What's new in Swift 4

SE-0156 Subtype existential

(https://www.boxueio.com/series/what-is-new-in-swift-4/ebook/239)

如何处理常见的JSON嵌套结构

(https://www.boxueio.com/series/what-is-new-in-swift-4/ebook/295)

使用Codable解析JSON

Back to series (/series/what-is-new-in-swift-4)

字号
字号
默认主题
金色主题
暗色主题

对于如何处理JSON，尽管之前也有很多优秀的第三方代码，但在Swift 4，Apple终于给出了官方答案。作为开始，我们先来看一些简单情况的处理。

感受下Swift 4的原生处理方式

假设，我们有一段表示泊学视频信息的JSON：

```
let response = """
{
    "title": "How to parse JSON in Swift 4",
    "series": "What's new in Swift 4",
    "creator": "Mars",
    "type": "free"
}
"""
```

为了把这段JSON存到本地的Model里，我们需要定义两个类型：

```
enum EpisodeType: String {
    case free,
    case paid
}

struct Episode {
    var title: String
    var series: String
    var creator: String
    var type: EpisodeType
}
```

接下来，为了可以把之前的 `response` 自动转换成 `Episode` 对象，需要满足两个条件：

- `Episode` 中的每个属性名，和JSON中的要一致，例如，它们都是 `title`， `series`，`creator` 和 `type`；
- 参与类型转换的所有的类型，都遵从 `Codable`，它的定义是这样的：`typealias Codable = Decodable & Encodable`。其中 `Decodable` 表示把JSON的字符串表示映射到一个Swift model，而 `Encodable` 则表示把Swift model映射成JSON的字符串表示；

于是，我们只要把 `Episode` 和 `EpisodeType` 的类型都改成这样：

```
enum EpisodeType: String, Codable { // ... }
struct Episode: Codable { // ... }
```

之前的 `response` 就可以自动映射到 `Episode` 对象了，无需我们再进行任何类型转换操作：

```
// 1. Create a data object
var data = response.data(using: .utf8)!
let decoder = JSONDecoder()

// 2. Decode the data
let episode = try! decoder.decode(
    Episode.self, from: data)

// 3. Get the result
print(episode)
```

在上面这段代码里，我先们创建了一个 `JSONDecoder` 对象，然后，最关键的部分，是我们直接调用 `decoder.decode` 方法完成字符串到类型的映射。只要把Model的类型作为第一个参数传递给它就好了，我们无须再根据每一个JSON中的Key，手动把 `AnyObject` 转换成对应的类型。

我们来看下 decode 的声明：

```
func decode<T>(_ type: T.Type,
    from data: Data) throws -> T where T : Decodable
```

可以看到，它返回的，就是第一个参数类型对应的对象。于是，在上面的例子里，就可以通过type inference得到变量 episode 的类型了。当然，由于这个转换是有可能失败的，因此， decode 可能抛出异常，这里，简单起见，我们直接使用了 try!，稍后，我们会专门提到这个异常的处理方法。

如何自定义JSON中的Key

怎么样，是不是很简单？至此一切工作顺利。但现实情况可不像例子中这么简单，有时候服务器返回的JSON中Key的命名方式，和Swift代码变量的命名方式并不一致，例如，在Swift里，我们习惯使用驼峰式的命名：

```
struct Episode: Codable {
    ...
    let createdBy: String
    ...
}
```

但服务器接受的JSON中，key通常是用下划线分割的：

```
let response = """
{
    ...
    "created_by": "Mars",
    ...
}
"""
```

为了解决这个问题，我们来想一下：为什么JSON key和Model属性名称一样的时候，就可以自动完成映射呢？实际上，为了实现这个过程，编译器会在遵从了 Codable 的类型中安插一个 enum CodingKeys: String, CodingKey 类型，并通过这个类型完成映射。

因此，为了自定义映射规则，我们只要重定义 CodingKeys 这个类型就好了：

```
struct Episode: Codable {
    // ...

    enum CodingKeys: String, CodingKey {
        case title
        case series
        case type
        case createdBy = "created_by"
    }
}
```

这样，就通过associated value的方式，完成了JSON key到model属性的映射。我们可以用下面的代码来试一下：

```
let episode = Episode(
    title: "How to parse JSON in Swift 4",
    series: "What's new in Swift 4",
    createdBy: "Mars",
    type: .free)

let encoder = JSONEncoder()
data = try! encoder.encode(episode)
print(String(data: data, encoding: .utf8)!)
```

执行一下就可以看到，当我们把 episode 编码成JSON字符串的时候，就会得到下面这样的结果了：

```
{"series":"What's new in Swift 4","title":"How to parse JSON in Swift 4",
"created_by":"Mars","type":"free"}
```

当然，为了让这个结果打印出来好看一些，我们还可以设置 JSONEncoder 的一些属性：

```
encoder.outputFormatting = .prettyPrinted
data = try! json.encode(episode)
```

这样，打印出来的结果就会变成这样：

```
{
  "series" : "What's new in Swift 4",
  "title" : "How to parse JSON in Swift 4",
  "created_by" : "Mars",
  "type" : "free"
}
```

处理JSON中的日期

至此，我们的JSON都还比较简单，因为它包含的值对应到model中，都是字符串类型。但实际情况并不如此，在JSON里还会用字符串的形式包含其他类型的值，例如时间。我们给 Episode 新增一个属性：

```
struct Episode: Codable {
    // ...
    let createdAt: Date

    enum CodingKeys: String, CodingKey {
        // ...
        case createdAt = "created_at"
    }
}
```

