The Swift Programming Language in Chinese

https://github.com/letsswift

http://letsswift.com/

June 15, 2014

目录

1 Swift 中文教程（六）函数

函数是执行特定任务的代码自包含块。给定一个函数名称标识, 当执行其任务时就可以用这个标识来进行” 调用”。

Swift 的统一的功能语法足够灵活来表达任何东西，无论是甚至没有参数名称的简单的 C 风格的函数表达式，还是需要为每个本地参数和外部参数设置复杂名称的 Objective-C 语言风格的函数。参数提供默认值，以简化函数调用，并通过设置在输入输出参数，在函数执行完成时修改传递的变量。

Swift 中的每个函数都有一个类型，包括函数的参数类型和返回类型。您可以方便的使用此类型像任何其他类型一样, 这使得它很容易将函数作为参数传递给其他函数, 甚至从函数中返回函数类型。函数也可以写在其他函数中来封装一个嵌套函数用以范围内有用的功能。

1.1 函数的声明与调用

当你定义一个函数时，你可以为其定义一个或多个不同名称、类型值作为函数的输入（称为参数），当该函数完成时将传回输出定义的类型（称为作为它的返回类型）。

每一个函数都有一个函数名，用来描述了函数执行的任务。要使用一个函数的功能时，你通过使用它的名称进行 “调用”，并通过它的输入值（称为参数）来匹配函数的参数类型。一个函数的提供的参数必须始终以相同的顺序来作为函数参数列表。

例如在下面的例子中被调用的函数 greetingForPerson，像它描述的那样 — 它需要一个人的名字作为输入并返回一句问候给那个人。

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

所有这些信息都汇总到函数的定义中，并以 func 关键字为前缀。您指定的函数的返回类型是以箭头 -> （一个连字符后跟一个右尖括号）以及随后类型的名称作为返回的。

该定义描述了函数的作用是什么，它期望接收什么，以及当它完成返回的结果是什么。该定义很容易让该函数可以让你在代码的其他地方以清晰、明确的方式来调用:

```
println(sayHello("Anna"))
// prints "Hello, Anna!"
println(sayHello("Brian"))
// prints "Hello, Brian!"
```

通过括号内 String 类型参数值调用 sayHello 的函数，如的 sayHello(”Anna”)。由于该函数返回一个字符串值，sayHello 的可以被包裹在一个 println 函数调用中来打印字符串，看看它的返回值，如上图所示。

在 sayHello 的函数体开始定义了一个新的名为 greeting 的 String 常量，并将其设置加上 personName 个人姓名组成一句简单的问候消息。然后这个问候函数以关键字 return 来传回。只要问候函数被调用时，函数执行完毕是就会返回问候语的当前值。

你可以通过不同的输入值多次调用 sayHello 的函数。上面的例子显示了如果它以”Anna” 为输入值，以”Brian” 为输入值会发生什么。函数的返回在每种情况下都是量身定制的问候。

为了简化这个函数的主体，结合消息创建和 return 语句用一行来表示：

```
func sayHello(personName: String) -> String {
    return "Hello again, " + personName + "!"
}
println(sayHello("Anna"))
// prints "Hello again, Anna!"
```

1.2 函数的参数和返回值

在 swift 中函数的参数和返回值是非常具有灵活性的。你可以定义任何东西无论是一个简单的仅仅有一个未命名的参数的函数还是那种具有丰富的参数名称和不同的参数选项的复杂函数。

1.2.1 多输入参数

函数可以有多个输入参数，把他们写到函数的括号内，并用逗号加以分隔。下面这个函数设置了一个开始和结束索引的一个半开区间，用来计算在范围内有多少元素包含：

```
func halfOpenRangeLength(start: Int, end: Int) -> Int {
```

```
    return end - start
}
println(halfOpenRangeLength(1, 10))
// prints "9"
```

1.2.2 无参函数

函数并没有要求一定要定义的输入参数。下面就一个没有输入参数的函数，任何时候调用时它总是返回相同的字符串消息：

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

该函数的定义在函数的名称后还需要括号，即使它不带任何参数。当函数被调用时函数名称也要跟着一对空括号。

1.2.3 没有返回值的函数

函数也不需要定义一个返回类型。这里有一个版本的 sayHello 的函数，称为 waveGoodbye，它会输出自己的字符串值而不是函数返回：

```
func sayGoodbye(personName: String) {
    println("Goodbye, \(personName)!")
}
sayGoodbye("Dave")
// prints "Goodbye, Dave!"
```

