理解值语义的自定义类型

都是修改对象属性惹的祸 (https://www.boxueio.com/series/understand-value-types/ebook/169)

不再只是“值替身”的enum (https://www.boxueio.com/series/understand-value-types/ebook/171)

定义更复杂的值 - struct

Back to series (/series/understand-value-types)

字号
字号
默认主题
金色主题
暗色主题

Swift提供了多种可以结构化存储数据的方式，我们在之前的视频中其实也都用过了，它们是: `struct` 、`enum` 和 `class` 。但和你在其他编程语言中的经验不同的是，Swift标准库中的绝大多数类型都是 `struct` ，甚至Foundation中的一些类都提供了它们在Swift中的 `struct` 版本，而 `enum` 和 `class` 只占据了其中份额很小的一部分。当然，这和Swift标准库中提供的数据类型有一部分关系。但无论如何，这说明了一个事实，就是 `struct` 在Swift中，占有举足轻重的地位。

应该在什么地方使用struct

在深入了解 `struct` 的各种细节之前，我们先来了解应该把 `struct` 用在哪些场景里。在平时的编程中，按照对象的生命周期形态，可以把我们使用的类型分成两大类：

一类是必须有明确生命周期的，它们必须被明确的初始化、使用、最后明确的被释放。例如：文件句柄、数据库连接、线程同步锁等等。这些类型的初始化和释放都不是拷贝内存这么简单，通常，这类内容，我们会选择使用 `class` 来实现。

另一类，则是没有那么明显的生命周期，例如：整数、字符串、URL等等。这些对象一旦被创建之后，就很少被修改，我们只是需要使用这些对象的值，用完之后，我们也无需为这些对象的销毁做更多额外的工作，只是把它们占用的内存回收就好了。这类内容，通常我们会选择使用 `struct` 或 `enum` 来实现。

在这一节，我们先来看 `struct` 。虽然之前已经使用过很多次了，但我们还是把和 `struct` 相关的内容整理一下。

struct的定义和初始化

我们从定义一个 `struct` 开始。例如，为了定义一个表示二维空间坐标的类型，我们可以这样：

```
struct Point {
    var x: Double
    var y: Double
}
```

这里，我们定义了两个属性， `x` 和 `y` ，表示 `Point` 的xy轴坐标，它们占用的内存空间，决定了一个 `Point` 对象的大小，因此，叫做stored properties，稍后，我们还会看到更多形式的属性。

定义好 `struct` 之后，我们就要了解初始化 `struct` 对象的方法了。在Swift里，这是一套被严谨设计的规则。

Memberwise initializer

首先，如果你不创建任何 `init` 方法，Swift编译器就会为你自动创建一个，让你可以逐个初始化 `struct` 中的每一个属性，这个 `init` 方法也叫做memberwise initializer。于是，我们可以像这样直接创建 `Point` 对象：

```
var pointA = Point(x: 100, y: 200)
```

Default initializer

其次，如果我们希望在创建 `Point` 对象的时候，可以自动给属性设置默认值，有两种方法可以实现这个目的。第一种，就是把每一个属性的默认值直接写在属性定义的地方：

```
struct Point {
    var x: Double = 0.0
    var y: Double = 0.0
}
```

这样，我们就可以在创建 `Point` 时，不指定参数了：

```
let origin = Point()
```

这种用法的一个要求就是，**必须为每一个属性都指定默认值**，因为Swift要求 `init` 方法必须初始化自定义类型的每一个属性。如果我们无法做到这一点，就只能自定义memberwise initializer，然后，给它的参数设置默认值：

```
struct Point {
    var x: Double
    var y: Double

    init(x: Double = 0.0, y: Double = 0.0) {
        self.x = x
        self.y = y
    }
}
```

这和之前我们直接给属性设置默认值的方法，效果是一样的。

Other initializer

第三，如果我们还希望可以这样创建 `Point`：

```
var pointB = Point((200, 200))
```

我们就只能依照期望的参数类型，来自定义对应的 `init` 方法：

```
struct Point {
    var x: Double
    var y: Double

    // ...
    init(_ pt: (Double, Double)) {
        self.x = pt.0
        self.y = pt.1
    }
}
```

