

Swift 编程语言

可能是最用心的翻译了吧。

闭包

快速检索 [\[点击展开\]](#)

*闭包*是可以在你的代码中被传递和引用的功能性独立模块。Swift 中的闭包和 C 以及 Objective-C 中的 blocks 很像，还有其他语言中的匿名函数也类似。

闭包能够捕获和存储定义在其上下文中的任何常量和变量的引用，这也就是所谓的*闭合并包裹*那些常量和变量，因此被称为“闭包”，Swift 能够为你处理所有关于捕获的内存管理的操作。

注意

不必担心你不熟悉“捕获”这个概念，在后面的[捕获值](#)中会对其进行详细介绍。

在[函数](#)章节中有介绍的全局和内嵌函数，实际上是特殊的闭包。闭包符合如下三种形式中的一种：

- 全局函数是一个有名字但不会捕获任何值的闭包；
- 内嵌函数是一个有名字且能从其上层函数捕获值的闭包；
- 闭包表达式是一个轻量级语法所写的可以捕获其上下文中常量或变量值的没有名字的闭包。

Swift 的闭包表达式拥有简洁的风格，鼓励在常见场景中实现简洁，无累赘的语法。常见的优化包括：

- 利用上下文推断形式参数和返回值的类型；
- 单表达式的闭包可以隐式返回；
- 简写实际参数名；
- 尾随闭包语法。

闭包表达式

内嵌函数，在内嵌函数中有介绍，一种在较复杂的函数中方便命名和定义独立代码块的手段。总之，有时候对于写更简短的没有完整定义和命名的类函数构造非常有用，尤其是在你处理一些函数时调用其他函数作为该函数的参数时。

闭包表达式是一种在简短行内就能写完闭包的语法。闭包表达式为了缩减书写长度又不失易读明晰而提供了一系列的语法优化。下边的闭包表达式栗子通过使用几次迭代展示 `sorted(by:)` 方法的精简来展示这些优化，每一次都让相同的功能性更加简明扼要。

Sorted 方法

Swift 的标准库提供了一个叫做 `sorted(by:)` 的方法，会根据你提供的排序闭包将已知类型的数组的值进行排序。一旦它排序完成，`sorted(by:)` 方法会返回与原数组类型大小完全相同的一个新数组，该数组的元素是已排序好的。原始数组不会被 `sorted(by:)` 方法修改。

下面这个闭包表达式的栗子使用 `sorted(by:)` 方法按字母排序顺序来排序一个 `String` 类型的数组。这是将被排序的初始数组：

```
1 let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
```

`sorted(by:)` 方法接收一个接收两个与数组内容相同类型的实际参数的闭包，然后返回一个 `Bool` 值来说明第一个值在排序后应该出现在第二个值的前边还是后边。如果第一个值应该出现在第二个值之前，排序闭包需要返回 `true`，否则返回 `false`

这个栗子对一个 `String` 类型的数组进行排序，因此排序闭包需为一个 `(String, String) -> Bool` 的类型函数。

提供排序闭包的一个方法是写一个符合其类型需求的普通函数，并将它作为 `sorted(by:)` 方法的形式参数传入：

```
1 func backward(_ s1: String, _ s2: String) -> Bool {
2     return s1 > s2
3 }
4 var reversedNames = names.sorted(by: backward)
5 // reversedNames is equal to ["Ewa", "Daniella", "Chris", "
  Barry", "Alex"]
```

如果第一个字符串 `s1` 大于第二个字符串 `s2`，`backwards(_:_:)` 函数将返回 `true`，这意味着 `s1` 应该在排序数组中排在 `s2` 的前面。对于在字符串中的字符，“比谁大”意思是“比较谁排在字母顺序的后面”。这意味着字母“`B`”是“大于”字母“`A`”的，并且字符串“`Tom`”大于字符串“`Tim`”。如果按照相反的字母顺序表的话，“`Barry`”应该处于“`Alex`”的前面，依次类推。

总之，这样来写本质上只是一个单一表达式函数(`a > b`)是非常啰嗦的。在这个栗子中，我们更倾向于使用闭包表达式在行内写一个简单的闭包。

闭包表达式语法

闭包表达式语法有如下的一般形式：

```
1 { (parameters) -> (return type) in
2     statements
3 }
```

