**函数**

函数是执行特定任务的代码自包含块。通过给定一个函数名称标识它是什么，并在需要的时候使用该名称来调用函数以执行任务。

Swift的统一的功能语法足够灵活的，可表达任何东西，无论是不带参数名称的简单的样式函数，还是带本地和外部参数名称的复杂的Objective-C样式方法。参数可为简单函数调用提供默认值，并且可以被作为输入/输出参数传递，在函数执行完成时修改传递来的变量。

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。你可以像使用Swift中其他类型一样使用该类型,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。函数也可以被写入其他函数中以在函数作用于中封装有用的功能。

**定义和调用函数**

当你定义一个函数时，你可以选择性地定义一个或多个名称，类型值作为函数的输入（称为形参），或者定义一个函数结束后返回值的类型（称之为返回型）。每一个函数都有一个函数名，用来描述了函数执行的任务。要使用一个函数时，可使用它的名称进行“调用”，并通过它的输入值（称为实参--argument）来匹配函数的参数类型。一个函数的实参（arguments）必须始终和函数形参（parameter）顺序一致。

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针对那个人的问候语。为了实现该功能，你定义了一个输出参数--一个名为personName的字符串值，以及一个String返回类型，包含一个针对那个人的问候语：

1. func sayHello(personName: String) -> String {
2. let greeting = "Hello, " + personName + "!"
3. **return** greeting
4. }

所有这些信息都汇总到以func关键字为前缀的函数定义中。使用箭头->来指明函数的返回类型（一个连字符后跟一个向右的箭头），后边跟着返回的类型名称。

该定义描述了函数的作用是什么，它期望接收什么，以及完成后返回的结果。该定义可轻易地让你在代码中的其他地方清晰明确地调用该函数：

1. println(sayHello("Anna"))
2. // prints "Hello, Anna!"
3. println(sayHello("Brian"))
4. // prints "Hello, Brian!"

你可以通过给它传递一个圆括号内String实参值来调用sayHello函数，例如sayHello（"Anna"）。由于该函数返回一个String值，sayHello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。

sayHello的函数主体首先定义了一个新的名为greeting的String常量，并将其设置加上personName组成一句简单的问候消息。然后这个greeting以关键字return来传回到函数外部。只要return greeting被调用，函数执行完毕后就会返回greeting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使用"Anna"输入值调用它会发生什么，以及以"Brian"输入值调用时会发生什么。函数为每种情况量身定制了问候语。

为了简化这个函数的主体，可把消息创建和return语句合并成一行：

1. func sayHello(personName: String) -> String {
2. **return** "Hello again, " + personName + "!"
3. }
4. println(sayHello("Anna"))
5. // prints "Hello again, Anna!"

**函数的形参和返回值**

在swift中，函数的形参和返回值是非常具有灵活性的。你可以定义任何事情，无论是一个简单的仅有一个未命名形参的工具函数，还是那种具有丰富的参数名称和不同的形参选项的复杂函数。

**多输入形参**

函数可以有多个输入形参，把它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半开区间的开始和结束索引，用来计算在范围内有多少元素：

1. func halfOpenRangeLength(start: Int, end: Int) -> Int {
2. **return** end - start
3. }
4. println(halfOpenRangeLength(1, 10))
5. // prints "9"

**无形参函数**

函数并没有要求一定要定义的输入形参。下面就是一个没有输入形参的函数，任何时候调用时它总是返回相同的String消息：

1. func sayHelloWorld() -> String {
2. **return** "hello, world"
3. }
4. println(sayHelloWorld())
5. // prints "hello, world"

该函数的定义还需要在函数的名称后跟一对儿圆括号，即使它不带任何形参。当函数被调用时函数名称也要跟着一对儿空括号。

**无返回值的函数**

函数不需要定义一个返回类型。这里有一个版本的sayHello函数，称为waveGoodbye，它会打印自己的String值而不是返回它：

1. func sayGoodbye(personName: String) {
2. println("Goodbye, \(personName)!")
3. }
4. sayGoodbye("Dave")
5. // 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函数确实还返回一个值，即使没有定义返回值。没有定义返回类型的函数返回了一个Void类型的特殊值。这仅是一个空元组，这里边没有元素，可以被写成()。 |