因为它并不需要返回一个值，该函数的定义不包括返回箭头（- >）和返回类型。

> 提示严格地说，sayGoodbye 功能确实还返回一个值，即使没有返回值定义。函数没有定义返回类型但返回了一个 void 返回类型的特殊值。它是一个简直是空的元组，实际上零个元素的元组，可以写为 ()。当一个函数调用时它的返回值可以忽略不计：

```
func printAndCount(stringToPrint: String) -> Int {
    println(stringToPrint)
    return countElements(stringToPrint)
}
func printWithoutCounting(stringToPrint: String) {
    printAndCount(stringToPrint)
}
printAndCount("hello, world")
// prints "hello, world" and returns a value of 12
printWithoutCounting("hello, world")
// prints "hello, world" but does not return a value
```

第一个函数 printAndCount，打印了一个字符串，然后并以 Int 类型返回它的字符数。第二个函数 printWithoutCounting，调用的第一个函数，但忽略它的返回值。当第二函数被调用时，字符串消息由第一函数打印了回来，去没有使用其返回值。

> 提示返回值可以忽略不计，但对一个函数来说，它的返回值即便不使用还是一定会返回的。在函数体底部返回时与定义的返回类型的函数不能相容时，如果试图这样做将导致一个编译时错误。

1.2.4 多返回值函数

你可以使用一个元组类型作为函数的返回类型返回一个有多个值组成的一个复合作为返回值。

下面的例子定义了一个名为 count 函数，用它计来算字符串中基于标准的美式英语中设定使用的元音、辅音以及字符的数量：

func count(string: String) -> (vowels: Int, consonants: Int, others: Int) { var vowels = 0, consonants = 0, others = 0 for character in string { switch String(character).lowercaseString { case “a”, “e”, “i”, “o”, “u”: ++vowels case “b”, “c”, “d”, “f”, “g”, “h”, “j”, “k”, “l”, “m”, “n”, “p”, “q”, “r”, “s”, “t”, “v”, “w”, “x”, “y”, “z”: ++consonants default: ++others } } return (vowels, consonants, others) }

您可以使用此计数函数来对任意字符串进行字符计数，并检索统计总数的元组三个指定 Int 值：

```
let total = count("some arbitrary string!")
```

```
println("\(total.vowels) vowels and \(total.consonants) consonants")
// prints "6 vowels and 13 consonants"
```

需要注意的是在这一点上元组的成员不需要被命名在该该函数返回的元组中，因为他们的名字已经被指定为函数的返回类型的一部分。

1.3 函数参数名

所有上面的函数都为参数定义了参数名称:

```
func someFunction(parameterName: Int) {
    // function body goes here, and can use parameterName
    // to refer to the argument value for that parameter
}
```

然而，这些参数名的仅能在函数本身的主体内使用，在调用函数时，不能使用。这些类型的参数名称被称为本地的参数，因为它们只适用于函数体中使用。

1.3.1 外部参数名

有时当你调用一个函数将每个参数进行命名是非常有用的，以表明你传递给函数的每个参数的目的。

如果你希望用户函数调用你的函数时提供参数名称, 除了设置本地地的参数名称，也要为每个参数定义外部参数名称。你写一个外部参数名称在它所支持的本地参数名称之前, 之间用一个空格来分隔:

func someFunction(externalParameterName localParameterName: Int) { // function body goes here, and can use localParameterName // to refer to the argument value for that parameter }

> 注意如果您为参数提供一个外部参数名称，调用该函数时外部名称必须始终被使用。作为一个例子，考虑下面的函数，它通过插入他们之间的第三个”joiner” 字符串来连接两个字符串:

```
func join(s1: String, s2: String, joiner: String) -> String {
    return s1 + joiner + s2
}
```

当你调用这个函数，你传递给函数的三个字符串的目的就不是很清楚了：

```
join("hello", "world", ", ")
// returns "hello, world"
```