闭包表达式语法能够使用常量形式参数、变量形式参数和输入输出形式参数，但不能提供默认值。可变形式参数也能使用，但需要在形式参数列表的最后面使用。元组也可被用来作为形式参数和返回类型。

下面这个栗子展示一个之前 `backward(_:_:)` 函数的闭包表达版本：

```
1 reversedNames = names.sorted(by: { (s1: String, s2: String)
2     -> Bool in
```

```
3     return s1 > s2
    })
```

需要注意的是行内闭包的形式参数类型和返回类型的声明与 `backwards(_:_:)` 函数的申明相同。在这两个方式中，都书写成 `(s1: String, s2: String) -> Bool`。总之对于行内闭包表达式来说，形式参数类型和返回类型都应写在花括号内而不是花括号外面。

闭包的函数整体部分由关键字 `in` 导入，这个关键字表示闭包的形式参数类型和返回类型定义已经完成，并且闭包的函数体即将开始。

闭包的函数体特别短以至于能够只用一行来书写：

```
1 reversedNames = names.sorted(by: { (s1: String, s2: String)
    -> Bool in return s1 > s2 } )
```

示例中 `sorted(by:)` 方法的整体部分调用保持不变，一对圆括号仍然包裹函数的所有实际参数。然而，这些实际参数中的一个变成了现在的行内闭包。

从语境中推断类型

因排序闭包为实际参数来传递给函数，故 Swift 能推断它的形式参数类型和返回类型。`sorted(by:)` 方法期望它的第二个形式参数是一个 `(String, String) -> Bool` 类型的函数。这意味着 `(String, String)` 和 `Bool` 类型不需要被写成闭包表达式定义中的一部分，因为所有的类型都能被推断，返回箭头 (`->`) 和围绕在形式参数名周围的括号也能被省略：

```
1 reversedNames = names.sorted(by: { s1, s2 in return s1 > s2
    } )
```

当把闭包作为行内闭包表达式传递给函数，形式参数类型和返回类型都可以被推断出来。所以说，当闭包被用作函数的实际参数时你都不需要

用完整格式来书写行内闭包。

然而，如果你希望的话仍然可以明确类型，并且在读者阅读你的代码时如果它能避免可能存在的歧义的话还是值得的。在这个 `sorted(by:)` 方法的栗子中，闭包的目的很明确，即排序被替换。对读者来说可以放心的假设闭包可能会使用 `String` 值，因为它正帮一个字符串数组进行排序。

从单表达式闭包隐式返回

单表达式闭包能够通过从它们的声明中删掉 `return` 关键字来隐式返回它们单个表达式的结果，前面的栗子可以写作：

```
1 reversedNames = names.sorted(by: { s1, s2 in s1 > s2 } )
```

这里，`sorted(by:)` 方法的第二个实际参数的函数类型已经明确必须通过闭包返回一个 `Bool` 值。因为闭包的结构包涵返回 `Bool` 值的单一表达式 (`s1 > s2`)，因此没有歧义，并且 `return` 关键字能够被省略。

简写的实际参数名

Swift 自动对行内闭包提供简写实际参数名，你也可以通过 `$0`，`$1`，`$2` 等名字来引用闭包的实际参数值。

如果你在闭包表达式中使用这些简写实际参数名，那么你可以在闭包的参数列表中忽略对其的定义，并且简写实际参数名的数字和类型将会从期望的函数类型中推断出来。`in` 关键字也能被省略，因为闭包表达式完全由它的函数体组成：

```
1 reversedNames = names.sorted(by: { $0 > $1 } )
```

这里，`$0` 和 `$1` 分别是闭包的第一个和第二个 `String` 实际参数。

运算符函数

实际上还有一种更简短的方式来撰写上述闭包表达式。Swift 的 `String` 类型定义了关于大于号（`>`）的特定字符串实现，让其作为一个有两个 `String` 类型形式参数的函数并返回一个 `Bool` 类型的值。这正好与 `sorted(by:)` 方法的第二个形式参数需要的函数相匹配。因此，你能简单地传递一个大于号，并且 Swift 将推断你想使用大于号特殊字符串函数实现：

```
1 reversedNames = names.sorted(by: >)
```

