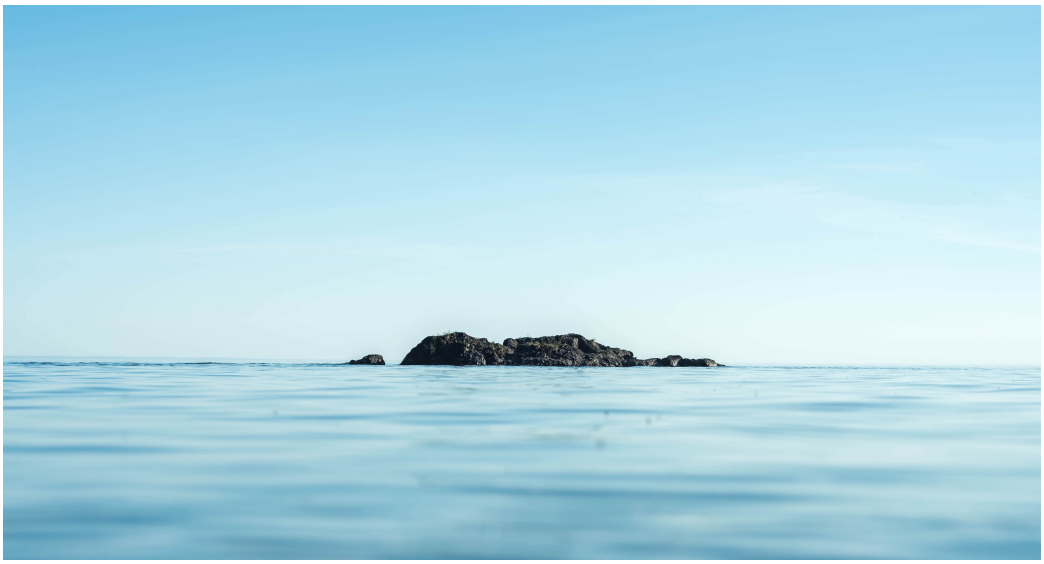
Swift 4 JSON 解析指南

2017-06-30 11:39 编辑：枣泥布丁 分类：Swift 来源：BigNerdCoding 5 4169

Swift 4 Codable API



Apple 终于在 Swift 4 的 Foundation 的模块中添加了对 JSON 解析的原生支持。

虽然已经有很多第三方类库实现了 JSON 解析，但是能够看到这样一个功能强大、易于使用的官方实现还是不免有些兴奋。

值得注意的是，官方的实现方式适用于任何 Encoder/Decoder ，例如 PropertyListEncoder 。当然如果你需要 XML 格式的内容，可以进行自定义实现。在接下来的内容中，我们将专注于 JSON 格式的解析，因为这是 iOS 开发中最常见的数据格式。

基础

如果你的 JSON 数据结构和你使用的 Model 对象结构一致的话，那么解析过程将会非常简单。

下面是一个 JSON 格式的啤酒说明：

```
{
    "name": "Endeavor",
    "abv": 8.9,
    "brewery": "Saint Arnold",
    "style": "ipa"
}
```

热门资讯


多年iOS开发经验总结(一)
点击量 12320


33 款主宰 2017 iOS 开发的开源库
点击量 11858


iOS:怎样创建一个好的App目录结构
点击量 6086


Stack Overflow 2017 开发者调查报告（程序
点击量 4523


我们来谈谈iOS 11第二个开发者测试版值不值
点击量 3786


iOS面试之坎坷之路到放弃
点击量 3183


丁香园iOS电话面试问题总结
点击量 2969


对比 | Android与IOS职位需求，哪个更大?
点击量 2927


Swift 4 JSON 解析指南
点击量 2759


iOS模式详解runtime面试工作
点击量 2360

综合评论

作者说的很良心。看一些评论，不知道有些人是不是傻，哈哈
孔思哲 评论了 你还在使用百度搜索? ...

装个xp虚拟机就行了，何必这么麻烦
yan127422 评论了 Mac下也能用抓包工具Fiddler...

mk
Esvn 评论了 多年iOS开发经验总结(一)...

那播放进度是怎么实现的,能否解答一下
tatatata 评论了 音频框架TheAmazingAudioEngine...

