
    "Kaylee": "Mechanic",
]
occupations["Jayne"] = "Public Relations"
```

用初始化语法创建一个空的数组或字典：

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

如果元素类型可推导，你可以用 [] 初始化数组，用 [:] 初始化字典，就像你给变量赋值或者给函数传参数一样：

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

控制流程

`if` 和 `switch` 用来做条件判断，and use `for - in` ， `while` 和 `repeat - while` 用来做循环。判断条件和循环变量的圆括号可以省略，但是语句体的大括号不能：

```
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，判断条件必须是一个返回布尔值的表达式——也就是说 `if score { ... }` 这类写法是错误的，因为编译器不会隐式地与零值做比较。

你可以兼用 `if` 和 `let` 来处理赋值的变量可能为空的情况，这些赋值的变量表现为可选类型。可选类型或有确定值，或用 `nil` 来表示空值。在类型后添加 `?` 来表示变量为可选类型：

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

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

> 小试身手
>
> 把 `optionalName` 值改为 `nil` ，greeting输出什么？添加 `else` 分句：如果 `optionalName` 值为 `nil` 输出不同的greeting。

如果可选类型值为 nil ，由于判断条件为 false ，花括号中的代码就被跳过了。反之，可选类型的值就会被解包并赋给 let 声明的常量，这样，解包的值能够在花括号中可用了。

另一个处理可选类型值的方式是用 ?? 来提供一个默认值，如果可选类型值为空，就使用默认值：

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

Switch语句支持任意数据类型的各种比较操作——不拘泥于整数及等式检测：

```
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.")
}
```

> 小试身手
>
> 把 `default` 结果删掉，看看报什么错？

留意一下上例中 `let` 的用法：通过将匹配式的值赋给常量的方式使得在判断分支中可以调用匹配式的值。

运行完 `switch` 语句中 `case` 匹配的代码后，程序就会结束整个选择。由于不在自动执行下一个判断，所以每个分支中的代码不需要加 `break` 来结束选择。

你可以 for-in 遍历代表键值对的一对值的方式来遍历字典。字典是无序的，所以遍历也是无序的：

```
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)
```

> 小试身手
>
> 添加另一个变量来记录最大值的类型，同时仍然记录这个最大值。

用 `while` 设定一个条件来循环执行一段代码。如果把循环条件写在结尾，可以保证循环至少执行一次：

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

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

通过 ..< 可以在循环中设定一个索引来约束范围：

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

用 ..< 约束的范围不包括上界的值，用 ... 可以设定同时包含上下界值的范围。

函数和闭包

用 func 来声明一个函数，用函数名和括号内的参数来调用这个函数。用 `->` 分割参数名和函数返回类型：

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

> 小试身手
>
> 去掉 day 参数，在问候语中加上一个具体午餐的参数。

一般情况下，函数用形参作为实参标签。实参标签也可以通过写在形参前的标签来自定义，或者用 `_` 来表示无实参标签：

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

用元组来创建复合值——例如，让函数来返回多个值。元组中的元素可以用名称或者索引来表示：

```
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)
```

函数是可以嵌套的。嵌套函数可以访问声明在外层函数中的参数。可以通过嵌套函数来简化那些在一个函数中又长又复杂的代码。

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

函数是第一类对象（First-class object：在计算机科学中指可以在执行期创造并作为参数传递给其他函数或存入一个变数的实体 - 摘自维基百科）。就是说函数可以把另一个函数作为自己的返回值：

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

函数也可以把另一个函数作为自己的参数：

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

函数实际上是一种特殊的闭包：一段延后调用的代码。闭包中的代码可以存取在闭包内创建的变量和函数，即使闭包是在其他空间执行的——正如之前的嵌套函数一般。你可以将代码写在 ({}) 中来创建一个匿名闭包，用 in 来分隔函数名和返回值：

