SwiftGG
走心的 Swift 翻译组

手把手教你封装网络层

2017-04-25 2368

作者：Tomasz Szulc，原文链接，原文日期：2016-07-30
译者：智多芯；校对：Crystal Sun；定稿：CMB

同时负责两个项目是个探索应用架构的好机会，可以在项目中试验一下已有的想法或刚学到的知识。我最近学习了如何封装一个网络层框架，说不定对你有所帮助。

如今的移动应用几乎都是“客户端-服务端（client-server）”架构，在应用里都会有网络层，大小不同而已。我见过很多种实现方式，但都有一些缺陷。当然这并不是说，我最近实现的这个一点缺陷也没有，但至少在目前的两个项目上都运行的很不错。*测试覆盖率也将近百分百。*

本文涉及的网络层仅限发送 JSON 请求给后端，也不会太复杂。这网络层会和亚马逊 AWS 的功能。

保存后端URL

首先，*后端 URL 相关的代码放在哪？* 系统的其他部分代码如何知道在哪里发送请求？我倾向于创建一个 `BackendConfiguration` 类用来保存这些信息。

```
import Foundation

public final class BackendConfiguration {
    let baseURL: NSURL

    public init(baseURL: NSURL) {
        self.baseURL = baseURL
    }

    public static var shared: BackendConfiguration!
}
```

这样易于测试，也易于配置。可以在网络层的任何地方读写静态变量 `shared`，而不必到处传递。

```
let backendURL = NSURL(string: "https://szulctomasz.com")!
BackendConfiguration.shared = BackendConfiguration(baseURL: backendURL)
```

代码中硬编码端点。也
。不过这些都不是想要

哪个端点发送请求，知
消息体和头部。

]

`PUT`，`DELETE` 几种

```
final class SignUpRequest: BackendAPIRequest {
    private let firstName: String
```

```
    private let lastName: String
    private let email: String
    private let password: String

    init(firstName: String, lastName: String, email: String, password: String) {
        self.firstName = firstName
        self.lastName = lastName
        self.email = email
        self.password = password
    }

    var endpoint: String {
        return "/users"
    }

    var method: NetworkService.Method {
        return .POST
    }

    var parameters: [String: AnyObject]? {
        return [
            "first_name": firstName,
            "last_name": lastName,
            "email": email,
```

t 定义一个

有必需的参数都传给了

单，因为实际上只要把

```
init(id: String, ...) {
  self.id = id
```

```
}

var endpoint: String {
  return "/users/\(id)"
}
```

请求方法不变、参数易于构建和维护，头部也一样，这样就很容易对它们进行测试了。

执行请求

是否需要使用第三方库和后端通信?

有很多人都在用 AFNetworking(Objective-C) 和 Alamofire(Swift)。我也用过很多次，但有时候我就不使用它们了。毕竟有 `NSURLSession` 可以很好地实现需求，就没必要使用第三方库了。在我看来，这些依赖会导致应用架构越来越复杂。

目前的解决方案由两个类组成： `NetworkService` 和 `BackendService` 。

`NetworkService` ：可以执行HTTP请求，它内部集成了 `NSURLSession` 。每个网络服务一次只能执行一个请求，也能够取消请求（很大的优势），而且请求成功和失败时都会有回调。

请求（就是上面提到的
当前使用的版本中，尝
各式的数据。

```
onseCode: Int) -> Void)

olicy: .ReloadIgnoringL
: 10.0)

taWithJSONObject(params

        }

        let session = NSURLSession.sharedSession()
```

```
        task = session.dataTaskWithRequest(mutableRequest, completionHandler: { data, re
            // 判断调用是否成功
            // 回调处理
        })

        task?.resume()
    }

    func cancel() {
        task?.cancel()
    }
}
```

```
class BackendService {
    private let conf: BackendConfiguration
    private let service: NetworkService!

    init(_ conf: BackendConfiguration) {
        self.conf = conf
        self.service = NetworkService()
    }

    func request(request: BackendAPIRequest,
                 success: (AnyObject? -> Void)? = nil,
```

```
st.endpoint)
```

```
: request.parameters, he
```

```
ta(data, options: [])
```

`BackendService` 可以在 `headers` 中设置认证令牌（authentication token）。其中 `BackendAuth` 只是个简单的对象，用来将令牌保存到 `UserDefaults` 中。在必要的时候，也可以将令牌保存在 `Keychain` 中。

`BackendService` 将 `BackendAPIRequest` 作为 `request(_:success:failure:)` 方法的参数从 `request` 对象中提取出必要的信息，这保持了很好的封装性。

```
public final class BackendAuth {

    private let key = "BackendAuthToken"
    private let defaults: NSUserDefaults

    public static var shared: BackendAuth!

    public init(defaults: NSUserDefaults) {
        self.defaults = defaults
    }

    public func setToken(token: String) {
        defaults.setValue(token, forKey: key)
    }

    public var token: String? {
        return defaults.valueForKey(key) as? String
    }

    public func deleteToken() {
        defaults.removeObjectForKey(key)
```

可以很容易地测试和维

一次执行多次请求呢?