只能说作者对MVVM的理念理解不够，把网络请求，数据库操作业务处理等都

对应的 Swift 数据结构如下：

```
enum BeerStyle : String {
    case ipa
    case stout
    case kolsch
    // ...
}

struct Beer {
    let name: String
    let brewery: String
    let style: BeerStyle
}
```

为了将 JSON 字符串转化为 Beer 类型的实例，我们需要将 Beer 类型标记为 Codable。

Codable 实际上是 Encodable & Decodable 两个协议的组合类型，所以如果你只需要单向转换的话，你可以只选用其中一个。该功能也是 Swift 4 中引入的最重要新特性之一。

Codable 带有默认实现，所以在大多数情形下，你可以直接使用该默认实现进行数据转换。

```
enum BeerStyle : String, Codable {
   // ...
}

struct Beer : Codable {
   // ...
}
```

下面只需要创建一个解码器：

```
let jsonData = jsonString.data(encoding: .utf8)!
let decoder = JSONDecoder()
let beer = try! decoder.decode(Beer.self, for: jsonData)
```

这样我们就将 JSON 数据成功解析为了 Beer 实例对象。因为 JSON 数据的 Key 与 Beer 中的属性名一致，所以这里不需要进行自定义操作。

需要注意的是，这里直接使用了 try! 操作。因为这里只是简单示例，所以在真实程序中你应该对错误进行捕获并作出对应的处理。

但是，现实中不可能一直都是完美情形，很大几率存在 Key 值与属性名不匹配的情形。

自定义键值名

通常情形下，API 接口设计时会采用 snake-case 的命名风格，但是这与 Swift 中的编程风格有着明显的差异。

为了实现自定义解析，我们需要先去看下 Codable 的默认实现机制。

默认情形下 Keys 是由编译器自动生成的枚举类型。该枚举遵守 CodingKey 协议并建立了属性和编码后格式之间的关系。

为了解决上面的风格差异需要对其进行自定义，实现代码：

```
struct Beer : Codable {
      // ...
```

```
    enum CodingKeys : String, CodingKey {
        case name
        case abv = "alcohol_by_volume"
        case brewery = "brewery_name"
        case style
    }
}
```

现在我们将 Beer 实例转化为 JSON ，看看自定义之后的 JSON 数据格式：

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

输出如下：

```
{"style":"ipa","name":"Endeavor","alcohol_by_volume":8.8999996185302734,"brewery_name":"Saint Arnol
```

上面的输出格式对阅读起来并不是太友好。不过我们可以设置 JSONEncoder 的 **outputFormatting** 属性来定义输出格式。

默认 **outputFormatting** 属性值为 **.compact**，输出效果如上。如果将其改为 **.prettyPrinted** 后就能获得更好的阅读体检。

```
encoder.outputFormatting = .prettyPrinted
```

效果如下：

```
{
  "style" : "ipa",
  "name" : "Endeavor",
  "alcohol_by_volume" : 8.8999996185302734,
  "brewery_name" : "Saint Arnold"
}
```

JSONEncoder 和 JSONDecoder 其实还有很多选项可以自定义设置。其中有一个常用的需求就是自定义时间格式的解析。

时间格式处理

JSON 没有数据类型表示日期格式，因此需要客户端和服务端对序列化进行约定。通常情形下都会使用 ISO 8601 日期格式并序列化为字符串。

提示：nsdateformatter.com 是一个非常有用的网站，你可以查看各种日期格式的字符串表示，包括 ISO 8601。

其他格式可能是参考日期起的总秒（或毫秒）数，并将其序列化为 JSON 格式中的数字类型。

之前，我们必须自己处理这个问题。在数据结构中使用属性接收该字符串格式日期，然后使用 DateFormatter 将该属性转化为日期，反之亦然。

不过 JSONEncoder 和 JSONDecoder 自带了该功能。默认情况下，它们使用 .deferToDate 处理日期，如下：

```
struct Foo : Encodable {
    let date: Date
}

let foo = Foo(date: Date())
try! encoder.encode(foo)
```

```
{
  "date" : 519751611.12542897
}
```

当然，我们也可以选用 **.iso8601** 格式：

```
encoder.dateEncodingStrategy = .iso8601
{
  "date" : "2017-06-21T15:29:32Z"
}
```

其他日期编码格式选择如下：

- **.formatted(DateFormatter)** - 当你的日期字符串是非标准格式时使用。需要提供你自己的日期格式化器实例。
- **.custom((Date, Encoder) throws** -> Void) - 当你需要真正意义上的自定义时，使用一个闭包进行实现。
- **.millisecondsSince1970**、 **.secondsSince1970** - 这在 API 设计中不是很常见。 由于时区信息完全不在编码表示中，所以不建议使用这样的格式，这使得人们更容易做出错误的假设。

对日期进行 Decoding 时基本上是相同的选项，但是 .custom 形式是 .custom((Decoder) throws -> Date)，所以我们给了一个解码器并将任意类型转换为日期格式。

浮点类型处理

浮点是 JSON 与 Swift 另一个存在不匹配情形的类型。如果服务器返回的事无效的 "NaN" 字符串会发生什么？无穷大或者无穷大？这些不会映射到 Swift 中的任何特定值。

默认的实现是 .throw，这意味着如果上述数值出现的话就会引发错误，不过对此我们可以自定义映射。

```
{
   "a": "NaN",
   "b": "+Infinity",
   "c": "-Infinity"
}
struct Numbers {
  let a: Float
  let b: Float
  let c: Float
}
decoder.nonConformingFloatDecodingStrategy =
  .convertFromString(
      positiveInfinity: "+Infinity",
      negativeInfinity: "-Infinity",
      nan: "NaN")

let numbers = try! decoder.decode(Numbers.elf, from: jsonData)
dump(numbers)
```

