RxSwift - step by step

为什么RxSwift也需要flatMap (https://www.boxueio.com/series/rxswift-101/ebook/267)

App demo II 使用map/flatMap简化代码 (https://www.boxueio.com/series/rxswift-101/ebook/270)

App demo I 一个Alamofire router的实现

Back to series (/series/rxswift-101)

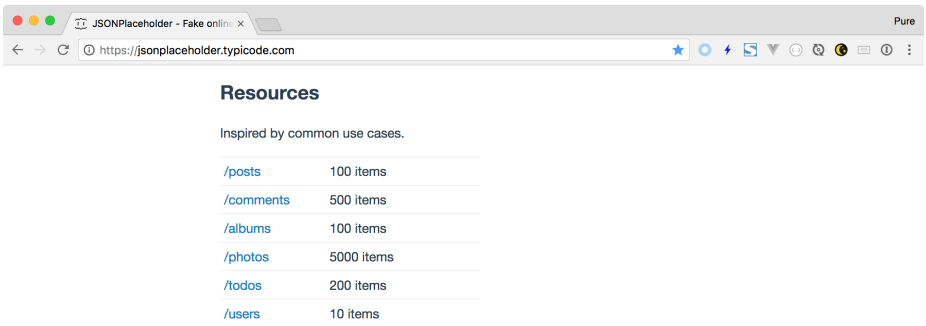
字号
字号
默认主题
金色主题
暗色主题

在接下来的两节内容里，我们通过一个App demo，加深对*Transform operators*用法的了解。

访问源代码 (https://github.com/puretears/my-todo-rx-demo/tree/master/Starter)

JSONPlaceholder

在开始之前，先介绍一个可以帮助我们测试REST接口的网站：JSONPlaceholder (https://jsonplaceholder.typicode.com/)：



就像图中所示的这样，它提供了很多常用功能需要的REST接口。例如，直接GET请求 `/todos` ，就会得到类似下面这样的JSON结果：

```
[
  {
    "userId": 1,
    "id": 1,
    "title": "delectus aut autem",
    "completed": false
  },
  {
    "userId": 1,
    "id": 2,
    "title": "quis ut nam facilis et officia qui",
    "completed": false
  },
  ...
]
```

当然，这些信息没有什么实际的意义，仅仅是为了测试接口，其中：

- `id` - 表示保存todo内容的主键；
- `userId` - 表示创建todo的不同用户；
- `title` - 表示Todo的标题；
- `completed` - 表示Todo的完成状态；

它们都很好理解，我们就不多说了。除此之外，大家还可以在网站上找到对应URL的不同REST请求方式，大家可以自己研究下。在我们的例子里，简单起见，我们只是通过 `GET /todos` 得到所有的todo列表，然后，用一个table view把这些todo显示出来。

App Demo Template

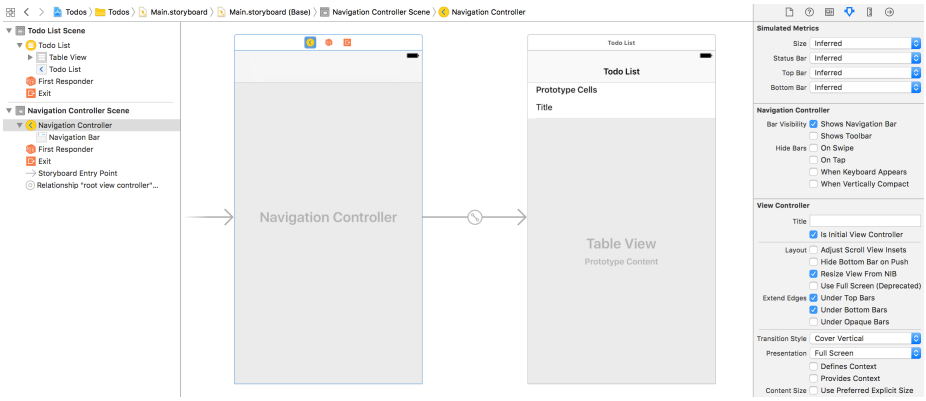
接下来，是项目的起始模板，我们创建了一个Single View Application，并做了以下修改：

