

函数

4.2 ▾

Swift 编程语言 / 函数

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。

函数是执行一个具体任务的一段独立代码块，你可以通过为函数命名来标识其任务功能，当需要执行这个任务时，函数名就可以用来「调用」该函数。

Swift 的统一的函数语法非常灵活，以致于从一个简单、无参数名的 C 风格函数，到一个复杂、多参数名的 Objective-C 风格方法都可以灵活表达。参数可以通过设置一个默认值，以简化函数的调用。也可以传递可修改参数，一旦函数完成执行，传递的参数值就会被修改。

Swift 中的函数类型由参数值类型和返回值类型共同组成。你可以像其它类型一样来使用这个类型，这样把一个函数做为参数传递给另一个函数就会非常容易，并且可以从其它函数来返回函数。另外，一个封装了具体功能的函数能直接嵌套在另一个函数的代码块中。

函数的定义和调用

在定义一个函数时，你可以可选地提供一个或多个输入值作为参数。当函数执行完成时，你也能可选地提供一个值作为返回值。

每一个函数的 函数名 描述了这个这个函数需要做的事情。你可以通过这个函数名去调用它并为它提供符合参数类型的参数值。函数的实参值必须按形参的参数列表顺序依次传入。

下面我们定义一个 greet(person:) 方法，它表示向一个人打招呼，它接受一个 String 类型的值做为输入并返回一个 String 类型的值。

```
func greet(person: String) -> String {
    let greeting = "Hello, " + person + "!"
    return greeting
}
```

100%

翻译进度

21

分块数量

2

参与人数



RecherJ 翻译于 5个月

0

重译

由 Aufree 审阅



RecherJ 翻译于 5个月

0

重译

由 Aufree 审阅



函数名的定义应该清晰地指明这个函数的功能，函数的函数名前面必须要加 `func` 关键字。函数 `->` (向右的箭头) 后面是返回值， `->` 后面跟返回值的类型。

上面方法的方法名描述了这个方法要做的事情、需要的参数值和当执行完成时返回的值。在其他地方调用时，这个方法清晰的表达了它的作用

```
print(greet(person: "Anna"))
// Prints "Hello, Anna!"

print(greet(person: "Brian"))
// Prints "Hello, Brian!"
```

调用 `greet(person:)` 时，传递一个 `String` 类型的人名作为值，例如这样 `greet(person: "Anna")` 。因为这个方法有一个返回值，所以像上面这样放在 `print()` 方法中，可以直接打印出执行完的结果。

注意：

`print(_:separator:terminator:)` 方法在调用时除第一个参数外，其他值可以不传，因为其他参数有默认值。函数语法变化的讨论在这里

[Function Argument Labels and Parameter Names](#)

[Default Parameter Values](#)

`greet(person:)` 方法定义了一个 `greeting` 变量，并设置了一条简单的打招呼信息。当函数执行完时，这个值通过函数的 `return` 关键字返回出去。

你可以多次调用 `greet(person:)` 方法通过传递不同值，上面例子展示了调用时分别传递 "Anna" 和 "Brian" 的情况。

为了简化函数体书写，我们可以把消息的创建和返回合并在一条语句中：

```
func greetAgain(person: String) -> String {
    return "Hello again, " + person + "!"
}

print(greetAgain(person: "Anna"))
// Prints "Hello again, Anna!"
```

函数的参数和返回值

在 `Swift` 中，函数的参数和返回值是非常灵活的。你能定义任何事 无论是一个单一参数的简单函数 还是有着多个参数和不同参数选项的复杂函数。



RecherJ 翻译于 5个月

0

重译

由 Aufree 审阅



RecherJ 翻译于 5个月

0

重译

由 Aufree 审阅

无参函数

函数可以没有参数。下面是一个没有参数的函数，当调用时，它总是返回同一个

`String` 类型的值：

```
func sayHelloWorld() -> String {
    return "hello, world"
}

print(sayHelloWorld())

// Prints "hello, world"
```

函数在定义时，即使函数没有任何参数，函数名后也需要跟 `()` 圆括号。调用时，函数名后面跟一对空的 `()` 圆括号。

多个参数

函数也可以有多个参数，这些参数写在函数名后面的 `()` 内，参数之间通过 `,` 逗号分隔。

下面方法接受一个 `String` 类型的人名值和是否已经打过招呼的 `Bool` 值作为输入，返回一个给这个人打招呼的信息：

```
func greet(person: String, alreadyGreeted: Bool) -> String {
    if alreadyGreeted {
        return greetAgain(person: person)
    } else {
        return greet(person: person)
    }
}

print(greet(person: "Tim", alreadyGreeted: true))

// Prints "Hello again, Tim!"
```