当一个函数调用时它的返回值可以忽略不计：

1. func printAndCount(stringToPrint: String) -> Int {
2. println(stringToPrint)
3. **return** countElements(stringToPrint)
4. }
5. func printWithoutCounting(stringToPrint: String) {
6. printAndCount(stringToPrint)
7. }
8. printAndCount("hello, world")
9. // prints "hello, world" and returns a value of 12
10. printWithoutCounting("hello, world")
11. // 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，消息由第一函数打印了回来，但没有使用其返回值。

|  |
| --- |
| 提示：返回值可以忽略，但一个定义了返回值的函数则必须有返回值。对于一个定义了返回类型的函数来说，如果没有返回值，那么将不允许控制流离开函数的底部。如果试图这样做将出现一个编译时错误。 |

**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，来返回一个由多个值组成的复合返回值。

下面的例子定义了一个名为count函数，用来计算字符串中基于标准美式英语的元音、辅音以及字符的数量：

1. func count(string: String) -> (vowels: Int, consonants: Int, others: Int) {
2. **var** vowels = 0, consonants = 0, others = 0
3. **for** character **in** string {
4. **switch** String(character).lowercaseString {
5. **case** "a", "e", "i", "o", "u":
6. ++vowels
7. **case** "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
8. "n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":
9. ++consonants
10. **default**:
11. ++others
12. }
13. }
14. **return** (vowels, consonants, others)
15. }

您可以使用此计数函数来对任意字符串进行字符计数，以检索一个包含三个指定Int值的元素统计总数：

1. let total = count("some arbitrary string!")
2. println("\(total.vowels) vowels and \(total.consonants) consonants")
3. // prints "6 vowels and 13 consonants"

注意：这一点上元组的成员不需要被命名，元组是从函数中返回的，因为它们的名字已经被指定为函数的返回类型的一部分。

**函数形参名**

所有上面的函数都为其形参定义了形参名：

1. func someFunction(parameterName: Int) {
2. // function body goes here, and can use parameterName
3. // to refer to the argument value for that parameter
4. }

然而，这些参数名的仅能在函数本身的主体内使用，不能在调用函数时使用。这种形参类型名称被称之为本地形参名（local parameter name），因为它们只能在函数的主体中使用。

**外部形参名**

有时当你调用一个函数将每个形参进行命名是非常有用的，以表明你把每个实参传递给函数的目的。

如果你希望使用你函数的人在调用函数时提供形参名称，那除了本地形参名外，你还要为每个形参定义一个外部形参名称。你写一个外部形参名称在它所支持的本地形参名称之前,之间用一个空格来分隔:

1. func someFunction(externalParameterName localParameterName: Int) {
2. // function body goes here, and can use localParameterName
3. // to refer to the argument value for that parameter
4. }

|  |
| --- |
| 提示：如果您为形参提供一个外部形参名称，那么外部形参名必须在调用时使用。 |

举一个例子，考虑下面的函数，通过在它们之间插入第三个"joiner"字符串来连接两个字符串：

1. func join(s1: String, s2: String, joiner: String) -> String {
2. **return** s1 + joiner + s2
3. }

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：

1. join("hello", "world", ", ")
2. // returns "hello, world"

为了使这些字符串值的目的更为清晰,为每个join函数形参定义外部形参名称:

1. func join(string s1: String, toString s2: String, withJoiner joiner: String)
2. -> String {
3. **return** s1 + joiner + s2
4. }

在这个版本的join函数中，第一个形参的外部名称string，本地名称s1;第二个形参的外部名称toString，本地名称s2;第三个形参的外部名称是withJoiner，本地名称为joiner。

现在，您可以使用这些外部形参名称清楚明确地调用该函数：

1. join(string: "hello", toString: "world", withJoiner: ", ")
2. // returns "hello, world"

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

|  |
| --- |
| 注意：在别人第一次阅读你的代码不知道你函数形参目的时候，就要考虑到使用外部形参名称了。在调用函数的时候，如果每个形参的目的清晰明确的话，那你就无需指定外部形参名。 |

