闭包是功能性自包含模块，可以在代码中被传递和使用。 Swift 中的闭包与 C 和 Objective-C中的 blocks 以及其他一些编程语言中的 lambdas 比较相似。

闭包可以**捕获**和存储其所在上下文中任意常量和变量的引用。 这就是所谓的闭合并包裹着这些常量和变量，俗称闭包。Swift会为您管理在**捕获**过程中涉及到的内存操作。

|  |
| --- |
| 注意：如果您不熟悉 捕获 (capturing) 这个概念也不用担心，后面会详细对其进行介绍。 |

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：

1. 全局函数是一个有名字但不会捕获任何值的闭包

2. 嵌套函数是一个有名字并可以捕获其封闭函数域内值的闭包

3. 闭包表达式是一个利用轻量级语法所写的可以捕获其上下文中变量或常量值的没有名字的闭包

Swift的闭包表达式拥有简洁的风格，并鼓励在常见场景中以实现语法优化，主要优化如下：

\* 利用上下文推断参数和返回值类型

\* 单表达式（single-expression）闭包可以省略 return 关键字

\* 参数名称简写

\* Trailing 闭包语法

**闭包表达式**

嵌套函数是一种在较复杂函数中方便进行命名和定义自包含代码模块的方式。 当然，有时候撰写小巧的没有完整定义和命名的类函数结构也是很有用处的，尤其是在处理一些函数并需要将另外一些函数作为该函数的参数时。

闭包表达式是一种利用简洁语法构建内联闭包的方式。 闭包表达式提供了一些语法优化，使得撰写闭包变得简单明了。 下面闭包表达式的例子通过使用几次迭代展示了 sort 函数定义和语法优化的方式。 每一次迭代都用更简洁的方式描述了相同的功能。

**sort 函数**

Swift 标准库提供了 sort 函数，会根据您提供的排序闭包将已知类型数组中的值进行排序。 一旦排序完成，函数会返回一个与原数组大小相同的新数组，该数组中包含已经正确排序的同类型元素。

下面的闭包表达式示例使用 sort 函数对一个**String**类型的数组进行字母逆序排序，以下是初始数组值：

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

排序函数有两个参数：

1. 已知类型值的数组。

2. 一个闭包，采用相同类型的数组的内容的两个参数，并返回一个布尔值来表示是否将第一个值在排序时放到第二个值的前面或是后面。如果第一个值应该出现第二个值之前，闭包需要返回true，否则返回false。

该例子对一个**String**类型的数组进行排序，因此排序闭包需为 (String, String) -> Bool 类型的函数。

提供排序闭包的一种方式是撰写一个符合其类型要求的普通函数，并将其作为 sort 函数的第二个参数传入：

1. func backwards(s1: String, s2: String) -> Bool {
2. **return** s1 > s2
3. }
4. **var** reversed = sort(names, backwards)
5. // reversed 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 函数则返回 true，表示在新的数组中 s1 应该出现在 s2 前。 字符中的 "大于" 表示 "按照字母顺序后出现"。 这意味着字母 "B" 大于字母 "A", 字符串 "Tom" 大于字符串 "Tim"。 其将进行字母逆序排序，"Barry" 将会排在 "Alex" 之后,一次类推。

然而，这是一个相当冗长的方式，本质上只是写了一个单表达式函数 (a > b)。 在下面的例子中，利用闭合表达式语法可以更好的构造一个内联排序闭包。

**闭包表达式语法**

闭包表达式语法有如下一般形式：

1. { (parameters) -> returnType **in**
2. statements
3. }

闭包表达式语法可以使用常量、变量和 inout 类型作为参数，但不提供默认值。 也可以在参数列表的最后使用可变参数。元组也可以作为参数和返回值。

下面的例子展示了之前 backwards 函数对应的闭包表达式版本的代码：

1. reversed = sort(names, { (s1: String, s2: String) -> Bool **in**
2. **return** s1 > s2
3. })

需要注意的是内联闭包参数和返回值类型声明与 backwards 函数类型声明相同。 在这两种方式中，都写成了 (s1: String, s2: String) -> Bool类型。 然而在内联闭包表达式中，函数和返回值类型都写在大括号内，而不是大括号外。