你能调用 `greet(person:alreadyGreeted:)` 函数 通过传递一个 `String` 类型的人名值 和一个 `Bool` 类型的值。用来和之前的 `greet(person:)` 函数做区分。尽管他们函数名都是 `greet`， 但一个接受一个参数，而另一个接受两个。

无返回值函数

函数也可以没有返回值。这个 `greet(person:)` 函数版本就没有返回值，而是将结果直接打印出来：



RecherJ 翻译于 5个月

👍 0 重译

由 Aufree 审阅



RecherJ 翻译于 5个月

👍 0 重译

由 Aufree 审阅

```
func greet(person: String) {  
    print("Hello, \ \(person)!")  
}  
  
greet(person: "Dave")  
  
// Prints "Hello, Dave!"
```

因为它不需要返回一个值，所以函数的定义可以不写 `->` 箭头及后面的返回值类型。

注意

严格意义来说，`greet(person:)` 函数 仍然 返回一个值，只是这个返回值没有被定义。函数返回值没有定义时，默认是返回 `Void` 类型。它是一个简单的空元祖，可以被写做 `()`。

调用函数时，它的返回值可以被忽略：

```
func printAndCount(string: String) -> Int {  
    print(string)  
    return string.count  
}  
  
func printWithoutCounting(string: String) {  
    let _ = printAndCount(string: string)  
}  
  
printAndCount(string: "hello, world")  
// prints "hello, world" and returns a value of 12  
  
printWithoutCounting(string: "hello, world")  
// prints "hello, world" but does not return a value
```

第一个函数 `printAndCount(string:)` 打印一个字符串，然后返回这个字符串的字符集数量。第二个函数 `printWithoutCounting(string:)` 调用第一个函数，忽略了第一个函数的返回值。所以当调用第二个函数时，信息仍然被第一个函数打印了，但第一个函数的返回值确没被使用。

注意

返回值可以被忽略，但函数的返回值还是需要接收。一个有返回值的函数的返回值不允许直接丢弃不接收，如果你尝试这样做，编译器将给你抛出错误。

多返回值函数



RecherJ 翻译于 5个月

👍 0

重译

由 Aufree 审阅



RecherJ 翻译于 5个月

👍 0

重译

由 Aufree 审阅

你可以用一个元祖类型包装多个值来作为一个函数的返回值。

下面这个 `minMax(array:)` 函数，它找出参数数组中的最大整数和最小整数：

```
func minMax(array: [Int]) -> (min: Int, max: Int) {
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..
```

这个 `minMax(array:)` 函数返回一个包含了两个整型 `Int` 的元祖。它们的键名分别是 `Max` 和 `Min`，所以你可以通过这个键名来访问这两个值。

`minMax(array:)` 方法先定义了两个变量 `currentMin` 和 `currentMax`，分别存储这数组中第一个元素。然后迭代数组，检查每一个值是否比最小值小或比最大值大，如果是则分别记录这个值。最后，找出的最小值和最大值被包装在一个元祖中返回。

因为这个元祖的成员值被作为函数返回值的一部分，最大值和最小值能通过点语法来直接被访问。

```
let bounds = minMax(array: [8, -6, 2, 109, 3, 71])
print("min is \(bounds.min) and max is \(bounds.max)")
// Prints "min is -6 and max is 109"
```

注意这个从函数返回的元祖的成员不需要在被指定键名，因为它们的键名已经被作为函数返回类型的一部分而指定。

可选元祖返回类型

如果从函数返回的元祖类型有可能为空，你可以用一个可选元祖来指示这个返回值可能为

`nil`。你可以在元祖返回类型后面加上一个 `?` 问号来表示返回值可能为空，例如 `(Int, Int)?` 或 `(String, Int, Bool)?`。

注意

一个可选的元祖类型例如 `(Int, Int)?` 和元祖值可选例如 `(Int?, Int?)` 是不



RecherJ 翻译于 5个月

👍 0

重译

由 Aufree 审阅



RecherJ 翻译于 5个月

👍 0

重译

由 Aufree 审阅

同的。对于前者是整个元祖可能为空，而后者则是元祖内每一个独立的元素可能为空。

上面这个 `minMax(array:)` 函数返回了一个包含两个 `Int` 值的元祖。然而，这个函数没有对传递进来的数组进行任何安全性的检查。如果这个数组为空， 这个 `minMax(array:)` 函数在尝试访问数组第一个元素时，将在运行时触发一个数组越界的错误。

