The Swift Programming Language 中文版

作者：极客学院

3.0 更新说明

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: crayygy - 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！

1 欢迎使用 Swift

在本章中您将了解 Swift 的特性和开发历史，并对 Swift 有一个初步的了解。

关于 Swift（About Swift）

1.0 翻译：numbbbbb
校对：yeahdongcn

2.0 翻译+校对：xtymichael

3.0 翻译+校对：shanks，2016-10-06 3.0.1 review : 2016-11-09

Swift 是一种新的编程语言，用于编写 iOS，macOS，watchOS 和 tvOS 的应用程序。Swift 结合了 C 和 Objective-C 的优点并且不受 C 兼容性的限制。Swift 采用安全的编程模式并添加了很多新特性，这将使编程更简单，更灵活，也更有趣。Swift 是基于成熟而且倍受喜爱的 Cocoa 和 Cocoa Touch 框架，它的降临将重新定义软件开发。

Swift 的开发从很久之前就开始了。为了给 Swift 打好基础，苹果公司改进了编译器，调试器和框架结构。我们使用自动引用计数（Automatic Reference Counting, ARC）来简化内存管理。我们在 Foundation 和 Cocoa 的基础上构建框架栈使其完全现代化和标准化。 Objective-C 本身支持块、集合语法和模块，所以框架可以轻松支持现代编程语言技术。正是得益于这些基础工作，我们现在才能发布这样一个用于未来苹果软件开发的新语言。

Objective-C 开发者对 Swift 并不会感到陌生。它采用了 Objective-C 的命名参数以及动态对象模型，可以无缝对接到现有的 Cocoa 框架，并且可以兼容 Objective-C 代码。在此基础之上，Swift 还有许多新特性并且支持过程式编程和面向对象编程。

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 将现代编程语言的精华和苹果工程师文化的智慧结合了起来。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 既可以用于开发 “hello, world” 这样的小程序，也可以用于开发一套完整的操作系统。所有的这些特性让 Swift 对于开发者和苹果来说都是一项值得的投资。

Swift 是编写 iOS，macOS，watchOS 和 tvOS 应用的极佳手段，并将伴随着新的特性和功能持续演进。我们对 Swift 充满信心，你还在等什么！

Swift 初见（A Swift Tour）

1.0 翻译：numbbbbb 校对：shinyzhu, stanzhai

2.0 翻译+校对：xtymichael

2.2 翻译：175，2016-04-09 校对：SketchK，2016-05-11

3.0 翻译+校对：shanks，2016-10-06

3.0.1 review: 2016-11-09

本页内容包括：

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： 最佳实践是，在 Xcode 作为 playground 打开本章，Playgrounds允许你编辑你的代码并且立即得到结果。
>
> 下载 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。也就是说你可以用常量来表示这样一个值：你只需要决定一次，但是需要使用很多次。

```
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，声明的同时赋值的话，编译器会自动推断类型。在上面的例子中，编译器推断出`myVariable`是一个整数（integer）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"
```

要创建一个空数组或者字典，使用初始化语法。

```
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`、`for`、`while`和`repeat-while`来进行循环。包裹条件和循环变量括号可以省略，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 0 做对比。

你可以一起使用`if`和`let`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`中匹配到的子句之后，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，需要两个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不满足条件。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 = n * 2
}
print(n)

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

你可以在循环中使用`..<`来表示范围。

```
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)
```

函数可以带有可变个数的参数，这些参数在函数内表现为数组的形式：

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

练习：写一个计算参数平均值的函数。

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所建作用域中能得到的变量和函数，即使闭包是在一个不同的作用域被执行的 - 你已经在嵌套函数例子中所看到。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，比如作为一个回调函数，你可以忽略参数的类型和返回值。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.sort { $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。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`）。

如果你需要在删除对象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，一个是半径一个是名称，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 triagle 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。

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:)初始化构造器在原始值和枚举值之间进行转换。

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

枚举的成员值是实际值，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，有两种方式可以引用Hearts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的值已经知道是一个suit。已知变量类型的情况下你可以使用缩写。

一个枚举成员的实例可以有实例值。相同枚举成员的实例可以有不同的值。创建实例的时候传入值即可。实例值和原始值是不同的：枚举成员的原始值对于所有实例都是相同的，而且你是在定义枚举的时候设置原始值。

例如，考虑从服务器获取日出和日落的时间。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):
    let serverResponse = "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，比如方法和构造器。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实现协议。

