使用func和closure加工数据

Swift 3关于函数类型的一项重要提议
(https://www.boxueio.com/series/functions-and-closure/ebook/149)

通过Local function捕获变量共享资源
(https://www.boxueio.com/series/functions-and-closure/ebook/151)

函数和Closure真的是不同的类型么？

Back to series (/series/functions-and-closure)

字号
字号
默认主题
金色主题
暗色主题

提起closure，如果你有过其他编程语言的经历，你可能会立即联想起一些类似的事物，例如：匿名函数、或者可以捕获变量的一对 {}，等等。但实际上，我们很容易搞混两个概念：Closure expression和Closure。它们究竟是什么呢？我们先从closure expression开始。

理解Closure Expressions

简单来说，closure expression就是函数的一种简写形式。例如，对于下面这个计算参数平方的函数：

```
func square(n: Int) -> Int {
    return n * n
}
```

我们也可以这样来定义：

```
let squareExpression = { (n: Int) -> Int in
    return n * n
}
```

而调用 square 和 squareExpression 的方法，是完全相同的：

```
square(2) // 4
squareExpression(2) // 4
```

并且，它们也都可以当作函数参数来使用：

```
let numbers = [1, 2, 3, 4, 5]
numbers.map(square) // [1, 4, 9, 16, 25]
numbers.map(squareExpressions) // [1, 4, 9, 16, 25]
```

在我们这个例子里，用于定义 squareExpression 的 {} 就叫做closure expression，它只是把函数参数、返回值以及实现统统写在了一个 {} 里。

没错，此时的 {} 以及 squareExpression 并不能叫closure，它只是一个closure expression。那么，为什么要有两种不同的方式来定义函数呢？最直接的理由就是，为了写起来更简单。Closure expression可以在定义它的上下文里，被不断简化，让代码尽可能呈现出最自然的语义形态。

例如，当我们把一个完整的closure expression定义在 map 参数里，是这样的：

```
numbers.map({ (n: Int) -> Int in
    return n * n
})
```

首先，Swift可以根据 numbers 的类型，自动推导出 map 中的函数参数以及返回值的类型，因此，我们可以在closure expression中去掉它：

```
numbers.map({ n in return n * n })
```

其次，如果closure expression中只有一条语句，Swift可以自动把这个语句的值作为整个expression的值返回，因此，我们还可以去掉 return 关键字：

```
numbers.map({ n in n * n })
```

第三，如果你觉得在closure expression中为参数起名字是个意义不大的事情，我们还可以使用Swift内置的 $0/1/2/3/4 这样的形式作为closure expression的参数替代符，这样，我们连参数声明和 in 关键字也都可以省略了：

```
numbers.map({ $0 * $0 })
```

第四，如果函数类型的参数在参数列表的最后一个，我们还可以把closure expression写在 () 外面，让它和其它普通参数更明显的区分开：

```
numbers.map() { $0 * $0 }
```

最后，如果函数只有一个函数类型的参数，我们甚至可以在调用的时候，去掉 ()：

```
numbers.map { $0 * $0 }
```

看到这里，你就应该知道当我们把closure expression用在它的上下文里，究竟有多方便了，相比一开始的定义，或者单独定义一个函数，然后传递给它，都好太多。但事情至此还没结束，相比这样：

```
numbers.sorted(by: { $0 > $1 }) // [5, 4, 3, 2, 1]
```

Closure expression还有一种更简单的形式：

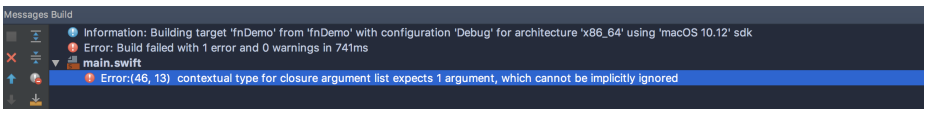
```
numbers.sorted(by: >) // [5, 4, 3, 2, 1]
```

这是因为， `numbers.sorted(by:)` 的函数参数是这样的： `(Int, Int) -> Bool` ，而Swift为 `Int` 类型定义的 `>` 操作符也正好和这个类型相同，这样，我们就可以直接把操作符传递给它，本质上，这和我们传递函数名是一样的。

另外，除了写起来更简单之外，closure expression还有一个副作用，就是默认情况下，我们无法忽略它的参数，编译器会对这种情况报错。来看个例子，如果我们要得到一个包含10个随机数的 `Array` ，最简单的方法，就是对一个 `CountableRange` 调用 `map` 方法：

```
(0...9).map { arc4random() } // !!! Error in swift !!!
```

这样看似很好，但是由于 `map` 的函数参数默认是带有一个参数的，在我们的例子里，表示range中的每个值，因此，如果我们在整个closure expression里都没有使用这个参数，Swift编译器就会提示我们下面的错误：



我们不能默认忽略closure expression中的参数，如果坚持如此，我们必须用 _ 明确表明这个意图：

```
(0...9).map { _ in arc4random() }
```

这也算是Swift为了类型和代码安全，利用编译器，为我们提供的一层保障。以上，就是和closure expression有关的内容，如你看到的一样，它就是函数的另外一种在上下文中更简单的写法，和用 `func` 定义的函数没有任何区别。

那么，究竟什么是closure呢?

