函数

快速检索 [点击展开]

*函数*是一个独立的代码块，用来执行特定的任务。通过给函数一个名字来定义它的功能，并且在需要的时候，通过这个名字来“调用”函数执行它的任务。

Swift 统一的函数语法十分灵活，可以表达从简单的无形式参数的 C 风格函数到复杂的每一个形式参数都带有局部和外部形式参数名的 Objective-C 风格方法的任何内容。形式参数能提供一个默认的值来简化函数的调用，也可以被当作输入输出形式参数被传递，它在函数执行完成时修改传递来的变量。

Swift 中的每一个函数都有类型，由函数的形式参数类型和返回类型组成。你可以像 Swift 中其他类型那样来使用它，这使得你能够方便的将一个函数当作一个形式参数传递到另外的一个函数中，也可以在一个函数中返回另一个函数。函数同时也可以写在其他函数内部来在内嵌范围封装有用的功能。

定义和调用函数

当你定义了一个函数的时候，你可以选择定义一个或者多个命名的分类的值作为函数的输入（所谓的*形式参数*），并且/或者定义函数完成后将要传回作为输出的值的类型（所谓它的*返回类型*）。

每一个函数都有一个*函数名*，它描述函数执行的任务。要使用一个函数，你可以通过“调用”函数的名字并且传入一个符合函数形式参数类型的输入值（所谓*实际参数*）来调用这个函数。给函数提供的实际参数的顺序必须符合函数的形式参数列表顺序。

下边示例中的函数叫做 `greet(person:)` ，跟它的功能一致——它接收一个人的名字作为输入然后返回对这个人的问候。要完成它，你需要定义一个输入形式参数——一个叫做 `person` 的 `String` 类型值——并且返回一个 `String` 类型，它将会包含对这个人的问候：

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

这些信息都被包含在了函数的*定义*中，它使用一个 `func` 的关键字前缀。你可以用一个*返回箭头* `->` (一个连字符后面跟一个向右的尖括号)来明确函数返回的类型。

定义描述了函数会做什么，接收什么和它结束的时候会返回什么。定义能够帮助你更容易的从你代码的其他地方准确的调用到函数：

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

你可以通过在 `person` 标签后边给函数 `greet(person:)` 传入一个用圆括号包裹的 `String` 实际参数值来调用它，例如 `greet(person: "Anna")` 。如同上边示例中展示的一样，因为函数返回一个 `String` 值，所以 `greet(person:)` 可以包裹在 `print(_:separator:terminator:)` 函数中来打印字符串并查看它的返回值。

> 注意
>
> 函数 `print(_:separator:terminator:)` 的第一个实际参数并没有标签，并且它的其他实际参数是可选的，是因为他们都有默认值。这些函数语法的变化在下边函数实际参数标签和形式参数名以及默认形式参数值（此处应有链接）小节中讨论。

函数 `greet(person:)` 的主体从定义一个新的叫做 `greeting` 的 `String` 常量开始，它被设置成简单问候信息。之后这个问候被 `return` 关键字传

递出函数。一旦执行到 `return greeting` 这句代码，函数就会结束执行并返回 `greeting` 的当前值。

你可以多次调用 `greet(person:)` 并给它传入不同的值，上面的栗子演示了我们用 `"Anna"` 值和 `"Brian"` 值作为输入值调用函数会发生什么。函数为每种情况定制了一个问候语。

为了简化这个函数的主体，我们将创建信息和返回语句组合到一行：

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

函数的形式参数和返回值

在 Swift 中，函数的形式参数和返回值非常灵活。你可以定义从一个简单的只有一个未命名形式参数的工具函数到那种具有形象的参数名称和不同的形式参数选项的复杂函数之间的任何函数。

