翻译：honghaoz　校对：LunaticM

函数（Functions）

本页包含内容：

函数是用来完成特定任务的独立的代码块。你给一个函数起一个合适的名字，用来标示函数做什么，并且当函数需要执行的时候，这个名字会被“调用”。

Swift 统一的函数语法足够灵活，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。参数也可以即当做传入参数，也当做传出参数，也就是说，一旦函数执行结束，传入的参数值可以被修改。

在 Swift 中，每个函数都有一种类型，包括函数的参数值类型和返回值类型。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，parameters），也可以定义某种类型的值作为函数

执行结束的输出（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，你用函数名“调用”，并传给它匹配的输入值（称作实参，arguments）。一个函数的实参必须与函数参数表里参数的顺序一致。

在下面例子中的函数叫做`"greetingForPerson"`，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，并返回给这个人的问候语。为了完成这个任务，你定义一个输入参数-一个叫做 `personName` 的 `String` 值，和一个包含给这个人问候语的 `String` 类型的返回值：

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

该定义描述了函数做什么，它期望接收什么和执行结束时它返回的结果是什么。这样的定义使的函数可以在别的地方以一种清晰的方式被调用：

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

调用 `sayHello` 函数时，在圆括号中传给它一个 `String` 类型的实参。因为这个函数返回一个 `String` 类型的值，`sayHello` 可以被包含在 `println` 的调用中，用来输出这个函数的返回值，正如上面所示。

在 `sayHello` 的函数体中，先定义了一个新的名为 `greeting` 的 `String` 常量，同时赋值了给 `personName` 的一个简单问候消息。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 `sayHello`。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 sayHelloAgain(personName: String) -> String {
    return "Hello again, " + personName + "!"
}
println(sayHelloAgain("Anna"))
// prints "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中极为灵活。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。

多重输入参数（Multiple Input Parameters）

函数可以有多个输入参数，写在圆括号中，用逗号分隔。

下面这个函数用一个半开区间的开始点和结束点，计算出这个范围内包含多少数字：

```
func halfOpenRangeLength(start: Int, end: Int) -> Int
{
    return end - start
}
println(halfOpenRangeLength(1, 10))
// prints "9
```

无参函数（Functions Without Parameters）

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

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

```
}
println(sayHelloWorld())
// prints "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。

无返回值函数（Functions Without Return Values）

函数可以没有返回值。下面是 `sayHello` 函数的另一个版本，叫 `waveGoodbye`，这个函数直接输出 `String` 值，而不是返回它：

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

严格上来说，虽然没有返回值被定义，`sayGoodbye` 函数依然返回了值。没有定义返回类型的函数会返回特殊的值，叫 `Void`。它其实是一个空的元组（tuple），没有任何元素，可以写成`()`。

被调用时，一个函数的返回值可以被忽略：

```
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`，输出一个字符串并返回 `Int` 类型的字符数。第二个函数 `printWithoutCounting`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，这叫导致编译错误（compile-time error）。

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。

下面的这个例子中，`count` 函数用来计算一个字符串中元音，辅音和其他字母的个数（基于美式英语的标准）。

```
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)
}
```

你可以用 `count` 函数来处理任何一个字符串，返回的值将是一个包含三

个 `Int` 型值的元组（tuple）：

```
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，元组的成员不需要在函数中返回时命名，因为它们的名字已经在函数返回类型有有了定义。

函数参数名称（Function Parameter Names）

以上所有的函数都给它们的参数定义了`参数名（parameter name）`：

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

但是，这些参数名仅在函数体中使用，不能在函数调用时使用。这种类型的参数名被称作`局部参数名（local parameter name）`，因为它们只能在函数体中使用。

外部参数名（External Parameter Names）

有时候，调用函数时，给每个参数命名是非常有用的，因为这些参数名可以指出各个实参的用途是什么。

如果你希望函数的使用者在调用函数时提供参数名字，那就需要给每个参数除了局部参数名外再定义一个`外部参数名`。外部参数名写在局部参数名之前，用空格分隔。

