功能

*函数*是执行特定任务的自包含代码块。你给一个函数一个名字来标识它做了什么，这个名字用来“调用”函数在需要时执行它的任务。

Swift的统一函数语法足够灵活，可以将任何内容从一个没有参数名称的简单C风格函数转换为一个复杂的Objective-C风格方法，每个参数都有名称和参数标签。参数可以提供默认值以简化函数调用，并且可以作为输入参数传递，这些参数在函数完成执行后修改传递的变量。

Swift中的每个函数都有一个类型，由函数的参数类型和返回类型组成。你可以像Swift中的任何其他类型一样使用这种类型，这使得将函数作为参数传递给其他函数以及从函数返回函数变得容易。也可以在其他函数中编写函数，以在嵌套的函数范围内封装有用的功能。

定义和调用函数

当你定义一个函数时，你可以选择定义一个或多个命名的类型值作为输入函数，称为*参数*。你也可以选择定义一个函数在完成时作为输出*返回的值的类型*，称为它的*返回类型*。

每个函数都有一个*函数名称*，它描述了函数执行的任务。要使用某个函数，需要使用其名称“调用”该函数，并传递与函数参数类型相匹配的输入值（称为*参数*）。函数的参数必须始终以与函数的参数列表相同的顺序提供。

下面的例子中的函数被调用`greet(person:)`，因为它就是这样做的 - 它将一个人的名字作为输入并返回该人的问候语。要做到这一点，你需要定义一个输入参数 - 一个`String`名为`person`- 的值和一个返回类型`String`，它将包含该人的问候语：

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

所有这些信息都汇总到函数的*定义中*，该*定义*以`func`关键字为前缀。使用*返回箭头* `->`（连字符后跟右括号）指示函数的返回类型，紧接着是要返回的类型的名称。

该定义描述了函数的功能，期望得到的内容以及完成后返回的内容。该定义使您可以轻松地从代码中的其他位置明确调用该函数：

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

你`greet(person:)`可以通过`String`在`person`参数标签后面传递一个值来调用函数，比如`greet(person: "Anna")`。因为该函数返回一个`String`值，`greet(person:)`所以可以通过调用`print(_:separator:terminator:)`函数来打包该字符串并查看其返回值，如上所示。

> 注意
>
> 该`print(_:separator:terminator:)`函数没有第一个参数的标签，其他参数是可选的，因为它们有一个默认值。函数语法的这些变体在下面的函数参数标签和参数名称以及默认参数值中讨论。

`greet(person:)`函数 的主体首先定义一个新的`String`常量，`greeting`并将其设置为一个简单的问候消息。然后使用`return`关键字将该问候语传回功能。在代码行中`return greeting`，该函数完成其执行并返回当前值`greeting`。

您可以`greet(person:)`使用不同的输入值多次调用该函数。上面的例子显示了如果使用输入值`"Anna"`和输入值来调用会发生什么`"Brian"`。该函数在每种情况下返回一个定制的问候语。

为了使该函数的主体更短，可以将消息创建和return语句组合到一行中：

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

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

函数参数和返回值

函数参数和返回值在Swift中非常灵活。您可以使用具有单个未命名参数的简单实用程序函数来定义任何具有表达参数名称和不同参数选项的复杂函数。

没有参数的函数

函数不需要定义输入参数。这里有一个没有输入参数的函数，`String`当它被调用时它总是返回相同的消息：

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

函数定义仍然需要函数名称后面的圆括号，即使它没有使用任何参数。函数名称后面跟着一对空括号，当调用该函数时。

具有多个参数的函数

函数可以有多个输入参数，它们写在函数的括号内，用逗号分隔。

这个功能需要一个人的姓名，以及他们是否已经被输入，并返回适当的问候语给该人：

```
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!"
```

您`greet(person:alreadyGreeted:)`可以通过传递一个`String`标记`person`为`Bool`参数值的参数值和一个标记`alreadyGreeted`为括号的参数值来调用该函数，并以逗号分隔。请注意，此功能`greet(person:)`与前面部分中显示的功能不同。虽然这两个功能有打头的名称`greet`，该`greet(person:alreadyGreeted:)`函数有两个参数，但该`greet(person:)`功能只需要一个。

