

Swift 概览

4.2 ▾

Swift 编程语言 / Swift 概览

这是一篇社区协同翻译的文章，你可以点击右边区块信息里的『改进』按钮向译者提交改进建议。

按照传统，新语言的第一个程序都应该是在屏幕上打印「Hello, world!」，在 Swift 中可以用这行代码来实现：

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

如果你以前写过 C 或 Objective-C，那你应该对 Swift 的语法不会感到陌生，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而去导入一个单独的库。全局作用域中的代码会自动作为程序的入口，因此不需要 `main()` 函数。同样的，你也不需要 在句尾写分号。

本教程会用 Swift 向您展示如何完成各种编程任务，从而为您提供足够的信息以开始编写 Swift 代码。

如果有不明白的地方，请不要担心 —— 这里介绍的内容在本书剩余部分会有更详细的说明。

注意

为了获得最佳体验，推荐使用 Xcode 的 Playground，Playground 允许你对代码进行编辑并实时看到运行结果。

[Download Playground](#)

简单值

用 `let` 声明常量，用 `var` 声明变量。常量在编译时不需要赋初值，但后续只能对它赋值一次。也就是说你可以用常量来命名一个值，这个值可以在多个地方使用。

```
var myVariable = 42
```

Infinity

翻译进度

16
分块数量

10
参与人数



v2panda 翻译于 5个月

0

重译

由 Aufree 审阅



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

常量或变量的类型必须和赋值的类型保持一致。但是，你不用明确的声明类型，因为编译器会根据你所创建的常量或变量来推断它们的类型。在上面的例子中，编译器推断

`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)，则可以在变量后声明类型，用冒号分割。

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

练习

创建一个 `Float` 类型且值为 `4` 的常量。

Swift 中变量或常量的值永远不会被隐式的转换成其它类型。如果你需要把一个值的类型转换为其它类型，则需要显式为实例指定类型：

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

练习

尝试删除代码最后一行的 `String` 。看看你会得到什么错误提示？

有一种更简单的方式能让值转换为字符串：把值写在括号中，在括号之前再加一个反斜杠（`\`）。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 `\()` 让一个浮点运算包含在一个字符串中，并且在该字符串中加入某个人的名

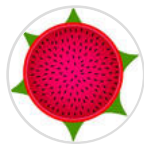


v2panda 翻译于 5个月前

👍 0

重译

由 Aufree 审阅



chai 翻译于 5个月前

👍 0

重译

由 Summer 审阅

字，跟他打下招呼。

对于占用多行的字符串请使用三个引号（`"""`）。每个引用行开头的缩进都要跟右引号的缩进相匹配。例如：

```
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，并且使用下标或者键来访问它们的元素。其中最后一个元素后面允许有逗号。

```
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]()
```

如果类型信息能够被推断出来，则可以用 `[]` 创建空数组，用 `[:]` 创建空字典 --- 就像你给一个变量赋值或给函数传递一个参数一样。

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

控制流

使用 `if` 和 `switch` 来创建条件语句，使用 `for` - `in`，`while`，以及 `repeat` - `while` 来创建循环语句。包裹条件或循环变量的括号是可选的。语句体的大括号是必不可缺的。

```
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，条件语句必须是布尔表达式---这意味着类似 `if score { ... }` 的代码将会报错，而不会与 0 做隐式的比较。

你可以一起使用 `if` 和 `let` 来处理值缺失的情况。这些值由可选值来代表。可选值要么包含一个值，要么为 `nil` 表示值缺失。在值的类型后面跟随一个（`?`）则表示这个值是可选的。

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

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

练习

更改 `optionalName` 为 `nil`。添加一个 `else` 子句来处理 `optionalName` 为 `nil` 的情况，看看 `greeting` 会等于什么？

如果可选值为 `nil`，条件语句值为 `false`，则大括号中的代码会被跳过。否则可选值将被解包，并分配给 `let` 后的常量，这样在代码块中就可以使用这个值。