```
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。你可以使用扩展在别处修改定义，甚至是从外部库或者框架引入的一个类型，使得这个类型遵循某个协议。

```
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功能。

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，编译器会把它的类型当做ExampleProtocol。这表示你不能调用类在它实现的协议之外实现的方法或者属性。

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`来抛出一个错误并使用`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`。否则的话，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 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

在尖括号里写一个名字来创建一个泛型函数或者类型。

```
func repeatItem<Item>(repeating item: Item, numberOfTimes: Int) -> [Item] {
    var result = [Item]()
    for _ in 0..<numberOfTimes {
        result.append(item)
    }
    return result
}
repeatItem(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>`是等价的。

Swift 文档修订历史

1.0 翻译：成都老码团队翻译组-Arya 校对：成都老码团队翻译组-Oberyn changkun

1.1 翻译：成都老码团队翻译组-Arya 校对：成都老码团队翻译组-Oberyn changkun

1.2 翻译：成都老码团队翻译组-Arya 校对：成都老码团队翻译组-Oberyn changkun

2.0 翻译+校对：changkun

2.1 翻译+校对：changkun

2.2 翻译+校对：changkun

3.0 翻译+校对：shanks，2016-10-06

3.0.1 翻译+校对：shanks，2016-11-10

本页面根据 Document Revision History 进行适配更新。

本页内容包括： - Swift 3.1 更新 - Swift 3.0 更新 - Swift 2.2 更新 - Swift 2.1 更新 - Swift 2.0 更新 - Swift 1.2 更新 - Swift 1.1 更新 - Swift 1.0 更新

Swift 3.1 更新

发布日期	语法变更记录
2016-10-27	• 更新至 Swift 3.0.1。 • 更新自动引用计数章节中关于 weak 和 unowned 引用的讨论。 • 增加声明标识符章节中关于新的标识符`unowned`，`unowend(safe)`和`unowned(unsafe)`的描述。 • 增加Any 和 AnyObject 的类型转换一节中关于使用类型`Any`作为可选值的描述。 • 更新表达式章节，把括号表达式和元组表达式的描述分开。

Swift 3.0 更新

发布日期	语法变更记录
2016-09-13	• 更新至 Swift 3.0。 • 更新函数章节中关于函数的讨论，在函数定义一节中，标明所有函数参数默认都有函数标签。 • 更新高级操作符章节中关于操作符的讨论，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。 • 增加访问控制章节中关于对新的访问级别描述符`open`和`fileprivate`的信息 • 更新函数定义一节中关于`inout`的讨论，标明它放在参数类型的前面，替代之前放在参数名称前面的方式。 • 更新逃逸闭包和自动闭包还有属性章节中关于`@noescape`和`@autoclosure`的讨论，现在他们是类型属性，而不是定义属性。 • 增加高级操作符章节中自定义中缀操作符的优先级一节和定义章节中优先级组声明一节中关于操作符优先级组的信息。 • 更新一些讨论：使用 macOS 替换掉 OS X， Error 替换掉 ErrorProtocol，和更新一些协议名称，比如使用 ExpressibleByStringLiteral 替换掉 StringLiteralConvertible。 • 更新泛型章节中泛型 Where 语句一节和泛型形参和实参章节，现在泛型的 where 语句写在一个声明的最后。 • 更新逃逸闭包一节，现在闭包默认为非逃逸的(noescaping)。 • 更新基础部分章节中可选绑定一节和语句章节中While 语句一节，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。 • 增加控制流章节中Switch一节和语句章节中Switch 语句一节关于 switch cases 可以使用多模式的信息。 • 更新函数类型一节，现在函数参数标签不包含在函数类型中。 • 更新协议章节中协议组合一节和类型章节中协议组合类型一节关于使用新的 Protocol1 & Protocol2 语法的信息。 • 更新动态类型表达式一节使用新的 type(of) 表达式的信息。 • 更新行控制表达式一节使用 #sourceLocation(file:line:) 表达式的信息。 • 更新永不返回函数一节使用 新的 Never 类型的信息。 • 增加字面量表达式一节关于 playground 字面量的信息。 • 更新In-Out 参数一节，标明只有非逃逸闭包能捕获 in-out 参数。 • 更新默认参数值一节，现在默认参数不能在调用时候重新排序。 • 更新属性章节中关于属性参数使用分号的说明。 • 增加重新抛出函数和方法一节中关于在 catch 代码块中抛出错误的重新抛出函数的信息。 • 增加Selector 表达式一节中关于访问 Objective-C 中 Selector 的 getter 和 setter 的信息。 • 增加类型别名声明一节，标明函数类型作为参数类型必须使用括号包裹。 • 增加函数类型一节中关于泛型类型别名和在协议内使用类型别名的信息。 • 更新属性章节，标明 @IBAction，@IBOutlet 和 @NSManaged 隐式含有 @objc 属性。 • 增加声明属性一节中关于 @GKInspectable 的信息。 • 更新可选协议要求一节中关于只能在与 Objective-C 交互的代码中才能使用可选协议要求的信息。 • 删除函数声明一节中关于显式使用 let 关键字作为函数参数的信息。 • 删除语句一节中关于 Boolean 协议的信息， 现在这个协议已经被 Swift 标准库删除。 • 更正声明一节中关于 @NSApplicationMain 协议的信息。