没有返回值的函数

函数不需要定义返回类型。下面是该`greet(person:)`函数的一个版本，它打印自己的`String`值而不是返回它：

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

因为它不需要返回值，所以函数的定义不包含返回箭头（`->`）或返回类型。

> 注意
>
> 严格地说，这个版本的`greet(person:)`功能*确实*还是返回一个值，即使没有返回值的定义。没有定义返回类型的函数返回一个特殊的类型值`Void`。这只是一个空的元组，写成`()`。

调用函数的返回值时可以忽略：

```
func printAndCount(string: String) -> Int {
    pri
    retu
}
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，然后将其字符数作为一个返回`Int`。第二个函数`printWithoutCounting(string:)`调用第一个函数，但忽略其返回值。当第二个函数被调用时，消息仍然由第一个函数打印，但是不使用返回的值。

> 注意
>
> 返回值可以被忽略，但是一个表示它将返回一个值的函数必须始终这样做。具有定义的返回类型的函数不允许控件在没有返回值的情况下掉到函数的底部，并且尝试这样做会导致编译时错误。

具有多个返回值的函数

您可以使用元组类型作为函数的返回类型，以返回多个值作为一个复合返回值的一部分。

下面的例子定义了一个叫做的函数`minMax(array:)`，它在一个`Int`值数组中找到最小和最大的数字：

```
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)
}
```

该`minMax(array:)`函数返回一个包含两个`Int`值的元组。这些值是标记的`min`，`max`以便在查询函数的返回值时可以按名称访问它们。

该`minMax(array:)`函数的主体通过设置两个调用的工作变量`currentMin`以及`currentMax`数组中第一个整数的值开始。然后，该函数在阵列并检查在每个值的剩余值进行迭代，看它是否比的值较小或较大`currentMin`和`currentMax`分别。最后，总体最小值和最大值作为两个`Int`值的元组返回。

因为元组的成员值被命名为函数返回类型的一部分，所以可以用点语法访问它们以检索最小值和最大值。

```
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"
```

请注意，元组的成员不需要在从函数返回元组的位置命名，因为它们的名称已经被指定为函数返回类型的一部分。

可选的元组返回类型

如果要从函数返回的元组类型有可能对整个元组具有“无值”，那么可以使用*可选的*元组返回类型来反映整个元组可能存在的事实`nil`。您可以通过在元组类型的右括号后面放置问号来编写可选的元组返回类型，例如`(Int, Int)?`或`(String, Int, Bool)?`。

> 注意
>
> 一个可选的元组类型，例如`(Int, Int)?`不同于包含可选类型的元组，例如`(Int?, Int?)`。使用可选的元组类型，整个元组是可选的、而不仅仅是元组中的每个单独的值。

`minMax(array:)`上面 的函数返回一个包含两个`Int`值的元组。但是，函数不会对它传递的数组执行任何安全检查。如果`array`参数包含一个空数组，则`minMax(array:)`如上所定义的函数将在尝试访问时触发运行时错误`array[0]`。

要安全地处理空数组，请`minMax(array:)`使用可选的元组返回类型编写该函数，并`nil`在数组为空时返回值：

```
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)
}
```

您可以使用可选绑定来检查此版本的`minMax(array:)`函数是否返回实际的元组值或`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"
```

函数参数标签和参数名称

每个函数参数都有一个*参数标签*和一个*参数名称*。参数标签在调用函数时使用; 每个参数都在函数调用中用它的参数标签写入。参数名称用于实现该功能。默认情况下，参数使用其参数名称作为参数标签。

```
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)
```

所有参数必须具有唯一的名称。尽管多个参数可能具有相同的参数标签，但唯一参数标签有助于使代码更具可读性。

指定参数标签

您在参数名称前面写一个参数标签，并用空格分隔：

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

下面是一个`greet(person:)`函数的变体，它带有一个人的名字和家乡，并返回一个问候语：

```
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."
```