为了处理空数组这种情况，将 `minMax(array:)` 方法的返回值标记为可选类型。如果数组为空，将返回 `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..<array.count] {
        if value < currentMin {
            currentMin = value
        } else if value > currentMax {
            currentMax = value
        }
    }
    return (currentMin, currentMax)
}
```

你也能用可选值绑定的方法来检查 `minMax(array:)` 方法是否返回一个有效值：

```
if let bounds = minMax(array: [8, -6, 2, 109, 3, 71]) {
    print("min is \(bounds.min) and max is \(bounds.max)")
}

// Prints "min is -6 and max is 109"
```

函数的参数标签和参数名

每一个参数都由一个 参数标签 和一个 参数名 构成。参数标签被用在调这个方法时；每一个参数标签写在参数的前面。参数名被用在函数的具体实现中。默认参数的参数标签可以不写，用参数名来作为参数标签。

```
func someFunction(firstParameterName: Int, secondParameterName: Int) {
    // 在函数体内， 变量 firstParameterName 和 变量 secondParameterName 所对应的
```



RecherJ 翻译于 5个月

👍 0

重译

由 Aufree 审阅



RecherJ 翻译于 5个月

👍 0

重译

由 Aufree 审阅


```
}  
  
someFunction(firstParameterName: 1, secondParameterName: 2)
```

所有的参数必须有一个唯一的名称。 尽管多个参数可以有相同的参数标签， 但唯一的参数标签将使你的代码可读性更好。

明确参数标签

参数标签在参数名前面， 通过一个空格 来分隔：

```
func someFunction(argumentLabel parameterName: Int) {  
    // 在函数体内，变量 parameterName 的值对应是参数传递进来的  
}
```

这个修改后的 `greet(person:)` 函数接收一个人的姓名和家乡名称，并返回一个 `String` 类型的 `greeting`：

```
func greet(person: String, from hometown: String) -> String {  
    return "Hello \$(person)! Glad you could visit from \$(hometown)."  
}  
  
print(greet(person: "Bill", from: "Cupertino"))  
  
// Prints "Hello Bill! Glad you could visit from Cupertino."
```

多使用参数标签可以使函数在被调用时表达更清晰，可读性更好。

参数标签省略

如果一个参数不需要参数标签，可以用下划线 `_` 来代替之前的参数标签。

```
func someFunction(_ firstParameterName: Int, secondParameterName: Int) {  
    // 在函数体中，变量 firstParameterName 和 secondParameterName 分别对应第一个和第二个参数  
}  
  
someFunction(1, secondParameterName: 2)
```

如果一个参数有一个明确的参数标签，在调用时这个标签 必须 被明确写明。

参数默认值

你可以给函数的任何参数提供一个默认值，通过写在参数类型后面。如果提供了默认值，你就可以在调用时省略给这个参数传值。



RecherJ 翻译于 5个月

👍 0

重译

由 Aufree 审阅

```
func someFunction(parameterWithoutDefault: Int, parameterWithDefault: Int = 12) {
    // 调用时如果你没有给第二个参数传值，那么变量 parameterWithDefault 的值默认就是 12
}

someFunction(parameterWithoutDefault: 3, parameterWithDefault: 6) // 变量 parameterWithDefault 的值是 6
someFunction(parameterWithoutDefault: 4) // 变量 parameterWithDefault 的值是 12
```

把没有默认值的参数放在有默认值的参数前面。 没有默认值的参数对函数更重要 --- 当调用时，把它们写在前面更容易区分具有部分相同参数的函数，无论默认参数是否被忽略。

可变参数

可变参数接受 0 个或多个具体相同类型的值。调用时，你可以用一个可变参数来代表这些有着不确定数量的多个参数。在参数类型后面跟上 3 个点 `...` 来表示参数的数量可变。

传入函数体内的可变参数可以被当做一个数组类型来使用。下面这个例子中，变量名 `numbers` 表示的一系列可变参数（每一个的类型为 `Double`）被合并成更合适的数组类型 `[Double]`。

函数 `arithmeticMean()` 为传入的一系列数字计算出平均值：