Swift 2.2 更新

发布日期	语法变更记录
2016-03-21	• 更新至 Swift 2.2。 • 增加了编译配置语句一节中关于如何根据 Swift 版本进行条件编译。 • 增加了显示成员表达式一节中关于如何区分只有参数名不同的方法和构造器的信息。 • 增加了选择器表达式一节中针对 Objective-C 选择器的 `#selector` 语法。 • 更新了关联类型和协议关联类型声明，使用 `associatedtype` 关键词修改了对于关联类型的讨论。 • 更新了可失败构造器一节中关于当构造器在实例完全初始化之前返回 `nil`的相关信息。 • 增加了比较运算符一节中关于比较元组的信息。 • 增加了关键字和标点符号一节中关于使用关键字作为外部参数名的信息。 • 增加了声明特性一节中关于`@objc`特性的讨论，并指出枚举(Enumeration)和枚举用例(Enumaration Case)。 • 增加了操作符一节中对于自定义运算符的讨论包含了 . 。 • 增加了重新抛出错误的函数和方法一节中关于重新抛出错误函数不能直接抛出错误的笔记。 • 增加了属性观察器一节中关于当作为 in-out 参数传递属性时，属性观察器的调用行为。 • 增加了Swift 初见一节中关于错误处理的内容。 • 更新了弱引用一节中的图片用以更清楚的展示重新分配过程。 • 删除了 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。 • 删除了对变量函数参数和柯里化函数的特殊语法的讨论。

Swift 2.1 更新

发布日期	语法变更记录

发布日期	语法变更记录
2015-10-20	• 更新至 Swift 2.1。 • 更新了字符串插值(String Interpolation)和字符串字面量(String Literals)小节，现在字符串插值可包含字符串字面量。 • 增加了在非逃逸闭包(Nonescaping Closures)一节中关于 `@noescape` 属性的相关内容。 • 更新了声明特性(Declaration Attributes)和编译配置语句(Build Configuration Statement)小节中与 tvOS 相关的信息。 • 增加了 In-Out 参数(In-Out Parameters)小节中与 in-out 参数行为相关的信息。 • 增加了在捕获列表(Capture Lists)一节中关于指定闭包捕获列表被捕获时捕获值的相关内容。 • 更新了通过可选链式调用访问属性(Accessing Properties Through Optional Chaining)一节，阐明了如何通过可选链式调用进行赋值。 • 改进了自闭包(Autoclosure)一节中对自闭包的讨论。 • 在Swift 初见(A Swift Tour)一节中更新了一个使用`??`操作符的例子。