首先，为了方便请求，在 `Podfile` 中，除了RxSwift，我们还引入了Alamofire：

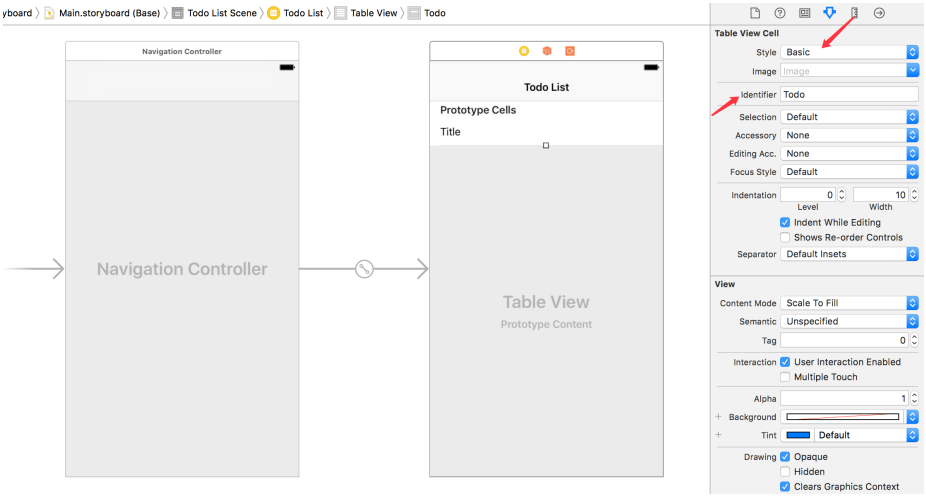
```
platform :ios, '10.0'

target 'Todos' do
  use_frameworks!
  pod 'RxSwift',    '~> 3.0'
  pod 'RxCocoa',    '~> 3.0'
  pod 'Alamofire',  '~> 4.4'
end
```

其次，在*Main.storyboard*中，删掉了默认的view controller，并添加了一个嵌套在navigation controller中的 `UITableViewController`；



第三，在 `UITableView` 中，把cell的style改成了*Basic*，identifier设置成*Cell*；



第四，添加了一个*Todo.swift*，在这里，定义了和 `/todos` 接口返回内容对应的Model：

```
class Todo {
    var id: UInt?
    var title: String
    var completed: Bool

    init(id: UInt, title: String, completed: Bool) {
        self.id = id
        self.title = title
        self.completed = completed
    }
}
```

> 简单起见，我们忽略掉了返回结果中的 `userId` 部分，假设所有的Todo都是同一个用户创建的。

除了这个默认的memberwise init方法之外，为了根据Alamofire返回的内容直接创建 Todo 对象，我们还定义了另外一个init方法。它根据服务器返回的JSON中的字段值，来初始化 Todo 的每个属性。

```
required init?(json: [String: Any]) {
    guard let todoId = json["id"] as? UInt,
        let title = json["title"] as? String,
        let completed = json["completed"] as? Bool else {
        return nil
    }

    self.id = todoId
    self.title = title
    self.completed = completed
}
```

为了方便调试，我们让它遵从了 `CustomStringConvertible` ，它把 Todo 的所有属性放在一行上，返回了一个字符串对象；

```
extension Todo: CustomStringConvertible {
    var description: String {
        return "ID: \(self.id ?? 0), " +
                "title: \(self.title), " +
                "completed: \(self.completed)"
    }
}
```

第五，删掉了默认的*ViewController.swift*，创建了一个*TodoListViewController.swift*，并在其中添加了一个 `TodoListViewController` 作为和UI对应的view controller；

第六，给 `TodoListViewController` 添加了一个属性，作为table view的data source。并实现了对应的初始化table cell的方法：

```
class TodoListViewController: UITableViewController {
    var todoList = [Todo]()

    /// ...
}
```

最后，添加了一个*TodoRouter.swift*，它包含了一个 `enum TodoRouter` ，用来路由所有到 `/todos` 的REST请求，它可以有效简化我们的网络编程代码。你可能还不太了解这种基于router的思路，别担心，接下来，我们就仔细讲一下这个 `TodoRouter` 。