处理可选值的另一种方法是使用 `'??'` 操作符来提供默认值。如果缺少可选值，则使用默认值。

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



VonZen 翻译于 5个月

👍 0

重译

由 Summer 审阅

'Switch' 语句支持任何类型的数据以及各种各样的比较操作——不仅仅局限于整数和测试相等。

```
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

尝试删除 default 语句，看看会得到什么错误？

注意在模式中是如何使用 `let` 将匹配值赋给常量的。

运行完 switch 语句中与 case 匹配的代码后，程序会直接从 switch 语句退出。因为下一个 case 语句不会被执行，因此没有必要在每个 case 语句后面显式的添加 break 来跳出执行。

你可以为字典中的键值对起一组名字，并用 `for` - `in` 语句来遍历字典。由于字典是无序的，所以它的遍历也是无序的。

```
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)
```



Espresso 翻译于 5个月

👍 0

重译

由 Summer 审阅

练习

添加另一个变量来记录最大值的类型（kind），同时仍然记录最大值。

使用 `while` 来重复执行一段代码，直至条件改变。循环条件可以放在循环结尾，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)
```

你可以使用 `..` 来限定索引范围，并在循环中遍历该索引范围。

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

使用 `..` 约束的范围不包括上界，使用 `...` 约束的范围包括上界。

函数和闭包

使用 `func` 来声明一个函数。使用函数名和参数来调用函数。使用 `->` 来指定函数返回值类型。

```
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。



Usan 翻译于 5个月前

👍 0

重译

由 Summer 审阅

查看其他 1 个版本

默认情况下，函数会使用它们的参数名称作为参数标签，在参数名称前可以自定义参数标签，或使用 `_` 来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")
```

使用元祖来生成复合值，例如使用元组来让一个函数返回多个值。该元组的元素可以通过名称或数字来获取。

```
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)
```

函数间可互相嵌套。被嵌套函数可以访问外部函数中声明的变量，你可以使用嵌套函数来重构一个过于冗长或复杂的函数。

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

```
}  
  
returnFifteen()
```

函数是一级类型。这意味着函数可以作为其它函数的返回值。

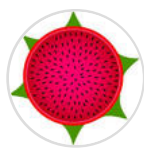
```
func makeIncrementer() -> ((Int) -> Int) {  
    func addOne(number: Int) -> Int {  
        return 1 + number  
    }  
    return addOne  
}  
  
var increment = makeIncrementer()  
increment(7)
```

一个函数也可以作为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)
```

函数其实是一种特殊的闭包：它是可以在之后被调用的一段代码。在闭包里的代码可以访问到闭包作用域范围内的变量和函数，即使闭包是在不同的作用域被执行---你已经在之前的嵌套函数见到过类似例子。你可以通过使用 `{}` 来创建一个匿名闭包。使用 `in` 将参数和返回值类型与闭包函数体分离。

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



chai 翻译于 5个月前

👍 0

重译

由 Summer 审阅

练习

重写上面的闭包，让它对所有的奇数返回 0。

写出更简洁的闭包有很多种方法。当我们已知一个闭包的类型，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方法中非常有用。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时，则可以完全忽略括号。

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

对象和类

通过在类名前加 `class` 关键字的方法来创建一个类。类中的属性声明和变量的属性声明相同，唯一不同的是，类的属性声明上下文是类。类似的，方法和函数也是用同样方式来声明。

```
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，并再添加一个带单个参数的方法。

通过在类名称后面插入括号来创建类的实例。使用 `.` 语法的方式来访问实例中的属性和方法。

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



liuzhanjing 翻译于 5个月

👍 0

重译

由 Summer 审阅

```
var shapeDescription = shape.simpleDescription()
```

这个版本的 `Shape` 类少了一些重要的东西：一个在类实例被创建时进行初始化的构造器。使用 `init` 来创建一个。

```
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，`self` 被用来区分 `name` 属性和构造器的 `name` 参数。当你创建类实例时，会像传入函数参数一样，给类传入构造器的参数。每个属性都要指定一个值 ---无论是在声明中（如 `numberOfSides`）还是在构造器里（如 `name`）。

如果你需要在对象被释放之前执行一些清理行为，可以用 `deinit` 来创建一个析构器。

子类会在其类名后面加上父类的名字，并用冒号分割。创建类的时候，并不需要一个标准根类，因此你可以根据自己需求，添加或省略父类的声明。

子类如果要重写父类的方法，则需要使用 `override` 来标记---不使用

`override` 关键字来标记会导致编译器报错。编译器同样也会检测 `override` 标记的方法是否存在父类当中。

