使用func和closure加工数据

是delegate protocol，还是callback?
(https://www.boxueio.com/series/functions-and-closure/ebook/156)

什么时候需要把参数自动转化为closure?
(https://www.boxueio.com/series/functions-and-closure/ebook/158)

被绝大多数人误会了的inout参数

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

字号
字号
默认主题
金色主题
暗色主题

在前面，讲到通过函数参数返回内容的时候，我们提到了函数参数默认都是常量，如果我们要在函数内修改参数，或者要通过参数返回内容，就要使用 inout 来修饰参数。例如这样：

```
func incremental(_ i: inout Int) -> Int {
    i = i + 1
    return i
}
```

然后，在调用 incremental 的时候，要在参数前加一个 & ：

```
var i = 0
incremental(&i) // 1
```

这样， i 的值，就会变成1。对于 & ，如果你之前有过C/C++的编程经验，一定会把它理解为是在传递变量 i 的地址，或者叫 i 的引用。这样，我们才能让修改在函数外生效。但事实并不如此，在Swift的官文文档里，我们可以找到一段这样的描述：

> An in-out parameter has a value that is passed in to the function, is modified by the function, and is passed back out of the function to replace the original value.

也就是说，管这个参数叫做 inout 而不是类似于ref之类的是有道理的，被 inout 修饰的参数只是被传递给函数，被修改后，再替换了初始值而已。当然，也许按引用传递是编译器采取的某种优化手段，但是你不能依赖这个特性。总之， inout 就真的只是in out而已。

哪些类型可以作为inout参数

理解了 inout 的工作方式之后，我们来看哪些类型的变量可以用作 inout 参数。用一句话总结就是：只有左值（l-value）才可以当作 inout 参数使用。那么，如何来区分左值（l-value）和右值（r-value）呢?

通常，左值需要用一块内存来表达，它的值就是内存区域中存放的内容，例如之前我们使用的变量 i 就需要一个4字节的内存空间；而右值则仅仅表示值本身，通常它们不需要特定的内存空间存储，编译器可以把它们优化成某种形式的符号在程序中被使用。

基于这样的规则，我们就不难理解哪些变量可以作为 inout 参数了。

变量可以、常量不行

如同之前的 incremental(&i) 一样，我们可以把 var 定义的变量作为 inout 参数，但 let 定义的常量却不行：

```
let i = 0
incremental(&i) // !!! Error !!!
```

对于上面的代码，编译器会给出类似：“Cannot pass immutable value as inout argument”的错误提示。

如果一个集合类型不是常量，它的下标操作符也可以作为 inout 参数：

```
var numbers = [1, 2, 3]
incremental(&numbers[0])
```

或者，如果一个自定义类型的属性同时有 get 和 set 方法，也可以作为 inout 参数：

```
struct Color {
    var r: Int
    var g: Int
    var b: Int
}

var red = Color(r: 254, g: 0, b: 0)
incremental(&red.r)
```

但是，对于一个只读的computed property，虽然它也是用 var 定义的，但是却不能作为 inout 参数：

```
struct Color {
    var r: Int
    var g: Int
    var b: Int

    var hex: Int {
        return r << 16 + g << 8 + b
    }
}

var red = Color(r: 255, g: 0, b: 0)

incremental(&red.hex) // !!! Error !!!
```

上面的代码里，尽管 hex 是通过 var 定义的，但它却是一个只读的computed property，因此，我们会得到和之前传递 let 类似的编译错误。

自定义操作符的inout参数

在Swift里， inout 参数还有一个特别照顾的情况，为了让操作符用起来更符合一般运算符号的特征，自定义操作符的 inout 参数在传递的时候，不需要使用 & 。

例如，对于已经在Swift 3中删除的 ++ 操作符，我们可以这样定义回来：

```
prefix func ++(i: inout Int) -> Int {
    i += 1
    return i
}
```

然后，就可以这样对 i 自增了：

```
++i // 2
```

此时， i 的值，就变成了2。

可以被修改，但却不能逃逸的inout参数

接下来，我们看一个在内嵌函数中访问 inout 参数的例子。我们可以内嵌函数中使用 inout 参数：

```
func doubleIncrement(_ i: inout Int) {
    func increment() {
        i += 1
    }

    [0, 1].forEach { _ in increment() }
}
```

