OneV's Den

上善若水，人淡如菊

2016-12-01 • 能工巧匠集

面向协议编程与 Cocoa 的邂逅 (下)

本文是笔者在 MDCC 16 (移动开发者大会) 上 iOS 专场中的主题演讲的文字整理。您可以在这里找到演讲使用的 Keynote，部分示例代码可以在 MDCC 2016 的官方 repo 中找到。

在上半部分主要介绍了一些理论方面的内容，包括面向对象编程存在的问题，面向协议的基本概念和决策模型等。本文 (下) 主要展示了一些笔者日常使用面向协议思想和 Cocoa 开发结合的示例代码，并对其进行了一些解说。

转・热恋 - 在日常开发中使用协议

WWDC 2015 在 POP 方面有一个非常优秀的主题演讲：#408 Protocol-Oriented Programming in Swift。Apple 的工程师通过举了画图表和排序两个例子，来阐释 POP 的思想。我们可以使用 POP 来解耦，通过组合的方式让代码有更好的重用性。不过在 #408 中，涉及的内容偏向理论，而我们每天的 app 开发更多的面临的还是和 Cocoa 框架打交道。在看过 #408 以后，我们就一直在思考，如何把 POP 的思想运用到日常的开发中?

我们在这个部分会举一个实际的例子，来看看 POP 是如何帮助我们写出更好的代码的。

基于 Protocol 的网络请求

网络请求层是实践 POP 的一个理想场所。我们在接下的例子中将从零开始，用最简单的面向协议的方式先构建一个不那么完美的网络请求和模型层，它可能包含一些不合理的设计和耦合，但是却是初步最容易得到的结果。然后我们将逐步捋清各部分的所属，并用分离职责的方式来进行重构。最后我们会为这个网络请求层进行测试。通过这个例子，我希望能够设计出包括类型安全，解耦合，易于测试和良好的扩展性等诸多优秀特性在内的 POP 代码。

> Talk is cheap, show me the code.

初步实现

首先，我们想要做的事情是从一个 API 请求一个 JSON，然后将它转换为 Swift 中可用的实例。作为例子的 API 非常简单，你可以直接访问 https://api.onevcat.com/users/onevcat 来查看返回:

```
{"name":"onevcat","message":"Welcome to MDCC 16!"}
```

我们可以新建一个项目，并添加 `User.swift` 来作为模型:

```
// User.swift
import Foundation

struct User {
    let name: String
    let message: String

    init?(data: Data) {
        guard let obj = try? JSONSerialization.jsonObject(with: data, options: []) as? [String: Any] else
            return nil
        }
        guard let name = obj?["name"] as? String else {
            return nil
        }
        guard let message = obj?["message"] as? String else {
            return nil
```

```
        self.name = name
        self.message = message
    }
}
```

`User.init(data:)` 将输入的数据 (从网络请求 API 获取) 解析为 JSON 对象，然后从中取出 `name` 和 `message`，并构建代表 API 返回的 `User` 实例，非常简单。

现在让我们来看看有趣的部分，也就是如何使用 POP 的方式从 URL 请求数据，并生成对应的 `User`。首先，我们可以创建一个 protocol 来代表请求。对于一个请求，我们需要知道它的请求路径，HTTP 方法，所需要的参数等信息。一开始这个协议可能是这样的：

```
enum HTTPMethod: String {
    case GET
    case POST
}

protocol Request {
    var host: String { get }
    var path: String { get }

    var method: HTTPMethod { get }
    var parameter: [String: Any] { get }
}
```

将 `host` 和 `path` 拼接起来可以得到我们需要请求的 API 地址。为了简化，`HTTPMethod` 现在只包含了 `GET` 和 `POST` 两种请求方式，而在我们的例子中，我们只会使用到 `GET` 请求。

现在，可以新建一个 `UserRequest` 来实现 `Request` 协议：

```
struct UserRequest: Request {
    let name: String

    let host = "https://api.onevcat.com"
    var path: String {
        return "/users/\(name)"
    }
    let method: HTTPMethod = .GET
    let parameter: [String: Any] = [:]
}
```

`UserRequest` 中有一个未定义初始值的 `name` 属性，其他的属性都是为了满足协议所定义的。因为请求的参数用户名 `name` 会通过 URL 进行传递，所以 `parameter` 是一个空字典就足够了。有了协议定义和一个满足定义的具体请求，现在我们需要发送请求。为了任意请求都可以通过同样的方法发送，我们将发送的方法定义在 `Request` 协议扩展上：

```
extension Request {
    func send(handler: @escaping (User?) -> Void) {
        // ... send 的实现
    }
}
```