```
func someFunction(externalParameterName
localParameterName: Int) {
    // function body goes here, and can use
```

```
localParameterName
    // to refer to the argument value for that
parameter
}
```

注意：

如果你提供了外部参数名，那么函数在被调用时，必须使用外部参数名。

以下是个例子，这个函数使用一个结合者（joiner）把两个字符串联在一起：

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

当你调用这个函数时，这三个字符串的用途是不清楚的：

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

为了让这些字符串的用途更为明显，我们为 `join` 函数添加外部参数名：

```
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，第一个参数有一个叫 `string` 的外部参数名和 `s1` 的局部参数名，第二个参数有一个叫 `toString` 的外部参数名和 `s2` 的局部参数名，第三个参数有一个叫 `withJoiner` 的外部参数名和 `joiner` 的局部参数名。

现在，你可以使用这些外部参数名以一种清晰地方式来调用函数了：

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

使用外部参数名让第二个版本的 `join` 函数的调用更为有表现力，更为通顺，同时还保持了函数体是可读的和有明确意图的。

注意：

当其他人在第一次读你的代码，函数参数的意图显得不明显时，考虑使用外部参数名。如果函数参数名的意图是很明显的，那就不需要定义外部参数名了。

简写外部参数名（Shorthand External Parameter Names）

如果你需要提供外部参数名，但是局部参数名已经定义好了，那么你不需要写两次这些参数名。相反，只写一次参数名，并用井号（#）作为前缀就可以了。这告诉 Swift 使用这个参数名作为局部和外部参数名。

下面这个例子定义了一个叫 containsCharacter 的函数，使用井号（#）的方式定义了外部参数名：

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

这样定义参数名，使得函数体更为可读，清晰，同时也可以以一个不含糊的方式被调用：

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

默认参数值（Default Parameter Values）

你可以在函数体中为每个参数定义默认值。当默认值被定义后，调用这个函数时可以略去这个参数。

注意：

将带有默认值的参数放在函数参数表的最后。这样可以保证在函数调用时，非默认参数的顺序是一致的，同时使得相同的函数在不同情况下调用时显得更为清晰。

以下是另一个版本的`join`函数，其中`joiner`有了默认参数值：

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

像第一个版本的 `join` 函数一样，如果 `joiner` 被赋值时，函数将使用这个字符串值来连接两个字符串：

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

当这个函数被调用时，如果 `joiner` 的值没有被指定，函数会使用默认值（`" "`）：

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

默认值参数的外部参数名（External Names for Parameters with Default Values）

在大多数情况下，给带默认值的参数起一个外部参数名是很有用的。这样可以保证当函数被调用且带默认值的参数被提供值时，实参的意图是明显的。

为了使定义外部参数名更加简单，当你未给带默认值的参数提供外部参数名时，Swift 会自动提供外部名字。此时外部参数名与局部名字是一样的，就像你已经在局部参数名前写了`井号（#）`一样。

下面是 `join` 函数的另一个版本，这个版本中并没有为它的参数提供外部参数名，但是 `joiner` 参数依然有外部参数名：

```
func join(s1: String, s2: String, joiner: String = "
```

```
") -> String {
    return s1 + joiner + s2
}
```

在这个例子中，Swift 自动为 `joiner` 提供了外部参数名。因此，当函数调用时，外部参数名必须使用，这样使得参数的用途变得清晰。

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

注意：

你可以使用`下划线（_）`作为默认值参数的外部参数名，这样可以在调用时不用提供外部参数名。但是给带默认值的参数命名总是更加合适的。

可变参数（Variadic Parameters）

一个`可变参数（variadic parameter）`可以接受一个或多个值。函数调用时，你可以用可变参数来传入不确定数量的输入参数。通过在变量类型名后面加入`（...）`的方式来定义可变参数。

