

☰

📄 闭包

4.2 ▾

📖 Swift 编程语言 / 🇨🇳 闭包

这是一篇社区协同翻译的文章，你可以点击右边区块信息里的『改进』按钮向译者提交改进建议。

『闭包』是独立的代码块，可以在你代码中随意传递和使用。Swift 中的闭包与 Objective-C/C 中的 Block、其他编程语言中的匿名函数相似。

闭包可以从定义它们的代码的上下文中捕获和存储任何变量。这也被称为这些变量和常量被暂时关闭使用。并且 Swift 负责处理你所捕获的内存进行管理。

注意

不要担心，如果不能理解闭包的捕获的概念。这里解释了详细内容 [值捕获](#)。

在 [函数](#) 章节中介绍的全局和嵌套函数实际上也是特殊的闭包。闭包采取如下三种形式之一：

- 全局函数是一个有名字但不会捕获任何值的闭包。
- 嵌套函数是一个有名字并且可以捕获其封闭函数域内值的闭包。
- 闭包表达式是一个用轻量语法所写的可以捕获其上下文中变量或常量值的匿名闭包。

Swift 的闭包表达式具有干净、清晰的风格，并鼓励在常见场景中进行语法优化使其简明、不杂乱。这些优化主要包括：

- [利用上下文推断参数和返回值类型](#)
- [单语句表达式的闭包可以隐式返回结果](#)
- [参数名称缩写](#)
- [尾随闭包语法](#)

## 闭包表达式

[嵌套函数](#) 是一个在大功能中进行命名和定义自包含代码块的便捷方式。但是，有的时候它在

Infinity

翻译进度

📄 23

分块数量

👥 8

参与人数



ForC 翻译于 5个月前

👍 0

重译

由 Summer 审阅



chai 翻译于 5个月前

👍 0

重译

由 Aufree 审阅



写一些没有完整声明和命名的类似函数结构的更短版本时很有用。当你处理一些将函数作为它的一个或多个参数的函数时这种方式尤其有用。

闭包表达式 是一种用简短、集中的语法构建内联闭包的方式。闭包表达式提供了几种语法优化的方式，使其能够写出简短的闭包而又不失去闭包函数的可读性。下面的闭包表达式示例，通过在几次迭代中不断改善 `sorted(by:)` 方法的方式来说明这些优化，每一次迭代都用更简洁的方式描述了相同的功能。

## 方法排序

Swift的基础库提供了一个名字叫做 `sorted(by:)` API，它通过你编写的一个闭包来进行对数组进行排序。当完成所有排序代码，`sorted(by:)` 方法会返回一个与旧数组相同规格和相同类型的新数组，并且每个元素都会在正确的位置。最开始的数组也不会通过 `sorted(by:)` 被修改。

下面的闭包示例使用了 `sorted(by:)` 方法进行了对于 `String` 类型进行反向排序。这是最开始将要被排序的数组：

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

`sorted(by:)` 方法接受一个闭包表达式，闭包表达式接受两个相同类型的数组元素，并且返回 `Bool` 布尔值来告诉是否第一个值应该在第二个值的前面还是后面。如果这个第一个值应该在第二个值 前面 则返回 `true`，反之返回 `false`。

这个例子是对 `String` 类型的数组进行排序，因此这个闭包需要是 `(String, String) -> Bool` 类型的函数。

提供该排序闭包的一种方法是写正确类型的函数，并且作为参数传入 `sorted(by:)` 方法中。如下：

```
func backward(_ s1: String, _ s2: String) -> Bool {
    return s1 > s2
}

var reversedNames = names.sorted(by: backward)

// reversedNames 会等于 ["Ewa", "Daniella", "Chris", "Barry", "Alex"]
```

如果第一个字符串「s1」大与第二个字符串「s2」，`backward(_:_:)` 函数将返回 `true`，指示 `s1` 在这个排序数组中位置应该在 `s2` 的前面。对于字符串来说，“大于”的意思就是“在字母表中出现较后”。这意思是说字母 `"B"` 是大于字母 `"A"` 的，也就是说字符串 `"Tom"` 是大于 `"Tim"` 的。这是一个反向排序的例子 `"Barry"` 将出现在 `"Alex"` 前面，依此类推。