Swift 2.0 更新

发布日期	语法变更记录
2015-09-16	• 更新至 Swift 2.0。 • 增加了对于错误处理相关内容，包括 错误处理一章、Do 语句、Throw 语句、Defer 语句以及try 运算符 的多个小节。 • 更新了表示并抛出错误一节，现在所有类型均可遵循 `ErrorType` 协议。 • 增加了将错误转换成可选值一节 `try?` 关键字的相关信息。 • 增加了枚举一章的递归枚举一节和声明一章的任意类型用例的枚举一节中关于递归枚举的内容。 • 增加了控制流一章中a href="https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/ControlFlow.html#//apple_ref/doc/uid/TP40014097-CH9-ID523">检查 API 可用性一节和语句一章中可用性条件一节中关于 API 可用性检查的内容。 • 增加了控制流一章的早期退出一节和语句一章的guard语句中关于新 `guard` 语句的内容。 • 增加了协议一章中协议扩展一节中关于协议扩展的内容。 • 增加了访问控制一章中单元测试 target 的访问级别一节中关于单元测试的访问控制相关的内容。 • 增加了模式一章中可选模式一节中的新可选模式。 • 更新了 Repeat-While 一节中关于`repeat-while`循环的信息。 • 更新了字符串和字符一章，现在`String`在 Swift 标准库中不再遵循`CollectionType`协议。 • 增加了打印常量和变量一节中关于新 Swift 标准库中关于 `print(_:separator:terminator)` 的信息。 • 增加了枚举一章中原始值的隐式赋值一节和声明一章的包含原始值类型的枚举一节中关于包含`String`原始值的枚举用例的行为。 • 增加了自闭包一节中关于`@autoclosure`特性的相关信息，包括它的`@autoclosure(escaping)`形式。 • 更新了声明特性一节中关于`@avaliable`和`warn_unused_result`特性的相关内容。 • 更新了类型特性一节中关于`@convention`特性的相关信息。 • 增加了可选绑定一节中关于使用`where`子句进行多可选绑定的内容。 • 增加了字符串字面量一节中关于在编译时使用 `+` 运算符凭借字符串字面量的相关信息。 • 增加了元类型一节中关于元类型值的比较和使用它们通过构造器表达式构造实例。 • 增加了断言调试一节中关于用户定义断言是被警用的相关内容。 • 更新了声明特性一节中，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。 • 更新了可变参数一节，现在可变参数可以声明在函数参数列表的任意位置中。 • 增加了重写可失败构造器一节中，关于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。 • 增加了任意类型用例的枚举一节中关于枚举用例作为函数的内容。 • 增加了构造器表达式一节中关于显式引用一个构造器的内容。 • 更新了编译控制语句一节中关于编译信息以及行控制语句的相关信息。 • 更新了元类型一节中关于如何从元类型值中构造类实例。 • 更新了弱引用一节中关于弱引用作为缓存的显存的不足。 • 更新了类型特性一节，提到了存储型特性其实是懒加载。 • 更新了捕获类型一节，阐明了变量和常量在闭包中如何被捕获。 • 更新了声明特性一节用以描述如何在类中使用`@objc`关键字。 • 增加了错误处理一节中关于执行`throw`语句的性能的讨论。增加了 Do 语句一节中相似的信息。 • 更新了类型特性一节中关于类、结构体和枚举的存储型和计算型特性的信息。 • 更新了Break 语句一节中关于带标签的 break 语句。 • 更新了属性观察器一节，阐明了`willSet`和`didSet`观察器的行为。 • 增加了访问级一节中关于`private`作用域访问的相关信息。 • 增加了弱引用一节中关于若应用在垃圾回收系统和 ARC 之间的区别。 • 更新了字符串字面量中特殊字符一节中对 Unicode 标量更精确的定义。

Swift 1.2 更新

发布日期	语法变更记录
2015-4-8	• 更新至 Swift 1.2。 • Swift现在自身提供了一个`Set`集合类型，更多信息请看集合 • `@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`。更多信息，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，更多信息，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。更多信息，请看协议遵循性检查 • 增加了一个新的指导章节，它是关于字符串索引的 • 从溢出运算符中移除了溢出除运算符(&/)和求余溢出运算符(&%)。 • 更新了常量和常量属性在声明和构造时的规则，更多信息，请看常量声明 • 更新了字符串字面量中Unicode标量集的定义，请看字符串字面量中的特殊字符 • 更新了区间运算符章节来提示当半开区间运算符含有相同的起止索引时，其区间为空。 • 更新了闭包引用类型章节来澄清对于变量的捕获规则 • 更新了值溢出章节来澄清有符号整数和无符号整数的溢出行为 • 更新了协议声明章节来澄清协议声明时的作用域和成员 • 更新了捕获列表章节来澄清对于闭包捕获列表中的弱引用和无主引用的使用语法。 • 更新了运算符章节来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。更多的信息，请看常量声明 • 在构造器中，常量属性有且仅能被赋值一次。更多信息，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，更多信息，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。更多信息，请看类型转换运算符

Swift 1.1 更新

发布日期	语法变更记录