```
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  
    }  
}
```



LeonaHui 翻译于 5个月

👍 0

重译

由 Aufree 审阅

```

        override fun 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

创建一个名为 `Circle` 的 `NamedShape` 子类，其构造器会接收半径和名称两个参数。在 `Circle` 类里实现一个 `area()` 和一个 `simpleDescription()` 方法。

除了存储简单的属性，属性还可以拥有 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 fun 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，新值被隐式的命名为 `newValue` 。你可以在 `set` 的括号后面，显式的提供一个名字。

注意 `EquilateralTriangle` 类的初始化有三个不同的步骤:

- 1. 设定子类声明的属性值。
- 2. 调用父类的构造器。
- 3. 改变父类定义的属性值。其它的工作如调用方法，`getter` 或 `setter` 都可以在这时候完成。

如果你不需要计算属性，但是仍需要在设置一个新值之前或之后来执行代码，则可以使用 `willSet` 和 `didSet` 。代码会在属性值发生改变时被执行，在构造器中属性值发生改变的情况除外。例如，下面的类确保三角形的边长始终和正方形的边长相同。

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

在处理可选值时，你可以在如方法、属性和下标脚本等操作之前使用 `?` 。如果 `?` 前的值是 `nil` ，则 `?` 后面的所有内容都会被忽略，且整个表达式的值为 `nil` 。否则，可选项的值将被展开，然后 `?` 后边的代码会根据展开的值来执行。在这两种情况中，整个表达式的值是一个可选值。



LeonaHui 翻译于 5个月

0

重译

由 Aufree 审阅

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

枚举和结构体

使用 `enum` 来创建枚举。像类和其它所有命名类型一样，枚举也包含方法。

```
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 从 0 开始给原始值赋值，而后依次递增 1，你也可以通过指定一个特定值来改变这一行为。在上边的例子当中，`Ace` 的原始值被显式赋值为 `1`，其余的原始值会按照顺序来赋值。同样的，你也可以使用字符串或者浮点数来作为枚举的原始值。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拥有一个原始值的枚举实例。如果在 `Rank` 中有与该原始值相匹配的枚举实例则返回该实例，没有则返回 `nil`。



LeonaHui 翻译于 5个月



0

重译

由 Aufree 审阅

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

枚举成员的值是实际值，而不是原始值的另一种写法。事实上，如果没有一个有意义的原始值，你也没必要再提供一个。

```
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()
```

练习

给 `Suit` 添加一个 `color()` 方法，为黑桃（spades）和梅花（clubs）则返回「black」，为红桃（hearts）和方块（diamonds）则返回「red」。

注意在上面例子中用了两种方法来调用 `hearts` 成员：给 `hearts` 指定一个常量时，枚举成员 `Suit.hearts` 需要全名调用，因为常量没有显式指定类型。在 `switch` 语句中，枚举成员可以通过缩写的方式 `.hearts` 被调用，因为 `self` 的值已经确定是 `Suit` 类型。在值的类型已经被明确的情况下可以使用缩写。

如果一个枚举成员拥有原始值，那么这些值在声明时就会被确定，也即是说，每个不同枚举实例的枚举成员总有一个相同的原始值。另一种选择是为枚举成员设定关联值---这些值会在实例被创建时确定，这样它们在每一个实例中的原始值就不一样了。你可以将关联值想象成与枚举实例存储属性一样。例如，考虑在服务器上请求日出和日落的情况。服务器要么返回请求信息，要么返回错误信息。



LeonaHui 翻译于 5个月

👍 0

重译

由 Aufree 审阅

```
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。

注意日出日落时间是如何从 `ServerResponse` 值中进行提取，并与 `switch cases` 相匹配的。

使用 `struct` 来创建一个结构体。结构体提供了很多和类相似的行为，包括方法和构造器。类和结构体最重要的区别就是结构体在传递的时候会拷贝自身，而类则会传递引用。

```
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，来创建一副扑克牌，并把每张牌的 `rank` 和 `suit` 对应起来。

协议与扩展

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

编写一个遵循这个协议的枚举。

注意声明 `SimpleStructure` 时使用了关键字 `mutating` 来标记一个可修改结构体的方法。而声明 `SimpleClass` 时，则不需要标记任何方法，因为一个类中的方法总是可以修改类属性的。

使用 `extension` 可以为现有类型添加功能，例如新方法和计算属性。你可以使用扩展将协议一致性添加到其他地方声明的类型，甚至是你从其它库或框架导入的类型。

