如何为值类型实现Copy-On-Write？ - II

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

字号
字号
默认主题
金色主题
暗色主题

有了上一节末尾提供的思路之后，我们就可以动手优化MyArray的COW实现了。为了判断对象是否只有一个引用，我们可以调用Swift标准库中的 `isKnownUniquelyReferenced` 方法，它的用法是这样的：

首先，对于Swift原生类对象，只有单一引用时返回 `true` ，否则返回 `false`：

```
class Demo1 {}

var nativeClassObj1 = Demo1()
isKnownUniquelyReferenced(&nativeClassObj1) // true

var nativeClassObj2 = nativeClassObj1
isKnownUniquelyReferenced(&nativeClassObj1) // false
```

其次，对于Objective-C中的类对象，总是返回 `false`：

```
var arrayObj: NSArray = [1, 2, 3]
isKnownUniquelyReferenced(&arrayObj) // false
```

了解了它的用法之后就不难发现，我们还不能直接使用 `isKnownUniquelyReferenced` 来判断 `MyArray.data` 的引用，它是一个Objective-C的类对象。而解决的办法，就是把它封装在一个Swift `class` 里：

```
final class Box<T> {
    var unbox: T

    init(_ unbox: T) {
        self.unbox = unbox
    }
}
```

这里， `Box` 专门用来封装 `T` 类型的对象。因此，我们使用了 `final` 关键字修饰它，阻止它被其他类继承，这样可以帮助编译期产生更有效率的代码。然后，我们就可以正常判断 `arrayObj` 的引用了：

```
var boxed = Box(arrayObj)
isKnownUniquelyReferenced(&boxed) // true
```

有了这个思路之后，我们就可以如法炮制的把 `MyArray.data` “打包”起来，在复制 `MyArray.data` 之前先判断下它的引用。为此，我们需要把之前的 `MyArray` 做一番修改。

首先，把 `MyArray.data` “打包”起来，并修改对应的 `init` 方法：

```
struct MyArray {
    var data: Box<NSMutableArray>

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

其次，我们在 `dataCOW` 中，加入对 `data` 的判断：

```
struct MyArray {
    // ...

    var dataCOW: NSMutableArray {
        mutating get {
            if !isKnownUniquelyReferenced(&data) {
                data = Box(
                    data.unbox.mutableCopy() as! NSMutableArray)
            }

            return data.unbox
        }
    }
}
```

这样，我们之前的测试代码仍旧可以正常工作，并且，我们还可以在对象只有一个引用时，优化掉不必要的拷贝行为了。

一个原生Array的特殊实现

理解了COW的实现思想之后，我们来看一个在Swift集合实现中不太容易被注意到的小细节，有时候，拷贝行为会发生在我们意想不到的地方，对此，你应该保持警惕。

首先，我们定义一个Swift原生数组，让它包含一个我们之前创建的 MyArray 类型的对象：

```
var array: [MyArray] = [MyArray(data: [1, 2, 3])]
```

为了观察到COW的效果，我们先在 dataCOW 的实现中，打印一个字符串：

```
struct MyArray {
    // ...

    var dataCOW: NSMutableArray {
        mutating get {
            if !isKnownUniquelyReferenced(&data) {
                data = Box(data.unbox.mutableCopy() as! NSMutableArray)
                print("copied")
            }

            return data.unbox
        }
    }
}
```

然后，当我们通过数组下标修改元素的时候，可以看到并不会触发COW：

```
array[0].append(4)
```

但当我们把数组元素拷贝出来再修改的时候，COW就触发了：

```
var item0 = array[0]
item0.append(4) // copied
```

这看上去很好理解对不对？下标操作符直接修改的是 array 对象内的元素，而 item0 修改的是 array[0] 的一个拷贝。但是，当你再观察 Dictionary 的场景时，就不会觉得那么自然了。首先，我们定义一个值为"MyArray"的 Dictionary：

```
var dictionary = ["key": MyArray(data: [1, 2, 3])]
```

当我们通过 key 修改 MyArray 的时候，你猜会发生什么？

```
dictionary["key"]?.append(4) // copied
```

没错，这次，下标操作符的访问竟然会触发COW。之所以会造成这种差别，是因为除了 Array 之外，其它集合类型的 [] 实现返回的都是集合内值对象的一个拷贝。而只有 Array 对它的元素自身使用了COW技术。为了了解这个技术的实现细节，大家可以在Accessor.rst (https://github.com/apple/swift/blob/73841a643c087e854a2f62c7e073317bd43af310/docs/proposals/Accessors.rst) 中找到完整的介绍，我们就不展开了。

Conclusion

以上，就是关于COW执行机制以及实现的讨论。当一个值类型中包含其它引用类型时，应该说，COW并不是一个可有可无的优化手段，而是一个我们为了实现正确值语义必须要认真考虑和处理的细节。特别是，当我们把这样的类型用在集合中的时候，就更是要理解元素操作的语义和细节。这样，才能尽可能写出语义正确并执行高效的代码。

⏮ 如何为值类型实现Copy-On-Write？ - I (https://www.boxueio.com/series/understand-value-types/ebook/172)

返回视频 ⏭ (/series/understand-value-types)

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/)