理解值语义的自定义类型

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

如何为值类型实现Copy-On-Write？- II (https://www.boxueio.com/series/understand-value-types/ebook/173)

如何为值类型实现Copy-On-Write？- I

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

字号
字号
默认主题
金色主题
暗色主题

在之前我们讲到集合类型的时候，不止一次提到过，由于这些集合类型都是值类型，Swift对它们使用了copy on write的优化手段来避免了不必要拷贝行为。在这一节里，我们就通过为一个自定义的 `struct` 实现COW，来了解这个技术的实现原理。它听着有点儿复杂，但实现起来很简单。

COW适合用在什么场景?

在动手实现之前，我们先通过Swift中的 `Array` 对COW有一个感性的认识。当我们这样定义一个 `Array` 的时候：

```
var numbers1 = [1, 2, 3, 4, 5]
```

虽然 `Array` 是一个 `struct` ，但 `numbers1` 中的数字并不保存在 `numbers1` 对象里，它们会另外存储在系统堆内存中。 `numbers1` 对象里，只会保存指向堆内存的一个引用。我们可以用下面的代码确认这个事情：

```
MemoryLayout.size(ofValue: numbers1) // 8
```

可以看到，这段代码的返回的结果是8，也就是一个64位内存地址占用的空间，事实上，无论数组里存放多少元素，一个 `Array` 对象的大小都是一个内存地址的大小。

此时，如果我们复制一个 `numbers1` 对象：

```
var numbers2 = numbers1
```

由于 `Array` 实现了COW机制， `numbers1` 和 `numbers2` 会指向系统堆中的同一个位置。直到我们修改了其中的一个对象：

```
numbers2[0] = 11
```

这时， `numbers2` 就会拷贝 `numbers1` 的内容，并把数组第一个位置的值设置成11。但是，假设没有 `numbers2` ，分配出来的系统堆就只会有 `numbers1` 对象的一个引用。这时，先拷贝原数组的值，再进行修改就显得多余了，作为一项可以优化的手段，我们可以选择直接在 `numbers1` 的原始内存中修改。

以上谈到的这个机制，就是Swift中的copy on write。通过之前的描述你应该能感受到，不同的自定义类型很难存在通用的COW实现机制。因此，这不是一个Swift语言的福利，我们需要自己编写额外的代码。

并且，通过观察 `Array` 的工作机制不难发现，当一个值类型中包含引用类型的对象时，为了在拷贝对象时的值语义正确，我们必须明确处理被包含的引用对象的拷贝规则。因此，也可以说，COW并不是一个可有可无的优化手段，在某些情况下，还是我们必须思考和处理的问题。

接下来，我们就自己实现一个支持COW的 `MyArray` ，以此，加深对COW运行机制的了解。

一个简单粗暴的COW实现

首先，为了让所有的 `MyArray` 对象共享存储元素的空间，我们让它包含一个 `NSMutableArray` 对象：

```
struct MyArray {
    var data: NSMutableArray

    init(data: NSMutableArray) {
        self.data = data.mutableCopy() as! NSMutableArray
    }
}
```

然后，为了在操作 MyArray 对象时隐藏 data ，我们再给它添加一个插入元素的方法：

```
extension MyArray {
    func append(element: Any) {
        data.insert(element, at: self.data.count)
    }
}
```

在创建新的 MyArray 对象时，为了实现值语义，我们让 self.data 等于了 init 参数的一个拷贝。但这样做并没有COW的效果，我们看下面的例子：

```
let m = MyArray(data: [1, 2, 3])
let n = m

m.append(11)

m.data === n.data // true
```

在上面的代码里，尽管 m 和 n 都是常量，但 data 是一个引用类型，我们仍旧可以修改它引用的内容，这种修改并不会被认为是修改 MyArray 对象本身。不信，你回头去看，我们甚至都不需要使用 mutating 来修饰 append(:) 方法。

但是，把 m 拷贝到 n 之后，尽管我们修改了 m ，但 m 和 n 引用相等的比较结果，仍旧是 true 。也就是说， m 和 n 中的 data 仍旧是同一个对象。当然，这也不意外，毕竟我们没有特别处理拷贝 MyArray 对象时， data 引用内容的处理方式。

那么，究竟该如何实现COW的效果呢?

由于Swift并不像C++一样允许我们通过拷贝构造函数来明确定义对象的拷贝行为。我们只能在需要COW的属性上下功夫，把它用一个computed property封装起来。然后，把所有修改属性的操作，都交由这个computed property完成。

例如，我们给 MyArray.data 添加一个新的属性：

```
struct MyArray {
    var dataCOW: NSMutableArray {
        mutating get {
            data = data.mutableCopy() as! NSMutableArray
            return data
        }
    }
}
```

很简单，每当读取 dataCOW 的时候，我们就创建一个 data 的拷贝。但是，由于我们在 get 里修改了 data ，就像我们之前提到的，这也是一个修改 self 的行为，因此，我们也要使用 mutating 来修饰。

接下来，所有要对 data 的修改操作，我们可以使用 dataCOW 来完成。例如： append(:) 方法可以修改成这样：

```
extension MyArray {
    mutating func append(_ element: Any) {
        dataCOW.insert(element, at: self.data.count)
    }
}
```

这样， append(:) 就会在一个 data 的拷贝上添加元素了。并且，由于 append(:) 使用了 mutating 修饰，我们也无法再修改常量 MyArray 对象了。此时，编译器会对 m.append(11) 这行代码报错。这样就在实现了COW效果的同时，完美隐藏了 MyArray 内部使用了引用类型作为数据存储的事实。我们把 m 改成变量，这时之前的引用相等比较就会返回 false 了。

```
var m = MyArray(data: [1, 2, 3])
let n = m

m.append(11)

m.data === n.data // false here
```

怎么样，是不是这个实现方式比你想象的要简单的多？的确，这样可以工作，但是却很暴力，它存在一个明显的硬伤，当我们需要多次修改 MyArray 对象，而只需要最后的结果时，所有中间的拷贝操作就成了浪费。例如，我们通过 for 循环给 MyArray 添加内容：

```
for i in 1...10 {
    m.append(i)
}
```

如果你理解了 MyArray 的COW机制，就会立刻发现 for 循环每执行一次， MyArray.data 就会被拷贝一次。但中间过程的拷贝明显是没意义的。因此，这个方案有点儿过于简单粗暴了。

What's next?

其实，当 MyArray.data 只有一个引用的时候，我们可以直接在原始对象上修改，而不用拷贝整个对象。然而，究竟该怎么做呢？说起来虽然简单，但有不少细节的问题要处理，在下一节中，我们就来实现它。

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

如何为值类型实现Copy-On-Write？ - II ⏭
(https://www.boxueio.com/series/understand-value-types/ebook/173)