chai 翻译于 5个月前

👍 0

重译

由 Aufree 审阅



ForC 翻译于 5个月前

👍 0

重译

由 Aufree 审阅



ForC 翻译于 5个月前

👍 0

重译

由 Aufree 审阅

然而，使用单一表达式的函数「a>b」是一种相当冗长的方式。在本例中最好写法是使用闭包表达式内联的方式编写一个排序闭包。

## 闭包表达式语法

闭包表达式语法基本组成如下：

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

*parameters* 在闭包表达式中当作入参，但它们没有默认值。如果你参数类型为入参，则可以在闭包中使用可变类型的方法。元组也可以当做参数和返回值。

下面的例子展示了上面 `backward(_:_:)` 函数的闭包表达式版本：

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

注意这个内联闭包与 `backward(_:_:)` 函数的入参和返回值是相同的。在这两种情况下，它被写成 `(s1: String, s2: String) -> Bool`。然而，对于内联闭包表达式，参数和返回值被写在花括号‘内部’，而不是外部。

闭包表达式主体部分开始于关键字 `in`。这个关键字也代表这个闭包的入参和返回值已经声明结束，主体部分将要开始。

因为上面闭包表达式的主体部分比较短，甚至可以写成一行。

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

这也说明了 `sorted(by:)` 方法总体调用保持不变。一对括号仍然包含该方法的入参。然而，现在参数却是一个内联闭包了。

## 通过上下文推测类型

因为这个排序闭包是作为一个方法的参数，Swift 能够推断出这个闭包的参数和返回值。 `sorted(by:)` 方法将被一个字符串数组调用，以至于方法参数的函数的类型一定是 `(String, String) -> Bool`。这也意味 `(String, String)` 和 `Bool` 类型可以不需要作为闭包定义的一部分。也正因为推断出了所有入参和返回值，返回的符号 (`->`) 和入参周围的括号也可以被省略。简写后如下：



ForC 翻译于 5个月前

👍 0 重译

由 Aufree 审阅



ForC 翻译于 5个月前

👍 0 重译

由 Aufree 审阅

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

当闭包以内联的形式作为一个函数或者方法的参数传入时，始终可以被推断出入参和返回值的类型。因此，当内联闭包被方法或函数当作参数使用时，你也不需要以完整形式去写这个内联闭包。

尽管如此，你也按照个人意愿将参数类型显示出来。如果这样的做法可以让你的读者对你的代码不保持模棱两可，显示参数类型也是被鼓励使用的。在 `sorted(by:)` 方法的调用情况下，闭包的目的是帮助一个数组进行排序，也正因为这个闭包在字符串数组中被使用，可以让读者轻松推断出这个闭包正在使用 `String` 类型的值。

## 单一闭包表达式隐式返回

单一闭包表达式可以省略声明 `return` 关键字来返回单一表达式的结果，上一个事例省略后如下：

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

这里，这个 `sorted(by:)` 方法函数类型清晰的表明闭包必须返回一个 `Bool` 类型的值。也因为这个闭包内部包含一个单一表达式 ( `s1 > s2` ) 并且返回一个明确的 `Bool` 类型的值，因此关键字 `return` 可以被省略。

## 缩写参数名

Swift 自动为内联闭包提供了参数名缩写写法，这里可以使用 `$0`，`$1`，`$2` 等来代替闭包的参数。

如果你在闭包表达式中使用了缩写写法，你就可以省略闭包中的参数声明部分，并且这个缩写参数的值和类型也会通过函数预期类型推断出来。`in` 关键字也可以被省略，因为这个闭包表达式已经通过主体完全构建出来了。缩写后如下：

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

这里，`$0` 和 `$1` 的值指的是第一个和第二个 `String` 的参数

## 运算符方法

实际上还有一种 *更简短* 的方式来编写上面例子中的闭包表达式。Swift 的 `String` 类型将其大于运算符 ( `>` ) 的字符串特定实现为具有两个 `String` 类型参数的方法，并返回一个 `Bool` 类型的值。而这正好与 `sorted(by:)` 方法的参数需要的函数类型相符合。因此，你可以简单地传递一个大于运算符，Swift 可以自动推断出你想使用其特定于字



ForC 翻译于 5个月前

👍 0

重译

由 Aufree 审阅

字符串的实现：

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

更多关于运算符方法的内容请查看 [运算符方法](#).