参数标签的使用

省略参数标签

如果不需要参数的参数标签，请为该参数编写一个下划线（_）而不是显式参数标签。

```
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)
```

如果参数具有参数标签，则在调用该函数时*必须*标记该参数。

默认参数值

您可以通过在该参数的类型之后为该参数分配一个值来为函数中的任何参数定义*默认值*。如果定义了默认值，则可以在调用该函数时省略该参数。

```
func someFunction(parameterWithoutDefault: Int, parameterWithDefault: Int = 12) {
    // If you omit the second argument when calling this function, then
    // the value of parameterWithDefault is 12 inside the function body.
}
someFunction(parameterWithoutDefault: 3, parameterWithDefault: 6) //
  parameterWithDefault is 6
someFunction(parameterWithoutDefault: 4) // parameterWithDefault is 12
```

在具有默认值的参数之前，将没有默认值的参数放置在函数参数列表的开头。没有默认值的参数通常对函数的意义更重要 - 首先编写它们可以更容易地识别出调用了相同的函数，而不管是否省略任何默认参数。

变量参数

甲*可变参数参数*接受具有指定类型的零倍或更多的值。您可以使用可变参数来指定在调用函数时可以传递不同数量的输入值。通过`...`在参数的类型名称后面插入三个句点字符（）来写入可变参数。

传递给可变参数的值可以在函数体内作为适当类型的数组使用。例如，名称为`numbers`和类型的可变参数`Double...`在函数的主体中可用作称为`numbers`类型的常量数组`[Double]`。

以下示例计算任意长度的数字列表的*算术平均值*（也称为*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 3.0, 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
> 一个函数最多可以有一个可变参数。

输入输出参数

函数参数默认是常量。试图从该函数体内更改函数参数的值会导致编译时错误。这意味着你不能错误地改变参数的值。如果您想要一个函数来修改参数的值，并且希望这些更改在函数调用结束后保持不变，请将该参数定义为输入*输出参*

通过将inout关键字放置在参数类型的前面来编写输入参数。一个在出参数具有传递的值*中*，由函数修改的功能，并将该部分送回*出*的功能来代替原来的值。有关输入输出参数和相关编译器优化的详细讨论，请参阅输入输出参数。

您只能传递一个变量作为输入参数的参数。您不能传递常量或文字值作为参数，因为常量和文字不能被修改。当您将一个＆符号（&）作为参数传递给一个输入输出参数时，将其直接放在变量的名称前面，以表明它可以被该函数修改。

> 注意
>
> 输入输出参数不能具有默认值，并且可变参数不能标记为inout。

这里的一个调用的函数的示例swapTwoInts(_:_:)，其具有两个在出整数参数调用a和b：

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

该swapTwoInts(_:_:)函数只需将值b转换为a，并将值a转换为b。该函数通过将值存储a在一个名为临时常量中temporaryA，将值赋值为bto a，然后赋值temporaryA给执行此交换b。

你可以swapTwoInts(_:_:)用两个类型的变量来调用函数Int来交换它们的值。请注意，当它们传递给函数时，someInt和的名称前anotherInt加一个＆符号swapTwoInts(_:_:)：

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

上面的例子显示函数的原始值someInt和anotherInt被swapTwoInts(_:_:)函数修改，尽管它们最初是在函数之外定义的。

> 注意
>
> 输入输出参数与从函数返回值不同。在swapTwoInts上面的例子不限定返回类型或返回值，但它仍然修改的值someInt和anotherInt。输入输出参数是函数在函数体范围之外产生效果的替代方法。