在 `send(handler:)` 的参数中，我们定义了可逃逸的 `(User?) -> Void`，在请求完成后，我们调用这个 `handler` 方法来通知调用者请求是否完成，如果一切正常，则将一个 `User` 实例传回，否则传回 `nil`。

我们想要这个 `send` 方法对于所有的 `Request` 都通用，所以显然回调的参数类型不能是 `User`。通过在 `Request` 协议中添加一个关联类型，我们可以将回调参数进行抽象。在 `Request` 最后添加：

```
protocol Request {
    ...
    associatedtype Response
}
```

```
struct UserRequest: Request {
    ...
    typealias Response = User
}
```

现在，我们来重新实现 `send` 方法，现在，我们可以用 `Response` 代替具体的 `User`，让 `send` 一般化。我们这里使用 `URLSession` 来发送请求：

```
extension Request {
    func send(handler: @escaping (Response?) -> Void) {
        let url = URL(string: host.appending(path))!
        var request = URLRequest(url: url)
        request.httpMethod = method.rawValue

        // 在示例中我们不需要 `httpBody`，实践中可能需要将 parameter 转为 data
        // request.httpBody = ...

        let task = URLSession.shared.dataTask(with: request) {
            data, res, error in
            // 处理结果
            print(data)
        }
        task.resume()
    }
}
```

通过拼接 `host` 和 `path`，可以得到 API 的 entry point。根据这个 URL 创建请求，进行配置，生成 data task 并将请求发送。剩下的工作就是将回调中的 `data` 转换为合适的对象类型，并调用 `handler` 通知外部调用者了。对于 `User` 我们知道可以使用 `User.init(data:)`，但是对于一般的 `Response`，我们还不知道要如何将数据转为模型。我们可以在 `Request` 里再定义一个 `parse(data:)` 方法，来要求满足该协议的具体类型提供合适的实现。这样一来，提供转换方法的任务就被“下放”到了 `UserRequest`：

```
protocol Request {
    ...
    associatedtype Response
    func parse(data: Data) -> Response?
}

struct UserRequest: Request {
    ...
    typealias Response = User
    func parse(data: Data) -> User? {
        return User(data: data)
    }
}
```

有了将 `data` 转换为 `Response` 的方法后，我们就可以对请求的结果进行处理了：

```
extension Request {
    func send(handler: @escaping (Response?) -> Void) {
        let url = URL(string: host.appending(path))!
        var request = URLRequest(url: url)
        request.httpMethod = method.rawValue

        // 在示例中我们不需要 `httpBody`，实践中可能需要将 parameter 转为 data
        // request.httpBody = ...

        let task = URLSession.shared.dataTask(with: request) {
            data, _, error in
            if let data = data, let res = parse(data: data) {
                DispatchQueue.main.async { handler(res) }
            } else {
                DispatchQueue.main.async { handler(nil) }
            }
        }
        task.resume()
    }
}
```

现在，我们来试试看请求一下这个 API：

```
let request = UserRequest(name: "onevcat")
request.send { user in
    if let user = user {
        print("\(user.message) from \(user.name)")
    }
}

// Welcome to MDCC 16! from onevcat
```

重构，关注点分离

虽然能够实现需求，但是上面的实现可以说非常糟糕。让我们看看现在 `Request` 的定义和扩展：

```
protocol Request {
    var host: String { get }
    var path: String { get }

    var method: HTTPMethod { get }
    var parameter: [String: Any] { get }

    associatedtype Response
    func parse(data: Data) -> Response?
}

extension Request {
    func send(handler: @escaping (Response?) -> Void) {
        ...
    }
}
```

这里最大的问题在于，`Request` 管理了太多的东西。一个 `Request` 应该做的事情应该仅仅是定义请求入口和期望的响应类型，而现在 `Request` 不光定义了 `host` 的值，还对如何解析数据了如指掌。最后 `send` 方法被绑死在了 `URLSession` 的实现上，而且是作为 `Request` 的一部分存在。这是很不合理的，因为这意味着我们无法在不更改请求的情况下更新发送请求的方式，它们被耦合在了一起。这样的结构让测试变得异常困难，我们可能需要通过 stub 和 mock 的方式对请求拦截，然后返回构造的数据，这会用到 `NSURLProtocol` 的内容，或者是引入一些第三方的测试框架，大大增加了项目的复杂度。在 Objective-C 时期这可能是一个可选项，但是在 Swift 的新时代，我们有好得多的方法来处理这件事情。