## 尾随闭包

如果你需要将闭包表达式作为函数的最后一个参数传入函数，并且这个闭包非常长，这样的情况下使用“尾随闭包”这种写法会很有效。尾随闭包通常在函数调用的括号之后，即使他仍是一个参数。当你使用尾随闭包语法，你可以不用填写函数入参为闭包那部分的参数。

```
func someFunctionThatTakesAClosure(closure: () -> Void) {  
    // 函数主体部分  
}  
  
// 这里被调用函数没用后置闭包的写法：  
  
someFunctionThatTakesAClosure(closure: {  
    // 闭包主体部分  
})  
  
// 这里被调用函数使用后置闭包的写法：  
  
someFunctionThatTakesAClosure() {  
    // 尾随闭包主体部分  
}
```

上面的字符串排序闭包 [闭包表达式语法](#) 作为尾随闭包被写在 `sorted(by:)` 方法的括号外部，如下：

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

如果函数只有一个闭包类型入参，并且使用了尾随闭包的写法，当你调用这个函数的时候可以省略函数名称后面写 `()`，写法如下：

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

如果一个闭包代码很长以至于不能把它写在同一行上，这时使用后置闭包写法就比较有用了。例如，Swift 的 `Array` 类型的 `map(_:)` 方法就把闭包表达式作为唯一的参数传



chai 翻译于 5个月前

👍 0

重译

由 Aufree 审阅



ForC 翻译于 5个月前

👍 0

重译

由 Aufree 审阅



入。对数组中每个元素调用一次闭包，并为该元素返回一个映射值（可能是其他类型）。映射的值和类型由调用的闭包制定。

将提供的闭包应用到数组中的每个元素之后，`map(_:)` 方法会返回一个包含所有新映射值的新数组，并与原数组保持相同顺序。

下面介绍如何调用 `map(_:)` 方法并使用后置闭包的写法，来对一个 `Int` 类型的数组转换成 `String` 类型的数组。`[16, 58, 510]` 数组用于创建新数组 `["OneSix", "FiveEight", "FiveOneZero"]`：

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

这段代码上边创建了一个映射字典，整数数字作为键，以整数数字转换的英文作为值。并定义了一个整形数组，准备去转换成字符串数组。

你可以调用数组的 `map(_:)` 方法并传入一个后置闭包表达式，将 `numbers` 类型的数组转换为一个 `String` 类型的数组：

```
let strings = numbers.map { (numbers) -> String in
    var number = numbers
    var output = ""
    repeat {
        output = digitNames[number % 10]! + output
        number /= 10
    } while number > 0
    return output
}
// strings 被推测成 [String] 类型的数组
// 它的值为 ["OneSix", "FiveEight", "FiveOneZero"]
```

`map(_:)` 方法传入的闭包表达式会调用数组中每一个元素。你将不需要指定闭包的入参 `number` 类型，因为这个类型在数组映射的时候可以被推断出来。

这个例子中，局部变量 `number` 是在闭包主体中初始化并赋予闭包入参 `number` 的值，因此可以在闭包体内修改它的值。(函数或闭包的参数是常量)。闭包表达式也指定了一个 `String` 类型的返回值，指明了映射数组的返回值类型。

闭包表达式每次被调用时都会构建一个名为 `output` 的字符串。它通过取余操作



ForC 翻译于 5个月前

👍 0 重译

由 Aufree 审阅



ForC 翻译于 5个月前

👍 0 重译

由 Aufree 审阅

( `number % 10` ) 获取 `number` 最末位数字并通过这个数组去字典 `digitNames` 找到对应的字符串。闭包可以使用字典找到所有大于等于 `0` 的字符串。

注意

字典 `digitNames` 下标形式的调用之后是感叹号 ( `!` )，因为字典下标形式调用返回一个可选类型的值，表示对应键值不存可能查找失败的情况。上边的例子中 `number % 10` 获得的值一定作为 `digitNames` 字典的有效下标，因此使用感叹号强制解包字典索引下的 `String` 类型的值。

通过 `digitNames` 字典检索出得字符串会被存储在 `output` 字符串的前面，通过这种反向操作有效的构建了一个与原始值每个数字位置相同的字符串。( 取余表达式 `number % 10` ,在 `16` 值是 `6` ， 在 `58` 值是 `8` ， 在 `510` 值是 `0` )。

然后这个 `number` 的值除以 `10` 。因为是整形在除法时会被四舍五入，然后 `16` 变成 `1` ， `58` 变成 `5` ， `510` 变成 `51` 。