理解网络请求的执行逻辑

为什么需要一个router

当我们使用Alamofire发送网络请求的时候，有两不同的方式。

第一种，是我们普遍熟悉的用法。例如，要获取所有todo的列表，可以这样：

```
let todoList: String =
    "https://jsonplaceholder.typicode.com/todos"

Alamofire.request(todoList)
    .responseJSON { response in
        /// Handle response here
    }
```

这时，如果我们要获取每个Todo的具体内容，就得这样：

```
let todoDetail: String =
    "https://jsonplaceholder.typicode.com/todos/1"

Alamofire.request(todoDetail)
    .responseJSON { response in
        /// Handle response here
    }
```

或者，如果我们要使用POST方法创建Todo，就得这样：

```
let todos: String =
    "https://jsonplaceholder.typicode.com/todos"
let newTodo: [String: Any] = [
    "title": "My First Post",
    "completed": 0,
    "userId": 1]

Alamofire.request(todos,
    method: .post,
    parameters: newTodo,
    encoding: JSONEncoding.default)
    .responseJSON { response in
        /// Handle response here
    }
```

可以看到，为了发起不同的请求，我们要准备不同的地址，不同的请求参数，调用不同版本的 `Alamofire.request` 方法。于是，用不了多少请求，我们的代码看起来就不那么舒服了。

实际上， `Alamofire.request` 还有另外一个版本，它只接受一个 `URLRequestConvertible` 参数。看起来要比我们之前调用的版本统一。因此，只要我们提前把不同类型的请求封装成 `URLRequestConvertible` 再调用 `request` ，代码就会干净很多了。而这，就是我们打造一个router的思路。

创建一个REST router

由于我们的router要表达不同类型的请求，因此，把它定义成 enum 最合适了。在*TodoRouter.swift*里，添加下面的代码：

```
enum TodoRouter {
    static let baseURL: String =
        "https://jsonplaceholder.typicode.com/"

    case get(Int?)
    /// TODO: Add other HTTP methods here
    /// Such as case post([[String: Any]])
}
```

由于暂时我们只需要处理一个请求，因此，这里只定义了一个 `case get` ，它有一个 `Int?` 类型的关联值，为 `nil` 时，表示要获取所有todo的列表；为 `Int` 时，表示要获取某一个具体的todo信息。

接下来，为了让 `TodoRouter` 理解 `case get` 的含义，并且，让它成为 `Alamofire.request` 的参数，我们得让它遵从 `URLRequestConvertible` ，这个 `protocol` 只有一个要求，就是实现 `asURLRequest` 方法：

```
extension TodoRouter: URLRequestConvertible {
    func asURLRequest() throws -> URLRequest {
    }
}
```

从它的签名就可以看到，它没有参数，并返回一个 `URLRequest` 对象。因此，我们只要在 `asURLRequest` 里，返回各种不同请求需要的 `URLRequest` 对象就好了。这个过程分为四部分：

第一部分，是生成Alamofire中对应的 `HTTPMethod` 对象，由于现在我们只有一个GET请求，就可以这样：

```
func asURLRequest() throws -> URLRequest {
    var method: HTTPMethod {
        switch self {
        case .get:
            return .get
        /// TODO: Add other HTTP methods here
        }
    }

    /// ...
}
```

就像代码中 `TODO` 注释中说明的，当我们要添加其它HTTP verb的时候，直接在这里写多个 `case` 就好了。

第二部分，是根据不同的HTTP verb生成对应的数据。当然，我们的GET请求自然是不用这部分，但如果是POST请求，通常就会带有一个 `[String:Any]` 类型的数据。因此，我们把它定义成一个Optional：

```
func asURLRequest() throws -> URLRequest {
    /// ...

    var params: [String: Any]? {
        switch self {
        case .get:
            return nil
        /// TODO: Add other HTTP methods here
        }
    }

    /// ...
}
```

思路和之前是一样的，只是根据不同的请求类型生成对应的内容。

第二部分，是根据不同的HTTP verb生成对应的URL。之前我们已经定义了 baseURL ， 这里只要追加上不同的URL后缀就可以了：

```
func asURLRequest() throws -> URLRequest {
    /// ...

    var url: URL {
        var relativeUrl: String = "todos"

        switch self {
        case .get(let todoId):
            if todoId != nil {
                relativeUrl = "todos/\(todoId!)"
            }
        /// TODO: Add other HTTP methods here
        }

        let url = URL(string: TodoRouter.baseURL)!
            .appendingPathComponent(relativeUrl)

        return url
    }

    /// ...
}
```