想要了解更多有关运算符函数,请阅览[运算符函数](#)

尾随闭包

如果你需要将一个很长的闭包表达式作为函数最后一个实际参数传递给函数，使用尾随闭包将增强函数的可读性。尾随闭包是一个被书写在函数形式参数的括号外面（后面）的闭包表达式：

```
1 func someFunctionThatTakesAClosure(closure:() -> Void){
2     //function body goes here
3 }
4
5 //here's how you call this function without using a trailing closure
6
7
8 someFunctionThatTakesAClosure({
9     //closure's body goes here
10 })
11
12 //here's how you call this function with a trailing closure instead
13
14
15 someFunctionThatTakesAClosure() {
    // trailing closure's body goes here
}
```

}

注意

如果闭包表达式被用作函数唯一的实际参数并且你把闭包表达式用作尾随闭包，那么调用这个函数的时候你就不需要在函数的名字后面写一对圆括号（ ）。

来自于上文的 闭包表达式 一节的字符串排列闭包也可以作为一个尾随闭包被书写在 `sorted(by:)` 方法的括号外面：

```
1 reversedNames = names.sorted() { $0 > $1 }
```

如果闭包表达式作为函数的唯一实际参数传入，而你又使用了尾随闭包的语法，那你就不需要在函数名后边写圆括号了：

```
1 reversedNames = names.sorted { $0 > $1 }
```

当闭包很长以至于不能被写成一行时尾随闭包就显得非常有用。举个例子，Swift 的 `Array` 类型中有一个以闭包表达式为唯一的实际参数的 `map(_:)` 方法。数组中每一个元素调用一次该闭包，并且返回该元素所映射的值（有可能是其他类型）。具体的映射方式和返回值的类型有闭包来指定。

在给数组中每一个成员提供闭包时，`map(_:)` 方法返回一个新的数组，数组中包涵与原数组一一对应的映射值。

使用一个带有尾随闭包的 `map(_:)` 方法将一个包含 `Int` 值的数组转换成一个包含 `String` 值的数组。这个数组 `[16,58,510]` 被转换成一个新的数组 `["OneSix","FiveEight","FiveOneZero"]`：

```
1 let digitNames = [  
2     0: "Zero",1: "One",2: "Two", 3: "Three",4: "Four",  
3     5: "Five",6: "Six",7: "Seven",8: "Eight",9: "Nine"  
4 ]  
5  
6 let numbers = [16,58,510]
```

上面的代码创建了一个整数数字到它们英文名字之间的映射字典，同时定义了一个将被转换成字符串的整数数组。

你现在可以通过传递一个闭包表达式到数组的 `map(_:)` 方法中作为尾随闭包，利用 `number` 数组来创建一个 `String` 类型的数组。值得注意的是 `number.map` 不需要在 `map` 后面加任何圆括号，因为 `map(_:)` 方法仅仅只有一个形式参数，这个形式参数将以尾随闭包的形式来书写：

```
1 let strings = numbers.map {
2     (number) -> String in
3     var number = number
4     var output = ""
5     repeat {
6         output = digitNames[number % 10]! + output
7         number /= 10
8     } while number > 0
9     return output
10 }
11 // strings is inferred to be of type [String]
12 // its value is ["OneSix", "FiveEight", "FiveOneZero"]
```

`map(_:)` 方法在数组中为每一个元素调用了一次闭包表达式。你不需要指定闭包的输入形式参数 `number` 的类型，因为它能从数组中将被映射的值来做推断。

这个栗子中，变量 `number` 以闭包的 `number` 形式参数初始化，这样它就可以在闭包结构内部直接被修改。（函数和闭包的形式参数是常量。）闭包表达式同样指定返回 `String` 类型，来表明存储映射值的 `output` 数组也为 `String` 类型。

闭包表达式建立一个命名为 `output` 的字符串，每调用一次就命名一次。它通过利用余数运算符（`number % 10`）计算了 `number` 的最后一位数字，并且用这个数字在 `digitNames` 字典中找到相对应的字符串。这个闭包能够被用来创建一个任何一个大于零的整数表示的字符串。

