函数

*函数*是一段完成特定任务的独立代码片段。你可以通过给函数命名来标识某个函数的功能，这个名字可以被用来在需要的时候“调用”这个函数来完成它的任务。

Swift 统一的函数语法非常的灵活，可以用来表示任何函数，包括从最简单的没有参数名字的 C 风格函数，到复杂的带局部和外部参数名的 Objective-C 风格函数。参数可以提供默认值，以简化函数调用。参数也可以既当做传入参数，也当做传出参数，也就是说，一旦函数执行结束，传入的参数值将被修改。

在 Swift 中，每个函数都有一个由函数的参数值类型和返回值类型组成的类型。你可以把函数类型当做任何其他普通变量类型一样处理，这样就可以更简单地把函数当做别的函数的参数，也可以从其他函数中返回函数。函数的定义可以写在其他函数定义中，这样可以在嵌套函数范围内实现功能封装。

函数的定义与调用 {#Defining-and-Calling-Functions}

当你定义一个函数时，你可以定义一个或多个有名字和类型的值，作为函数的输入，称为*参数*，也可以定义某种类型的值作为函数执行结束时的输出，称为*返回类型*。

每个函数有个*函数名*，用来描述函数执行的任务。要使用一个函数时，用函数名来“调用”这个函数，并传给它匹配的输入值（称作*实参*）。函数的实参必须与函数参数表里参数的顺序一致。

下面例子中的函数的名字是 `greet(person:)`，之所以叫这个名字，是因为这个函数用一个人的名字当做输入，并返回向这个人问候的语句。为了完成这个任务，你需要定义一个输入参数——一个叫做 `person` 的 `String` 值，和一个包含给这个人问候语的 `String` 类型的返回值：

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

所有的这些信息汇总起来成为函数的*定义*，并以 `func` 作为前缀。指定函数返回类型时，用返回箭头 `->`（一个连字符后跟一个右尖括号）后跟返回类型的名称的方式来表示。

该定义描述了函数的功能，它期望接收什么作为参数和执行结束时它返回的结果是什么类型。这样的定义使得函数可以在别的地方以一种清晰的方式被调用：

```
print(greet(person: "Anna"))
// 打印“Hello, Anna!”
print(greet(person: "Brian"))
// 打印“Hello, Brian!”
```

调用 `greet(person:)` 函数时，在圆括号中传给它一个 `String` 类型的实参，例如 `greet(person: "Anna")`。正如上面所示，因为这个函数返回一个 `String` 类型的值，所以 `greet` 可以被包含在 `print(_:separator:terminator:)` 的调用中，用来输出这个函数的返回值。

> 注意
>
> `print(_:separator:terminator:)` 函数的第一个参数并没有设置一个标签，而其他的参数因为已经有了默认值，因此是可选的。关于这些函数语法上的变化详见下方关于 函数参数标签和参数名以及默认参数值。

在 `greet(person:)` 的函数体中，先定义了一个新的名为 `greeting` 的 `String` 常量，同时，把对 `personName` 的问候消息赋值给了 `greeting` 。然后用 `return` 关键字把这个问候返回出去。一旦 `return greeting` 被调用，该函数结束它的执行并返回 `greeting` 的当前值。

你可以用不同的输入值多次调用 `greet(person:)`。上面的例子展示的是用 `"Anna"` 和 `"Brian"` 调用的结果，该函数分别返回了不同的结果。

为了简化这个函数的定义，可以将问候消息的创建和返回写成一句：

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

函数参数与返回值 {#Function-Parameters-and-Return-Values}

函数参数与返回值在 Swift 中非常的灵活。你可以定义任何类型的函数，包括从只带一个未名参数的简单函数到复杂的带有表达性参数名和不同参数选项的复杂函数。

无参数函数 {#functions-without-parameters}

函数可以没有参数。下面这个函数就是一个无参数函数，当被调用时，它返回固定的 `String` 消息：

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

尽管这个函数没有参数，但是定义中在函数名后还是需要一对圆括号。当被调用时，也需要在函数名后写一对圆括号。

多参数函数 {#functions-with-multiple-parameters}

函数可以有多种输入参数，这些参数被包含在函数的括号之中，以逗号分隔。

下面这个函数用一个人名和是否已经打过招呼作为输入，并返回对这个人的适当问候语：

```
func greet(person: String, alreadyGreeted: Bool) -> String {
    if alreadyGreeted {
```

```
        return greetAgain(person: person)
    } else {
        return greet(person: person)
    }
}
print(greet(person: "Tim", alreadyGreeted: true))
// 打印“Hello again, Tim!”
```