这个过程被一直被重复直到 `number` 值为 `0` 。 `output` 字符串在闭包中被返回，并且添加到了 `map(_:)` 方法的输出数组中。

上边的例子中使用了后置闭包的写法让这个闭包被调用后代码立即巧妙的封装函数的功能，不需要在 `map(_:)` 方法的括号里包装闭包体。

## 值捕获

闭包可以 捕获 它所定义的上下文环境中的常量和变量。在闭包体内可以使用和修改这些常量和变量的值， 即使这些常量、变量的作用域已经不存在了。

在 `Swift` 中，闭包捕获值的最简单的形式是嵌套函数--写在另一个函数的函数体内。嵌套函数可以捕获外部函数中的任意参数，也可以捕获定义在函数外部的任意常量、变量。

这是一个例子，一个叫做 `makeIncrementer` 的函数内部包含了一个嵌套函数 `incrementer` 。嵌套函数 `incrementer()` 从它所在的上下文环境中，捕获了两个值， `runningTotal` 和 `amount` 。捕获这些值之后 `incrementer` 作为一个每调用一次就会让 `runningTotal` 和 `amount` 的值进行相加的闭包被 `makeIncrementer` 函数返回。

```
func makeIncrementer(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementer() -> Int {
```

ForC 翻译于 5个月前

👍 0

重译

由 Aufree 审阅

gjcbo 翻译于 5个月前

👍 0

重译

由 Aufree 审阅

```
        runningTotal += amount
    }
    return runningTotal
}
return incrementer
}
```

`makeIncrementer` 的返回值类型是 `() -> Int`。这意味着它返回的是一个 *函数*，而不是一个简单的值。这个函数没有参数，每次被调用后会返回一个 `Int` 类型的值。想了解一个函数如何返回其他函数，请参考 [函数类型和返回值类型](#)。

`makeIncrementer(forIncrement:)` 这个函数定义了一个整型变量

`runningTotal`，用来存储将被返回的计算后的总和。这个变量的初始值是 `0`。

`makeIncrementer(forIncrement:)` 函数有一个 `Int` 型的参数，参数标签是 `forIncrement`，参数名是 `amount`。传递给这个参数的参数值用来指定每一次增加函数被调用后 `runningTotal` 的值增加多少。`makeIncrementer` 函数定义了一个名为 `incrementer` 的嵌套函数，这个函数用来执行具体的增加操作。这个函数只进行简单的相加，并将结果返回。

当单独来看 `incrementer()` 这个嵌套函数时，会觉得有点不可思议：

```
func incrementer() -> Int {
    runningTotal += amount
    return runningTotal
}
```

`incrementer()` 是一个无参数的函数，但是在它的函数体内却使用了 `runningTotal` 和 `amount` 这两个变量。

通过从它周围的函数中将 `runningTotal`、`amount` 这两个变量进行捕获，并在自己的函数内使用。通过引用进行捕获来确保 `makeIncrementer` 函数调用结束后 `runningTotal` 和 `amount` 这两个变量不消失，同时也可以确保在下一次调用 `incrementer` 函数时 `runningTotal` 是可用的。

## 注意

作为优化，如果一个值在闭包内没被修改，或闭包创建后该值没被修改，Swift 可能会替换捕获而是存储这个值的一个 *拷贝*。

Swift 还会处理所有不再被使用的变量的内存管理问题。

这是一个 `makeIncrementer` 的例子：



gjcbo 翻译于 5个月前

👍 0

重译

由 RecherJ 审阅



gjcbo 翻译于 5个月前

👍 0

重译

由 chdzq 审阅



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

这里定义了一个常量 `incrementByTen`，每次调用加数器函数都会将 `10` 和 `runningTotal` 变量的值相加，最后将函数的返回值赋值给常量 `incrementByTen`。多次调用这个函数的打印结果如下：

```
incrementByTen()  
// 返回值是 10  
incrementByTen()  
// 返回值是 20  
incrementByTen()  
// 返回值是 30
```

如果你创建第二个加数器，它会存储一个新的、单独的 `runningTotal` 变量：

```
let incrementBySeven = makeIncrementer(forIncrement: 7)  
incrementBySeven()  
// 返回值是 7
```

继续调用原来的加数器（`incrementByTen`）会继续让它自己的 `runningTotal` 变量增加，不会影响到 `incrementBySeven` 捕获到的变量：

```
incrementByTen()  
// 返回值是 40
```

## 注意

如果你将闭包作为一个类实例的属性，闭包通过引用该实例或其他成员来捕获该实例，在闭包和实例之间可能会导致循环引用。Swift 使用 *捕获列表* 来打断循环引用。更多信息请参考 [闭包的循环引用](#)。

