快速游览

在本页

传统表明，用新语言编写的第一个程序应该在屏幕上打印“Hello，world！”。在Swift中，这可以在一行中完成：

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

如果你用C或Objective-C编写代码，这个语法看起来很熟悉 - 在Swift中，这行代码是一个完整的程序。您不需要为输入/输出或字符串处理等功能导入单独的库。在全局范围编写的代码被用作程序的入口点，所以你不需要一个`main()`函数。您也不需要在每个语句的末尾都写分号。

通过本教程，您将了解如何完成各种编程任务，从而为您开始在Swift中编写代码提供足够的信息。如果您不了解某些内容，请不要担心，本书其余部分详细介绍了本次导览中介绍的所有内容。

> 注意
>
> 为了获得最佳体验，请将本章作为Xcode中的游乐场打开。游乐场让你编辑代码清单，并立即看到结果。
>
> **下载游乐场**

简单的价值

使用`let`做一个常数，`var`使一个变量。在编译时不需要知道常量的值，但必须为其赋值一次。这意味着您可以使用常量来命名一次您确定一次但在许多地方使用的值。

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

常量或变量必须与要分配给它的值具有相同的类型。但是，您并不总是必须明确地写出类型。在创建常量或变量时提供值可让编译器推断其类型。在上面的例子中，编译器推断这`myVariable`是一个整数，因为它的初始值是一个整数。

如果初始值没有提供足够的信息（或者没有初始值），请在变量之后写入，并用冒号分隔来指定类型。

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

> 实验
>
> 用显式类型`Float`和值创建一个常量4。

值永远不会被隐式转换为另一种类型。如果您需要将值转换为其他类型，请显式创建所需类型的实例。

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

> 实验
>
> 尝试`String`从最后一行删除转换。你会得到什么错误?

在字符串中包含值的方法更为简单：将值写入括号中，并在括号`\`之前写入反斜杠（）。例如：

```
let apples = 3
let oranges = 5
let appleSummary = "I have \(apples) apples."
```

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

> 实验
> 用于\()在字符串中包含浮点计算，并在问候语中包含某人的姓名。

"""对于占用多行的字符串 使用三个双引号（）。只要符合收尾引号的缩进，每个引用行开头的缩进就会被删除。例如：

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

使用括号（[]）创建数组和字典，并通过将括号中的索引或键写入它们来访问它们的元素。在最后一个元素之后允许逗号。

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

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

要创建一个空数组或字典，请使用初始化程序语法。

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

如果可以推断出类型信息，那么可以将空数组写为[]空字典，[:]例如，当为变量设置新值或将参数传递给函数时。

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

控制流

使用if和switch制作条件语句和使用for- in，while和repeat- while进行循环。条件或循环变量的括号是可选的。身体周围的括号是必需的。

```
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声明中，条件必须是一个布尔表达式 - 这意味着代码如if score { ... }错误，而不是隐式比较为零。

您可以使用if并let一起使用可能缺失的值。这些值表示为可选项。可选值包含值或包含nil以指示缺少值。?在值的类型后面写一个问号（）以将该值标记为可选。

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

```
3
4  var optionalName: String? = "John Appleseed"
5  var greeting = "Hello!"
6  if let
7      greeting = "Hello, \(name)
8  }
```

> 实验
>
> 更改optionalName为nil。你有什么问候？添加else，设置不同的问候语，如果条款optionalName是nil。

如果可选值为nil，则条件为，false并且花括号中的代码被跳过。否则，可选值将被解包并分配给之后的常量let，这会使代码块中的解包值可用。

处理可选值的另一种方法是使用??运算符提供默认值。如果可选值缺失，则使用默认值。

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

交换机支持任何种类的数据和各种比较操作 - 它们不限于整数和相等性测试。