函数类型

每个函数都有一个特定的*函数类型*，由*函数*的参数类型和返回类型组成。

例如：

```
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叫做addTwoInts和multiplyTwoInts。这些函数每个都取两个Int值，并返回一个Int值，这是执行适当的数学运算的结果。

这两种功能的类型是(Int, Int) -> Int。这可以理解为：

“一个函数有两个参数，都是类型Int，并返回一个类型的值Int。”

下面是另一个例

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

这个函数的类型是() -> Void，或“一个没有参数并返回的函数Void”。

使用函数类型

您可以像使用Swift中的其他类型一样使用函数类型。例如，您可以将常量或变量定义为函数类型，并为该变量分配一个适当的函数：

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

这可以理解为：

“定义一个名为的变量mathFunction，它具有一个'具有两个Int值的函数，并返回一个Int值'。设置这个新变量来引用被调用的函数addTwoInts。“

该addTwoInts(_:_:)函数与mathFunction变量具有相同的类型，因此Swift的类型检查程序允许此分配。

您现在可以使用名称来调用指定的功能mathFunction：

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

具有相同匹配类型的不同函数可以分配给相同的变量，方式与非函数类型相同：

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

和任何其他类型一样，当你给函数赋予一个常量或变量时，你可以让Swift来推断函数类型：

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

函数类型作为参数类型

您可以将函数类型用作(Int, Int) -> Int另一个函数的参数类型。这使您可以为函数的调用者保留函数实现的某些方面，以提供该函数何时被调用。

下面是一个打印上面的数学函数结果的例子：

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

这个例子定义了一个叫做的函数printMathResult(_:_:_:)，它有三个参数。第一个参数被调用mathFunction，并且是类型的(Int, Int) -> Int。您可以传递该类型的任何函数作为第一个参数的参数。第二和第三参数被称为a和b，且都是类型的Int。这些被用作提供的数学函数的两个输入值。

当printMathResult(_:_:_:)被调用时，它会传递addTwoInts(_:_:)函数，以及整数值3和5。它与值调用提供的函数3和5，并打印结果8。

其作用printMathResult(_:_:_:)是将调用的结果打印到适当类型的数学函数中。这个函数的实现实际上并不重要，只是函数的类型是正确的。这样可以printMathResult(_:_:_:)以类型安全的方式将其某些功能交给功能的调用者。

函数类型作为返回类型

您可以使用函数类型作为另一个函数的返回类型。您可以通过在返回函数的返回箭头（->）之后立即编写完整的函数类型来完成。

下一个例子定义了两个简单的函数叫做`stepForward(_:)`和`stepBackward(_:)`。该`stepForward(_:)`函数返回一个比其输入值多一个值，并且该`stepBackward(_:)`函数返回一个小于其输入值的值。这两个函数都有一个类型`(Int) -> Int`：

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

这是一个叫做的函数`chooseStepFunction(backward:)`，它的返回类型是`(Int) -> Int`。该`chooseStepFunction(backward:)`函数基于一个布尔参数调用返回`stepForward(_:)`函数或`stepBackward(_:)`函数`backward`：

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

你现在可以`chooseStepFunction(backward:)`用来获得一个将朝着一个方向或另一个方向的功能：

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

上面的例子决定了是否需要一个正的或负的步骤来将一个称为`currentValue`逐渐接近零的变量移动。`currentValue`有一个初始值`3`，这意味着`currentValue > 0`返回`true`，导致`chooseStepFunction(backward:)`返回该`stepBackward(_:)`函数。对返回函数的引用存储在一个调用的常量中`moveNearerToZero`。

现在`moveNearerToZero`指的是正确的功能，它可以用来计数到零：

```
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

本章到目前为止所遇到的所有功能都是在全球范围内定义的*全局功能的*示例。您还可以在其他函数的主体内定义函数，称为*嵌套函数*。

嵌套函数在默认情况下对外部世界隐藏，但仍然可以通过其封闭函数调用和使用。封闭函数也可以返回其嵌套函数之一，以允许嵌套函数在另一个作用域中使用。

你可以重写`chooseStepFunction(backward:)`上面的例子来使用并返回嵌套函数：

```
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)
// move
while currentValue != 0 {
    print("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
print("zero!")
// -4...
// -3...
// -2...
// -1...
// zero!
```

在本页