```
extension Int: ExampleProtocol {  
    var simpleDescription: String {  
        return "The number \(self)"  
    }  
}
```



VonZen 翻译于 5个月

👍 0

重译

由 Aufree 审阅


```
}  
  
mutating func adjust() {  
    self += 42  
}  
  
}  
  
print(7.simpleDescription)
```

练习

为 `Double` 类型编写一个扩展，用于添加一个 `absoluteValue` 属性。

你可以像使用其它命名类型一样来使用协议---例如，创建一个具有不同类型但都遵循某一协议的对象集合。当你处理的类型为协议的值时，协议外定义的方法是不可用的。

```
let protocolValue: ExampleProtocol = a  
print(protocolValue.simpleDescription)  
// print(protocolValue.anotherProperty) // 取消注释看报什么错
```

尽管变量 `protocolValue` 在运行时类型为 `SimpleClass`，但编译器依旧会把它的类型当做 `ExampleProtocol`。这也就意味着，你不能随意访问在协议外的方法或属性。

错误处理

你可以使用任何遵循 `Error` 协议的类型来表示错误。

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

使用 `throw` 跑出异常并且用 `throws` 来标记一个可以抛出异常的函数。如果你在一个函数中抛出异常，这个函数会立即返回并且调用处理函数错误的代码。

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



LeonaHui 翻译于 5个月

👍 0

重译

由 Aufree 审阅

这里有几种方法可以处理异常。一种是使用 `do` - `catch` 。在 `do` 代码块里，你可以是用 `try` 在抛出的异常的函数前标记。在 `catch` 代码块里边，如果你不给定其他名字的话，错误会自动赋予名字为 `error` 。

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

练习

改变 `printer` 的名字为 `"Never Has Toner"` ，这样 `send(job:toPrinter:)` 就会抛出一个异常。

你可以提供多个 `catch` 代码块来处理特定的错误。你可以在 `catch` 后面一个模式，就像 `switch` 语句里面的 `case` 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)  
}
```

练习

加一些代码在 `do` 代码块里面抛出异常。你想要抛出什么样的异常才能让第一个 `catch` 代码块处理到？那第二个和第三个呢？

另外一种处理错误的方法是用 `try?` 去转换结果为可选项。如果这个函数抛出了异常，那么这个错误会被忽略并且结果为 `nil` 。否则，结果是一个包含了函数返回值的和选项。

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



LeonaHui 翻译于 5个月



0

重译

由 Aufree 审阅

使用 `defer` 来写在函数返回后也会被执行的代码块。无论这个函数是否抛出异常，这个代码都会被执行。即使他们需要再不同的时间段执行，你仍可以使用 `defer` 来简化代码。

```
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

把名字写在尖括号里来创建一个泛型方法或者类型。

```
func makeArray<Item>(repeating item: Item, numberOfTimes: Int) -> [Item] {
    var result = [Item]()
    for _ in 0..
```

你可以从函数和方法中，同时还有类，枚举以及结构体中创建泛型。

```
// 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)
```



LeonaHui 翻译于 5个月

👍 0

重译

由 Aufree 审阅

在类型名称后紧接 `where` 来明确一系列需求---例如，要求类型实现一个协议，要求两个类型必须相同，或者要求类必须继承来自特定的父类。

```
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(_:_:)` 函数来返回一个两个数组中共有元素的数组。

















写 `<T: Equatable>` 和 `<T> ... where T: Equatable` 是一回事。

本文中的所有译文仅用于学习和交流目的，转载请务必注明文章译者、出处、和本文链接

我们的翻译工作遵照 [CC 协议](#)，如果我们的工作有侵犯到您的权益，请及时联系我们。



👍 点赞





更多职位

讨论数量: 0

发起讨论

☐ 只看当前版本讨论

暂无话题~

兄弟社区



Laravel China



PythonCaff.com



GolangCaff.com



VuejsCaff.com

资源推荐

资源推荐

其他信息



软件外包



商务合作



联系站长

由 Summer 设计和编码 ❤