翻译：wh1100717　校对：lyuka

闭包（Closures）

本页包含内容：

闭包是自包含的函数代码块，可以在代码中被传递和使用。 Swift 中的闭包与 C 和 Objective-C 中的代码块（blocks）以及其他一些编程语言中的 lambdas 函数比较相似。

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所有内存操作。

注意：

如果您不熟悉捕获（capturing）这个概念也不用担心，您可以在 值捕获 章节对其进行详细了解。

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可以捕获其封闭函数域内值的闭包
- 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匿名闭包

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

- 利用上下文推断参数和返回值类型
- 隐式返回单表达式闭包，即单表达式闭包可以省略`return`关键字

- 参数名称缩写
- 尾随（Trailing）闭包语法

闭包表达式（Closure Expressions）

嵌套函数 是一个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，尤其是在您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（The Sort Function）

Swift 标准库提供了`sort`函数，会根据您提供的基于输出类型排序的闭包函数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：

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

`sort`函数需要传入两个参数：

- 已知类型的数组
- 闭包函数，该闭包函数需要传入与数组类型相同的两个值，并返回一个布尔类型值来告诉`sort`函数当排序结束后传入的第一个参数排在第二个参数前面还是后面。如果第一个参数值出现在第二个参数值前面，排序闭包函数需要返回`true`，反之返回`false`。

该例子对一个`String`类型的数组进行排序，因此排序闭包函数类型需为

`(String, String) -> Bool`。

提供排序闭包函数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：

```
func backwards(s1: String, s2: String) -> Bool {
    return s1 > s2
}
var reversed = sort(names, backwards)
// reversed 为 ["Ewa", "Daniella", "Chris", "Barry",
"Alex"]
```

如果第一个字符串 (`s1`) 大于第二个字符串 (`s2`)，`backwards`函数返回`true`，表示在新的数组中`s1`应该出现在`s2`前。 对于字符串中的字符来说，“大于” 表示 “按照字母顺序较晚出现”。 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。

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（Closure Expression Syntax）

闭包表达式语法有如下一般形式：

```
{ (parameters) -> returnType in
    statements
}
```

闭包表达式语法可以使用常量、变量和`inout`类型作为参数，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：

```
reversed = sort(names, { (s1: String, s2: String) ->
Bool in
```

```
    return s1 > s2
    })
```
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。 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：

```
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（Inferring Type From Context）

因为排序闭包函数是作为sort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：

```
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。

单表达式闭包隐式返回（Implicit Return From

Single-Expression Clossures）

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：

```
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。

参数名称缩写（Shorthand Argument Names）

Swift 自动为内联函数提供了参数名称缩写功能，您可以直接通过`$0`,`$1`,`$2`来顺序调用闭包的参数。

如果您在闭包表达式中使用参数名称缩写，您可以在闭包参数列表中省略对其的定义，并且对应参数名称缩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：

```
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（Operator Functions）

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，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：

```
reversed = sort(names, >)
```

更多关于运算符表达式的内容请查看 运算符函数。

尾随闭包（Trailing Closures）

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，可以使用尾随闭包来增强函数的可读性。 尾随闭包是一个书写在函数括号之后的闭包表达式，函数支持将其作为最后一个参数调用。

```
func someFunctionThatTakesAClosure(closure: () -> ())
{
    // 函数体部分
}

// 以下是不使用尾随闭包进行函数调用

someFunctionThatTakesAClosure({
    // 闭包主体部分
    })

// 以下是使用尾随闭包进行函数调用

someFunctionThatTakesAClosure() {
    // 闭包主体部分
}
```

注意:

如果函数只需要闭包表达式一个参数，当您使用尾随闭包时，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:

```
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，尾随闭包变得非常有用。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方法中使用尾随闭包将Int类型数组[16,58,510]转换为包含对应String类型的数组["OneSix", "FiveEight", "FiveOneZero"]:

```
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]
```

如上代码创建了一个数字位和他们名字映射的英文版本字典。 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尾随闭包给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，因为其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尾随方式进行撰写:

```
let strings = numbers.map {
    (var number) -> String in
    var output = ""
    while number > 0 {
        output = digitNames[number % 10]! + output
        number /= 10
    }
    return output
}
// strings 常量被推断为字符串类型数组，即 String[]
// 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常量参数和变量参数），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制解析 (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。

上例中尾随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。

捕获值（Capturing Values）

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的函数体内的函数。 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。

```
func makeIncrementor(forIncrement amount: Int) -> ()
-> Int {
    var runningTotal = 0
    func incrementor() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementor
}
```

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，请查看函数类型作为返回类型。

makeIncrementor函数定义了一个整型变量runningTotal(初始为0) 用来存储当前跑步总数。 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：

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

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。
下面代码为一个使用makeIncrementor的例子：

```
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：

```
incrementByTen()
// 返回的值为10
incrementByTen()
// 返回的值为20
incrementByTen()
// 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：

```
let incrementBySeven = makeIncrementor(forIncrement:
7)
incrementBySeven()
// 返回的值为7
incrementByTen()
// 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 闭包引起的循环强引用。

闭包是引用类型（Closures Are Reference Types）

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：

```
let alsoIncrementByTen = incrementByTen
alsoIncrementByTen()
// 返回的值为50
```