当我们重新编码 episode 对象的时候：

```
let episode = Episode(
    title: "How to parse JSON in Swift 4",
    series: "What's new in Swift 4",
    createdBy: "Mars",
    type: .free,
    createdAt: Date())

data = try! encoder.encode(episode)
print(String(data: data, encoding: .utf8)!)
```

就会看到下面这样的结果：

```
{
  "series" : "What's new in Swift 4",
  "title" : "How to parse JSON in Swift 4",
  "created_by" : "Mars",
  "type" : "free",
  "created_at" : 525144946.403841
}
```

可以看到，编码过的日期是个奇怪的浮点数，为了让它变成我们习惯阅读的格式，我们同样需要设定 JSONEncoder 的选项：

```
encoding.dateEncodingStrategy = .iso8601
```

这样，编码出来的结果就好看多了：

```
{
  "series" : "What's new in Swift 4",
  "title" : "How to parse JSON in Swift 4",
  "created_by" : "Mars",
  "type" : "free",
  "created_at" : "2017-08-23T01:42:42Z"
}
```

但是，在编码出来的结果中还包含了字符T和Z，如果我们要去掉它，可以还可以把 dateEncodingStrategy 设置成 .custom ，但是这需要我们额外掌握不少内容，因此等后面我们讲到Container概念的时候，再来回顾这个用法。

处理JSON中的浮点数

JSON中另外一类非字符串的内容，是浮点数。例如，我们给 Episode 再添加一个表示时长的属性 duration：

```
struct Episode: Codable {
    /// ...
    let duration: Float

    enum CodingKeys: String, CodingKey {
        /// ...
        case duration
    }
}
```

然后，当我们解码下面这段JSON的时候：

```
{
    "title": "How to parse JSON in Swift 4",
    "series": "What's new in Swift 4",
    "created_by": "Mars",
    "type": "free",
    "created_at": "2017-08-23T01:42:42Z",
    "duration": "6.5"
}
```

就会自动把它转换成一个 Episode 对象了：

```
data = response.data(using: .utf8)!
decoder.dateDecodingStrategy = .iso8601
let episode = decoder.decode(Episode.self, from: data)
```

但当我们要处理一些“特殊值”的时候，就不这么简单了，例如 NaN， +Infinity， -Infinity 等。当JSON字符串中存在着这类数值的时候，它们就无法自动转换成Swift中的 Float 或 Double 了。为此，我们也需要定义相关的转换规则：

```
decoder.nonConformingFloatDecodingStrategy =
    .convertFromString(
        positiveInfinity: "+Infinity",
        negativeInfinity: "-Infinity",
        nan: "NaN")
```

然后，当由于某种原因服务器返回的 duration 是 NaN 的时候，：

```
{
    "title": "How to parse JSON in Swift 4",
    "series": "What's new in Swift 4",
    "created_by": "Mars",
    "type": "free",
    "created_at": "2017-08-23T01:42:42Z",
    "duration": "NaN"
}
```

当我们查看 episode.duration 的时候，就会看到它的值是 nan 了：

```
dump(episode.duration)
// - nan
```

处理JSON中的base64编码

接下来要处理的一类数据是model中的二进制数据，它们通常是服务器返回的一段base 64编码。例如：

```
{
    ...
    "origin": "Ym94dWVpby5jb20="
}
```

其中， origin 是一段base 64编码过的值，它的原始值是boxueio.com。现在，为了在model中保存这个字段，我们给 Episode 添加一个属性：

```
struct Episode: Codable {
    /// ...
    var origin: Data

    enum CodingKeys: String, CodingKey {
        /// ...
        case origin
    }
}
```

接下来，为了能把JSON中的 `origin` 直接解码并保存到 `Episode.origin` ，我们同样需要配置下 `JSONDecoder`：

```
/// ...
decoder.dataDecodingStrategy = .base64
let v = try! decoder.decode(Episode.self, from: data)

dump(String(data: v.origin, encoding: .utf8)!)
```

这样，当我们查看 dump 结果的时候，就可以直接得到*boxueio.com*了。

处理URL

最后一个要介绍的，是JSON中的URL， `JSONDecoder` 可以直接把它转成Swift中的 URL 对象。例如，服务器返回的JSON中，包含一个URL值：

```
{
    ...
    "url": "boxueio.com"
}
```

然后，我们继续给 `Episode` 添加对应的属性：

```
struct Episode: Codable {
    /// ...
    let url: URL

    enum CodingKeys: String, CodingKey {
        /// ...
        case url
    }
}
```

现在， `JSONDecode` 就可以处理结果中的url key了。

What's next?

以上，就是这一节的内容，我们初步掌握了通过 `Codeable` 在JSON和各种Swift对象之间的映射方法。但至此，我们的JSON都还是“扁平”的，而实际环境中，服务器返回的JSON通常是包含各种嵌套结构的。下一节，我们就来看如何处理几种不同的嵌套结构。

SE-0156 Subtype existential
(https://www.boxueio.com/series/what-is-new-in-swift-4/ebook/239)

如何处理常见的JSON嵌套结构
(https://www.boxueio.com/series/what-is-new-in-swift-4/ebook/295)

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/)