关闭

*闭包*是自包含的功能块，可以在代码中传递和使用。Swift中的闭包与C和Objective-C中的块以及其他编程语言中的lambda类似。

闭包可以从定义的上下文中捕获和存储对任何常量和变量的引用。这被称为*关闭*那些常量和变量。Swift处理所有为你捕获的内存管理。

> 注意
> 如果您不熟悉捕捉概念，请不要担心。它在下面的捕获值中有详细的解释。

全球和嵌套函数，如推出的功能，实际上是封闭的特殊情况。闭包采取以下三种形式之一：

- 全局函数是具有名称并且不捕获任何值的闭包。
- 嵌套函数是具有名称的闭包，可以从其封闭函数中捕获值。
- Closure表达式是以轻量级语法编写的未命名的闭包，可以捕获周围环境中的值。

Swift的闭包表达式具有干净清晰的风格，优化可以在常见场景中促进简洁，无混乱的语法。这些优化包括：

- 从上下文中推断参数和返回值类型
- 来自单表达式闭包的隐式返回
- 速记参数名称
- 尾随闭包语法

闭包表达式

嵌套函数中引入的嵌套函数是一种方便的方式，可以将自包含的代码块作为更大函数的一部分进行命名和定义。但是，编写较短版本的类似功能的结构（没有完整的声明和名称）有时很有用。当您使用将函数作为一个或多个参数的函数或方法时，尤其如此。

*Closure表达式*是一种用简短的聚焦语法编写内联闭包的方法。闭包表达式提供了几种语法优化，用于以缩写形式编写闭包，而不会损失清晰度或意图。下面的闭包表达式示例通过`sorted(by:)`在几次迭代中完善该方法的单个示例来说明这些优化，每个迭代都以更简洁的方式表达相同的功能。

排序方法

Swift的标准库提供了一种称为的方法`sorted(by:)`，它基于您提供的排序闭包的输出对已知类型的值的数组进行排序。一旦完成排序过程，该`sorted(by:)`方法将返回一个与旧的相同类型和大小的新数组，其元素按正确的排序顺序排列。原始数组不被该`sorted(by:)`方法修改。

下面的闭包表达式示例使用该`sorted(by:)`方法以`String`反向字母顺序对值数组进行排序。这是要排序的初始数组：

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

该`sorted(by:)`方法接受一个闭包，该闭包接受与数组内容相同类型的两个参数，并返回一个`Bool`值，表示在排序值后第一个值应该出现在第二个值之前还是之后。`true`如果第一个值应该出现*在*第二个值*之前*，则排序闭包需要返回，`false`否则。

这个例子是排序`String`值的数组，因此排序闭包需要是类型的函数`(String, String) -> Bool`。

提供排序闭包的一种方法是编写正确类型的正常函数，并将其作为参数传递给`sorted(by:)`方法：

```
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），则该backward(_:_:)函数将返回true，指示s1应s2在排序数组之前出现。对于字符串中的字符，“大于”意味着“字母后面出现比”。这意味着该字母"B"“大于”该字母"A"，并且该字符串"Tom"大于该字符串"Tim"。这给出了一个反向字母排序，"Barry"放置在之前"Alex"，等等。

然而，这是写一
法来内联编写排序闭包。

闭包表达式语法

Closure表达式语法具有以下一般形式：