你可以通过在括号内使用逗号分隔来传递一个 `String` 参数值和一个标识为 `alreadyGreeted` 的 `Bool` 值，来调用 `greet(person:alreadyGreeted:)` 函数。注意这个函数和上面 `greet(person:)` 是不同的。虽然它们都有着同样的名字 `greet`，但是 `greet(person:alreadyGreeted:)` 函数需要两个参数，而 `greet(person:)` 只需要一个参数。

无返回值函数 {#functions-without-return-values}

函数可以没有返回值。下面是 `greet(person:)` 函数的另一个版本，这个函数直接打印一个 `String` 值，而不是返回它：

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

因为这个函数不需要返回值，所以这个函数的定义中没有返回箭头（->）和返回类型。

> 注意
>
> 严格地说，即使没有明确定义返回值，该 `greet(Person: )` 函数仍然返回一个值。没有明确定义返回类型的函数会返回一个 `Void` 类型特殊值，该值为一个空元组，写成 ()。

调用函数时，可以忽略该函数的返回值：

```
func printAndCount(string: String) -> Int {
    print(string)
    return string.count
}
func printWithoutCounting(string: String) {
    let _ = printAndCount(string: string)
}
printAndCount(string: "hello, world")
// 打印“hello, world”，并且返回值 12
printWithoutCounting(string: "hello, world")
// 打印“hello, world”，但是没有返回任何值
```

第一个函数 `printAndCount(string:)`，输出一个字符串并返回 `Int` 类型的字符数。第二个函数 `printWithoutCounting(string:)` 调用了第一个函数，但是忽略了它的返回值。当第二个函数被调用

时，消息依然会由第一个函数输出，但是返回值不会被用到。

> 注意
>
> 返回值可以被忽略，但定义了有返回值的函数必须返回一个值，如果在函数定义底部没有返回任何值，将导致编译时错误。

多重返回值函数 {#functions-with-multiple-return-values}

你可以用元组（tuple）类型让多个值作为一个复合值从函数中返回。

下例中定义了一个名为 `minMax(array:)` 的函数，作用是在一个 `Int` 类型的数组中找出最小值与最大值。

```
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)
}
```

`minMax(array:)` 函数返回一个包含两个 `Int` 值的元组，这些值被标记为 `min` 和 `max`，以便查询函数的返回值时可以通过名字访问它们。

在 `minMax(array:)` 的函数体中，在开始的时候设置两个工作变量 `currentMin` 和 `currentMax` 的值为数组中的第一个数。然后函数会遍历数组中剩余的值并检查该值是否比 `currentMin` 和 `currentMax` 更小或更大。最后数组中的最小值与最大值作为一个包含两个 `Int` 值的元组返回。

因为元组的成员值已被命名，因此可以通过 `.` 语法来检索找到的最小值与最大值：

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

需要注意的是，元组的成员不需要在元组从函数中返回时命名，因为它们的名字已经在函数返回类型中指定了。

可选元组返回类型 {#optional-tuple-return-types}

如果函数返回的元组类型有可能整个元组都“没有值”，你可以使用*可选的*元组返回类型反映整个元组可以是 `nil` 的事实。你可以通过在元组类型的右括号后放置一个问号来定义一个可选元组，例如 `(Int, Int)?` 或 `(String, Int, Bool)?`

> 注意

> 可选元组类型如 `(Int, Int)?` 与元组包含可选类型如 `(Int?, Int?)` 是不同的。可选的元组类型，整个元组是可选的，而不只是元组中的每个元素值。