无形式参数的函数

函数没有要求必须输入一个参数，这里有一个没有输入形式参数的函数，无论何时它被调用永远会返回相同的 `String` 信息：

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

函数的定义仍然需要在名字后边加一个圆括号，即使它不接受形式参数

也得这样做。当函数被调用的时候也要在函数的名字后边加一个空的圆括号。

多形式参数的函数

函数可以输入多个形式参数，可以写在函数后边的圆括号内，用逗号分隔。

这个函数以一个人的名字以及是否被问候过为输入，并返回对这个人的相应的问候：

```
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!"
```

通过在圆括号中传入带有 `person` 标签的 `String` 实际参数值和带有 `alreadyGreeted` 标签的 `Bool` 实际参数值来调用 `greet(person:alreadyGreeted:)` 这个函数，实际参数之间用逗号分隔。注意这个函数与之前展示的函数 `greet(person:)` 是明显不同的。尽管两个函数都叫做 `greet` ， `greet(person:alreadyGreeted:)` 接收两个实际参数但 `greet(person:)` 函数只接收一个。

无返回值的函数

函数定义中没有要求必须有一个返回类型。下面是另一个版本的 `greet(person:)` 函数，叫做 `sayGoodbye(_:)` ，它将自己的 `String` 值打印了出来而不是返回它：

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

正因为它不需要返回值，函数在定义的时候就没有包含返回箭头（ `->` ）或者返回类型。

> 注意
>
> 严格来讲，函数 `greet(person:)` 还是有一个返回值的，尽管没有定义返回值。没有定义返回类型的函数实际上会返回一个特殊的类型 `Void` 。它其实是一个空的元组，作用相当于没有元素的元组，可以写作 `()` 。

当函数被调用时，函数的返回值可以被忽略：

```
func printAndCount(string: String) -> Int {
    print(string)
    return string.characters.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
```

在第一个函数 `printAndCount(_:)` 中，打印了一个字符串，然后返回了一个 `Int` 类型的字符数统计。在第二个函数 `printWithoutCounting` 中，调用了第一个函数，但却忽略了它的返回值。当调用第二个函数时，第一个函数仍然会打印出信息，但是返回的值却没有被使用。

> 注意

> 返回值可以被忽略，但是如果一个函数需要返回值的时候就必须返回。如果一个函数有定义的返回类型，没有返回值的话就不会继续运行到函数的末尾，尝试这么做的话会得到编译时错误。

多返回值的函数

为了让函数返回多个值作为一个复合的返回值，你可以使用元组类型作为返回类型。

下面的栗子定义了一个叫做 `minMax(array:)` 的函数，它可以找到类型为 `Int` 的数组中最大数字和最小数字。

```
func minMax(array: [Int]) -> (min: Int, max: Int) {
    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)
}
```

函数 `minMax(array:)` 返回了一个包含两个 `Int` 值的元组。这两个值被 `min` 和 `max` 标记，这样当查询函数返回值的时候就可以通过名字访问了。

函数 `minMax(array:)` 的主体在给数组中的第一个整数的值设置两个名为 `currentMin` 和 `currentMax` 的操作变量的时候开始。然后函数遍历数组中剩下的值，并检查它们会不会比 `currentMin` 小或者比 `currentMax` 大。最后最终的最大值和最小值被当作一个包含两个 `Int` 值的元组被返回。

因为元组的成员值在函数的返回类型部分被命名，所以它们可以通过使用点语法取出最大值和最小值：

```
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"
```

需要注意的是，元组的成员值不必在函数返回元组的时候命名，因为它们的名字早已经在函数的返回类型部分被明确。

可选元组返回类型

如果元组在函数的返回类型中有可能“没有值”，你可以用一个*可选*元组返回类型来说明整个元组的可能是 `nil` 。书法是在可选元组类型的圆括号后边添加一个问号（ `?` ）例如 `(Int, Int)?` 或者 `(String, Int, Bool)?` 。

> 注意
>
> 类似 `(Int, Int)?` 的可选元组类型和包含可选类型的元组 `(Int?, Int?)` 是不同的。对于可选元组类型，整个元组是可选的，而不仅仅是元组里边的单个值。