让我们开始着手重构刚才的代码，并为它们加上测试吧。首先我们将 `send(handler:)` 从 `Request` 分离出来。我们需要一个单独的类型来负责发送请求。这里基于 POP 的开发方式，我们从定义一个可以发送请求的协议开始：

```
protocol Client {
    func send(_ r: Request, handler: @escaping (Request.Response?) -> Void)
}

// 编译错误
```

从上面的声明从语义上来说是挺明确的，但是因为 `Request` 是含有关联类型的协议，所以它并不能作为独立的类型来使用，我们只能够将它作为类型约束，来限制输入参数 `request`。正确的声明方式应当是：

```
protocol Client {
    func send<T: Request>(_ r: T, handler: @escaping (T.Response?) -> Void)

    var host: String { get }
}
```

除了使用 `<T: Request>` 这个泛型方式以外，我们还将 `host` 从 `Request` 移动到了 `Client` 里，这是更适合它的地方。现在，我们可以把含有 `send` 的 `Request` 协议扩展删除，重新创建一个类型来满足 `Client` 了。和之前一样，它将使用 `URLSession` 来发送请求：

```
struct URLSessionClient: Client {
    let host = "https://api.onevcat.com"
```

```
        let url = URL(string: host.appending(r.path))!
        var request = URLRequest(url: url)
        request.httpMethod = r.method.rawValue

        let task = URLSession.shared.dataTask(with: request) {
            data, _, error in
            if let data = data, let res = r.parse(data: data) {
                DispatchQueue.main.async { handler(res) }
            } else {
                DispatchQueue.main.async { handler(nil) }
            }
        }
        task.resume()
    }
}
```

现在发送请求的部分和请求本身分离开了，而且我们使用协议的方式定义了 `Client`。除了 `URLSessionClient` 以外，我们还可以使用任意的类型来满足这个协议，并发送请求。这样网络层的具体实现和请求本身就不再相关了，我们之后在测试的时候会进一步看到这么做所带来的好处。

现在这个的实现里还有一个问题，那就是 `Request` 的 `parse` 方法。请求不应该也不需要知道如何解析得到的数据，这项工作应该交给 `Response` 来做。而现在我们没有对 `Response` 进行任何限定。接下来我们将新增一个协议，满足这个协议的类型将知道如何将一个 `data` 转换为实际的类型：

```
protocol Decodable {
    static func parse(data: Data) -> Self?
}
```

`Decodable` 定义了一个静态的 `parse` 方法，现在我们需要在 `Request` 的 `Response` 关联类型中为它加上这个限制，这样我们可以保证所有的 `Response` 都可以对数据进行解析，原来 `Request` 中的 `parse` 声明也就可以移除了：

```
// 最终的 Request 协议
protocol Request {
    var path: String { get }
    var method: HTTPMethod { get }
    var parameter: [String: Any] { get }

    // associatedtype Response
    // func parse(data: Data) -> Response?
    associatedtype Response: Decodable
}
```

最后要做的就是让 `User` 满足 `Decodable`，并且修改上面 `URLSessionClient` 的解析部分的代码，让它使用 `Response` 中的 `parse` 方法：

```
extension User: Decodable {
    static func parse(data: Data) -> User? {
        return User(data: data)
    }
}

struct URLSessionClient: Client {
    func send<T: Request>(_ r: T, handler: @escaping (T.Response?) -> Void) {
        ...
     // if let data = data, let res = parse(data: data) {
        if let data = data, let res = T.Response.parse(data: data) {
            ...
        }
    }
}
```

最后，将 `UserRequest` 中不再需要的 `host` 和 `parse` 等清理一下，一个类型安全，解耦合的面向协议的网络层就呈现在我们眼前了。想要调用 `UserRequest` 时，我们可以这样写：

```
    if let user = user {
        print("\(user.message) from \(user.name)")
    }
}
```

当然，你也可以为 `URLSessionClient` 添加一个单例来减少请求时的创建开销，或者为请求添加 Promise 的调用方式等等。在 POP 的组织下，这些改动都很自然，也不会牵扯到请求的其他部分。你可以用和 `UserRequest` 类型相似的方式，为网络层添加其他的 API 请求，只需要定义请求所必要的内容，而不用担心会触及网络方面的具体实现。