```
1  let vegetable = "red pepper"
2  switch vegetable {
3  case "celery":
4      print("Add some raisins and make ants on a log.")
5  case "cucumber", "watercress":
6      print("That would make a good tea sandwich.")
7  case let x where x.hasSuffix("pepper"):
8      print("Is it a spicy \(x)?")
9  default:
10     print("Everything tastes good in soup.")
11 }
```

> 实验
>
> 尝试删除默认情况。你会得到什么错误？

请注意，如何let在模式中使用将与模式匹配的值分配给常量。

在匹配的switch case内部执行代码后，程序从switch语句中退出。执行不会延续到下一个案例，因此不需要在每个案例的代码末尾显式地跳出交换机。

您可以使用for- in通过提供一对用于每个键值对的名称来迭代字典中的项目。字典是无序集合，所以它们的键和值以任意顺序迭代。

```
1  let interestingNumbers = [
2      "Prime": [2, 3, 5, 7, 11, 13],
3      "Fibonacci": [1, 1, 2, 3, 5, 8],
4      "Square": [1, 4, 9, 16, 25],
5  ]
6  var largest = 0
7  for (kind, numbers) in interestingNumbers {
8      for number in numbers {
9          if number > largest {
10             largest = number
11         }
12     }
13 }
14 print(largest)
```

> 实验
> 添加另一个变量以跟踪哪种数字最大，以及最大数量是多少。

使用`while`重复的代码块，直到病情变化。循环的条件可以在最后，确保循环至少运行一次。

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

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

您可以通过使用`..<`一系列索引来保留索引。

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

使用`..<`使省略了其上限值的范围内，并用`...`做既包括值的范围。

功能和关闭

使用`func`声明函数。通过使用括号中的参数列表跟随其名称来调用函数。用于`->`从函数的返回类型中分离参数名称和类型。

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

> 实验
> 删除`day`参数。添加一个参数，以包含今日午餐特别的问候。

默认情况下，函数使用它们的参数名称作为其参数的标签。在参数名称之前写入自定义参数标签，或写入`_`以使用不带参数标签。

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

使用一个元组来创建一个复合值 - 例如，从一个函数返回多个值。元组的元素可以通过名字或数字来引用。

```
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。嵌套函数可以访问在外部函数中声明的变量。您可以使用嵌套函数来组织长或复杂函数中的代码。

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

函数是一流的类型。这意味着一个函数可以返回另一个函数作为它的值。

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

函数可以将另一个函数作为其参数之一。

```
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)
```

函数实际上是闭包的一个特例：可以稍后调用的代码块。闭包中的代码可以访问变量和函数，这些变量和函数可以在创建闭包的范围中使用，即使闭包在执行时处于不同的范围 - 您已经看到一个嵌套函数的例子。你可以使用花括号（{}）来编写一个没有名字的闭包。用于in从正文中分离参数和返回类型。

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

> 实验
>
> 重写闭包使所有奇数返回零。

您可以更简洁地编写闭包的几个选项。当闭包的类型已知时，例如代理的回调，可以省略其参数的类型，其返回类型或两者。单语句闭包隐式返回其唯一语句的值。

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

您可以通过数字而不是名称来引用参数 - 这种方法在非常短的闭包中特别有用。作为函数的最后一个参数传递的闭包可以在括号之后立即出现。当闭包是函数的唯一参数时，可以完全省略括号。

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

对象和类

使用`class`后跟班的名字来创建一个班级。类中的属性声明与常量或变量声明的写法相同，只不过它在类的上下文中。同样，方法和函数声明的写法也是一样的。

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

> 实验
>
> 添加一个常量属性`let`，并添加另一个接受参数的方法。

通过在类名后加括号来创建一个类的实例。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`类缺少一些重要的东西：初始化器在创建实例时设置类。使用`init`创建一个。

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

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

    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}
```

请注意，如何`self`将`name`属性`name`与初始值设定项的参数区分开来。在创建类的实例时，初始值设定项的参数会像函数调用一样传递。每个属性都需要赋值 - 无论是在其声明中（如同`numberOfSides`）还是在初始化程序中（与之一样`name`）。

`deinit`如果您需要在释放对象之前执行一些清理操作， 请使用此命令创建取消初始化程序。

子类在其类名后面包含它们的超类名称，并用冒号分隔。不要求类为任何标准根类继承，所以您可以根据需要包含或省略超类。

覆盖超类实现的子类上override的方法被标记为 - 无意中覆盖方法．如果没有override．则被编译器检测为错误。编译器还检