## 闭包引用类型

在上面的例子中，`incrementBySeven` 和 `incrementByTen` 是常量，但是这些常量闭包仍然能够增加它们捕获到的 `runningTotal` 变量。这因为闭包和函数是引用类型。

无论你分配变量还是常量给函数或闭包，实际上你是设置闭包或者函数引用该常量或变量。在上面的例子中，闭包 `incrementByTen` 选择引用的是常量，而不是闭包本身内容。



gjcbo 翻译于 5个月前

👍 0

重译

由 RecherJ 审阅

这也意味这如果你分配一个闭包两个不同的变量或常量, 这些常量和变量引用相同的闭包:

```
let alsoIncrementByTen = incrementByTen
alsoIncrementByTen()

// 返回值为 50
```

## 逃逸闭包

当一个闭包作为参数传递给函数时, 闭包被称为 *逃逸* 了函数, 但是会在函数返回后才调用。当您声明将闭包作为参数之一的函数时, 可以在参数的类型之前写入 `@ escape`, 用来表示允许闭包逃逸。

有一种闭包可以逃逸的方式是存储在函数外定义的变量中。比如, 许多有异步操作的函数以闭包参数作为 completion handler。函数在启动操作之后就已经返回, 但在操作完成之后才调用闭包——这种闭包就需要需要逃逸, 以便函数返回后调用。例如:

```
var completionHandlers: [() -> Void] = []
func someFunctionWithEscapingClosure(completionHandler: @escaping () -> Void)
    completionHandlers.append(completionHandler)
}
```

`someFunctionWithEscapingClosure(_:)` 函数将闭包作为它的参数, 并且添加到函数之外的数组中。如果你不将函数中的这个参数标记为 `@escaping`, 将为得到一个编译时的错误。

用 `@escaping` 标记闭包意味着你会在闭包中显式地使用 `self`。在刚才的例子中, 传递给 `someFunctionWithEscapingClosure(_:)` 的是一个逃逸闭包, 意味着需要显式地使用 `self`。相对来说, 传给 `someFunctionWithNonescapingClosure(_:)` 的是一个非逃逸闭包, 就意味着可以隐式地使用 `self`。

```
func someFunctionWithNonescapingClosure(closure: () -> Void) {
    closure()
}

class SomeClass {
    var x = 10
    func doSomething() {
        someFunctionWithEscapingClosure { self.x = 100 }
        someFunctionWithNonescapingClosure { x = 200 }
    }
}
```



ForC 翻译于 5个月前

👍 0 重译

由 Aufree 审阅



xixizhy 翻译于 5个月前

👍 0 重译

由 Summer 审阅



xixizhy 翻译于 5个月前

👍 0 重译

由 Summer 审阅

```
}

let instance = SomeClass()
instance.doSomething()
print(instance.x)
// 打印 "200"

completionHandlers.first?()
print(instance.x)
// 打印 "100"
```

## 自动闭包

*自动闭包* 自动包装书写的表达式，并将表达式作为一个闭包传入的函数。自动闭包不包含任何参数，当它被调用时会返回一个内部表达式包装的值。这种写法让你使用正常表达式而不是闭包的语法使你可以省略函数旁边的大括号。

*调用* 类型函数采用自动闭包是常见的，*实现* 该类型的函数并不常见。例

如， `assert(condition:message:file:line:)` 方法为 `condition` 和 `message` 参数采用了自动闭包的写法，参数 `condition` 只在编译时取值，并且 `message` 参数只在 `condition` 参数为 `false` 时取值。

一个自动闭包能让我们延时取值，因为内部代码没有运行，直到你调用这个闭包。延时取值对于较大计算量和具有副作用的代码是非常有效的，因为这样能够让你去控制何时计算该代码，下面的代码展示了闭包如何延时计算：

```
var customersInLine = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
print(customersInLine.count)
// 打印 "5"

let customerProvider = { customersInLine.remove(at: 0) }
print(customersInLine.count)
// 打印 "5"

print("Now serving \(customerProvider())!")
// 打印 "Now serving Chris!"
print(customersInLine.count)
// 打印 "4"
```

尽管在闭包内部 `customersInLine` 的第一个元素被移除了，但是在闭包被调用之前，该元素是不会被移除。如果这个闭包永远不会被调用，那么这个闭包内部的表达式将永远不



ForC 翻译于 5个月前

👍 0

重译

由 Aufree 审阅

会执行，这意味着这个数组中的元素永远不会被移除。注意 `customerProvider` 不是一个 `字符串` 类型而是 `() -> String` --- 一个没有参数且返回值是字符串的函数。