前面的 `minMax(array:)` 函数返回了一个包含两个 `Int` 值的元组。但是函数不会对传入的数组执行任何安全检查，如果 `array` 参数是一个空数组，如上定义的 `minMax(array:)` 在试图访问 `array[0]` 时会触发一个运行时错误。

为了安全地处理这个“空数组”问题，将 `minMax(array:)` 函数改写为使用可选元组返回类型，并且当数组为空时返回 `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)
}
```

你可以使用可选绑定来检查 `minMax(array:)` 函数返回的是一个存在的元组值还是 `nil`：

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

隐式返回的函数 {#functions-with-an-implicit-return}

如果一个函数的整个函数体是一个单行表达式，这个函数可以隐式地返回这个表达式。举个例子，以下的函数有着同样的作用：

```
func greeting(for person: String) -> String {
    "Hello, " + person + "!"
}
print(greeting(for: "Dave"))
// 打印 "Hello, Dave!"

func anotherGreeting(for person: String) -> String {
    return "Hello, " + person + "!"
}
print(anotherGreeting(for: "Dave"))
// 打印 "Hello, Dave!"
```

`greeting(for:)` 函数的完整定义是打招呼内容的返回，这就意味着它能使用隐式返回这样更简短的形式。`anothergreeting(for:)` 函数返回同样的内容，却因为 `return` 关键字显得函数更长。任何一个可以被写成一行 `return` 语句的函数都可以忽略 `return`。

正如你将会在 简略的 Getter 声明 里看到的， 一个属性的 getter 也可以使用隐式返回的形式。

> 注意
>
> 作为隐式返回值编写的代码需要返回一些值。例如，你不能使用 `print(13)` 作为隐式返回值。然而，你可以使用不返回值的函数（如 `fatalError("Oh no!")`）作为隐式返回值，因为 Swift 知道它们并不会产生任何隐式返回。

函数参数标签和参数名称 {#Function-Argument-Labels-and-Parameter-Names}

每个函数参数都有一个*参数标签（argument label）以及一个参数名称（parameter name）*。参数标签在调用函数的时候使用；调用的时候需要将函数的参数标签写在对应的参数前面。参数名称在函数的实现中使用。默认情况下，函数参数使用参数名称来作为它们的参数标签。

```
func someFunction(firstParameterName: Int, secondParameterName: Int) {
    // 在函数体内，firstParameterName 和 secondParameterName 代表参数中的第一个和第二个参数值
}
someFunction(firstParameterName: 1, secondParameterName: 2)
```

所有的参数都必须有一个独一无二的名字。虽然多个参数拥有同样的参数标签是可能的，但是一个唯一的参数标签能够使你的代码更具可读性。

指定参数标签 {#specifying-argument-labels}

你可以在参数名称前指定它的参数标签，中间以空格分隔：

```
func someFunction(argumentLabel parameterName: Int) {
    // 在函数体内，parameterName 代表参数值
}
```

这个版本的 `greet(person:)` 函数，接收一个人的名字和他的家乡，并且返回一句问候：

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

参数标签的使用能够让一个函数在调用时更有表达力，更类似自然语言，并且仍保持了函数内部的可读性以及清晰的意图。

忽略参数标签 {#omitting-argument-labels}

如果你不希望为某个参数添加一个标签，可以使用一个下划线（_）来代替一个明确的参数标签。

```
func someFunction(_ firstParameterName: Int, secondParameterName: Int) {
    // 在函数体内，firstParameterName 和 secondParameterName 代表参数中的第一个和第二个参数值
}
someFunction(1, secondParameterName: 2)
```

如果一个参数有一个标签，那么在调用的时候必须使用标签来标记这个参数。

默认参数值 {#default-parameter-values}

你可以在函数体中通过给参数赋值来为任意一个参数定义*默认值（Deafult Value）*。当默认值被定义后，调用这个函数时可以忽略这个参数。