```
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()
```

> 实验
>
> 创建另一个NamedShape被调用的子类Circle，它将一个半径和一个名称作为其初始值设定项的参数。实施area()和simpleDescription()对方法Circle的类。

除了存储的简单属性外，属性还可以有一个getter和一个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setter中perimeter，新值具有隐式名称newValue。你可以在后面的括号中提供一个明确的名字set。

请注意，`EquilateralTriangle`该类的初始化器有三个不同的步骤：

1. 设置子类声明的属性的值。
2. 调用超类的初始化器。
3. 更改由超类定义的属性的值。使用方法、获取器或设置器的任何其他工作也可以在这个时候完成。

如果您不需要计算属性，但仍需要提供在设置新值之前和之后运行的代码，请使用`willSet`和`didSet`。只要初始值设定项外的值发生更改，您提供的代码就会运行。例如，下面的类确保其三角形的边长总是与其正方形的边长相同。

```
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)
```

使用可选值时，可以`?`在诸如方法，属性和下标之类的操作之前编写代码。如果之前的值`?`是`nil`，`?`则忽略之后的所有内容以及整个表达式的值为`nil`。否则，可选值将被解包，并且所有的`?`行为都会被解包。在这两种情况下，整个表达式的值都是一个可选值。

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

枚举和结构

使用`enum`创建一个枚举。像类和所有其他命名类型一样，枚举可以具有与它们相关的方法。

```
enum Rank: Int {
    case ace = 1
    case two, three, four, five, six, seven, eight, nine, ten
    case jack, queen, king
    func simpleDescription() -> String {
        switch self {
        case .ace:
            return "ace"
        case .jack:
            return "jack"
        case .queen:
            return "queen"
        case .king:
            return "king"
        default:
            return String(self.rawValue)
        }
    }
```

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

> 实验
>
> 编写一个比较两个Rank值的函数，比较它们的原始值。

默认情况下，Swift分配原始值从零开始，每次递增1，但您可以通过显式指定值来更改此行为。在上面的例子中，Ace明确给出了一个原始值1，其余的原始值是按顺序分配的。您也可以使用字符串或浮点数作为枚举的原始类型。使用该rawValue属性来访问枚举个案的原始值。

使用init?(rawValue:)初始化程序从原始值创建枚举的实例。它返回匹配原始值的枚举情况或者nil没有匹配的情况Rank。

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

枚举的实例值是实际值，而不仅仅是编写其原始值的另一种方式。事实上，在没有有意义的原始价值的情况下，您不必提供一个。

```
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()
```

> 实验
>
> 添加一个黑桃和黑桃返回“黑色”的color()方法Suit，并返回“红色”用于心脏和钻石。

注意hearts上面引用枚举情况的两种方式：当为hearts常量赋值时，枚举大小写Suit.hearts由全名引用，因为该常量没有指定显式类型。在交换机内部，枚举案由缩写形式引用，.hearts因为self已知该值是套装。只要该值的类型已知，您就可以使用缩写形式。

如果枚举具有原始值，则这些值将作为声明的一部分来确定，这意味着特定枚举案例的每个实例始终具有相同的原始值。枚举案例的另一种选择是将值与案例相关联 - 这些值在创建实例时确定，并且对于枚举案例的每个实例它们可以不同。您可以将关联的值视为像枚举大小写实例的存储属性一样工作。例如，考虑从服务器请求日出和日落时间的情况。服务器或者以请求的信息作出响应，或者以对出错的描述做出响应。