。在继承

e 。

```
}
```

```
        get { return _executing }
        set { update({ self._executing = newValue }, key: "isExecuting") }
    }
```

```
    private var _finished: Bool
    public override var finished: Bool {
        get { return _finished }
        set { update({ self._finished = newValue }, key: "isFinished") }
    }

    private var _cancelled: Bool
    public override var cancelled: Bool {
        get { return _cancelled }
        set { update({ self._cancelled = newValue }, key: "isCancelled") }
    }

    private func update(change: Void -> Void, key: String) {
        willChangeValueForKey(key)
        change()
        didChangeValueForKey(key)
    }

    override init() {
        _ready = true
        _executing = false
        _finished = false
        _cancelled = false
```

```
        self.cancelled = true
    }
}
```

接着，因为想通过 `BackendService` 执行网络调用，所以继承了 `NetworkOperation`，并创建了 `ServiceOperation`。

```
public class ServiceOperation: NetworkOperation {
    let service: BackendService

    public override init() {
        self.service = BackendService(BackendConfiguration.shared)
        super.init()
    }

    public override func cancel() {
        service.cancel()
        super.cancel()
    }
}
```

这个类已经在它内部创建了 `BackendService`，所以就没必要每次都在子类中创建一次。

下面是 *`SignInOperation`* 的代码：

```
public class SignInOperation: ServiceOperation {
    private let request: SignInRequest

    public var success: (SignInItem -> Void)?
```

```
                                                )

                                                : handleFailure)

                                                )
    }
}
```

在 `SignInOperation` 初始化时创建了登录请求，随后在 `start` 方法中执行它。`handleSuccess` 和 `handleFailure` 两个方法作为回调传递给了服务的 `request(_:success:failure:)` 方法。我觉得这让代码看起来更干净，可读性更强。

将 `Operations` 传给 `NetworkQueue` 对象。`NetworkQueue`对象是一个单例，可以将每个 `Operation` 入队。暂时尽量让代码保持简洁吧：

```
public class NetworkQueue {
    public static var shared: NetworkQueue!

    let queue = NSOperationQueue()

    public init() {}

    public func addOperation(op: NSOperation) {
        queue.addOperation(op)
    }
}
```

那么，在同一个地方执行 `Operation` 都有什么好处呢?

- 方便取消所有的网络请求。
- 为了给用户更好的体验，当网络不好的时候，取消所有正在下载图像或请求非必需数据的操作。
- 可以构建一个优先级队列用于提前执行一些请求，以便更快地得到结果。

和Core Data共处

这是我不得不推迟发表这篇文章的原因。在之前的几个网络层版本中，`Operation` 都会返回 Core Data 对象。接收到的响应会被解析并转换成 Core Data 对象。可是这种方案远远不够完美。

- `SignInOperation`需要知道 Core Data 是个什么东西。由于我把数据模型独立出来了，因此网络库也需要知晓数据模型。
- 每个 `SignInOperation` 都需要增加一个额外的 `NSManagedObjectContext` 参数，用来决定在什么上下文执行操作。
- 每次接收到响应并准备调用 `success` 的代码之前，都会在 Core Data 上下文中查找对象，然后访问磁盘并将其提取出来。我觉得这是个不足的地方，并不是每次都想创建 Core Data 对象。

所以我想到应该把 Core Data 完完全全地从网络层中分离出去。于是创建了一个中间层，其实也就是一些在解析响应时创建的对象。

- 这样一来，解析和创建对象就很快了，而且不用访问磁盘。
- 不再需要将 `NSManagedObjectContext` 传给 `SignInOperation` 了。

- 可以在 `success` 代码块中使用解析过的数据来更新 Core Data 对象，然后引用之前可能保存在某处的 Core Data 对象——这是我在将 `SignInOperation` 入队时会碰到的情况。

映射响应

响应映射器的思想主要是将解析逻辑和 JSON 映射逻辑分成多个有用的单项。

可以两种不同的解析器区分开来，第一种只解析一个特定类型的对象，第二种用来解析对象数组。

首先定义一个通用协议：

```
public protocol ParsedItem {}
```

下面是映射器的映射结果：

```
public struct SignInItem: ParsedItem {
    public let token: String
    public let uniqueId: String
}

public struct UserItem: ParsedItem {
    public let uniqueId: String
    public let firstName: String
    public let lastName: String
    public let email: String
    public let phoneNumber: String?
}
```