上面的函数 `minMax(array:)` 返回了一个包含两个 `Int` 值的元组。总之，函数不会对传入的数组进行安全性检查。如果 `array` 的实际参数包含了一个空的数组，上面定义的函数 `minMax(array:)` 在尝试调用 `array[0]` 的时候就会触发一个运行时错误。

为了安全的处理这种“空数组”的情景，就需要把 `minMax(array:)` 函数的返回类型写做可选元组，当数组是空的时候返回一个 `nil` 值：

```
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)
}
```

你可以利用可选项绑定去检查这个版本的 `minMax(array:)` 函数返回了一个实际的元组值还是是 `nil` 。

```
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"
```

函数实际参数标签和形式参数名

每一个函数的形式参数都包含*实际参数标签*和*形式参数名*。实际参数标签用在调用函数的时候；在调用函数的时候每一个实际参数前边都要写实际参数标签。形式参数名用在函数的实现当中。默认情况下，形式参数使用它们的形式参数名作为实际参数标签。

```
func someFunction(firstParameterName: Int, secondParameterName: Int) {
    // In the function body, firstParameterName and secondParameterName
    // refer to the argument values for the first and second parameters.
}
someFunction(firstParameterName: 1, secondParameterName: 2)
```

左右的形式参数必须有唯一的名字。尽管有可能多个形式参数拥有相同

的实际参数标签，唯一的实际参数标签有助于让你的代码更加易读。

指定实际参数标签

在提供形式参数名之前写实际参数标签，用空格分隔：

```
func someFunction(argumentLabel parameterName: Int) {
    // In the function body, parameterName refers to the argument value
    // for that parameter.
}
```

> 注意
>
> 如果你为一个形式参数提供了实际参数标签，那么这个实际参数就必须在调用函数的时候使用标签。

这里有另一个函数 `greet(person:)` 的版本，接收一个人名字和家乡然后返回对这个的问候：

```
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."
```

实际参数标签的使用能够让函数的调用更加明确，更像是自然语句，同时还能提供更可读的函数体并更清晰地表达你的意图。

省略实际参数标签

如果对于函数的形式参数不想使用实际参数标签的话，可以利用下划线（ _ ）来为这个形式参数代替显式的实际参数标签。

```
func someFunction(_ firstParameterName: Int, secondParameterName: Int) {
    // In the function body, firstParameterName and secondParameterName
    // refer to the argument values for the first and second parameters.
}
someFunction(1, secondParameterName: 2)
```

默认形式参数值

你可以通过在形式参数类型后给形式参数赋一个值来给函数的任意形式参数定义一个*默认值*。如果定义了默认值，你就可以在调用函数时候省略这个形式参数。

```
func someFunction(parameterWithDefault: Int = 12) {
    // In the function body, if no arguments are passed to the function
    // call, the value of parameterWithDefault is 12.
}
someFunction(parameterWithDefault: 6) // parameterWithDefault is 6
someFunction() // parameterWithDefault is 12
```

把不带有默认值的形式参数放到函数的形式参数列表中带有默认值的形式参数前边，不带有默认值的形式参数通常对函数有着重要的意义——把它们写在前边可以便于让人们看出来无论是否省略带默认值的形式参数，调用的都是同一个函数。

可变形式参数

一个*可变形式参数*可以接受零或者多个特定类型的值。当调用函数的时候你可以利用可变形式参数来声明形式参数可以被传入值的数量是可变的。可以通过在形式参数的类型名称后边插入三个点符号（ `...` ）来书写可变形式参数。

传入到可变参数中的值在函数的主体中被当作是对应类型的数组。举个栗子，一个可变参数的名字是 `numbers` 类型是 `Double...` 在函数的主体中它会被当作名字是 `numbers` 类型是 `[Double]` 的常量数组。