```
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)
// returns 3, 5 个数的平均值是 3

arithmeticMean(3, 8.25, 18.75)
// returns 10.0, 3 个数的平均值是 10.0
```

注意

一个函数至多只能有一个可变参数。

传入传出参数

函数参数默认是常量，不能直接修改其值。编译器会报错如果你尝试在函数体内修改传入参数的值。但如果你执意要修改这个参数值，并希望在函数执行完成后修改的值仍然有效，那么用 `传入传出参数` 来代替普通参数。



RecherJ 翻译于 5个月

👍 0

重译

由 Aufree 审阅

传入传出参数通过在参数类型前加上 `inout` 关键字来定义。传入传出参数可以有一个初始值，传入函数后值将被修改，在函数执行完传出后，这个变量的初始值就会被替换完成。更多有关传入传出参数的行为和编译器优化的详细讨论，请移步

In-Out Parameters.

传入传出参数只支持变量。常量或字面量将不被允许做为参数传递，因为它们都不能被修改。传值时，在参数名前面加上 `&` 符号，来表示它能在函数体内被修改。

注意

传入传出参数不能有默认值，并且可变参数也不能被标记 `inout`。

下面这个 `swapTwoInts(_:_:)` 函数，有两个参数名分别为 `a` 和 `b` 的传入传出参数：

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

`swapTwoInts(_:_:)` 函数简单地交换两个传入参数 `a` 和 `b` 的值。首先将参数 `a` 的值存储于临时变量 `temporaryA` 中，然后将 `b` 的值赋值给 `a`，最后将临时变量 `temporaryA` 的值再赋值给 `b`。

你可以通过传递两个 `Int` 类型的参数来调用 `swapTwoInts(_:_:)` 函数来交换彼此的值。需要注意的是，在调用 `swapTwoInts(_:_:)` 方法时，变量 `someInt` 和 `anotherInt` 需要加上 `&` 符号：

```
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"
```

上面这个例子中，即便变量 `someInt` 和 `anotherInt` 被定义在函数体外部，但通过参数传递，`swapTwoInts(_:_:)` 函数还是修改了彼此的初始值。

注意

在同一个函数中，传入传出参数和返回值不一定要同时存在。上面这个例子中，



RecherJ 翻译于 5个月

👍 0

重译

由 Aufree 审阅



RecherJ 翻译于 5个月

👍 0

重译

由 Aufree 审阅

`swapTwoInts` 函数没有返回值，但变量 `someInt` 和 `anotherInt` 的初始值仍然被修改了。传入传出参数为函数影响函数体外部的作用域提供了一种可选的方式。

函数类型

每个函数的具体 函数类型 由它的参数类型和返回类型共同决定。

举个例子：

```
func addTwoInts(_ a: Int, _ b: Int) -> Int {
    return a + b
}

func multiplyTwoInts(_ a: Int, _ b: Int) -> Int {
    return a * b
}
```

上面分别定义了两个叫 `addTwoInts` 和 `multiplyTwoInts` 的运算函数。分别接受两个 `Int` 类型的参数，经过各自合适地运算再返回一个 `Int` 值。

这两个函数的类型都是 `(Int, Int) -> Int`。你可以理解为：

「这个函数接收两个都是 `Int` 的值，返回一个 `Int` 类型的值。」

再例如，下面这个函数没有参数也没有返回值：

```
func printHelloWorld() {
    print("hello, world")
}
```

那么它的函数类型就是 `() -> Void`，你可以理解为没有参数返回值是空 `Void`。

使用函数类型

`Swift` 中，函数类型的使用和其他类型一样。举个例子，你可以定义一个函数类型的常量或变量，并给其赋一个函数类型的值：

```
var mathFunction: (Int, Int) -> Int = addTwoInts
```

你可以理解为：



RecherJ 翻译于 5个月

👍 0 重译

由 Aufree 审阅



RecherJ 翻译于 5个月

👍 0 重译

由 Aufree 审阅

『定义了一个函数类型的 `mathFunction` 变量，「它接收两个 `Int` 类型的参数，并返回一个 `Int` 值。」并把 `addTwoInts` 函数关联给这个变量。』

`mathFunction` 的类型和 `addTwoInts(_:_:)` 函数的类型相同，所以Swift 的类型检查器将允许这样的赋值。

现在，你就可以用 `mathFunction` 变量来调用 `addTwoInts(_:_:)` 函数了：

```
print("Result: \(mathFunction(2, 3))")
// 打印 "Result: 5"
```

有着相同类型的不同方法可以赋值给同一个变量：

```
mathFunction = multiplyTwoInts
print("Result: \(mathFunction(2, 3))")
// 打印 "Result: 6"
```

像其他类型一样，当然你给一个变量或常量赋一个函数类型的值时，`Swift` 将帮你自动推导出值的真实类型：

