使用func和closure加工数据

OC中水土不服的运行时特性

(https://www.boxueio.com/series/functions-and-closure/ebook/152)

在复杂排序中处理optional

(https://www.boxueio.com/series/functions-and-closure/ebook/154)

如何通过类型系统模拟OC的运行时特性?

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

字号
字号
默认主题
金色主题
暗色主题

在上一节中，我们了解了在Swift中使用OC运行时特性的一些问题，并提供了一个解决问题的方向。这一节，我们就来实现这个过程。

从排序函数开始

为了模拟 NSSortDescriptor 的实现，我们得先从它的排序函数做起。简单来说，这就是一个接受两个同类型的参数，并且返回 Bool 的函数，我们可以用一个 typealias 来表示：

```
typealias SortDescriptor<T> = (T, T) -> Bool
```

于是，两个比较 String 的descriptor可以写成：

```
let stringDescriptor: SortDescriptor<String> = {
    $0.localizedCompare($1) == .orderedAscending
}
```

但有时，我们实际上要比较的内容，不是 T ，而是 T 的某个属性，例如，我们要比较上一节中 Episode 的长度：

```
let lengthDescriptor: SortDescriptor<Episode> = {
    $0.length < $1.length
}
```

观察这两个例子，如果我们要抽象 SortDescriptor 的创建过程，要解决两个问题：

首先，对于要排序的值，不能简单的认为就是 SortDescriptor 泛型参数的对象，它还有可能是这个对象的某个属性。因此，我们应该用一个函数来封装获取排序属性这个过程；

其次，对于排序的动作，有可能是 localizedCompare 这样的方法，也有可能是系统默认的 < 操作符，因此，我们同样要用一个函数来抽象这个比较的过程；

理解了这两点之后，我们就可以试着为 SortDescriptor ，创建一个工厂函数了：

```
func makeDescriptor<Key, Value>(
    key: @escaping (Key) -> Value,
    _ isAscending: @escaping (Value, Value) -> Bool
) -> SortDescriptor<Key> {

        return { isAscending(key($0), key($1)) }
}
```

在上面的代码里，我们使用 @escaping 修饰了用于获取 Value 以及排序的函数参数，这是因为在我们返回的函数里，使用了 key 以及 isAscending ，这两个函数都逃离了 makeDescriptor 作用域，而Swift 3里，作为参数的函数类型默认是不能逃离的，因此我们需要明确告知编译器这种情况。

然后，我们就可以这样来定义用于按 type 和 length 排序的descriptor：

```
let lengthDescriptor: SortDescriptor<Episode> =
    makeDescriptor(key: { $0.length }, <)

let typeDescriptor: SortDescriptor<Episode> =
    makeDescriptor(key: { $0.type }, {
        $0.localizedCompare($1) == .orderedAscending
    })
```

在上面这段代码里，相比 NSSortDescriptor 的版本，Swift的实现有了一点改进。我们使用了 { $0.length } 和 { $0.type } 这样的形式指定了要比较的属性。这样，当指定的属性和后面用于排序的方法使用的参数类型不一致的时候，编译器就会报错，避免了在运行时因为类型问题带来的错误。

有了这些descriptors，就离 NSSortDescriptor 的替代方案更进一步了。我们先试一下其中一个 descriptor：

```
episodes.sorted(by: typeDescriptor)
    .forEach { print($0) }
```

就可以在控制台看到已经按 type 进行排序了：

```
title 1 Free    520
title 2 Free    330
title 3 Free    240
title 4 Paid    500
title 5 Paid    260
title 6 Paid    390
```

合并多个排序条件

接下来，我们要继续模拟通过一个数组来定义多个排序条件的功能。怎么做呢？我们有两种选择：

- 通过 `extension Sequence`，添加一个接受 `[SortDescriptor<T>]` 为参数的 `sorted(by:)` 方法；
- 定义一个可以把 `[SortDescriptor<T>]` 合并为一个 `SortDescriptor<T>` 的方法。这样，就可以先合并，再调用 `sorted(by:)` 进行排序；