闭包的函数体部分由关键字 in 引入。 该关键字表示闭包的参数和返回值类型定义已经完成，闭包函数体即将开始。

因为这个闭包的函数体部分如此短以至于可以将其改写成一行代码：

1. reversed = sort(names, { (s1: String, s2: String) -> Bool **in** **return** s1 > s2 } )

这说明 sort 函数的整体调用保持不变，一对圆括号仍然包裹住了函数中整个参数集合。而其中一个参数现在变成了内联闭包 (相比于 backwards 版本的代码)。

**根据上下文推断类型**

因为排序闭包是作为函数的参数进行传入的，Swift可以推断其参数和返回值的类型。 sort 期望第二个参数是类型为 (String, String) -> Bool 的函数，因此实际上 String, String 和 Bool 类型并不需要作为闭包表达式定义中的一部分。 因为所有的类型都可以被正确推断，返回箭头 (->) 和 围绕在参数周围的括号也可以被省略：

1. reversed = sort(names, { s1, s2 **in** **return** s1 > s2 } )

实际上任何情况下，通过内联闭包表达式构造的闭包作为参数传递给函数时，都可以推断出闭包的参数和返回值类型，这意味着您几乎不需要利用完整格式构造任何内联闭包。

然而，你也可以使用明确的类型，如果你想它避免读者阅读可能存在的歧义，这样还是值得鼓励的。这个排序函数例子，闭包的目的是很明确的，即排序被替换，而且对读者来说可以安全的假设闭包可能会使用字符串值,因为它正协助一个字符串数组进行排序。

**单行表达式闭包可以省略 return**

单行表达式闭包可以通过隐藏 return 关键字来隐式返回单行表达式的结果，如上版本的例子可以改写为：

1. reversed = sort(names, { s1, s2 **in** s1 > s2 } )

在这个例子中，sort 函数的第二个参数函数类型明确了闭包必须返回一个 Bool 类型值。 因为闭包函数体只包含了一个单一表达式 (s1 > s2)，该表达式返回 Bool 类型值，因此这里没有歧义，return关键字可以省略。

**参数名简写**

Swift 自动为内联函数提供了参数名称简写功能，您可以直接通过 $0,$1,$2等名字来引用的闭包的参数的值。

如果您在闭包表达式中使用参数名称简写，您可以在闭包参数列表中省略对其的定义，并且对应参数名称简写的类型会通过函数类型进行推断。 in 关键字也同样可以被省略，因为此时闭包表达式完全由闭包函数体构成：

1. reversed = sort(names, { $0 > $1 } )

在这个例子中，$0 和 $1 表示闭包中第一个和第二个 String 类型的参数。

**运算符函数**

实际上还有一种更简短的方式来撰写上面例子中的闭包表达式。 Swift的 String 类型定义了关于大于号 (>) 的字符串实现，让其作为一个函数接受两个 String 类型的参数并返回 Bool 类型的值。 而这正好与 sort 函数的第二个参数需要的函数类型相符合。 因此，您可以简单地传递一个大于号，Swift可以自动推断出您想使用大于号的字符串函数实现：

1. reversed = sort(names, >)

更多关于运算符表达式的内容请查看 [Operator Functions](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/AdvancedOperators.html#//apple_ref/doc/uid/TP40014097-CH27-XID_43)。

**Trailing 闭包**

如果您需要将一个很长的闭包表达式作为最后一个参数传递给函数，可以使用 trailing 闭包来增强函数的可读性。

Trailing 闭包是一个书写在函数括号之外(之后)的闭包表达式，函数支持将其作为最后一个参数调用。

1. func someFunctionThatTakesAClosure(closure: () -> ()) {
2. // 函数体部分
3. }
5. // 以下是不使用 trailing 闭包进行函数调用
7. someFunctionThatTakesAClosure({
8. // 闭包主体部分
9. })
11. // 以下是使用 trailing 闭包进行函数调用
13. someFunctionThatTakesAClosure() {
14. // 闭包主体部分
15. }

|  |
| --- |
| 注意：如果函数只需要闭包表达式一个参数，当您使用 trailing 闭包时，您甚至可以把 () 省略掉。 |

在上例中作为 sort 函数参数的字符串排序闭包可以改写为：

1. reversed = sort(names) { $0 > $1 }

当闭包非常长以至于不能在一行中进行书写时，Trailing 闭包就变得非常有用。 举例来说，Swift 的 **Array** 类型有一个 map 方法，其获取一个闭包表达式作为其唯一参数。 数组中的每一个元素调用一次该闭包函数，并返回该元素所映射的值(也可以是不同类型的值)。 具体的映射方式和返回值类型由闭包来指定。