当我们调用 doubleIncrement 时， i 的值就变成了4。

```
doubleIncrement(&i) // 4
```

但是，我们却不能通过内嵌函数，让 inout 参数逃离函数的作用域：

```
// !!! Error !!!
func increment(_ i: inout Int) -> () -> Void {
    return {
        i += 1
    }
}
```

对于上面的 `increment`，编译器会把我们拒之门外。在了解具体的原因之前，我们也不难想象为何会如此：我们在一开始的时候就提到了，`inout` 参数的语义并不是直接传递了外部变量的引用，而是在函数内部修改完成之后，再拷贝回去的。如果我们返回了一个closure带走了这个变量，当这个closure返回的时候，我们还需要重新再写回参数变量么？如果closure返回的时候，原先的参数变量已经不存在了呢？面对这些问题，最好的方式，就是编译器不允许你如此。

真的不能传递对象的引用么?

这一节最后，我们来讨论下对象引用的问题。既然 `inout` 并不如我们想象的按引用传递，`&` 也不像Objective-C一样表示获取对象的地址，在Swift里，就真的没办法获取对象引用了么?

实则不然，在Swift里，有一种情况，`&` 表示读取对象的引用，而不是执行 `inout` 语义。来看下面的例子:

```
func incrementByReference(_ pointer: UnsafeMutablePointer<Int>) {
    pointer.pointee += 1
}
```

这里，可以把 `UnsafeMutablePointer<Int>` 理解为C语言中的 `Int *`，于是，当我们要传递一个 `Int` 变量的地址时，就要使用 `&`：

```
incrementByReference(&i) // 5
```

看到了吧，这个时候 `&` 就是获取对象地址的含义了。但是，对于接受这类参数的函数，你要警惕一种情况，尽量不要让它们返回一个函数类型，否则，你的App就时刻面临崩溃的风险。我们继续看下面的例子，把之前的 `incrementByReference` 改成下面的样子:

```
func incrementByReference(
    _ pointer: UnsafeMutablePointer<Int>) -> () -> Int {
    return { _ in
        pointer.pointee += 1
        return pointer.pointee
    }
}
```

这次，它返回一个函数用来把参数指向的内存地址的值加1，然后，我们来看下面的用法:

```
let boom: () -> Int

if true {
    var j = 0
    boom = incrementByReference(&j)
}

boom() // !!! BOOM here
```

如果你不是运气特别好，通常当执行到 `boom()` 的时候，程序就会崩溃了。因为 `boom` 访问了一个可能已经不存在的地址空间。其实，这个例子，和刚才我们提到的编译器不允许返回的函数捕获 `inout` 参数是类似的。只不过这次，由于我们使用了类似原生指针这样的底层类型，逃过了编译器的检查，当然，就得自己为这些不安分的代码负责了。

所以，为了少给自己找麻烦，不要让接受指针参数的函数返回另外一个函数。

What's next?

以上就是和 `inout` 参数和 `&` 操作符有关的话题。它们的功能、用法以及最佳实践，应该比我们想象的复杂一些。但现在，你应该可以胸有成竹的处理各种和 `inout` 参数有关的应用了。接下来，我们将讨论函数参数的另外两个属性：`@autoclosure` 和 `@escaping`。它们到底解决什么问题呢?

⏮ 是delegate protocol，还是callback?
(https://www.boxueio.com/series/functions-and-closure/ebook/156)

什么时候需要把参数自动转化为closure? ⏭
(https://www.boxueio.com/series/functions-and-closure/ebook/158)

职场漂泊的你，每天多学一点。

从开发、测试到运维，让技术不再成为你成长的绊脚石。我们用打磨产品的精神去传播知识，把最新的移动开发技术，通过简单的图表， 清晰的视频，简明的文字和切实可行的例子一一向你呈现。让学习不仅是一种需求，也是一种享受。

泊学动态

一个工作十年PM终创业的故事（二） (https://www.boxueio.com/after-the-full-upgrade-to-swift3)
Mar 4, 2017

人生中第一次创业的"10有" (https://www.boxueio.com/founder-chat)
Jan 9, 2016

猎云网采访报道泊学 (http://www.lieyunwang.com/archives/144329)
Dec 31, 2015

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/)