哪种方法更好呢？为了尽可能使用统一的方式使用Swift集合类型，我们还是决定采用第二种方式。

那么，如何合并多个descriptors呢？核心思想有三条，在合并 `[SortDescriptor]` 的过程中：

1. 如果某个descriptor可以比较出大小，那么后面的所有descriptor就都不再比较了；
2. 只有某个descriptor的比较结果为相等时，才继续用后一个descriptor进行比较；
3. 如果所有的descriptor的比较结果都相等，则返回 `false`；

我们来看代码：

```
func combine<T>(rules: [SortDescriptor<T>]) -> SortDescriptor<T> {
    return { l, r in
        for rule in rules {
            if rule(l, r) {
                return true
            }

            if rule(r, l) {
                return false
            }
        }

        return false
    }
}
```

在上面的代码里，只有一个技巧，就是我们使用了 `rule(l, r)` 和 `rule(r, l)` 同时为 `false` 的情况，模拟了 `r` 和 `l` 相等的情况。其余，就是我们之前提到的三点核心思想的实现，很简单。有了 `combine` 方法，我们就可以把之前的 `typeDescriptor` 和 `lengthDescriptor` 合并起来了：

```
let mixDescriptor = combine(rules:
    [typeDescriptor, lengthDescriptor])
```

然后，我们可以使用合并后的结果，对 `episodes` 进行排序：

```
episodes.sorted(by: mixDescriptor)
    .forEach { print($0) }
```

这样，我们就可以得到和之前 NSSortDescriptor 同样的结果了：

```
title 3 Free    240
title 2 Free    330
title 1 Free    520
title 5 Paid    260
title 6 Paid    390
title 4 Paid    500
```

阶段性总结

回顾下我们的Swift实现，整体过程是这样的：

首先，在Swift里，我们使用函数类型替代了OC中的 NSSortDescriptor 类，表示了一个排序规则：

```
typealias SortDescriptor<T> = (T, T) -> Bool
```

其次，我们使用函数类型替代了OC中的Key-Value coding和selector，来获取要排序的属性，和执行排序的selector：

```
func makeDescriptor<Key, Value>(
    key: @escaping (Key) -> Value,
    _ isAscending: @escaping (Value, Value) -> Bool
) -> SortDescriptor<Key> {

        return { isAscending(key($0), key($1)) }
}
```

第三，我们用类似的方式，创建了一个 [SortDescriptor<T>] 。不同的是，我们没有直接把这个数组传递给排序方法，而是把数组中所有的descriptor合并成了一个排序逻辑之后，再进行排序：

```
// 1. Create descriptors
let lengthDescriptor: SortDescriptor<Episode> =
    makeDescriptor(key: { $0.length }, >)

let typeDescriptor: SortDescriptor<Episode> =
    makeDescriptor(key: { $0.type }, {
        $0.localizedCompare($1) == .orderedAscending
    })

// 2. Combine descriptor array
let mixDescriptor = combine(rules:
    [typeDescriptor, lengthDescriptor])

// 3. Sort
episodes.sorted(by: mixDescriptor)
```

这样，我们不仅保留了 NSSortDescriptor 的编程思想，也充分利用了Swift是一门强类型语言的特性，尽可能在编译期保障代码安全。另外，通过这种方案，我们还去掉了对要排序类型的限制，现在，它可以是任意一个Swift的原生类型：

```
struct Episode: CustomStringConvertible {
    // The same as before
}
```

我们之前说过，类似 Episode 这样的类型，更适合用一个 struct ，现在，我们也终于可以如愿了。

What's next?

这一节，我们最核心的一个思想，就是函数除了可以当作加工数据的过程之外，函数类型自身，还可以当作一种数据类型来使用，通过合理利用函数类型，我们可以把一些运行时才能完成的事情，抽象到编译期进行安全检查，进而提高代码的可靠性。但我们的方案还有值得改进的地方：

1. 在 makeDescriptor 中，如果 key 返回的是optional，我们并没有处理这种情况；
2. 先 combine 再调用 sorted 或多或少有点儿不自然，毕竟这也算是一个实现的细节，如果我们在写一个library，这显然是没必要暴露给使用者的；

怎么办呢？在下一节中，我们就来改进它们。

◀ OC中水土不服的运行时特性
(https://www.boxueio.com/series/functions-and-closure/ebook/152)

在复杂排序中处理optional ▶
(https://www.boxueio.com/series/functions-and-closure/ebook/154)

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/)