当提供给数组闭包函数后，map 方法将返回一个新的数组，数组中包含了与原数组一一对应的映射后的值。

下例介绍了如何在 map 方法中使用 trailing 闭包将 **Int** 类型数组 [16,58,510] 转换为包含对应 **String** 类型的数组 ["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. let numbers = [16, 58, 510]

上面的代码创建了整数数字到他们的英文名字之间映射字典。 同时定义了一个准备转换为字符串的整型数组。

您现在可以通过传递一个 trailing 闭包给 numbers 的 map 方法来创建对应的字符串版本数组。 需要注意的时调用 numbers.map 不需要在 map 后面包含任何括号，因为只需要传递闭包表达式这一个参数，并且该闭包表达式参数通过 trailing 方式进行撰写：

1. let strings = numbers.map {
2. (**var** number) -> String **in**
3. **var** output = ""
4. **while** number > 0 {
5. output = digitNames[number % 10]! + output
6. number /= 10
7. }
8. **return** output
9. }
10. // strings 常量被推断为字符串类型数组，即 String[]
11. // 其值为 ["OneSix", "FiveEight", "FiveOneZero"]

map 在数组中为每一个元素调用了闭包表达式。 您不需要指定闭包的输入参数 number 的类型，因为可以通过要映射的数组类型进行推断。

闭包 number 参数被声明为一个变量参数 (变量的具体描述请参看[Constant and Variable Parameters](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Functions.html#//apple_ref/doc/uid/TP40014097-CH10-XID_224))，因此可以在闭包函数体内对其进行修改。 闭包表达式制定了返回值类型为 **String**，以表明存储映射值的新数组类型为**String**。

闭包表达式在每次被调用的时候创建了一个字符串并返回。 其使用求余运算符 (number % 10) 计算最后一位数字并利用 digitNames 字典获取所映射的字符串。

|  |
| --- |
| 注意：  字典 digitNames 下标后跟着一个叹号 (!)，因为字典下标返回一个可选值 (optional value)，表明即使该 key不存在也不会查找失败。 在上例中，它保证了 number % 10 可以总是作为一个 digitNames 字典的有效下标 key。 因此叹号可以用于强展开 (force-unwrap) 存储在可选下标项中的 String 类型值。 |

从 digitNames 字典中获取的字符串被添加到输出的前部，逆序建立了一个字符串版本的数字。 (在表达式 number % 10中，如果number为16，则返回6，58返回8，510返回0)。

number 变量之后除以10。 因为其是整数，在计算过程中未除尽部分被忽略。 因此 16变成了1，58变成了5，510变成了51。

整个过程重复进行，直到 number /= 10 为0，这时闭包会将字符串输出，而map函数则会将字符串添加到所映射的数组中。

上例中 trailing 闭包语法在函数后整洁封装了具体的闭包功能，而不再需要将整个闭包包裹在 map 函数的括号内。

**捕获 (Caputure)**

闭包可以在其定义的上下文中捕获常量或变量。 即使定义这些常量和变量的原作用域已经不存在，闭包仍然可以在闭包函数体内引用和修改这些值。

Swift最简单的闭包形式是嵌套函数，也就是定义在其他函数体内的函数。 嵌套函数可以捕获其外部函数所有的参数以及定义的常量和变量。

下例为一个叫做 makeIncrementor 的函数，其包含了一个叫做 incrementor 嵌套函数。 嵌套函数 incrementor 从上下文中捕获了两个值，runningTotal 和 amount。 之后 makeIncrementor 将 incrementor 作为闭包返回。 每次调用 incrementor 时，其会以 amount 作为增量增加 runningTotal 的值。