再定义一个错误类型，以便在解析发生错误时抛出。

```
internal enum ResponseMapperError: ErrorType {
    case Invalid
    case MissingAttribute
}
```

- `Invalid`：当解析到的 JSON 为 nil 且不该为 nil，或者是一个对象数组而不是期望的只含单个对象的 JSON 时抛出。
- `MissingAttribute`：名字本身就能说明它的作用了。当 key 在 JSON 中不存在，或者解析后值为 nil 且不该为 nil 时抛出。

`ResponseMapper` 的实现如下：

```
class ResponseMapper<A: ParsedItem> {
    static func process(obj: AnyObject?, parse: (json: [String: AnyObject]) -> A?) throw
        guard let json = obj as? [String: AnyObject] else { throw ResponseMapperError.Ir
        if let item = parse(json: json) {
            return item
```

```
        } else {
            L.log("Mapper failure (\(self)). Missing attribute.")
            throw ResponseMapperError.MissingAttribute
        }
    }
}
```

其中 `process` 静态方法的参数分别是 `obj`（也就是从后端返回的JSON）和 `parse` 方法（该方法会解析 `obj` 并返回一个 `ParsedItem` 类型的 `A` 对象）。

既然有了这个通用的映射器，接着就可以创建具体的映射器了。先来看看用于解析 *`SignInOperation`* 响应的映射器：

```
protocol ResponseMapperProtocol {
    associatedtype Item
    static func process(obj: AnyObject?) throws -> Item
}

final class SignInResponseMapper: ResponseMapper<SignInItem>, ResponseMapperProtocol {
    static func process(obj: AnyObject?) throws -> SignInItem {
        return try process(obj, parse: { json in
            let token = json["token"] as? String
            let uniqueId = json["unique_id"] as? String
            if let token = token, let uniqueId = uniqueId {
                return SignInItem(token: token, uniqueId: uniqueId)
            }
            return nil
        })
    }
}
```

`ResponseMapperProtocol` 协议为具体的映射器定义了用于解析响应的方法。

接着，这样的映射器就可以用在 `operation` 的 `success` 代码块中了。而且可以直接操作指定类型的具体对象，而不是字典。这样一切都可以很容易地进行测试了。

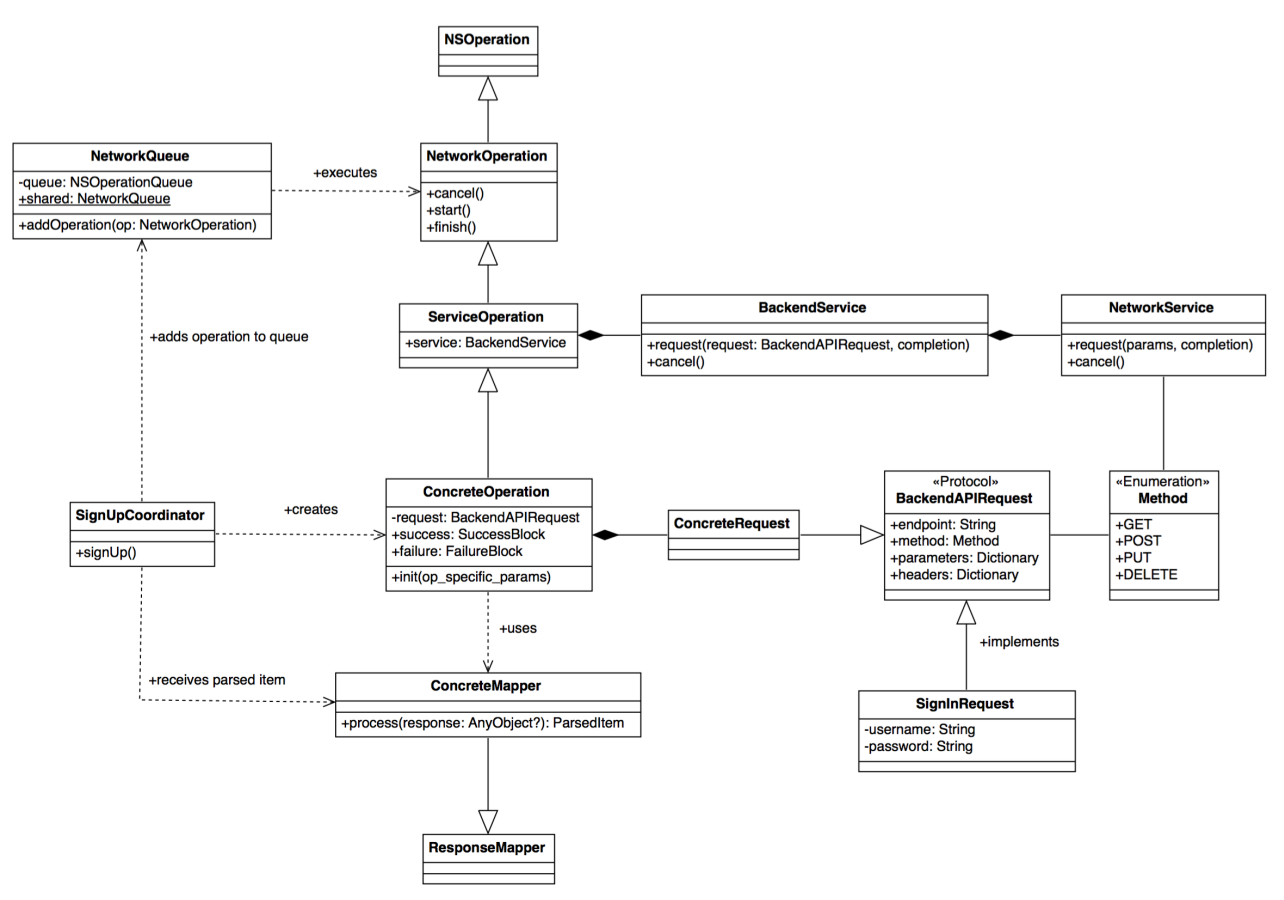
下面是解析数组的映射器：

```
final class ArrayResponseMapper<A: ParsedItem> {
    static func process(obj: AnyObject?, mapper: (AnyObject? throws -> A)) throws -> [A]
        guard let json = obj as? [[String: AnyObject]] else { throw ResponseMapperError

        var items = [A]()
        for jsonNode in json {
            let item = try mapper(jsonNode)
            items.append(item)
        }
        return items
    }
}
```