传入可变参数的值在函数体内当做这个类型的一个数组。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：

```
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：

一个函数至多能有一个可变参数，而且它必须是参数表中最后的一个。这样做是为了避免函数调用时出现歧义。
如果函数有一个或多个带默认值的参数，而且还有一个可变参数，那么把可变参数放在参数表的最后。

常量参数和变量参数（Constant and Variable 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。

但是，有时候，如果函数中有传入参数的变量值副本将是很有用的。你可以通过指定一个或多个参数为变量参数，从而避免自己在函数中定义新的变量。变量参数不是常量，你可以在函数中把它当做新的可修改副本来使用。

通过在参数名前加关键字 `var` 来定义变量参数：

```
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"
```

这个例子中定义了一个新的叫做 `alignRight` 的函数，用来右对齐输入的字符串到一个长的输出字符串中。左侧空余的地方用指定的填充字符填充。这个例子中，字符串`"hello"`被转换成了`"-----hello"`。

`alignRight` 函数将参数 `string` 定义为变量参数。这意味着 `string` 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操作。

该函数首先计算出多少个字符需要被添加到 `string` 的左边，以右对齐到总的字符串中。这个值存在局部常量 `amountToPad` 中。这个函数然后将 `amountToPad` 多的填充（pad）字符填充到 `string` 左边，并返回结果。它使用了 `string` 这个变量参数来进行所有字符串操作。

注意：

对变量参数所进行的修改在函数调用结束后变消息了，并且对于函数体外是不可见的。变量参数仅仅存在于函数调用的生命周期中。

输入输出参数（In-Out Parameters）

变量参数，正如上面所述，仅仅能在函数体内被更改。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。

你只能传入一个变量作为输入输出参数。你不能传入常量或者字面量（literal value），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，而且变量参数不能用 `inout` 标记。如果你用 `inout` 标记一个参数，这个参数不能别 `var` 或者 `let` 标记。
下面是例子，`swapTwoInts` 函数，有两个分别叫做 `a` 和 `b` 的输出输出参数：

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

```
    b = temporaryA
}
```
这个 `swapTwoInts` 函数仅仅交换 `a` 与 `b` 的值。该函数先将 `a` 的值存到一个暂时常量 `temporaryA` 中，然后将 `b` 的值赋给 `a`，最后将 `temporaryA` 幅值给 `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)
println("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
// prints "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：

输出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，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`。这两个函数都传入两个 `Int` 类型， 返回一个合适的 `Int`值。

这两个函数的类型是 `(Int, Int) -> Int`，可以读作“这个函数类型，它有两个 `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() {
    println("hello, world")
}
```

这个函数的类型是：`() -> ()`，或者叫“没有参数，并返回 `Void` 类型的函数。”。没有指定返回类型的函数总返回 `Void`。在Swift中，`Void` 与空的元组是一样的。

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，你可以定义一个常量或变量，它的类型是函数，并且可以幅值为一个函数：

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

这个可以读作：

“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 类型检查中是允许的。

现在，你可以用 `mathFunction` 来调用被赋值的函数了：

```
println("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
println("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 来推测其函数类型：

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

函数类型作为参数类型（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交由给函数的调用者。

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) {
    println("Result: \(mathFunction(a, b))")
}
printMathResult(addTwoInts, 3, 5)
// prints "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合适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，它只关心这个传入的函数类型是正确的。这使得 `printMathResult` 可以以一种类型安全（type-safe）的方式来保证传入函数的调用是正确的。

函数类型作为返回类型（Function Type 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一的值。`stepBackward` 函数返回一个比输入值小一的值。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
}
```

下面这个叫做 `chooseStepFunction` 的函数，它的返回类型是 `(Int) -> Int` 的函数。`chooseStepFunction` 根据布尔值 `backwards` 来返回 `stepForward` 函数或 `stepBackward` 函数：

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

你现在可以用 `chooseStepFunction` 来获得一个函数，不管是那个方向：

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

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` 是真的（`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`0`：

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

嵌套函数（Nested Functions）

这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，但是可以被他们封闭函数（enclosing function）来调用。一个封闭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` 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!
```