1. func makeIncrementor(forIncrement amount: Int) -> () -> Int {
2. **var** runningTotal = 0
3. func incrementor() -> Int {
4. runningTotal += amount
5. **return** runningTotal
6. }
7. **return** incrementor
8. }

makeIncrementor 返回类型为 () -> Int。 这意味着其返回的是一个函数，而不是一个简单类型值。 该函数在每次调用时不接受参数只返回一个**Int** 类型的值。 关于函数返回其他函数的内容，请查看[Function Types as Return Types](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Functions.html#//apple_ref/doc/uid/TP40014097-CH10-XID_232)。

makeIncrementor 函数定义了一个整型变量 runningTotal (初始为0) 用来存储当前增加总数。 该值通过 incrementor 返回。

makeIncrementor 有一个 **Int** 类型的参数，其外部命名为 forIncrement， 内部命名为 amount，表示每次 incrementor 被调用时 runningTotal 将要增加的量。

incrementor 函数用来执行实际的增加操作。 该函数简单地使 runningTotal 增加 amount，并将其返回。

如果我们单独看这个函数，会发现看上去不同寻常：

1. func incrementor() -> Int {
2. runningTotal += amount
3. **return** runningTotal
4. }

incrementor 函数并没有获取任何参数，但是在函数体内访问了 runningTotal 和 amount 变量。这是因为其通过捕获在包含它的函数体内已经存在的 runningTotal 和 amount 变量而实现。

由于没有修改 amount 变量，incrementor 实际上捕获并存储了该变量的一个副本，而该副本随着 incrementor 一同被存储。

然而，因为每次调用该函数的时候都会修改 runningTotal 的值，incrementor 捕获了当前 runningTotal 变量的引用，而不是仅仅复制该变量的初始值。捕获一个引用保证了当 makeIncrementor 结束时候并不会消失，也保证了当下一次执行 incrementor 函数时，runningTotal 可以继续增加。

|  |
| --- |
| 注意：  Swift 会决定捕获引用还是拷贝值。 您不需要标注 amount 或者 runningTotal 来声明在嵌入的 incrementor 函数中的使用方式。 Swift 同时也处理 runingTotal 变量的内存管理操作，如果不再被 incrementor 函数使用，则会被清除。 |

下面为一个使用 makeIncrementor 的例子：

1. let incrementByTen = makeIncrementor(forIncrement: 10)

该例子定义了一个叫做 incrementByTen 的常量，该常量指向一个每次调用会加10的 incrementor 函数。 调用这个函数多次可以得到以下结果：

1. incrementByTen()
2. // 返回的值为10
3. incrementByTen()
4. // 返回的值为20
5. incrementByTen()
6. // 返回的值为30

如果您创建了另一个 incrementor，其会有一个属于自己的独立的 runningTotal 变量的引用。 下面的例子中，incrementBySevne 捕获了一个新的 runningTotal 变量，该变量和 incrementByTen 中捕获的变量没有任何联系：

1. let incrementBySeven = makeIncrementor(forIncrement: 7)
2. incrementBySeven()
3. // 返回的值为7
4. incrementByTen()
5. // 返回的值为40

|  |
| --- |
| 注意：  如果您闭包分配给一个类实例的属性，并且该闭包通过指向该实例或其成员来捕获了该实例，您将创建一个在闭包和实例间的强引用环。 Swift 使用捕获列表来打破这种强引用环。更多信息，请参考 [Strong Reference Cycles for Closures](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/AutomaticReferenceCounting.html#//apple_ref/doc/uid/TP40014097-CH20-XID_61)。 |

**闭包是引用类型**

上面的例子中，incrementBySeven 和 incrementByTen 是常量，但是这些常量指向的闭包仍然可以增加其捕获的变量值。 这是因为函数和闭包都是引用类型。

无论您将函数/闭包赋值给一个常量还是变量，您实际上都是将常量/变量的值设置为对应函数/闭包的引用。 上面的例子中，incrementByTen 指向闭包的引用是一个常量，而并非闭包内容本身。

这也意味着如果您将闭包赋值给了两个不同的常量/变量，两个值都会指向同一个闭包：

1. let alsoIncrementByTen = incrementByTen
2. alsoIncrementByTen()
3. // 返回的值为50