```
{ (参数) -> 返回类型 在
    声明
}
```

该*参数*在封闭表达式语法可以在输出参数，但是他们不能有一个默认值。如果命名可变参数，则可以使用变量参数。元组也可以用作参数类型和返回类型。

下面的例子显示了backward(_:_:)上面函数的闭包表达式版本：

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

请注意，此内联关闭的参数声明和返回类型与backward(_:_:)函数声明相同。在这两种情况下，它都被写为(s1: String, s2: String) -> Bool。但是，对于内联闭包表达式，参数和返回类型写*在*花括号内，而不是外部。

in关键字 引入了封闭体的开始。这个关键字表示闭包的参数和返回类型的定义已经完成，闭包的主体即将开始。

由于封闭体的体积非常短，所以它甚至可以写在一行上：

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

这说明对该方法的整体调用sorted(by:)保持不变。一对圆括号仍然包含了该方法的整个参数。但是，这个论点现在是一个内联关闭。

从上下文推断类型

因为排序闭包作为参数传递给方法，所以Swift可以推断它的参数类型和它返回的值的类型。该sorted(by:)方法在一个字符串数组上被调用，所以它的参数必须是一个类型的函数(String, String) -> Bool。这意味着(String, String)和Bool类型不需要被写为闭包表达式定义的一部分。由于可以推断所有类型，->因此也可以省略返回箭头（）和参数名称周围的括号。

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

在将闭包作为内联闭包表达式传递给函数或方法时，始终可以推断参数类型和返回类型。因此，当闭包用作函数或方法参数时，您绝不需要以最充分的形式编写内联闭包。

尽管如此，如果您愿意的话，仍然可以明确类型，如果避免代码读者含糊不清，那么可以这样做。在该sorted(by:)方法的情况下，封闭的目的从排序发生的事实中清楚，并且读者认为封闭可能与String值一起工作是安全的，因为它正在帮助排序的字符串数组。

来自单表达式闭包的隐式返回

单个表达式闭包可以通过return从声明中省略关键字来隐式地返回单个表达式的结果，就像前面示例的这个版本一样：

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

在这里，`sorted(by:)`方法参数的函数类型清楚地表明了一个`Bool`值必须由闭包返回。因为闭包的主体包含一个`s1 > s2`返回`Bool`值的单个表达式（），所以没有歧义，`return`关键字可以省略。

速记参数名称

雨燕自动提供速记参数名内联闭包，它可以使用的名称，指的是关闭的参数值`$0`，`$1`，`$2`，等等。

如果在闭包表达式中使用这些简写参数名称，则可以从其定义中省略闭包的参数列表，并且可以从期望的函数类型中推断简写参数名称的数量和类型。的`in`关键字也可以被省略，因为封闭件表达是由完全其身体的：

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

在这里，`$0`并`$1`请参阅关闭的第一和第二个`String`参数。

运算符方法

实际上有一个更*短的*方法来编写上面的闭包表达式。Swift的`String`类型定义了它`>`作为具有两个类型参数的方法的大于运算符（）的字符串特定实现`String`，并返回一个类型值`Bool`。这与该方法所需的方法类型完全匹配`sorted(by:)`。因此，你可以简单地传入大于运算符，Swift会推断你想使用它的字符串特定实现：

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

欲了解更多有关操作方法，请参阅操作方法。

追踪关闭

如果需要将闭包表达式作为函数的最终参数传递给函数，并且闭包表达式很长，那么将其作为*尾部闭包*编写可能会很有用。尾随闭包在函数调用的括号后面写入，尽管它仍然是该函数的参数。在使用尾随闭包语法时，不要将闭包的参数标签作为函数调用的一部分写入。

```
func someFunctionThatTakesAClosure(closure: () -> Void) {
    // function body goes here
}

// Here's how you call this function without using a trailing closure:

someFunctionThatTakesAClosure(closure: {
    // closure's body goes here
})

// Here's how you call this function with a trailing closure instead:

someFunctionThatTakesAClosure() {
    // trailing closure's body goes here
}
```

上面 的Closure Expression Syntax部分的字符串排序闭包可以在`sorted(by:)`方法的括号之外写为尾部闭包：

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

如果将闭包表达式作为函数或方法的唯一参数提供，并且将该表达式作为尾随闭包提供，那么`()`在调用该函数时，无需在函数或方法名称后面编写一对括号。

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

当封闭足够长以至于不可能将它写在一行上时，尾随封闭非常有用。作为一个例子，Swift的`Array`类型有一个`map(_:)`方法，它把一个闭包表达式作为它的单个参数。对于数组中的每个项目都调用一次闭包，并为该项目返回一个替代映射值（可能是某种其他类型）。映射的本质和返回值的类型留给闭包来指定。

将提供的闭包应用于每个数组元素后，该`map(_:)`方法返回一个包含所有新映射值的新数组，其顺序与原始数组中的对应值相同。

以下是如何使用`map(_:)`具有尾随闭包的方法将`Int`值数组转换为值数组`String`。该数组`[16, 58, 510]`用于创建新数组`["OneSix", "FiveEight", "FiveOneZero"]`：

```
let dig
    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]
```

上面的代码创建了一个整数数字和英文版名称之间映射的字典。它还定义了一个整数数组，可以将其转换为字符串。

您现在可以使用该`numbers`数组创建一个`String`值数组，方法是将闭包表达式`map(_:)`作为尾部闭包传递给数组的方法：