当你将闭包作为参数传递给函数时，你能获得同样的延时求值行为：

```
// customersInLine 数组是 ["Alex", "Ewa", "Barry", "Daniella"]
func serve(customer customerProvider: () -> String) {
    print("Now serving \(customerProvider())!")
}
serve(customer: { customersInLine.remove(at: 0) } )
// 输出 "Now serving Alex!"
```

上面代码中的 `serve(customer:)` 函数接受一个返回顾客姓名的显示闭包。下面这个版本的 `serve(customer:)` 完成同样的功能，不过它并没有接受一个显示闭包，而是接受一个自动闭包，使用 `@autoclosure` 关键字标记参数。现在你可以将这个函数当作一个接受 `String` 类型的参数的函数而不是一个闭包一样调用。这个参数会自动转换成一个闭包，因为参数 `customerProvider` 的类型被标记成 `@autoclosure` 属性。

```
// customersInLine 数组是 ["Ewa", "Barry", "Daniella"]
func serve(customer customerProvider: @autoclosure () -> String) {
    print("Now serving \(customerProvider())!")
}
serve(customer: customersInLine.remove(at: 0))
// 输出 "Now serving Ewa!"
```

## 注意

自动闭包使用过多会让你的代码很难阅读。函数和上下文应该明确表明代码被延迟执行。

如果你想一个自动闭包也允许逃逸闭包的功能，你需要同时标记 `@autoclosure` 和 `@escaping` 关键字。`@escaping` 关键字描述在下面 [逃逸闭包](#)。

```
// customersInLine 数组值为 ["Barry", "Daniella"]
var customerProviders: [() -> String] = []
func collectCustomerProviders(_ customerProvider: @autoclosure @escaping () -> String) {
    customerProviders.append(customerProvider)
}
collectCustomerProviders(customersInLine.remove(at: 0))
collectCustomerProviders(customersInLine.remove(at: 0))
```



ForC 翻译于 5个月前

👍 0

重译

由 chdzq 审阅



ForC 翻译于 5个月前

👍 0

重译

由 Aufree 审阅

```
print("Collected \{(customerProviders.count) closures.\}")

// 打印 "Collected 2 closures."

for customerProvider in customerProviders {
    print("Now serving \{(customerProvider())!\}")
}

// 打印 "Now serving Barry!"
// 打印 "Now serving Daniella!"
```



在上边代码中，闭包作为 `customerProvider` 参数传入方法，`collectCustomerProviders(_:)` 函数为 `customerProviders` 数组拼接传入的闭包。数组 `customerProviders` 在函数外部定义，这意味着数组中的闭包允许在函数返回之后执行。综上所述，函数使用范围内必须允许 `customerProvider` 参数逃逸。

本文中的所有译文仅用于学习和交流目的，转载请务必注明文章译者、出处、和本文链接

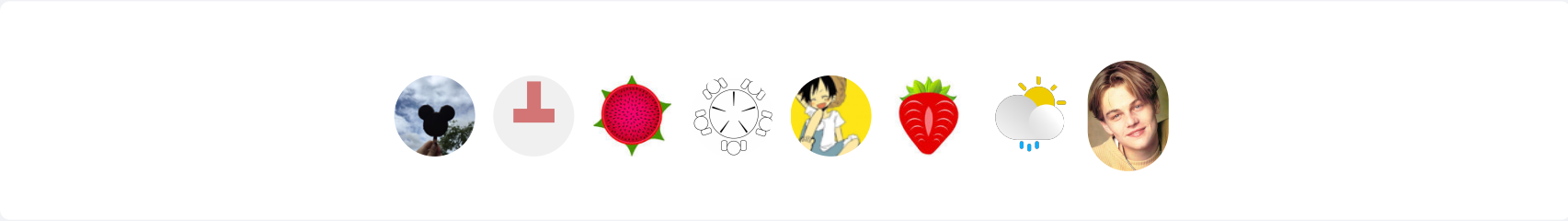
我们的翻译工作遵照 [CC 协议](#)，如果我们的工作有侵犯到您的权益，请及时联系我们。

-----

点赞



参与译者：8



更多职位

讨论数量：0

发起讨论

☐ 只看当前版本讨论



暂无话题~

## 兄弟社区



Laravel China



PythonCaff.com



GolangCaff.com



VuejsCaff.com

## 资源推荐

## 资源推荐

## 其他信息



软件外包



商务合作



联系站长

由 Summer 设计和编码 ❤️