上述处理后：

```
__lldb_expr_71.Numbers
  - a: inf
  - b: -inf
  - c: nan
```

当然，我们也可以使用 JSONEncoder 的 nonConformingFloatEncodingStrategy 进行反向操作。

虽然大多数情形下上述处理不太可能出现，但是以防万一也不给过。

Data 处理

有时候服务端 API 返回的数据是 base64 编码过的字符串。

对此，我们可以在 JSONEncoder 使用以下策略：

- .base64
- .custom((Data, Encoder) throws -> Void)

反之，编码时可以使用：

- .base64
- .custom((Decoder) throws -> Data)

显然，.base64 时最常见的选项，但如果需要自定义的话可以采用 block 方式。

Wrapper Keys

通常 API 会对数据进行封装，这样顶级的 JSON 实体 始终是一个对象。

例如：

```
{
  "beers": [ {...} ]
}
```

在 Swift 中我们可以进行对应处理：

```
struct BeerList : Codable {
    let beers: [Beer]
}
```

因为键值与属性名一致，所有上面代码已经足够了。

Root Level Arrays

如果 API 作为根元素返回数组，对应解析如下所示：

```
let decoder = JSONDecoder()
let beers = try decoder.decode([Beer].self, from: data)
```

需要注意的是，我们在这里使用 Array 作为类型。只要 T 可解码，Array 就可解码。

Dealing with Object Wrapping Keys

另一个常见的场景是，返回的数组对象里的每一个元素都被包装为字典类型对象。

```
[
  {
    "beer" : {
      "id": "uuid12459078214",
      "name": "Endeavor",
      "abv": 8.9,
      "brewery": "Saint Arnold",
      "style": "ipa"
    }
```

```
  }
]
```

你可以使用上面的方法来捕获此 Key 值，但最简单的方式就是认识到该结构的可编码的实现形式。

如下：

```
[[String:Beer]]
```

或者更易于阅读的形式：

```
Array
```

与上面的 Array 类似，如果 K 和 T 是可解码 Dictionary就能解码。

```
let decoder = JSONDecoder()
let beers = try decoder.decode([[String:Beer]].self, from: data)
dump(beers)
 1 element
  ? 1 key/value pair
    ? (2 elements)
      - key: "beer"
      ? value: __lldb_expr_37.Beer
        - name: "Endeavor"
        - brewery: "Saint Arnold"
        - abv: 8.89999962
        - style: __lldb_expr_37.BeerStyle.ipa
```

更复杂的嵌套

有时候 API 的响应数据并不是那么简单。顶层元素不一定只是一个对象，而且通常情况下是多个字典结构。

例如：

```
{
    "meta": {
        "page": 1,
        "total_pages": 4,
        "per_page": 10,
        "total_records": 38
    },
    "breweries": [
        {
            "id": 1234,
            "name": "Saint Arnold"
        },
        {
            "id": 52892,
            "name": "Buffalo Bayou"
        }
    ]
}
```

在 Swift 中我们可以进行对应的嵌套定义处理：

```
struct PagedBreweries : Codable {
    struct Meta : Codable {
        let page: Int
        let totalPages: Int
        let perPage: Int
        let totalRecords: Int
        enum CodingKeys : String, CodingKey {
            case page
            case totalPages = "total_pages"
            case perPage = "per_page"
            case totalRecords = "total_records"
        }
    }

    struct Brewery : Codable {
        let id: Int
        let name: String
    }

    let meta: Meta
    let breweries: [Brewery]
}
```

该方法的最大优点就是对同一类型的对象做出不同的响应（可能在这种情况下，“brewery” 列表响应中只需要 id 和 name 属性，但是如果查看详细内容的话则需要更多属性内容）。因为该情形下 Brewery 类型是嵌套的，我们依旧可以在其他地方进行不同的 Brewery 类型实现。

结论

Swift 4 中基础 Codable API 的内容已经介绍差不多了。更多的内容可以查看 Codable.swift、Using JSON with Custom Types 。

上一篇：Swift 开发中，为什么要远离 Heap?

相关资讯

回顾Swift 3, 展望Swift 4

我来说两句

您还没有登录！请 登录 或 注册

所有评论（5）

jsonbon 2017-07-04 10:25:43

用我家霉霉做图片 立马把我吸引过来了

0 0 回复

ljz2012008 2017-07-04 01:53:06

老婆，么么哒

0 0 回复

HL灬蓝莓 2017-07-01 03:19:55

NND。。。。这让我等了多久啊？？？？ 让我期待了多久的啊。。。。。。终。。。终于。。。出来了啊

1 0 回复

wy001 2017-06-30 20:54:40

struct？不能加方法吗？

0 0 回复

真时光人 2017-06-30 08:52:09

这tmd。。。太tmd给力了。。。这省多少事啊

0 21 回复