```
func someFunction(parameterWithoutDefault: Int, parameterWithDefault: Int = 12) {
    // 如果你在调用时候不传第二个参数，parameterWithDefault 会值为 12 传入到函数体中。
}
someFunction(parameterWithoutDefault: 3, parameterWithDefault: 6) // parameterWithDefault = 6
someFunction(parameterWithoutDefault: 4) // parameterWithDefault = 12
```

将不带有默认值的参数放在函数参数列表的最前。一般来说，没有默认值的参数更加的重要，将不带默认值的参数放在最前保证在函数调用时，非默认参数的顺序是一致的，同时也使得相同的函数在不同情况下调用时显得更为清晰。

可变参数 {#variadic-parameters}

一个*可变参数（variadic parameter）*可以接受零个或多个值。函数调用时，你可以用可变参数来指定函数参数可以被传入不确定数量的输入值。通过在变量类型名后面加入（...）的方式来定义可变参数。

可变参数的传入值在函数体中变为此类型的一个数组。例如，一个叫做 numbers 的 Double... 型可变参数，在函数体内可以当做一个叫 numbers 的 [Double] 型的数组常量。

下面的这个函数用来计算一组任意长度数字的 *算术平均数（arithmetic mean）*：

```
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)
// 返回 3.0, 是这 5 个数的平均数。
arithmeticMean(3, 8.25, 18.75)
// 返回 10.0, 是这 3 个数的平均数。
```

一个函数能拥有多个可变参数。可变参数后的第一个行参前必须加上实参标签。实参标签用于区分实参是传递给可变参数，还是后面的行参。

输入输出参数 {#in-out-parameters}

函数参数默认是常量。试图在函数体中更改参数值将会导致编译错误。这意味着你不能错误地更改参数值。如果你想要一个函数可以修改参数的值，并且想要在这些修改在函数调用结束后仍然存在，那么就应该把这个参数定义为*输入输出参数（In-Out Parameters）*。

定义一个输入输出参数时，在参数定义前加 `inout` 关键字。一个 输入输出参数有传入函数的值，这个值被函数修改，然后被传出函数，替换原来的值。想获取更多的关于输入输出参数的细节和相关的编译器优化，请查看 输入输出参数 一节。

你只能传递变量给输入输出参数。你不能传入常量或者字面量，因为这些量是不能被修改的。当传入的参数作为输入输出参数时，需要在参数名前加 `&` 符，表示这个值可以被函数修改。

> 注意
>
> 输入输出参数不能有默认值，而且可变参数不能用 `inout` 标记。

下例中，`swapTwoInts(_:_:)` 函数有两个分别叫做 `a` 和 `b` 的输入输出参数：

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

`swapTwoInts(_:_:)` 函数简单地交换 `a` 与 `b` 的值。该函数先将 `a` 的值存到一个临时常量 `temporaryA` 中，然后将 `b` 的值赋给 `a`，最后将 `temporaryA` 赋值给 `b`。

你可以用两个 `Int` 型的变量来调用 `swapTwoInts(_:_:)`。需要注意的是，`someInt` 和 `anotherInt` 在传入 `swapTwoInts(_:_:)` 函数前，都加了 `&` 的前缀：

```
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”
```

从上面这个例子中，我们可以看到 `someInt` 和 `anotherInt` 的原始值在 `swapTwoInts(_:_:)` 函数中被修改，尽管它们的定义在函数体外。

> 注意
>
> 输入输出参数和返回值是不一样的。上面的 `swapTwoInts` 函数并没有定义任何返回值，但仍然修改了 `someInt` 和 `anotherInt` 的值。输入输出参数是函数对函数体外产生影响的另一种方式。

函数类型 {#Function-Types}

每个函数都有种特定的*函数类型*，函数的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
}
```

这个例子中定义了两个简单的数学函数：`addTwoInts` 和 `multiplyTwoInts`。这两个函数都接受两个 `Int` 值， 返回一个 `Int` 值。

这两个函数的类型是 `(Int, Int) -> Int`，可以解读为：

“这个函数类型有两个 `Int` 型的参数并返回一个 `Int` 型的值”。

下面是另一个例子，一个没有参数，也没有返回值的函数：

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

这个函数的类型是：`() -> Void`，或者叫“没有参数，并返回 `Void` 类型的函数”。

使用函数类型 {#using-function-types}

在 Swift 中，使用函数类型就像使用其他类型一样。例如，你可以定义一个类型为函数的常量或变量，并将适当的函数赋值给它：

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

这段代码可以被解读为：

”定义一个叫做 `mathFunction` 的变量，类型是‘一个有两个 `Int` 型的参数并返回一个 `Int` 型的值的函数’，并让这个新变量指向 `addTwoInts` 函数”。