**外部参数名称速记**

如果你想为一个函数提供一个外部形参名，然而本地形参名已经使用了一个合适的名称了，那你就不需要两次书写该形参的名称。相反，你可以写一次名字，并用一个hash符号（＃）作为名称的前缀。这就告诉Swift使用名称相同的本地行参名称和外部形参名称。

这个例子定义了一个名为containsCharacter的函数,通过在本地形参名前添加hash符号(#)来定义外部形参名称。

1. func containsCharacter(#string: String, #characterToFind: Character) -> Bool {
2. **for** character **in** string {
3. **if** character == characterToFind {
4. **return** **true**
5. }
6. }
7. **return** **false**
8. }

该函数对形参名的选择使得其函数主题更加清晰易读，并且在调用该函数时也不会有歧义：

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

**默认形参值**

你可以为任何形参定义默认值以作为函数定义的一部分。如果已经定义了默认值，那么调用函数时就可以省略该行参。

|  |
| --- |
| 注意：请在函数形参列表的末尾放置带默认值的形参。这将确保所有函数调用都使用顺序相同的无默认值实参，并让在每种情况下清晰地调用相同的函数。 |

这里有一个早期的join函数，并为参数joiner设置了默认值：

1. func join(string s1: String, toString s2: String,
2. withJoiner joiner: String = " ") -> String {
3. **return** s1 + joiner + s2
4. }

如果在join函数调用时为joiner提供了字符串值，那么该字符串值可以用来连接两个字符串，就跟以前一样：

1. join(string: "hello", toString: "world", withJoiner: "-")
2. // returns "hello-world"

但是，如果函数调用时没有为joiner提供值，就会使用单个空格（" "）的默认值：

1. join(string: "hello", toString: "world")
2. // returns "hello world"

**有默认值的外部形参名**

在大多数情况下,为所有形参提供一个带默认值的外部名是非常有用的(因此要求)。如果在调用函数的时候提供了一个值，那么这将确保形参对应的实参有着明确的目的。

为了使这个过程更容易，当你自己没有提供外部名称时，Swift将为你定义的任何默认形参提供一个自动外部名。这个自动外部名和本地名一样，就像你已经在本地名前添加了hash符号（#）一样。

这里有一个早期join函数版本，没有为任何外部形参提供外部名，但仍然提供了joiner形参的默认值：

1. func join(s1: String, s2: String, joiner: String = " ") -> String {
2. **return** s1 + joiner + s2
3. }

在这种情况下，Swift为带默认值的形参提供了外部形参名，当调用该函数的时候，外部形参名必须让形参的目的明确无歧义：

1. join("hello", "world", joiner: "-")
2. // returns "hello-world"

|  |
| --- |
| 注意：在定义形参时，你可以通过使用下划线(\_)来代替显示外部名称。不过在适当的情况下，带有默认值形参的外部名通常是优先推荐的。 |

**可变形参**

一个可变形参可接受零个或多个指定类型的值。当函数被调用时，你可以使用可变形参来指定--形参可以用来传递任意数量的输入值。可通过在形参的类型名后边插入三个点符号（...）来编写可变形参。

传递至可变形参的值在函数主体内是以适当类型的数组存在的。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。

下边示例为任何长度的数字列表计算算术平均值：

1. func arithmeticMean(numbers: Double...) -> Double {
2. **var** total: Double = 0
3. **for** number **in** numbers {
4. total += number
5. }
6. **return** total / Double(numbers.count)
7. }
8. arithmeticMean(1, 2, 3, 4, 5)
9. // returns 3.0, which is the arithmetic mean of these five numbers
10. arithmeticMean(3, 8, 19)
11. // returns 10.0, which is the arithmetic mean of these three numbers

|  |
| --- |
| 注意：函数最多可以有一个可变形参，而且它必须出现在参数列表的最后，以避免使用多个形参调用函数引发歧义。如果你的函数有一个或多个带有默认值的形参，并且还有可变形参，请将可变形参放在所有默认形参之后，也就是的列表的最末尾。 |