```
let anotherMathFunction = addTwoInts
// anotherMathFunction 被会自动推导成 `(Int, Int) -> Int` 类型
```

函数类型作为参数

你可以把 `(Int, Int) -> Int` 类型的函数作为一个参数传递给另一个函数。当这个函数被调用时，这使得具体实现逻辑被当做一个函数传递给了这个函数的调用者。

下面这个例子，打印 `math functions` 函数相加后的结果：

```
func printMathResult(_ mathFunction: (Int, Int) -> Int, _ a: Int, _ b: Int) {
    print("Result: \(mathFunction(a, b))")
}

printMathResult(addTwoInts, 3, 5)
// 打印 "Result: 8"
```

这个例子中，定义了一个 `printMathResult(_:_:_:)` 的函数，它接收三个参数。第一个参数是一个叫 `mathFunction` 的函数，其类型是 `(Int, Int) -> Int`。你可以为第一个参数传递一个类型是 `(Int, Int) -> Int` 的函数作为参数。第二个和第三个参数 `a` 和 `b` 都是 `Int` 类型。这两个变量被当做第一个函数参数的输入值传入给了第一个参数。



RecherJ 翻译于 5个月

👍 0 重译

由 Aufree 审阅

当调用 `printMathResult(_:_:_:)` 时，分别传递 `addTwoInts(_:_:)` 函数 和 另外两个值 `3` 和 `5`。`3` 和 `5` 被当做第一个函数的参数传递给它做了相加，最终打印结果 `8`。

函数 `printMathResult(_:_:_:)` 的作用是打印 `addTwoInts(_:_:)` 函数的返回值。它不关心传入函数的具体实现 — 只关心传入函数的正确类型。这使得 `printMathResult(_:_:_:)` 函数把一些具体的功能逻辑实现推给了它的调用者。

返回类型为函数类型

你可以把一个函数类型作为另一个函数的返回类型。在这个返回箭头后面 (`->`) 跟上你要返回的具体函数类型。

下面这个例子分别定义了两个 `stepForward(_:)` 和 `stepBackward(_:)` 简单的函数。`stepForward(_:)` 函数返回一个在其输入值上 +1 后的值，`stepBackward(_:)` 方法返回一个在其输入值上 -1 后的值，两个方法的函数类型都是 `(Int) -> Int`：

```
func stepForward(_ input: Int) -> Int {
    return input + 1
}

func stepBackward(_ input: Int) -> Int {
    return input - 1
}
```

`chooseStepFunction(backward:)` 函数的返回类型是 `(Int) -> Int`。该函数根据一个 `Bool` 类型值来判断是返回 `stepForward(_:)` 还是 `stepBackward(_:)` 函数：

```
func chooseStepFunction(backward: Bool) -> (Int) -> Int {
    return backward ? stepBackward : stepForward
}
```

现在你可以通过调用 `chooseStepFunction(backward:)` 并为其输入一个 `Bool` 类型的值来获得一个递增或递减的函数：

```
var currentValue = 3

let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)

// 变量 moveNearerToZero 现在引用着 stepBackward() 函数
```

在上面这个例子中，最终返回递增还是递减函数由 `currentValue` 变量的值来决定。变



RecherJ 翻译于 5个月

👍 0 重译

由 Aufree 审阅

量 `currentValue` 的初始值是 `3`，`currentValue > 0` 比较结果就为 `true`，所以调用 `chooseStepFunction(backward:)` 后返回 `stepBackward(_:)` 函数。常量 `moveNearerToZero` 存储着该函数的返回函数。

现在 `moveNearerToZero` 表示这个递减函数，从输入值递减至 0：

```
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，在本章中你遇到的这些函数例子都是 *全局函数*，它们被定义在全局作用域上。当然，你也可以在函数体内定义一个函数，来作为该函数的 *嵌套函数*。

虽然嵌套函数默认对函数体外部是透明的，但仍然可以被该函数调用。函数也可以通过返回其内部的嵌套函数来使这个被嵌套的函数在外部作用域可以被使用。

你可以重写上面的 `chooseStepFunction(backward:)` 函数，来返回和使用其内部的嵌套函数：

```
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` 变量引用着 `stepForward()` 函数
while currentValue != 0 {
    print("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
```



RecherJ 翻译于 5个月

👍 0

重译

由 Aufree 审阅

```
print("zero!")

// -4...
// -3...
// -2...
// -1...
// zero!
```

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。

← 上一篇

下一篇 →

👍 点赞



👥 参与译者：2



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

资源推荐

资源推荐

其他信息

-  Laravel China
-  PythonCaff.com
-  GolangCaff.com
-  VuejsCaff.com

-  软件外包
-  商务合作
-  联系站长