和之前的代码稍有不同的是，这次我们提取到了 case let 的关联值，根据它是否为 nil 生成了不同的 URL 对象。

最后， method 、 params 和 url 就都齐备了，我们直接调用Alamofire中的 JSONEncoding.encode 方法，生成 URLRequest：

```
func asURLRequest() throws -> URLRequest {
    /// ...

    var request = URLRequest(url: url)
    request.httpMethod = method.rawValue
    let encoding = JSONEncoding.default

    return try encoding.encode(request, with: params)
}
```

至此，这个简单的router就完成了。有了它之后，我们可以把所有和HTTP请求有关的准备工作都统一放到这里。接下来，我们就该用它完成请求了。

获取并显示Todo列表

打开*TodoListViewController.swift*，简单起见，我们只是把这部分代码先放在 viewDidLoad 方法里：

```
override func viewDidLoad() {
    super.viewDidLoad()
    Alamofire.request(TodoRouter.get(nil))
        .responseJSON { response in
            /// TODO: Handle request here
        }
}
```

可以看到，我们直接传递了 TodoRouter.get(nil) 表示我们要获取Todo列表。可以想象的是，我们可以用类似的方式发起其它类型的HTTP请求，这要比我们直接把请求信息和 request 写在一起更直观一些。在 responseJSON 的closure里，我们要处理三种不同的情况：

第一种情况，是服务器返回错误：

```
Alamofire.request(TodoRouter.get(nil))
    .responseJSON { response in
        guard response.result.error == nil else {
            print(response.result.error!)
            return
        }

        /// ...
    }
```

第二种情况，是虽然正常收到了服务器的响应，但无法转换成对应的数据类型：

```
Alamofire.request(TodoRouter.get(nil))
    .responseJSON { response in
        /// ...
        guard let todos =
            response.result.value as? [[String: Any]] else {
                print("Cannot read the Todo list from the server.")
                return
            }
        /// ...
    }
```

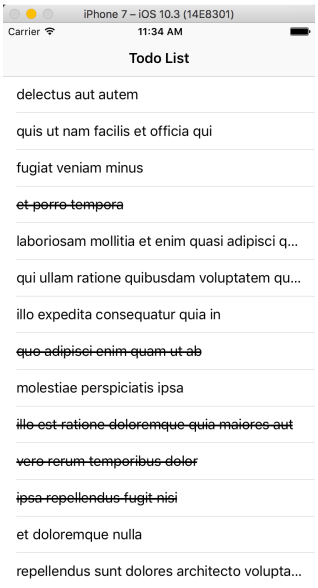
最后一种情况，当然就是正常收到结果了，我们在这里处理返回值：

```
Alamofire.request(TodoRouter.get(nil))
    .responseJSON { response in
        /// ...
        todos.reversed().forEach {
            guard let todo = Todo(json: $0) else {
                print("Cannot create a todo from the JSON.")
                return
            }

            self.todoList.append(todo)
        }

        DispatchQueue.main.async {
            self.tableView.reloadData()
        }
    }
```

这里，由于Alamofire发出的网络请求是异步的，因此，在它的closure里，我们要明确在主线程中更新UI。最后，执行一下，就能看到加载出来的列表了。



What's next?

在这个简单的Demo里，我们介绍了通过Router改进Alamofire请求的方法，但就像你看到的，`viewDidLoad` 方法知道的事情太多了，其实它只应该处理和UI有关的逻辑，而不应该关心如何发起请求，处理哪些请求错误，以及要切换到主线程更新UI等这些技术细节。在下一节，我们就通过RxSwift中的*Transform operators*，改进 `viewDidLoad` 的实现。

⏮ 为什么RxSwift也需要flatMap

(https://www.boxueio.com/series/rxswift-101/ebook/267)

App demo II 使用map/flatMap简化代码 ⏭

(https://www.boxueio.com/series/rxswift-101/ebook/270)

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/)