这样，之前的初始化方式就可以正常工作了。但这里有一点是要特别注意的：**当我们自定义了 `init` 方法之后**，Swift就会认为我们要接手 `struct` 的创建工作，就不会再自动为我们创建默认的memberwise initializer了。所以，如果我们使用了之前直接为属性设置默认值的方式创建对象，此时就会导致编译错误。这时，我们就必须自己定义默认的 `init` 方法。

Type property

其实，对于一个 `struct` 经常会使用的“特殊值”，除了每次我们自己定义之外，还可以在 `struct` 中定义成type property。它们**不是 `struct` 对象的一部分**，因此，不会增加 `Point` 对象的大小，还可以产生方便优美的代码：

```
struct Point {
    // ...
    static let origin = Point((0, 0))
}
```

然后，当我们要使用原点时，就可以这样：

```
print(Point.origin) // Point(x: 0.0, y: 0.0)
```

怎么样？是不是看起来还不错。接下来，我们来了解 `struct` 作为一个值类型，究竟意味着什么。

理解struct的值语义

首先，我们定义一个 `Point` 变量：

```
var pointC = Point(x: 100, y: 100)
```

其次，为了观察这个变量被修改的事件，我们可以给它添加一个 `didSet` block。这样，只要 `pointC` 发生变化，我们就会在控制台看到通知：

```
var pointC = Point(x: 100, y: 100) {
    didSet {
        print("pointC changed: \(pointC)")
    }
}
```

第三，我们先直接让 pointC 变量等于一个其他变量：

```
pointC = pointB
```

就会在控制台看到：*pointC changed: Point(x: 200.0, y: 200.0)*的提示。这很好理解，因为我们修改了 pointC 变量的值。但是，下面这段代码也会触发 didSet 语句：

```
pointC.x += 100
```

此时，我们会在控制台看到：*pointC changed: Point(x: 300.0, y: 200.0)*的提示，你怎么理解呢？为什么修改了 pointC 的某一个属性Swift也会认为整个 pointC 变量被修改了呢?

> 如果 Point 是一个 class ，修改 x 是不会看到 didSet 结果的。

这就是值语义的本质，即便是字面上我们直接修改了 pointC 变量的某个属性，但实际执行的逻辑则是我们重新给 pointC 赋值了一个新的 Point 对象。所以，即便一个 struct 属性的类型是 var ，修改它在语义上也是创建一个新的 struct 对象。

当然，编译器可以选择直接在 pointC 的内存地址上完成修改，用来消除看似没必要的拷贝。但这只是编译器可以采取的一种优化手段，你并不能依赖这个特性。况且，稍后我们还会看到，对于实现了copy on write的值类型，修改它的属性就又是另外一个故事了。

为struct添加方法

除了为 struct 添加属性之外，我们还可以添加方法，只不过 struct 的方法，默认都是只读的。例如，计算两个 Point 之间的距离：

```
extension Point {
    func distance(to: Point) -> Double {
        let distX = self.x - to.x
        let distY = self.y - to.y

        return sqrt(distX * distX + distY * distY)
    }
}
```

然后，我们就可以这样计算某个 Point 对象到原点的距离了：

```
pointC.distance(to: Point.origin) // 360.56
```

而当我们定义一个直接移动X轴坐标点的方法时，会导致编译错误：

```
extension Point {
    func move(to: Point) {
        self = to
    }
}
```

我们必须使用 mutating 来修饰这样的方法：

```
extension Point {
    mutating func move(to: Point) {
        self = to
    }
}
```

这样，Swift编译器就会在所有的 mutating 方法第一个参数的位置，自动添加一个 inout Self 参数：

```
extension Point {
    mutating func moveTo(
        /*self: inout Self, */
        to: Point) {
        self = to
    }
}
```

因此，当我们把 pointC 移动到 pointA 时：

```
pointC.move(to: pointA)
```

这种对 struct 对象的修改仅仅是字面上的，至于 move 之后的 pointC 是否还是内存中之前的 pointC 对象，则完全取决于编译器对 inout 的实现了。

What's next?

以上，就是关于 struct 类型我们要了解的内容。其中最重要的两点，就是 init 方法的合成规则，以及值语义在 struct 上的表现。理解了它们之后，在下一节，我们来看Swift中的另外一个值类型：enum 。

◀ 都是修改对象属性惹的祸

(https://www.boxueio.com/series/understand-value-types/ebook/169)

不再只是“值替身”的enum ▶

(https://www.boxueio.com/series/understand-value-types/ebook/171)