下面的栗子计算了一组任意长度的数字的算术平均值（也叫做平均数）。

```
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, 8.25, 18.75)
// returns 10.0, which is the arithmetic mean of these three numbers
```

> 注意
> 一个函数最多只能有一个可变形式参数。

输入输出形式参数

就像上面描述的，可变形式参数只能在函数的内部做改变。如果你想函数能够修改一个形式参数的值，而且你想这些改变在函数结束之后依然生效，那么就需要将形式参数定义为*输入输出形式参数*。

在形式参数定义开始的时候在前边添加一个 `inout` 关键字可以定义一个输入输出形式参数。输入输出形式参数有一个能*输入*给函数的值，函数能对其进行修改，还能*输出*到函数外边替换原来的值。

你只能把变量作为输入输出形式参数的实际参数。你不能用常量或者字面量作为实际参数，因为常量和字面量不能修改。在将变量作为实际参数传递给输入输出形式参数的时候，直接在它前边添加一个和符合（ `&` ）来明确可以被函数修改。

> 注意
>
> 输入输出形式参数不能有默认值，可变形式参数不能标记为 `inout` ，如果你给一个形式参数标记了 `inout` ，那么它们也不能标记 `var` 和 `let` 了。

这里有一个 `swapTwoInts(_:_:)` 函数，它有两个输入输出整数形式参数 `a` 和 `b` ：

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

函数 `swapTwoInts(_:_:)` 只是简单的将 `b` 和 `a` 的值进行了调换。函数将 `a` 的值储存在临时常量 `temporaryA` 中，将 `b` 的值赋给 `a` ，然后再将 `temporaryA` 的值赋给 `b` 。

你可以通过两个 `Int` 类型的变量来调用函数 `swapTwoInts(_:_:)` 去调换它们两个的值，需要注意的是 `someInt` 的值和 `anotherInt` 的值在传入函数 `swapTwoInts(_:_:)` 时都添加了和符号。

```
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
print("someInt is now \(someInt), and anotherInt is now \(a
notherInt)")
```

```
// prints "someInt is now 107, and anotherInt is now 3"
```

上边的栗子显示了 `someInt` 和 `anotherInt` 的原始值即使是在函数的外部定义的，也可被函数 `swapTwoInts(_:_:)` 修改。

> 注意
>
> 输入输出形式参数与函数的返回值不同。上边的 `swapTwoInts` 没有定义返回类型和返回值，但它仍然能修改 `someInt` 和 `anotherInt` 的值。输入输出形式参数是函数能影响到函数范围外的另一种替代方式。

函数类型

每一个函数都有一个特定的*函数类型*，它由形式参数类型，返回类型组成。

举个栗子：

```
1 func addTwoInts(_ a: Int, _ b: Int) -> Int {
2     return a + b
3 }
4 func multiplyTwoInts(_ a: Int, _ b: Int) -> Int {
5     return a * b
6 }
```

上边的栗子定义了两个简单的数学函数 `addTwoInts` 和 `multiplyTwoInts` 。这两个函数每个传入两个 `Int` 值，返回一个 `Int` 值，就是函数经过数学运算得出的结果。

这两个函数的类型都是 `(Int, Int) -> Int` 。也读作：
“有两个形式参数的函数类型，它们都是 `Int` 类型，并且返回一个 `Int` 类型的值。”

下边的另外一个栗子，一个没有形式参数和返回值的函数。

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

这个函数的类型是 `() -> Void`，或者“一个没有形式参数的函数，返回 `Void`。”

使用函数类型

你可以像使用 Swift 中的其他类型一样使用函数类型。例如，你可以给一个常量或变量定义一个函数类型，并且为变量指定一个相应的函数。

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

这可以读作：

“定义一个叫做 `mathFunction` 的变量，它的类型是‘一个能接受两个 `Int` 值的函数，并返回一个 `Int` 值。’将这个新的变量指向 `addTwoInts` 函数。”