```
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):
    pri
}
```

> 实验
> ServerResponse向交换机添加第三种情况。

注意日出和日落时间是如何从ServerResponse值中提取的，作为将值与开关情况相匹配的一部分。

使用struct创建的结构。结构支持许多与类相同的行为，包括方法和初始化器。结构和类之间最重要的差异之一就是结构在代码中传递时总是被复制，但类是通过引用传递的。

```
struct Card {
    var rank: Rank
    var suit: Suit
    func simpleDescription() -> String {
        return "The \(rank.simpleDescription()) of \(suit.simpleDescription())"
    }
}
let threeOfSpades = Card(rank: .three, suit: .spades)
let threeOfSpadesDescription = threeOfSpades.simpleDescription()
```

> 实验
> 添加一个方法来Card创建一套完整的卡片，每张卡片组合一个等级和套装。

协议和扩展

使用protocol申报的协议。

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

类，枚举和结构都可以采用协议。

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

> 实验
>
> 写一个符合这个协议的枚举。

注意在声明中使用`mutating`关键字`SimpleStructure`来标记修改结构的方法。声明`SimpleClass`不需要任何标记为变异的方法，因为类上的方法总是可以修改类。

用于`extension`向现有类型添加功能，例如新方法和计算属性。您可以使用扩展来将协议一致性添加到其他位置声明的类型，或者甚至添加到从库或框架导入的类型。

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

> 实验
>
> 为`Double`添加`absoluteValue`属性的类型编写扩展名。

您可以像使用任何其他命名类型一样使用协议名称，例如，创建具有不同类型的对象集合，但都符合单个协议。当您使用类型为协议类型的值时，协议定义之外的方法不可用。

```
let protocolValue: ExampleProtocol = a
print(protocolValue.simpleDescription)
// print(protocolValue.anotherProperty)  // Uncomment to see the error
```

即使变量`protocolValue`具有运行时类型`SimpleClass`，编译器也会将其视为给定的类型`ExampleProtocol`。这意味着除了协议一致性之外，您不能意外地访问类实现的方法或属性。

错误处理

使用任何采用`Error`协议的类型代表错误。

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

使用`throw`抛出一个错误，并`throws`标记，可以抛出一个错误的功能。如果在函数中抛出错误，函数会立即返回，并调用函数的代码处理错误。

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

有几种方法可以处理错误。一种方法是使用`do-catch`。在`do`块内部，您可以通过`try`在其前写入代码来标记可能导致错误的代码。在`catch`块内，错误会自动给出名称，`error`除非您给它一个不同的名称。

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

> 实验
>
> 将打印机名称更改为"Never Has Toner"，以便该send(job:toPrinter:)函数引发错误。

您可以提供catch处理特定错误的多个块。你在写完一个模式后catch就像case在切换之后一样。

```
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)
}
```

> 实验
>
> 添加代码以在do块内引发错误。你需要抛出什么样的错误，以便错误由第一个catch块处理？第二块和第三块呢？

处理错误的另一种方法是使用try?将结果转换为可选项。如果函数抛出一个错误，则丢弃该特定错误并返回结果nil。否则，结果是包含函数返回值的可选项。

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

用defer写的是在函数中的所有其它代码后执行代码块，只是在函数返回之前。无论函数是否引发错误，代码都会执行。defer即使需要在不同的时间执行它们，您也可以使用它们将设置和清理代码编写在一起。

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

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

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

泛型

在尖括号内写一个名字来创建一个通用函数或类型。

```
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)
```

您可以制作通用形式的函数和方法，以及类，枚举和结构。

```
// Reimplement the Swift standard library's optional type
enum OptionalValue<Wrapped> {
    case none
    case some(Wrapped)
}
var possibleInteger: OptionalValue<Int> = .none
possibleInteger = .some(100)
```

where在正文前面 使用指定需求列表 - 例如，要求类型实现协议，要求两种类型相同，或要求类具有特定的超类。

```
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])
```

> 实验
> 修改`anyCommonElements(_:_:)`函数以生成一个函数，该函数返回任意两个序列共有的元素数组。

写作`<T: Equatable>`与写作相同`<T> ... where T: Equatable`。