注意

`digitNames` 的下标紧跟着一个感叹号(`!`), 因为字典下标返回一个可选值, 表明即使该 `key` 不存在也不会查找失败。上述这个栗子中, 它保证了 `number % 10` 可以总是作为字典 `digitNames` 的下标 `key` , 因此一个感叹号可以被用作强制展开 (`force-unwrap`) 存储在可选返回值下标项的 `String` 值。

从字典 `digitNames` 获取的字符串被添加到 `output` 的前面, 有效的逆序建立了一个字符串版本的数字。(表达式 `number % 10` 得出 `16` 中的 `6` , `58` 中的 `8` , `510` 中的 `0`)

`number` 变量随后被 `10` 分开, 因为他是一个整数, 未除尽部分被忽略。因此 `16` 便得 `1` , `58` 得 `5` , `510` 得 `51` 。

整个过程反复运行, 直到 `number /= 10` 是等于 `0` , 这时闭包会将字符串 `output` 输出, 并且通过 `map(_:)` 方法添加到输出数组中。

在上面这个例子中尾随闭包语法在函数后整洁地封装了具体的闭包功能, 而不再需要将整个闭包包裹在 `map(_:)` 方法的括号内。

捕获值

一个闭包能够从上下文捕获已被定义的常量和变量。即使定义这些常量和变量的原作用域已经不存在, 闭包仍能够在其函数体内引用和修改这些值。

在 Swift 中, 一个能够捕获值的闭包最简单的模型是内嵌函数, 即被书写在另一个函数的内部。一个内嵌函数能够捕获外部函数的实际参数并且能够捕获任何在外部函数的内部定义了的常量与变量。

这里有个命名为 `makeIncrement` 的函数栗子, 其中包含了一个名叫 `incrementer` 一个内嵌函数。这个内嵌 `incrementer()` 函数能在它的上下文捕获两个值, `runningTotal` 和 `amount` 。在捕获这些值后, 通过 `makeIncrement` 将 `incrementer` 作为一个闭包返回, 每一次调用 `incrementer` 时, 将以 `amount` 作为增量来增加 `runningTotal` :

```
1 func makeIncrementer(forIncrement amount: Int) -> () -> Int
2 {
3     var runningTotal = 0
4     func incrementer() -> Int {
5         runningTotal += amount
6         return runningTotal
7     }
8     return incrementer
9 }
```

`makeIncrementer` 的返回类型是 `() -> Int`，意思就是比起返回一个单一的值，它返回的是一个函数。这个函数没有返回任何形式参数，每调用一次就返回一个 `Int` 值。想要了解更多关于函数是如何返回另一个函数的，请参照[函数类型作为返回类型](#)。

`makeIncrementer(forIncrement:)` 函数定义了一个叫 `runningTotal` 的整数变量，用来存储当前增加的总量，该值通过 `Incrementer` 返回。这个变量用 `0` 初始化。

`makeIncrementer(forIncrement:)` 函数只有一个在外部命名为 `Incrementer`，局部命名为 `amount` 的 `Int` 形式参数。在每次调用 `incrementer` 函数时，实际参数值通过形式参数指定 `runningTotal` 增加多少。

`makeIncrementer` 定义了一个名叫 `incrementer` 的内嵌函数，表明实际增加量，这个函数直接把 `amount` 增加到 `runningTotal`，并且返回结果。

当我们单看这个函数时，会发现内嵌函数 `incrementer()` 不同寻常：

```
1 func incrementer() -> Int {
2     runningTotal += amount
3     return runningTotal
4 }
```

`incrementer()` 函数是没有任何形式参数, `runningTotal` 和 `amount` 不是来自于函数体的内部, 而是通过捕获主函数的 `runningTotal` 和 `amount` 把它们内嵌在自身函数内部供使用。当调用 `makeIncrementer` 结束时通过引用捕获来确保不会消失, 并确保了在下次再次调用 `incrementer` 时, `runningTotal` 将继续增加。