```
let strings = numbers.map { (number) -> String in
    var number = number
    var output = ""
    repeat {
        output = digitNames[number % 10]! + output
        number /= 10
    } while number > 0
    return output
}
// strings is inferred to be of type [String]
// its value is ["OneSix", "FiveEight", "FiveOneZero"]
```

该`map(_:)`方法为数组中的每个项调用闭包表达式一次。您不需要指定闭包的输入参数`number`的类型，因为可以从要映射的数组中的值中推断出该类型。

在这个例子中，变量`number`是用闭包`number`参数的值初始化的，这样就可以在闭包体中修改该值。（函数和闭包的参数始终是常量。）闭包表达式还指定返回类型`String`，以指示将存储在映射的输出数组中的类型。

闭包表达式会在`output`每次调用时创建一个字符串。它`number`使用余数运算符（`number % 10`）计算最后一位数字，并使用此数字在`digitNames`字典中查找适当的字符串。闭包可以用来创建任何大于零的整数的字符串表示。

> 注意
>
> 对`digitNames`字典下标的调用后面跟着一个感叹号（`!`），因为字典下标返回一个可选值，表示如果该键不存在，字典查找可能会失败。在上面的例子中，保证字典`number % 10`总是一个有效的下标键`digitNames`，因此使用感叹号来强制解开`String`存储在下标可选返回值中的值。

从检索到的字符串`digitNames`辞典被添加到*前面*的`output`，有效地建立反向一数目的字符串版本。（该表达式`number % 10`给出了6for `16`，8for `58`和`0`for的值`510`。）

`number`然后 这个变量除以`10`。因为它是一个整数，它在划分期间被舍入，所以`16`变成`1`，`58`变成`5`，`510`变成`51`。

重复该过程直至`number`等于`0`，此时`output`字符串由闭包返回，并通过该`map(_:)`方法添加到输出数组中。

在上面的例子中，使用尾部闭包语法在闭包支持的函数后立即封闭闭包的功能，而不需要将整个闭包封装在`map(_:)`方法的外部圆括号内。

捕捉价值观

闭包可以*捕获*定义它的周围环境中的常量和变量。即使定义常量和变量的原始范围不再存在，闭包也可以引用并修改其正文中的那些常量和变量的值。

在Swift中，可以捕获值的闭包的最简单形式是嵌套函数，写在另一个函数的主体中。嵌套函数可以捕获任何外部函数的参数，也可以捕获外部函数中定义的任何常量和变量。

这是一个叫做函数的例子`makeIncrementer`，它包含一个叫做嵌套函数`incrementer`。嵌套`incrementer()`函数捕获两个值，`runningTotal`并`amount`从其周围的上下文中捕获。在捕获这些值后，作为闭包`incrementer`返回，每次调用时都会`makeIncrementer`递增。 `runningTotalamount`

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

        return runningTotal
    }
    return incrementer
}
```

返回类型`makeIncrementer`是`() -> Int`。这意味着它返回一个*函数*，而不是一个简单的值。它返回的函数没有参数，`Int`每次调用时都会返回一个值。要了解函数如何返回其他函数，请参阅函数类型作为返回类型。

该`makeIncrementer(forIncrement:)`函数定义了一个称为的整数变量`runningTotal`，用于存储将返回的增量器的当前运行总数。该变量的初始值为`0`。

该`makeIncrementer(forIncrement:)`函数具有单个`Int`参数，参数标签为`forIncrement`，参数名称为`amount`。传递给此参数的参数值指定`runningTotal`每次调用返回的增量函数时应递增多少。该`makeIncrementer`函数定义一个名为的嵌套函数`incrementer`，它执行实际递增。该功能仅添加`amount`到`runningTotal`，并返回结果。

当单独考虑时，嵌套`incrementer()`函数可能看起来很不寻常：

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

该`incrementer()`函数没有任何参数，但它指的是`runningTotal`和`amount`其函数体内。它通过捕获做到这一点*参考*，以`runningTotal`和`amount`从周围的功能和其自身的函数体中使用它们。通过参考捕捉保证`runningTotal`和`amount`不消失的时候调用`makeIncrementer`结束，而且也保证了`runningTotal`可用下一次`incrementer`函数被调用。

> 注意
>
> 作为一个优化，Swift可以取而代之地捕获并存储一个值的*副本*，如果该值没有被闭包变异，并且该值在闭包创建后没有变异。
>
> Swift还处理所有涉及处理变量时不再需要的内存管理。

以下是一个实例`makeIncrementer`：

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

这个例子设置了一个常量，`incrementByTen`用于引用一个增量函数，每次调用它时都会增加`10`它的`runningTotal`变量。多次调用该函数会显示此行为的实际行为：

```
incrementByTen()
// returns a value of 10
incrementByTen()
// returns a value of 20
incrementByTen()
// returns a value of 30
```

如果您创建了第二个增量器，它将拥有自己的对新的单独`runningTotal`变量的存储引用：

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

`incrementByTen`再次 调用原始增量（）再继续递增其自己的`runningTotal`变量，并且不影响通过`incrementBySeven`以下方式捕获的变量：

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

> 注意
>
> 如果将闭包分配给类实例的属性，并且闭包通过引用实例或其成员来捕获该实例，则将在闭包和实例之间创建一个强引用循环。Swift使用*捕获列表*来打破这些强大的参考周期。有关更多信息，请参阅闭合强参考周期。

闭包是参考类型

在上面的例子中，`incrementBySeven`和`incrementByTen`是常量，但是这些常量指的是封闭仍然能够递增`runningTotal`，他们已抓获的变量。这是因为函数和闭包是*引用类型*。