为了使这些字符串值的目的更为清晰, 为每个 join 函数参数定义外部参数名称:

```
func join(string s1: String, toString s2: String, withJoiner joiner: String)
    -> String {
    return s1 + joiner + s2
}
```

在这个版本的 join 函数中，第一个参数有一个外部名称 string 和一个本地名称 s1; 第二个参数有一个外部名称 toString 和一个本地名称 s2; 第三个参数有一个外部名称 withJoiner 和一个本地名称 joiner。

现在，您可以使用这些外部参数名称调用清楚明确的调用该函数：

```
join(string: "hello", toString: "world", withJoiner: ", ")
// returns "hello, world"
```

使用外部参数名称使 join 函数的第二个版本功能更富有表现力, 用户习惯使用 sentence-like 的方式, 同时还提供了一个可读的、意图明确的函数体。

> 注意考虑到使用外部参数名称的初衷就是为了在别人第一次阅读你的代码时并不知道你函数参数的目的是什么。但当函数调用时如果每个参数的目的是明确的和毫不含糊的，你并不需要指定外部参数名称。外部参数名称速记

如果你想为一个函数参数提供一个外部参数名，然而本地参数名已经使用了一个合适的名称了，你不需要为该参数写相同的两次名称。取而代之的是，写一次名字，并用一个 hash 符号（#）作为名称的前缀。这告诉 Swift 使用该名称同时作为本地参数名称和外部参数名称。

这个例子定义了一个名为 containsCharacter 的函数, 定义了两个参数的外部参数名称并通过放置一个散列标志在他们本地参数名称之前:

```
func containsCharacter(#string: String, #characterToFind: Character) -> Bool {
    for character in string {
        if character == characterToFind {
            return true
        }
    }
    return false
}
```

这个函数选择的参数名称清晰的、函数体极具可读性, 使的该函数被调用时没有歧义:

```
let containsAVee = containsCharacter(string: "aardvark", characterToFind: "v")
// containsAVee equals true, because "aardvark" contains a "v"
```

1.3.2 参数的默认值

可以为任何参数设定默认值来作为函数的定义的一部分。如果默认值已经定义，调用函数时就可以省略该参数的传值。

> 注意将使用默认值的参数放在函数的参数列表的末尾。这确保了所有调用函数的非默认参数使用相同的顺序, 并明确地表示在每种情况下相同的函数调用。这里有一个版本，是早期的 join 函数，并为参数 joiner 设置了默认值:

```
func join(string s1: String, toString s2: String,
        withJoiner joiner: String = " ") -> String {
    return s1 + joiner + s2
}
```

如果在 join 函数被调用时提供给 joiner 一个字符串值，该字符串是用来连接两个字符串，就跟以前一样:

```
join(string: "hello", toString: "world", withJoiner: "-")
// returns "hello-world"
```

但是，如果当函数被调用时提供了 joiner 的没有值，就会使用单个空格（” “）的默认值:

```
join(string: "hello", toString: "world")
// returns "hello world"
```

1.3.3 有默认值的外部名称参数

在大多数情况下, 为所有参数提供一个外部带有默认值的参数的名称是非常有用的 (因此要求)。这将确如果当函数被调用时提供的值时参数必须具有明确的目的。

为了使这个过程更容易，当你自己没有提供外部名称时，Swift 自动为所有参数定义了缺省的参数外部名称。自动外部名称与本地名称相同，就好像你在你的代码中的本地名称之前写了一个 hash 符号。

这里有一个早期 join 函数版本，它不为任何参数提供的外部名称，但仍然提供了 joiner 参数的默认值:

```
func join(s1: String, s2: String, joiner: String = " ") -> String {
    return s1 + joiner + s2
}
```

在这种情况下，Swift 自动为一个具有默认值的参数提供了外部参数名称。调用函数时，为使得参数的目的明确、毫不含糊，因此必须提供外部名称:

```
join("hello", "world", joiner: "-")
// returns "hello-world"
```

> 注意你可以通过编写一个下划线 (_) 有选择进行这种行为, 而不是一个明确的定义外部参数名称。然而，在适当情况下有默认值的外部名称参数总是优先被使用。

1.3.4 可变参数

一个可变参数的参数接受零个或多个指定类型的值。当函数被调用时，您可以使用一个可变参数的参数来指定该参数可以传递不同数量的输入值。写可变参数的参数时，需要参数的类型名称后加上点字符（...）。