注意

作为一种优化, 如果一个值没有改变或者在闭包的外面, Swift 可能会使用这个值的拷贝而不是捕获。

Swift也处理了变量的内存管理操作, 当变量不再需要时会被释放。

这有个使用 `makeIncrementer` 的栗子:

```
1 let incrementByTen = makeIncrementer(forIncrement: 10)
```

这个例子定义了一个叫 `incrementByTen` 的常量, 该常量指向一个每次调用会加 10 的函数。调用这个函数多次得到以下结果:

```
1 incrementByTen()  
2 //return a value of 10  
3 incrementByTen()  
4 //return a value of 20  
5 incrementByTen()  
6 //return a value of 30
```

如果你建立了第二个 `incrementer`, 它将会有一个新的、独立的 `runningTotal` 变量的引用:

```
1 let incrementBySeven = makeIncrementer(forIncrement: 7)  
2 incrementBySeven()  
3 // returns a value of 7
```

再次调用原来增量器 (`incrementByTen`) 继续增加它自己的变量 `runningTotal` 的值, 并且不会影响 `incrementBySeven` 捕获的变量

`runningTotal` 值:

```
1   incrementByTen()  
2   //returns a value of 40
```

注意

如果你分配了一个闭包给类实例的属性，并且闭包通过引用该实例或者它的成员来捕获实例，你将在闭包和实例间建立一个强引用环。

Swift将使用捕获列表来打破这种强引用环。更多信息请参考[闭包的强引用环](#)。

闭包是引用类型

在上面例子中，`incrementBySeven` 和 `incrementByTen` 是常量，但是这些常量指向的闭包仍可以增加已捕获的变量 `runningTotal` 的值。这是因为函数和闭包都是引用类型。

无论你在什么时候赋值一个函数或者闭包给常量或者变量，你实际上都是将常量和变量设置为对函数和闭包的引用。这上面这个例子中，闭包选择 `incrementByTen` 指向一个常量，而不是闭包它自身的内容。这也意味着你赋值一个闭包到两个不同的常量或变量中，这两个常量或变量都将指向相同的闭包：

```
1   let alsoIncrementByTen = incrementByTen  
2   alsoIncrementByTen()  
3   //return a value of 50
```

逃逸闭包

当闭包作为一个实际参数传递给一个函数的时候，我们就说这个闭包逃逸了，因为它可以在函数返回之后被调用。当你声明一个接受闭包作为形式参数的函数时，你可以在形式参数前写 `@escaping` 来明确闭包是允许逃逸的。

闭包可以逃逸的一种方法是被储存在定义于函数外的变量里。比如说，很多函数接收闭包实际参数来作为启动异步任务的回调。函数在启动任务后返回，但是闭包要直到任务完成——闭包需要逃逸，以便于稍后调用。举例来说：

```
1 var completionHandler: [() -> Void] = []
2 func someFunctionWithEscapingClosure(completionHandler: @es
3   caping () -> Void) {
4     completionHandler.append(completionHandler)
5 }
```

函数 `someFunctionWithEscapingClosure(_:)` 接收一个闭包作为实际参数并且添加它到声明在函数外部的数组里。如果你不标记函数的形式参数为 `@escaping`，你就会遇到编译时错误。

让闭包 `@escaping` 意味着你必须在闭包中显式地引用 `self`，比如说，下面的代码中，传给 `someFunctionWithEscapingClosure(_:)` 的闭包是一个逃逸闭包，也就是说它需要显式地引用 `self`。相反，传给 `someFunctionWithNonescapingClosure(_:)` 的闭包是非逃逸闭包，也就是说它可以隐式地引用 `self`。

```
1 func someFunctionWithNonescapingClosure(closure: () -> Voi
2   d) {
3     closure()
4 }
5
6 class SomeClass {
7     var x = 10
8     func doSomething() {
9         someFunctionWithEscapingClosure { self.x = 100 }
10        someFunctionWithNonescapingClosure { x = 200 }
11    }
12 }
13
14 let instance = SomeClass()
```

```
15 instance.doSomething()
16 print(instance.x)
17 // Prints "200"
18
19 completionHandler.first?()
20 print(instance.x)
    // Prints "100"
```