这个 `addTwoInts(_:_:)` 函数和 `mathFunction` 函数有相同的类型，所以这个赋值是可以通过 Swift 的类型检查的。

你可以利用名字 `mathFunction` 来调用指定的函数。

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

不同的函数如果有相同的匹配的类型的话，就可以指定相同的变量，和非函数的类型一样：

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

和其他的类型一样，当你指定一个函数为常量或者变量的时候，可以将它留给 Swift 来对类型进行推断：

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

函数类型作为形式参数类型

你可以利用使用一个函数的类型例如 `(Int, Int) -> Int` 作为其他函数的形式参数类型。这允许你预留函数的部分实现从而让函数的调用者在调用函数的时候提供。

下面的栗子打印出了上文中数学函数执行后的结果：

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

这个栗子定义了一个叫做 `printMathResult(_:_:_:)` 的函数，它有三个形式参数。第一个形式参数叫做 `mathFunction`，类型是 `(Int, Int) -> Int`。你可以传入任何这个类型的函数作为第一个形式参数的实例参数。第二个和第三个形式参数叫做 `a` 和 `b`，它们都是 `Int` 类型。它们被用作提供的数学函数的两个传入值。

但函数 `printMathResult(_:_:_:)` 被调用的时候，它传入了 `addTwoInts(_:_:)` 函数和两个整数值 `3` 和 `5`。它利用 `3` 和 `5` 的值调用了提供的函数，打印出了结果 `8`。

函数 `printMathResult(_:_:_:)` 的作用就是当调用一个相应类型的数学函数的时候打印出结果。它并不关心函数在实现过程中究竟做了些什

么，它只关心函数是不是正确的类型。这使得函数 `printMathResult(_:_:_:)` 以一种类型安全的方式把自身的功能传递给调用者。

函数类型作为返回类型

你可以利用函数的类型作为另一个函数的返回类型。写法是在函数的返回箭头（ `->` ）后立即写一个完整的函数类型。

下边的栗子定义了两个简单函数叫做 `stepForward(_:)` 和 `stepBackward(_:)` 。函数 `stepForward(_:)` 返回一个大于输入值的值，而 `stepBackward(_:)` 返回一个小于输入值的值。这两个函数的类型都是 `(Int) -> Int` ：

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

这里有一个函数 `chooseStepFunction(backward:)` ，它的返回类型是“一个函数的类型 `(Int) -> Int` ”。函数 `chooseStepFunction(backward:)` 返回了 `stepForward(_:)` 函数或者一个基于叫做 `backwards` 的布尔量形式参数的函数 `stepBackward(_:)` ：

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

现在你可以用 `chooseStepFunction(backward:)` 来得到一个向某方向前进或者其他的函数：

```
var currentValue = 3
```

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

上面的栗子显示了使变量趋近于零需要一个整数还是一个负数。`currentValue` 有一个 `3` 的初始值，也就是说 `currentValue > 0` 返回 `true`，导致 `chooseStepFunction(backward:)` 返回 `stepBackward(_:)` 函数。一个返回函数的引用存储在名为 `moveNearerToZero` 的常量里。

现在这个 `moveNearerToZero` 指向了正确的函数，它可以用来进行到零的计算了：

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

内嵌函数

到目前为止，你在本章中遇到的所有函数都是*全局函数*，都是在全局的范围内进行定义的。你也可以在函数的内部定义另外一个函数。这就是*内嵌函数*。

内嵌函数在默认情况下在外部是被隐藏起来的，但却仍然可以通过包裹它们的函数来调用它们。包裹的函数也可以返回它内部的一个内嵌函数来在另外的范围里使用。

你可以重写上边的栗子 `chooseStepFunction(backward:)` 来使用和返回内嵌函数：

```
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: curren
tValue > 0)
// moveNearerToZero now refers to the nested stepForward()
function
while currentValue != 0 {
    print("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
print("zero!")
// -4...
// -3...
// -2...
// -1...
// zero!
```