网络层测试

将 `Client` 声明为协议给我们带来了额外的好处，那就是我们不在局限于使用某种特定的技术 (比如这里的 `URLSession`) 来实现网络请求。利用 POP，你只是定义了一个发送请求的协议，你可以很容易地使用像是 AFNetworking 或者 Alamofire 这样的成熟的第三方框架来构建具体的数据并处理请求的底层实现。我们甚至可以提供一组“虚假”的对请求的响应，用来进行测试。这和传统的 stub & mock 的方式在概念上是接近的，但是实现起来要简单得多，也明确得多。我们现在来看一看具体应该怎么做。

我们先准备一个文本文件，将它添加到项目的测试 target 中，作为网络请求返回的内容：

```
// 文件名：users:onevcat
{"name":"Wei Wang", "message": "hello"}
```

接下来，可以创建一个新的类型，让它满足 `Client` 协议。但是与 `URLSessionClient` 不同，这个新类型的 `send` 方法并不会实际去创建请求，并发送给服务器。我们在测试时需要验证的是一个请求发出后如果服务器按照文档正确响应，那么我们应该也可以得到正确的模型实例。所以这个新的 `Client` 需要做的事情就是从本地文件中加载定义好的结果，然后验证模型实例是否正确：

```
struct LocalFileClient: Client {
    func send<T : Request>(_ r: T, handler: @escaping (T.Response?) -> Void) {
        switch r.path {
        case "/users/onevcat":
            guard let fileURL = Bundle(for: ProtocolNetworkTests.self).url(forResource: "users:onevcat", \
                fatalError()
            }
            guard let data = try? Data(contentsOf: fileURL) else {
                fatalError()
            }
            handler(T.Response.parse(data: data))
        default:
            fatalError("Unknown path")
        }
    }

    // 为了满足 `Client` 的要求，实际我们不会发送请求
    let host = ""
}
```

`LocalFileClient` 做的事情很简单，它先检查输入请求的 `path` 属性，如果是 `/users/onevcat` (也就是我们需要测试的请求)，那么就从测试的 bundle 中读取预先定义的文件，将其作为返回结果进行 `parse`，然后调用 `handler`。如果我们需要增加其他请求的测试，可以添加新的 `case` 项。另外，加载本地文件资源的部分应该使用更通用的写法，不过因为我们这里只是示例，就不过多纠结了。

在 `LocalFileClient` 的帮助下，现在可以很容易地对 `UserRequest` 进行测试了：

```
func testUserRequest() {
    let client = LocalFileClient()
    client.send(UserRequest(name: "onevcat")) {
        user in
        XCTAssertNotNil(user)
        XCTAssertEqual(user!.name, "Wei Wang")
    }
}
```

以进行请求测试了。保持简单的代码和逻辑，对于项目维护和发展是至关重要的。

可扩展性

因为高度解耦，这种基于 POP 的实现为代码的扩展提供了相对宽松的可能性。我们刚才已经说过，你不必自行去实现一个完整的 `Client`，而可以依赖于现有的网络请求框架，实现请求发送的方法即可。也就是说，你也可以很容易地将某个正在使用的请求方式替换为另外的方式，而不会影响到请求的定义和使用。类似地，在 `Response` 的处理上，现在我们定义了 `Decodable`，用自己手写的方式在解析模型。我们完全也可以使用任意的第三方 JSON 解析库，来帮助我们迅速构建模型类型，这仅仅只需要实现一个将 `Data` 转换为对应模型类型的方法即可。

如果你对 POP 方式的网络请求和模型解析感兴趣的话，不妨可以看看 APIKit 这个框架，我们在示例中所展示的方法，正是这个框架的核心思想。

合・陪伴 – 使用协议帮助改善代码设计

通过面向协议的编程，我们可以从传统的继承上解放出来，用一种更灵活的方式，搭积木一样对程序进行组装。每个协议专注于自己的功能，特别得益于协议扩展，我们可以减少类和继承带来的共享状态的风险，让代码更加清晰。

高度的协议化有助于解耦、测试以及扩展，而结合泛型来使用协议，更可以让我们免于动态调用和类型转换的苦恼，保证了代码的安全性。

提问环节

主题演讲后有几位朋友提了一些很有意义的问题，在这里我也稍作整理。有可能问题和回答与当时的情形会有小的出入，仅供参考。

我刚才在看 demo 的时候发现，你都是直接先写 `protocol`，而不是 `struct` 或者 `class`。是不是我们在实践 POP 的时候都应该直接先定义协议？