`addTwoInts` 和 `mathFunction` 有同样的类型，所以这个赋值过程在 Swift 类型检查（type-check）中是允许的。

现在，你可以用 `mathFunction` 来调用被赋值的函数了：

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

有相同匹配类型的不同函数可以被赋值给同一个变量，就像非函数类型的变量一样：

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

就像其他类型一样，当赋值一个函数给常量或变量时，你可以让 Swift 来推断其函数类型：

```
let anotherMathFunction = addTwoInts
// anotherMathFunction 被推断为 (Int, Int) -> Int 类型
```

函数类型作为参数类型 {#function-types-as-parameter-types}

你可以用 `(Int, Int) -> Int` 这样的函数类型作为另一个函数的参数类型。这样你可以将函数的一部分实现留给函数的调用者来提供。

下面是另一个例子，正如上面的函数一样，同样是输出某种数学运算结果：

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

这个例子定义了 `printMathResult(_:_:_:)` 函数，它有三个参数：第一个参数叫 `mathFunction`，类型是 `(Int, Int) -> Int`，你可以传入任何这种类型的函数；第二个和第三个参数叫 `a` 和 `b`，它们的类型都是 `Int`，这两个值作为已给出的函数的输入值。

当 `printMathResult(_:_:_:)` 被调用时，它被传入 `addTwoInts` 函数和整数 `3` 和 `5`。它用传入 `3` 和 `5` 调用 `addTwoInts`，并输出结果：`8`。

`printMathResult(_:_:_:)` 函数的作用就是输出另一个适当类型的数学函数的调用结果。它不关心传入函数是如何实现的，只关心传入的函数是不是一个正确的类型。这使得 `printMathResult(_:_:_:)` 能以一种类型安全（type-safe）的方式将一部分功能转给调用者实现。

函数类型作为返回类型 {#function-types-as-return-types}

你可以用函数类型作为另一个函数的返回类型。你需要做的是在返回箭头（->）后写一个完整的函数类型。

下面的这个例子中定义了两个简单函数，分别是 `stepForward(_:)` 和 `stepBackward(_:)`。`stepForward(_:)` 函数返回一个比输入值大 `1` 的值。`stepBackward(_:)` 函数返回一个比输入值小 `1` 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
}
```

如下名为 `chooseStepFunction(backward:)` 的函数，它的返回类型是 `(Int) -> Int` 类型的函数。`chooseStepFunction(backward:)` 根据布尔值 `backwards` 来返回 `stepForward(_:)` 函数或 `stepBackward(_:)` 函数：

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

你现在可以用 `chooseStepFunction(backward:)` 来获得两个函数其中的一个：

```
var currentValue = 3
let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)
// moveNearerToZero 现在指向 stepBackward() 函数。
```

上面这个例子中计算出从 `currentValue` 逐渐接近到0是需要向正数走还是向负数走。`currentValue` 的初始值是 `3`，这意味着 `currentValue > 0` 为真（true），这将使得 `chooseStepFunction(_:)` 返回 `stepBackward(_:)` 函数。一个指向返回的函数的引用保存在了 `moveNearerToZero` 常量中。

现在，`moveNearerToZero` 指向了正确的函数，它可以被用来数到零：

```
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 {#Nested-Functions}

到目前为止本章中你所见到的所有函数都叫*全局函数（global functions）*，它们定义在全局域中。你也可以把函数定义在别的函数体中，称作 *嵌套函数（nested functions）*。

默认情况下，嵌套函数是对外界不可见的，但是可以被它们的外围函数（enclosing function）调用。一个外围函数也可以返回它的某一个嵌套函数，使得这个函数可以在其他域中被使用。

你可以用返回嵌套函数的方式重写 `chooseStepFunction(backward:)` 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 now refers to the nested stepForward() function
while currentValue != 0 {
    print("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
print("zero!")
// -4...
// -3...
// -2...
// -1...
// zero!
```