**常量形参和变量形参**

函数的形参默认是常量。试图在函数体内改变函数形参的值会引发一个编译时错误。这意味着你不能错误地改变形参的值。

但是有时候，函数有一个形参值的变量副本是非常有用的。您可以指定一个或多个形参作为变量形参，从而避免在函数内部为自己定义一个新的变量。变量参数是变量而非常量,并给函数一个可修改的形参值副本。

在参数名称前用关键字var定义变量参数：

1. func alignRight(**var** string: String, count: Int, pad: Character) -> String {
2. let amountToPad = count - countElements(string)
3. **for** \_ **in** 1...amountToPad {
4. string = pad + string
5. }
6. **return** string
7. }
8. let originalString = "hello"
9. let paddedString = alignRight(originalString, 10, "-")
10. // paddedString is equal to "-----hello"
11. // originalString is still equal to "hello"

这个例子定义了一个新函数叫做alignRight,用于将一个输入字符串和更长的输出字符串右边缘对齐。所有左侧的空白使用指定的占位符来填充。在这个例子中，字符串"hello"被转化为字符串"-----hello"。

alignRight函数把输入的形参字符串定义成一个变量形参。这意味着字符串现在可以作为一个本地变量，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。

函数首先要找出有多少字符需要被添加到字符串的左侧，从而在整个字符串中靠右对齐。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。

|  |
| --- |
| 注意：你对变量形参所做的改变不会比调用函数更持久，并且在函数体外是不可见的。变量形参仅存在于函数调用的声明周期中。 |

**In-Out 形参**

如上描述，变量形参只能在函数本身内改变。如果你想让函数改变形参值，并想要在函数调用结束后保持形参值的改变，那你可以把形参定义为in-out形参。

通过在形参定义的开始添加inout关键字来编写in-out形参。In-Out形参有一个传递至函数的值，由函数修改，并从函数返回来替换原来的值。

你只能传递一个变量作为in-out形参对应的实参。你不能传递一个常量或者字面量作为实参，因为常量和字面量不能被修改。当你把变量作为实参传递给in out形参时，需要在直接在变量前添加 & 符号，以表明它可以被函数修改。

|  |
| --- |
| 提示：in-out参数不能有默认值，可变参数的参数也不能被标记为inout。如果您标记参数为inout，它不能同时被标记为var或let。 |

这里的一个叫做swapTwoInts函数，它有两个称为a和b的in-out整型形参：

1. func swapTwoInts(inout a: Int, inout b: Int) {
2. let temporaryA = a
3. a = b
4. b = temporaryA
5. }

swapTwoInts函数只是简单地交换a、b的值。该函数通过存储一个名为temporaryA临时常量的值，指定b的值到a，然后分配temporaryA到b执行该交换。

你可以通过两个Int类型的变量调用swapTwoInts函数，从而交换它们的值。需要注意的是当它们被传递给swapTwoInts函数时，someInt和anotherInt名称前要加上前缀符号&：

1. **var** someInt = 3
2. **var** anotherInt = 107
3. swapTwoInts(&someInt, &anotherInt)
4. println("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
5. // prints "someInt is now 107, and anotherInt is now 3"

上面的例子表明，someInt和anotherInt的原始值由swapTwoInts函数进行了修改，即使它们定义在函数外部。

|  |
| --- |
| 注意：In-out形参不同于从函数返回一个值。上边swapTwoInts例子没有定义返回类型或者返回值，但它仍然会修改someInt和anotherInt的值。对函数来说，In-out形参是一个影响函数主体范围外的可选方式。 |

**函数类型**

每一个函数都有特定的函数类型，由函数的形参类型和返回类型组成。例如：

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. }

这个例子中定义了两个简单的数学函数addTwoInts和multiplyTwoInts。每个函数接受两个int值，并返回一个int值，执行适当的数学运算并返回结果。

这两个函数的类型都是(Int, Int)->Int。可以解读为："这个函数类型，它有两个Int类型形参，并返回一个Int类型的值。"