> 我直接写 `protocol` 是因为我已经对我要做什么有充分的了解，并且希望演讲不要超时。但是实际开发的时候你可能会无法一开始就写出合适的协议定义。建议可以像我在 demo 中做的那样，先“粗略”地进行定义，然后通过不断重构来得到一个最终的版本。当然，你也可以先用纸笔勾勒一个轮廓，然后再去定义和实现协议。当然了，也没人规定一定需要先定义协议，你完全也可以从普通类型开始写起，然后等发现共通点或者遇到我们之前提到的困境时，再回头看看是不是面向协议更加合适，这需要一定的 POP 经验。

既然 POP 有这么多好处，那我们是不是不再需要面向对象，可以全面转向面向协议了？

> 答案可能让你失望。在我们的日常项目中，每天打交道的 Cocoa 其实还是一个带有浓厚 OOP 色彩的框架。也就是说，可能一段时期内我们不可能抛弃 OOP。不过 POP 其实可以和 OOP “和谐共处”，我们也已经看到了不少使用 POP 改善代码设计的例子。另外需要补充的是，POP 其实也并不是银弹，它有不好的一面。最大的问题是协议会增加代码的抽象层级 (这点上和类继承是一样的)，特别是当你的协议又继承了其他协议的时候，这个问题尤为严重。在经过若干层的继承后，满足末端的协议会变得困难，你也难以确定某个方法究竟满足的是哪个协议的要求。这会让代码迅速变得复杂。如果一个协议并没有能描述很多共通点，或者说能让人很快理解的话，可能使用基本的类型还会更简单一些。

谢谢你的演讲，想问一下你们在项目中使用 POP 的情况

> 我们在项目里用了很多 POP 的概念。上面 demo 里的网络请求的例子就是从实际项目中抽出来的，我们觉得这样的请求写起来非常轻松，因为代码很简单，新人进来交接也十分惬意。除了模型层之外，我们在 view 和 view controller 层也用了一些 POP 的代码，比如从 nib 创建 view 的 `NibCreatable`，支持分页请求 tableview controller 的 `NextPageLoadable`，空列表时显示页面的 `EmptyPage` 等等。因为时间有限，不可能展开一一说明，所以这里我只挑选了一个具有代表性，又不是很复杂的网络的例子。其实每个协议都让我们的代码，特别是 View Controller 变短，而且使测试变为可能。可以说，我们的项目从 POP 受益良多，而且我们应该会继续使用下去。

推荐资料

- Protocol-Oriented Programming in Swift – WWDC 15 #408
- Protocols with Associated Types – @alexisgallagher
- Protocol Oriented Programming in the Real World – @_matthewpalmer
- Practical Protocol-Oriented-Programming – @natashatherobot

最近的文章

Swift 并行编程现状和展望 – async/await 和参与者模式

这篇文章不是针对当前版本 Swift 3 的，而是对预计于 2018 年发布的 Swift 5 的一些特性的猜想。如果两年后我还记得这篇文章，可能会回来更新一波。在此之前，请当作一篇对现代语言并行编程特性的不太严谨科普文来看待。CPU 速度已经很多年没有大的突破了，硬件行业更多地将重点放在多核心技术上，而与之对应，软件中并行编程的概念也越来越重要。如何利用多核心 CPU，以及拥有密集计算单元的 GPU，来进行快速的处理和计算，是很多开发者十分感兴趣的事情。在今年年初 Swift 4 的……

2016-12-20 • 能工巧匠集 继续阅读

更早的文章

面向协议编程与 Cocoa 的邂逅 (上)

本文是笔者在 MDCC 16 (移动开发者大会) 上 iOS 专场中的主题演讲的文字整理。您可以在这里找到演讲使用的 Keynote，部分示例代码可以在 MDCC 2016 的官方 repo 中找到。因为全部内容比较长，所以分成了上下两个部分，本文 (上) 主要介绍了一些理论方面的内容，包括面向对象编程存在的问题，面向协议的基本概念和决策模型等，下半部分主要展示了一些笔者日常使用面向协议思想和 Cocoa 开发结合的示例代码，并对其进行了一些解说。引子面向协议编程 (Protocol Or……

2016-11-29 • 能工巧匠集 继续阅读

用protocol做函数参数类型debug的时候有个尴尬的地方：知道传入的这个对象或者struct实例可以调用什么方法和property，但是不知道它是什么类型的class/struct，如果那个class/struct的方法有问题要debug，找出这个class/struct的类型必须要找到那个对象初始化的地方或者lldb打印出来

回复 · 分享 ›

zhang zhijie · 10个月前

喵神，请教一下，这个框架的请求都是从模型出发的，那如果有些接口不需要模型，只返回一个Int或是Bool值，或是需要在数据返回以后做一些操作，那应该怎样?