传递一个可变参数的参数的值时，函数体中是以提供适当类型的数组的形式存在。例如, 一个可变参数的名称为 numbers 和类型为 Double...在函数体内就作为名为 numbers 类型为 Double[] 的常量数组。

下面的示例计算任意长度的数字的算术平均值（也称为平均）:

```
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 3.0, which is the arithmetic mean of these five numbers
arithmeticMean(3, 8, 19)
// returns 10.0, which is the arithmetic mean of these three numbers
```

> 注意函数可以最多有一个可变参数的参数，而且它必须出现在参数列表的最后以避免多参数函数调用时出现歧义。

如果函数有一个或多个参数使用默认值，并且还具有可变参数，将可变参数放在列表的最末尾的所有默认值的参数之后。常量参数和变量参数

函数参数的默认值都是常量。试图改变一个函数参数的值会让这个函数体内部产生一个编译时错误。这意味着您不能错误地改变参数的值。

但是，有时函数有一个参数的值的变量副本是非常有用的。您可以通过指定一个或多个参数作为变量参数，而不是避免在函数内部为自己定义一个新的变量。变量参数可以是变量而不是常量, 并给函数中新修改的参数的值的提供一个副本。

在参数名称前用关键字 var 定义变量参数:

```
func alignRight(var string: String, count: Int, pad: Character) -> String {
    let amountToPad = count - countElements(string)
    for _ in 1...amountToPad {
        string = pad + string
    }
    return string
}
let originalString = "hello"
let paddedString = alignRight(originalString, 10, "-")
// paddedString is equal to "-----hello"
// originalString is still equal to "hello"
```

这个例子定义了一个新函数叫做 alignRight, 它对准一个输入字符串，以一个较长的输出字符串。在左侧的空间中填充规定的字符。在该示例中，字符串”hello” 被转换为字符串”—hello”。

该 alignRight 函数把输入参数的字符串定义成了一个变量参数。这意味着字符串现在可以作为一个局部变量，用传入的字符串值初始化，并且可以在函数体中进行相应操作。

函数首先找出有多少字符需要被添加到左边让字符串以右对齐在整个字符串中。这个值存储在本地常量 amountToPad 中。该函数然后将填充字符的 amountToPad 个字符拷贝到现有的字符串的左边，并返回结果。整个过程使用字符串变量参数进行字符串操作。

> 注意一个变量参数的变化没有超出了每个调用函数, 所以对外部函数体是不可见的。变量参数只能存在于函数调用的生命周期里。

1.3.5 输入 -输出参数

可变参数，如上所述，只能在函数本身内改变。如果你想有一个函数来修改参数的值，并且想让这些变化要坚持在函数调用结束后，你就可以定义输入 -输出参数来代替。

通过在其参数定义的开始添加 inout 关键字写用来标明输入 -输出参数。一个在输入 -输出参数都有一个传递给函数的值，由函数修改后，并从函数返回来替换原来的值。

1.4 函数类型

每个函数都有一个特定的类型，包括参数类型和返回值类型，比如:

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

这个例子定义了两个简单的数学函数 addTwoInts 和 multiplyTwoInts。每个函数接受两个 int 参数，返回一个 int 值，执行相应的数学运算然后返回结果

这两个函数的类型是 (Int, Int)->Int 可以解释为:

这个函数类型它有两个 int 型的参数，并返回一个 int 类型的值

下面这个例子是一个不带任何参数和返回值的函数:

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

这个函数的类型是 ()->()，或者函数没有参数，返回 void。函数没有显式地指定返回类型，默认为 void，在 Swift 中相当于一个空元组，记为 ()。

1.4.1 使用函数类型

在 swift 中你可以像任何其他类型一样的使用函数类型。例如，你可以定义一个常量或变量为一个函数类型，并指定适当的函数给该变量:

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

可以解读为:

“定义一个名为 mathFunction 变量，该变量的类型为’ 一个函数，它接受两个 int 值，并返回一个 int 值。’ 设置这个新的变量来引用名为 addTwoInts 函数的功能。”