无论何时将函数或闭包分配给常量或变量，实际上都是将该常量或变量设置为函数或闭包的*引用*。在上面的例子中，闭包的选择是`incrementByTen` *指*常量，而不是闭包本身的内容。

这也意味着，如果将一个闭包分配给两个不同的常量或变量，那么这两个常量或变量都将引用相同的闭包：

```
let alsoIncrementByTen = incrementByTen
alsoIncrementByTen()
// returns a value of 50
```

逃逸关闭

闭包是说*逃避*当封盖作为参数传递给函数，但在函数返回之后被调用的函数。当你声明一个将闭包作为其参数的函数时，你可以`@escaping`在参数的类型之前写入，以表明允许闭包。

闭包可以逃脱的一种方式是存储在函数外部定义的变量中。作为例子，许多启动异步操作的函数都将闭包参数作为完成处理程序。该函数在开始操作后返回，但在操作完成之前不会调用闭包 - 闭包需要转义，稍后调用。例如：

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

该`someFunctionWithEscapingClosure(_:)`函数将闭包作为参数，并将其添加到在函数外声明的数组。如果你没有标记这个函数的参数`@escaping`，你会得到一个编译时错误。

标记闭包`@escaping`意味着你必须`self`在闭包中明确提及。例如，在下面的代码中，传递给的闭包`someFunctionWithEscapingClosure(_:)`是一个转义闭包，这意味着它需要`self`明确引用。相比之下，传递给`someFunctionWithNonescapingClosure(_:)`它的闭包是一个nonescaping闭包，这意味着它可以`self`隐式引用。

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

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

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

completionHandlers.first?()
print(instance.x)
```

```
// Prints "100"
```

Autoclosures

一个*autoclosure*是自动创建来包装被真实作为参数传递给函数的表达式的封闭件。它不需要任何参数，当它被调用时，它会返回包装在其中的表达式的值。这种语法上的便利可以让你通过写一个普通的表达式而不是显式的闭包来省略函数参数的大括号。

通常*调用*采用自动屏蔽的函数，但*实现*这种功能并不常见。例如，该`assert(condition:message:file:line:)`函数为其参数`condition`和`message`参数进行autoclosure；它`condition`仅在调试参数进行评估，并建立其`message`仅在参数评估`condition`是`false`。

autoclosure让你延迟评估，因为在你调用闭包之前，里面的代码不会运行。延迟评估对于有副作用或计算成本较高的代码非常有用，因为它可以让您控制代码的评估时间。下面的代码显示了封闭延迟评估的方式。

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

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

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

即使`customersInLine`数组的第一个元素被闭包中的代码删除，数组元素也不会被删除，直到实际调用闭包为止。如果闭包永远不会被调用，闭包内的表达式永远不会被计算，这意味着数组元素永远不会被移除。请注意，类型`customerProvider`是不是`String`，但`() -> String`不带任何参数，返回一个字符串-a功能。

当您将闭包作为参数传递给函数时，您会得到延迟评估的相同行为。

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

`serve(customer:)`上面列表中 的函数使用显式闭包来返回客户的名字。下面的版本`serve(customer:)`执行相同的操作，但不是采用显式闭包，而是通过用参数的`@autoclosure`属性标记其参数类型来采用autoclosure 。现在你可以调用函数，就好像它使用了一个`String`参数而不是闭包。该参数会自动转换为闭包，因为该`customerProvider`参数的类型是用`@autoclosure`属性标记的。

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

> 注意
>
> 过度使用自动遮挡会使您的代码难以理解。上下文和函数名称应该明确表示评估正在推迟。

如果您想要允许转义的自动关闭，请使用`@autoclosure`和`@escaping`属性。该`@escaping`属性在上面的转义闭包中描述。

```
// customersInLine is ["Barry", "Daniella"]
var customerProviders: [() -> String] = []
```

在本页

```
func collectCustomerProviders(_ customerProvider: @autoclosure @escaping () ->
  String) {
    customerProviders.append(customerProvider)
}
collectCustomerProviders(customersInLine.remove(at: 0))
collectCustomerProviders(customersInLine.remove(at: 0))

print("Collected \(customerProviders.count) closures.")
// Prints "Collected 2 closures."
for customerProvider in customerProviders {
    print("Now serving \(customerProvider())!")
}
// Prints "Now serving Barry!"
// Prints "Now serving Daniella!"
```

在上面的代码，而不是调用传递给它作为它的闭合`customerProvider`参数，该`collectCustomerProviders(_:)`函数将所述封闭的`customerProviders`阵列。数组声明在函数范围之外，这意味着数组中的闭包可以在函数返回后执行。因此，`customerProvider`参数的值必须被允许转义该函数的作用域。