回复 · 分享 ›

onevcat 管理员 ➚ zhang zhijie · 10个月前

这只是为了示例做的很简化的 prototype。
Int 或者 Bool 也可以是 Decodable 啊..额外操作的话加 hook 就可以，并没有什么很难的东西

回复 · 分享 ›

zhang zhijie ➚ onevcat · 10个月前

多谢喵神~~~那如果返回的是数组呢，应该没有必要为数组再封装一个模型吧。。

回复 · 分享 ›

onevcat 管理员 ➚ zhang zhijie · 10个月前

需要的，因为要求所有的模型类型都是 Decodable。这里要注意的是 Array 里的对象也需要是 Decodable，否则将无法解析。
但是麻烦的地方在于现在 Swift 还不支持一个类型去带有 where 条件地满足某个 protocol。

也就是说，下面的代码现在是不能编译的：

extension Array: Decodable where Element: Decodable { ... }

理论上上面这种会是最好的做法，可能我们可以在今后的 Swift 版本中见到它。
现在暂时的话，可能会需要一个 wrapper 之类的绕过这个问题，把 Array 封装一下然后使用。大概看起来会是这样的：

```
struct DecodableArray<t: decodable="">: Decodable {
    let value: [T]
    static func parse(data: Data) -> DecodableArray<t> {
        var value = [T]()
        for subData in data {
            let result = T.parse(data: data)
            value.append(result)
        }
        return DecodableArray(value: value)
    }
}
```

查看更多

回复 · 分享 ›

zhang zhijie ➚ onevcat · 9个月前

果然是这样。。目前我们项目里就是同时有BaseModel和BasePageModel，后者是前者的Array封装。。AnyWay，还是多谢喵神解答~~~

回复 · 分享 ›

Junwei Zhuge · 10个月前

其实我是在赞叹喵神的主页背景图片的，好久不来，一打开亮瞎我双眼。

回复 · 分享 ›

lsan · 10个月前

此处应该是T.parse吧
回复 · 分享 ›

onevcat 管理员 ➚ lsan · 10个月前
这里应该是 r.parse。我改一下。谢谢指出。
回复 · 分享 ›

tdt · 10个月前
正在苦恼写的糟糕swift代码如何优化,看了这篇文章后就有了方向了.感谢喵神.
回复 · 分享 ›

lvlingsheng · 10个月前
求问POP如何运用在VC上呢？看起来POP更适合处理数据一些，不知道对于VC之类的UI该怎么用
回复 · 分享 ›

onevcat 管理员 ➚ lvlingsheng · 10个月前
也可以使用，可以考虑将 VC 的职责分解一些出来。比如这里 http://krakendev.io/blog/su... 有个最初级的例子。
回复 · 分享 ›

lvlingsheng ➚ onevcat · 10个月前
多谢瞄神，试了一下感觉没有感受到相对之前的继承baseVC的好处是啥。。。求指教
回复 · 分享 ›

onevcat 管理员 ➚ lvlingsheng · 10个月前
比如那篇文章里的 ErrorPopoverRenderer，最显而易见的好处就是限制了作用域。如果 baseVC 的话就意味着你的所有 VC 都能弹出错误，而这并不是必须的。你可以限定只有部分需要的有可能出现错误的 VC 遵守这个协议。每个本来写在 VC 里的小功能块其实都可以拆出来，然后把不同的功能组合起来成为一个 VC，这会让你的代码更加清楚，而且每个小功能块都是可以测试的。另外，baseVC 是无法维护的，你会不自觉地把所有垃圾扔到那里，然后有的 VC 继承了 base 有的又没有，子 VC 的行为可能会被 base 改变。最后的问题是 baseVC 只是 ViewController 吧，那 TableVC 和 CollectionVC 甚至 PageVC 又怎么办 (其实就是个横切点关注的问题)?

在这方面，使用协议来对功能进行组合，要远比继承更加清晰可维护。
2 回复 · 分享 ›

Zerrun · 10个月前
组合优于继承
3 回复 · 分享 ›