下面是另一个例子，该函数没有形参或返回值：

1. func printHelloWorld() {
2. println("hello, world")
3. }

这个函数的类型是（）->（），或者"没有形参的函数，并返回void。"没有指明返回值的函数通常会返回void，在swift中相当于一个空元组，显示为（）。

**使用函数类型**

在swift中您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，并为变量指定一个对应的函数：

1. **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。"

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

现在你可以使用mathFunction来调用指定的函数：

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

具有相同匹配类型的不同函数可以被赋给同一个变量，和非函数类型一样：

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

与其他类型一样,当你给函数赋一个常量或者变量时，你可以让Swift去推断函数的类型。

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

**作为形参类型的函数类型**

您可以使用一个函数类型，如(Int, Int)->Int作为另一个函数的形参类型。这使你预留了一个函数的某些方面的函数实现，让调用者提供的函数时被调用。

下边的例子打印了上边的数学函数的结果：

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

这个例子中定义了一个名为printMathResult函数，它有三个形参。第一个形参名为mathFunction，类型为(Int, Int)->Int。您可以传递任何同类型的函数作为第一个形参的实参。第二和第三个参数a、b都是int类型。被用来作为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调用了提供的函数，打印的结果是8。

printMathResult的作用是打印调用适当类型的数学函数的结果。该函数真正实现了什么并不重要--它只关心函数的类型是正确的。这使得printMathResult以一种安全类型的方式把自身的功能转换至函数的调用者。

**作为返回类型的函数类型**

你可以将一个函数类型作为另一个函数的返回类型。你可以在返回函数的返回箭头(->) 后立即编写一个完整的函数类型来实现。

下面的例子定义了两个简单的函数调用stepForward和stepBackward。stepForward函数返回一个输入值+1的结果，而stepBackward函数返回一个输入值-1的结果。这两个函数都有一个相同的(Int) -> Int类型 ：

1. func stepForward(input: Int) -> Int {
2. **return** input + 1
3. }
4. func stepBackward(input: Int) -> Int {
5. **return** input - 1
6. }

这里有一个chooseStepFunction函数，它的返回类型是"函数类型(Int) -> Int"。chooseStepFunction基于名为backwards的布尔形参返回stepBackward或stepForward函数:

1. func chooseStepFunction(backwards: Bool) -> (Int) -> Int {
2. **return** backwards ? stepBackward : stepForward
3. }

你现在可以使用chooseStepFunction获取一个函数,可能是递增函数或递减函数:

1. **var** currentValue = 3
2. let moveNearerToZero = chooseStepFunction(currentValue > 0)
3. // moveNearerToZero now refers to the stepBackward() function

前面的例子可以计算出是否需要通过递增或者递减来让currentValue变量趋于零。currentValue的初始值为3，这意味着currentValue > 0返回为真，并且chooseStepFunction返回stepBackward函数。返回函数的引用存储在一个名为moveNearerToZero的常量里。

如今moveNearerToZero执行了正确的功能，就可以用来计数到零：

1. println("Counting to zero:")
2. // Counting to zero:
3. **while** currentValue != 0 {
4. println("\(currentValue)... ")
5. currentValue = moveNearerToZero(currentValue)
6. }
7. println("zero!")
8. // 3...
9. // 2...
10. // 1...
11. // zero!

**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，在全局作用域中定义。其实你还可以在其他函数体中定义函数，被称为嵌套函数。

嵌套函数默认对外界是隐藏的，但仍然可以通过它们包裹的函数调用和使用它。enclosing function也可以返回一个嵌套函数，以便在其他作用域中使用嵌套函数。

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

1. func chooseStepFunction(backwards: Bool) -> (Int) -> Int {
2. func stepForward(input: Int) -> Int { **return** input + 1 }
3. func stepBackward(input: Int) -> Int { **return** input - 1 }
4. **return** backwards ? stepBackward : stepForward
5. }
6. **var** currentValue = -4
7. let moveNearerToZero = chooseStepFunction(currentValue > 0)
8. // moveNearerToZero now refers to the nested stepForward() function
9. **while** currentValue != 0 {
10. println("\(currentValue)... ")
11. currentValue = moveNearerToZero(currentValue)
12. }
13. println("zero!")
14. // -4...
15. // -3...
16. // -2...
17. // -1...
18. // zero!