究竟什么是Closure?

如果我们翻翻Wikipedia (https://en.wikipedia.org/wiki/Closure_(computer_programming))，就能找到下面的定义：*a closure is a record storing a function together with an environment*。

说的通俗一点，一个函数加上它捕获的变量一起，才算一个closure。来看个例子：

```
func makeCounter() -> () -> Int {
    var value = 0

    return {
        value += 1

        return value
    }
}
```

`makeCounter()` 返回一个函数，用来返回它被调用的次数。然后，我们分别定义两个计数器，并各自调用几次：

```
let counter1 = makeCounter()
let counter2 = makeCounter()

(0...2).forEach { _ in print(counter1()) } // 1 2 3
(0...5).forEach { _ in print(counter2()) } // 1 2 3 4 5 6
```

这样，三次调用 `counter1()` 会在控制台打印“123”，6次调用 `counter2()` 会打印“123456”。这说明什么呢?

首先，尽管从 makeCounter 返回后， value 已经离开了它的作用域，但我们多次调用 counter1 或 counter2 时， value 的值还是各自进行了累加。这就说明， makeCounter 返回的函数，捕获了 makeCounter 的内部变量 value 。

此时， counter1 和 counter2 就叫做closure，它们既有要执行的逻辑（把 value 加1），还带有其执行的上下文（捕获的 value 变量）。

其次， counter1 和 counter2 分别有其各自捕获的 value ，也就是其各自的上下文环境，它们并不共享任何内容。

理解了closure的含义之后，我们就知道了，closure expression和closure并不是一回事儿。然而，捕获变量是 {} 的专利么？实际上也不是，函数也可以捕获变量。

函数同样可以是一个Closure

还是之前 makeCounter 的例子，我们把返回的closure expression改成一个local function：

```
func makeCounter() -> () -> Int {
    var value = 0
    func increment() -> Int {
        value += 1
        return value
    }

    return increment
}
```

然后，就会发现，之前 counter1 和 counter2 的例子的执行结果，和之前是一样的：

```
(0...2).forEach { _ in print(counter1()) } // 1 2 3
(0...5).forEach { _ in print(counter2()) } // 1 2 3 4 5 6
```

所以，捕获变量这种行为，实际上，跟用 {} 定义函数也没关系。

What's next?

至此，我们应该对函数在Swift中的地位有更进一步的认识了。一方面， func 和closure expression都可以用来定义函数，它们只是形式上的不同；另一方面，无论是用哪种方式定义了函数，一旦其捕获了变量，函数和它捕获变量的上下文环境一起，就形成了一个closure。在我们通过几节了解了函数和Closure的基本用法之后。下一节，我们将看到内嵌函数捕获变量的另外一种用法：如何在多次函数调用之间，共享资源。

◀ Swift 3关于函数类型的一项重要提议
(https://www.boxueio.com/series/functions-and-closure/ebook/149)

通过Local function捕获变量共享资源 ▶
(https://www.boxueio.com/series/functions-and-closure/ebook/151)

What most schools do not teach (https://www.boxueio.com/what-most-schools-do-not-teach)
Dec 21, 2015

一个工作十年PM终创业的故事（一） (https://www.boxueio.com/founder-story)
May 8, 2015

泊学相关

关于泊学 ›

加入泊学 ›

泊学用户隐私以及服务条款 (HTTPS://WWW.BOXUEIO.COM/TERMS-OF-SERVICE)

版权声明 (HTTPS://WWW.BOXUEIO.COM/COPYRIGHT-STATEMENT)

联系泊学

Email: 10[AT]boxue.io (mailto:10@boxue.io)
QQ: 2085489246

友情链接 SwiftV (http://www.swiftv.cn) | Seay信息安全博客 (http://www.cnseay.com) | Swift.gg (http://swift.gg/) | Laravist (http://laravist.com/) | SegmentFault (https://segmentfault.com) | 靛青K的博客 (http://blog.dianqk.org/)