该 mathFunction 函数具有与 addTwoInts 函数相同类型的变量，所以这个赋值能通过 Swift 的类型检查。

现在你可以调用指定的函数名称为 mathFunction:

```
println("Result: \(mathFunction(2, 3))")
// prints "Result: 5"
```

不同的函数相同的匹配类型可以分配给相同的变量, 也同样的适用于非函数性类型:

```
mathFunction = multiplyTwoInts
println("Result: \(mathFunction(2, 3))")
// prints "Result: 6"
```

与其他类型一样, 你可以把它迅速定义成函数类型当你为常量或变量分配一个函数时:

```
let anotherMathFunction = addTwoInts
// anotherMathFunction is inferred to be of type (Int, Int) -> Int
```

1.4.2 函数类型的参数

可以使用一个函数类型，如 (Int, Int)->Int 作为另一个函数的参数类型。这使你预留了一个函数的某些方面的实现，让调用者调用函数时提供。

下面就以打印上面的数学函数的结果为例:

```
func printMathResult(mathFunction: (Int, Int) -> Int, a: Int, b: Int) {
    println("Result: \(mathFunction(a, b))")
}
printMathResult(addTwoInts, 3, 5)
// prints "Result: 8"
```

这个例子中定义了一个名为 printMathResult 函数，它有三个参数。第一个参数名为 mathFunction，类型为 (Int, Int)->Int。您可以传入符合条件的任何函数类型作为此函数的第一个参数。第二和第三个参数 a、b 都是 int 类型。被用作于提供数学函数的两个输入值。

当 printMathResult 被调用时，它传递 addTwoInt 函数，以及整数值 3 和 5。它使用 3 和 5，调用 addTwoInt 函数，并打印函数运行的结果 8。

printMathResult 的作用是调用一个适当类型的数学函数并打印相应结果。那是什么功能的实现其实并不重要，你只要给以正确的类型匹配就行。这使 printMathResult 以调用者类型安全的方式转换了函数的功能。

1.5 函数类型的返回值

可以使用一个函数类型作为另一个函数的返回类型。返回的函数 (->) 即你的返回箭头后，立即写一个完整的函数类型就做到这一点。

下面的例子定义了两个简单的函数，分别是 stepForward 和 stepBackward。stepForward 函数返回输入值自增 1，而 stepBackward 函数返回输入值自减 1。这两个函数都有一个相同的类型 (Int) -> Int:

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

这里有一个 chooseStepFunction 函数，它的返回类型是” 函数类型 (Int) -> Int”。chooseStepFunction 返回一个基于布尔参数的 stepBackward 或 stepForward 函数类型:

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

您现在可以使用 chooseStepFunction 选择一个函数, 可能是加一函数或另一个:

```
var currentValue = 3
let moveNearerToZero = chooseStepFunction(currentValue > 0)
// moveNearerToZero now refers to the stepBackward() function
```

□□□□□□□□□□□□□□□□□□□□□□□□currentValue□□□□□□□□currentV

□□moveNearerToZero□□□□□□□□,□□□□□□□□□:

```
println("Counting to zero:")
// Counting to zero:
while currentValue != 0 {
    println("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
println("zero!")
// 3...
// 2...
// 1...
// zero!
```

1.6 嵌套函数

迄今为止所有你在本章中遇到函数都是全局函数，在全局范围内定义。其实你还可以在其他函数中定义函数，被称为嵌套函数。

嵌套函数默认对外界是隐藏的，但仍然可以调用和使用其内部的函数。内部函数也可以返回一个嵌套函数，允许在嵌套函数内的另一个范围内使用。

你可以重写上面的 chooseStepFunction 例子使用并返回嵌套函数：

```
func chooseStepFunction(backwards: Bool) -> (Int) -> Int {
    func stepForward(input: Int) -> Int { return input + 1 }
    func stepBackward(input: Int) -> Int { return input - 1 }
    return backwards ? stepBackward : stepForward
}
var currentValue = -4
let moveNearerToZero = chooseStepFunction(currentValue > 0)
// moveNearerToZero now refers to the nested stepForward() function
while currentValue != 0 {
    println("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
println("zero!")
// -4...
// -3...
// -2...
// -1...
// zero!
```

本文部分内容来源于CocoaChina的翻译小组，感谢他们的辛勤付出 ~

参考文献