自动闭包

*自动闭包*是一种自动创建的用来把作为实际参数传递给函数的表达式打包的闭包。它不接受任何实际参数，并且当它被调用时，它会返回内部打包的表达式值。这个语法的好处在于通过写普通表达式代替显式闭包而使你省略包围函数形式参数的括号。

调用一个带有自动闭包的函数是很常见的，但实现这类函数就不那么常见了。比如说，`assert(condition:message:file:line:)` 函数为它的 `condition` 和 `message` 形式参数接收一个自动闭包；它的 `condition` 形式参数只有在调试构建时才评判，而且 `message` 形式参数只有在 `condition` 是 `false` 时才评判。

自动闭包允许你延迟处理，因此闭包内部的代码直到你调用它的时候才会运行。对于有副作用或者占用资源的代码来说很有用，因为它可以允许你控制代码何时才进行求值。下面的代码展示了闭包如何延迟求值。

```
1 var customersInLine = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
2
3 print(customersInLine.count)
4 // Prints "5"
5
6 let customerProvider = { customersInLine.remove(at: 0) }
7 print(customersInLine.count)
8 // Prints "5"
9
10 print("Now serving \(customerProvider())!")
```

```
11 // Prints "Now serving Chris!"
12 print(customersInLine.count)
    // Prints "4"
```

尽管 `customersInLine` 数组的第一个元素以闭包的一部分被移除了，但任务并没有执行直到闭包被实际调用。如果闭包永远不被调用，那么闭包里边的表达式就永远不会求值。注意 `customerProvider` 的类型不是 `String` 而是 `() -> String` ——一个不接受实际参数并且返回一个字符串的函数。

当你传一个闭包作为实际参数到函数的时候，你会得到与延迟处理相同的行为。

```
1 // customersInLine is ["Alex", "Ewa", "Barry", "Daniella"]
2 func serve(customer customerProvider: () -> String) {
3     print("Now serving \(customerProvider())!")
4 }
5 serve(customer: { customersInLine.remove(at: 0) } )
6 // Prints "Now serving Alex!"
```

上边的函数 `serve(customer:)` 接收一个明确的返回下一个客户名称的闭包。下边的另一个版本的 `serve(customer:)` 执行相同的任务但是不使用明确的闭包而是通过 `@autoclosure` 标志标记它的形式参数使用了自动闭包。现在你可以调用函数就像它接收了一个 `String` 实际参数而不是闭包。实际参数自动地转换为闭包，因为 `customerProvider` 形式参数的类型被标记为 `@autoclosure` 标记。

```
1 // customersInLine is ["Ewa", "Barry", "Daniella"]
2 func serve(customer customerProvider: @autoclosure () -> String) {
3     print("Now serving \(customerProvider())!")
4 }
5
6 serve(customer: customersInLine.remove(at: 0))
    // Prints "Now serving Ewa!"
```

注意

滥用自动闭包会导致你的代码难以读懂。上下文和函数名应该写清楚求值是延迟的。

如果你想要自动闭包允许逃逸，就同时使用 `@autoclosure` 和 `@escaping` 标志。`@escaping` 标志在上边的逃逸闭包里有详细的解释。

```
1 // customersInLine is ["Barry", "Daniella"]
2 var customerProviders: [() -> String] = []
3 func collectCustomerProviders(_ customerProvider: @autoclo
4 sure @escaping () -> String) {
5     customerProviders.append(customerProvider)
6 }
7 collectCustomerProviders(customersInLine.remove(at: 0))
8 collectCustomerProviders(customersInLine.remove(at: 0))
9
10 print("Collected \(customerProviders.count) closures.")
11 // Prints "Collected 2 closures."
12 for customerProvider in customerProviders {
13     print("Now serving \(customerProvider())!")
14 }
15 // Prints "Now serving Barry!"
    // Prints "Now serving Daniella!"
```

上边的代码中，不是调用传入后作为 `customerProvider` 实际参数的闭包，`collectCustomerProviders(_:)` 函数把闭包追加到了 `customerProviders` 数组的末尾。数组声明在函数的生效范围之外，也就是说数组里的闭包有可能在函数返回之后执行。结果，`customerProvider` 实际参数的值必须能够逃逸出函数的生效范围。