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

> 小试身手
>
> 重写闭包，让它对所有奇数返回0。

有许多方式可以让闭包更简洁。当闭包类型已知的时候，比如代理的回调，可以忽略其参数、或者其返回值，甚至两样都忽略掉。单语句闭包隐式地返回执行结果。

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

你可以通过参数位置而不是名字来引用参数——这种方式在很短的闭包中非常高效。当闭包作为函数最后一个参数时，它可以直接跟在圆括号后。而如果此时闭包又是函数的唯一参数，圆括号就可以省略掉了：

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

类与对象

用 `class` + 类名来创建一个类。类中属性的声明写法和常量或者变量的声明写法是一样的，只是属性的作用域在类中。同理，类方法和函数的声明也是同样的写法：

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

> 小试身手
>
> 用 let 添加一个类属性，添加一个带一个参数的类方法

EXPERIMENT

Add a constant property with `let`, and add another method that takes an argument.

Create an instance of a class by putting parentheses after the class name. Use dot syntax to access the properties and methods of the instance.

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

This version of the `Shape` class is missing something important: an initializer to set up the class when an instance is created. Use `init` to create one.

1. `class NamedShape {`
2. `var numberOfSides : Int = 0`
3. `var name : String`
4.
5. `init ( name : String ) {`
6. `self . name = name`
7. `}`
8.
9. `func simpleDescription () - > String {`
10. `return "A shape with \( numberOfSides ) sides."`
11. `}`
12. `}`

Notice how `self` is used to distinguish the `name` property from the `name` argument to the initializer. The arguments to the initializer are passed like a function call when you create an instance of the class. Every property needs a value assigned—either in its declaration (as with `numberOfSides`) or in the initializer (as with `name`).

Use `deinit` to create a deinitializer if you need to perform some cleanup before the object is deallocated.

Subclasses include their superclass name after their class name, separated by a colon. There is no requirement for classes to subclass any standard root class, so you can include or omit a superclass as needed.

Methods on a subclass that override the superclass's implementation are marked with `override` —overriding a method by accident, without `override` , is detected by the compiler as an error. The compiler also detects methods with `override` that don't actually override any method in the superclass.

```
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 ()
```

EXPERIMENT

Make another subclass of `NamedShape` called `Circle` that takes a radius and a name as arguments to its initializer. Implement an `area()` and a `simpleDescription()` method on the `Circle` class.

In addition to simple properties that are stored, properties can have a getter and a setter.

```
class EquilateralTriangle : NamedShape {
var sideLength : Double = 0.0

```

```
4. init ( sideLength : Double , name : String ) {
5. self . sideLength = sideLength
6. super . init ( name : name )
7. numberOfSides = 3
8. }
9.
10. var perimeter : Double {
11. get {
12. return 3.0 * sideLength
13. }
14. set {
15. sideLength = newValue / 3.0
16. }
17. }
18.
19. override func simpleDescription () - > String {
20. return "An equilateral triangle with sides of length \( sideLength ) ."
21. }
22. }
23. var triangle = EquilateralTriangle ( sideLength : 3.1 , name : "a triangle" )
24. print ( triangle . perimeter )
25. triangle . perimeter = 9.9
26. print ( triangle . sideLength )
```

In the setter for `perimeter`, the new value has the implicit name `newValue`. You can provide an explicit name in parentheses after `set`.

Notice that the initializer for the `EquilateralTriangle` class has three different steps:

1. Setting the value of properties that the subclass declares.
2. Calling the superclass's initializer.
3. Changing the value of properties defined by the superclass. Any additional setup work that uses methods, getters, or setters can also be done at this point.

If you don't need to compute the property but still need to provide code that is run before and after setting a new value, use `willSet` and `didSet`. The code you provide is run any time the value changes outside of an initializer. For example, the class below ensures that the side length of its triangle is always the same as the side length of its square.

```
1. class TriangleAndSquare {
2. 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 )
```

When working with optional values, you can write `?` before operations like methods, properties, and subscripting. If the value before the `?` is `nil`, everything after the `?` is ignored and the value of the whole expression is `nil`. Otherwise, the optional value is unwrapped, and everything after the `?` acts on the unwrapped value. In both cases, the value of the whole expression is an optional value.

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