其中 `process` 静态方法的参数分别是 `obj` 和 `mapper` 方法，成功解析之后会返回一个数组。如果有某一项解析失败，可以抛出一个错误，或者更糟地直接返回一个空数组作为该映射器的结果，你来决定。另外，这个映射器希望传给它的 `obj` 参数（从后端返回的响应数据）是个 JSON 数组。

下面是整个网络层的 UML 图：



diagram

示例项目

可以在GitHub上找的示例项目。该项目中用到了伪造的后端 URL，所以任何请求都不会有响应。提供这个示例只是想让你对这个网络层的结构有个大致的认识。

总结

我发现用这种方法封装的网络层不仅简单而且很有用：

- 最大的优点在于，可以很容易地新增类似上文提到的 `Operation`，而不用关心 Core Data 的存在。
- 可以轻易地让代码覆盖率接近100%，而无需考虑如何覆盖某个难搞的情形，因为根本就不存在这么难搞的情形！
- 可以在其他类似的复杂应用中很容易地复用它的核心代码。

> 本文由 SwiftGG 翻译组翻译，已经获得作者翻译授权，最新文章请访问 http://swift.gg。

Tomasz Szulc Swift 进阶

上一篇
如何将一个可选字符串转换为 NSString
下一篇
Swift 简洁之道(上)

Like Issue Page

Error: Comments Not Initialized

Write Preview Login with GitHub

Leave a comment

Styling with Markdown is supported Comment

Powered by Gitment

分类

APPVENTURE 13

Andyy Hope 4

AppCoda 39

Big O Note-Taking 2

Coding Explorer Blog 2

Crunchy Development [24]

Erica Sadun [67]

IOSCREATOR [29]

Jacob Bandes-Storch [2]

Jameson Quave [1]

JamesonQuave.com [18]

Jesse Squires [1]

KHANLOU [16]

Mike Ash [5]

Natasha The Robot [48]

Ole Begemann [30]

Open Source Swift [11]

Raj Kandathi [6]

Reinder de Vries [1]

Russ Bishop [7]

Soroush Khanlou [2]

Swift and Painless [11]

标签

Swift 进阶 166

Swift 入门 128

iOS 开发 74

Swift 67

Swift 跨平台 11

Swift 开源信息 11

Swift 3 7

WatchOS 2 7

iOS 入门 6

Apple TV 开发 5

iOS 9 4

Xcode 4

IOSCREATOR 4

Jesse Squires 3

Swift 2 3

社区问答 3

Swift 进化 2

SwiftyDB 1

Objective-C 1

推送通知 1

友情链接

C4iOS 教程

SwiftGG直播

T 沙龙

Code Build Me

//TODO:

chiba

Perfect Freeze

小锅的 swift 之路

Prayer 的博客

画渣程序猿 mmoaay

小铁匠的 swift 之路

ppppppmst 的简书博客

CMB 的博客

BridgeQ

walkingway 的博客

靛青K

JackAlan

SwiftConChina

Swift 中国

泊学

BearyChat

PHP-Z 论坛

官方文档

又拍云赞助图床

RSS 订阅

微信公众号

Powered by hexo and Theme by Jacman © 2017 SwiftGG | 浙ICP备14022870号-3