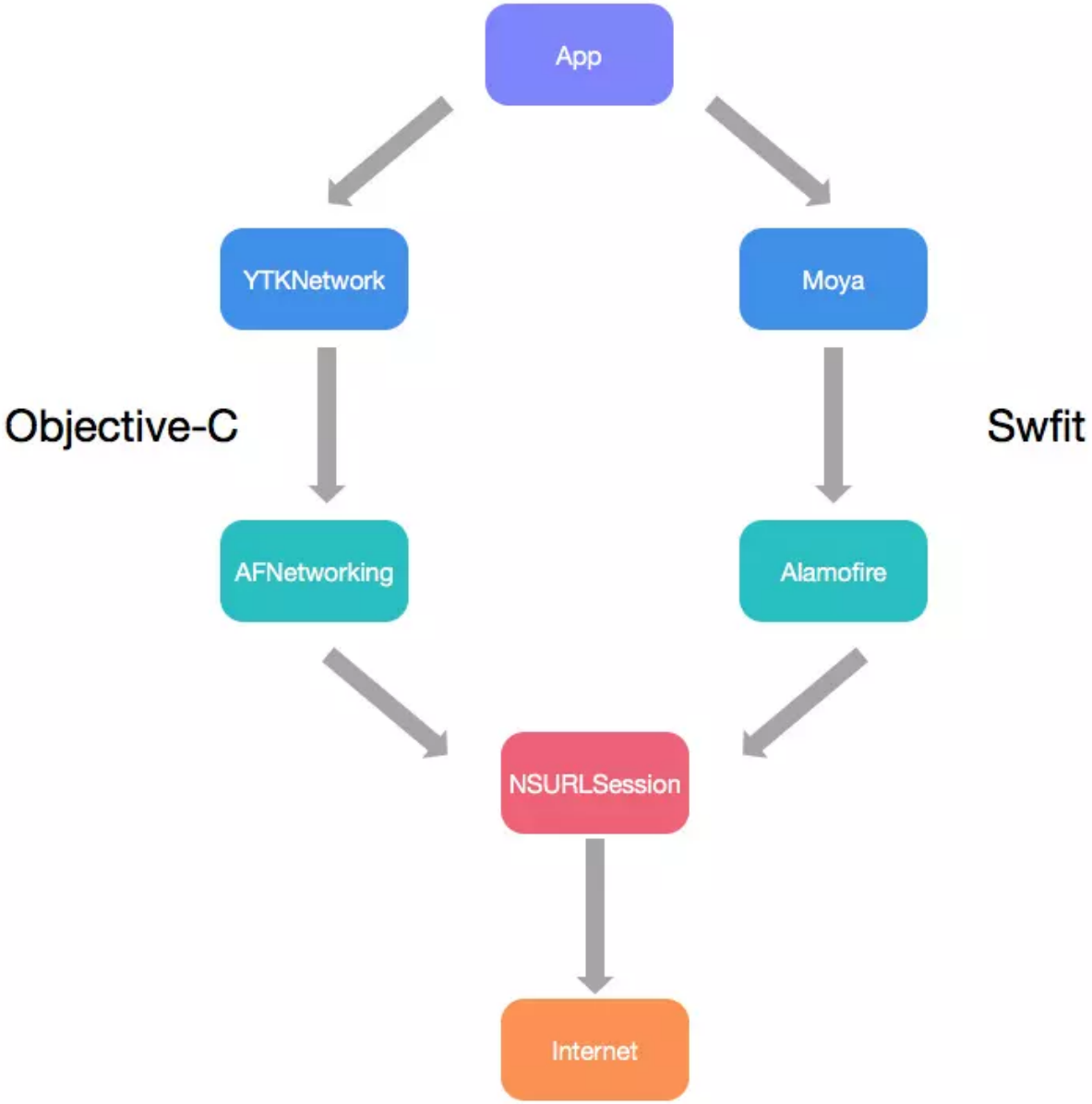
深入理解Moya设计

2018 年 01 月 26 日 YinTokey

关于Moya

Moya是一个网络抽象层，它在底层将Alamofire进行封装，对外提供更简洁的接口供开发者调用。在以往的Objective-C中，大部分开发者会使用AFNetwork进行网络请求，当业务复杂一些时，会对AFNetwork进行二次封装，编写一个适用于自己项目的网络抽象层。在Objective-C中，有著名的YTKNetwork，它将AFNetworking封装成抽象父类，然后根据每一种不同的网络请求，都编写不同的子类，子类继承父类，来实现请求业务。Moya在项目层次中的地位，有点类似于YTKNetwork。可以看下图对比



但是如果单纯把Moya等同于swift版的YTKNetwork，那就是比较错误的想法了。Moya的设计思路和YTKNetwork差距非常大。上面我在介绍YTKNetwork时在强调子类和父类，继承，是因为YTKNetwork是比较经典的利用OOP思想（面向对象）设计的产物。基于swift的Moya虽然也有使用到继承，但是它的整体上是以POP思想（Protocol Oriented Programming,面向协议编程）为主导的。

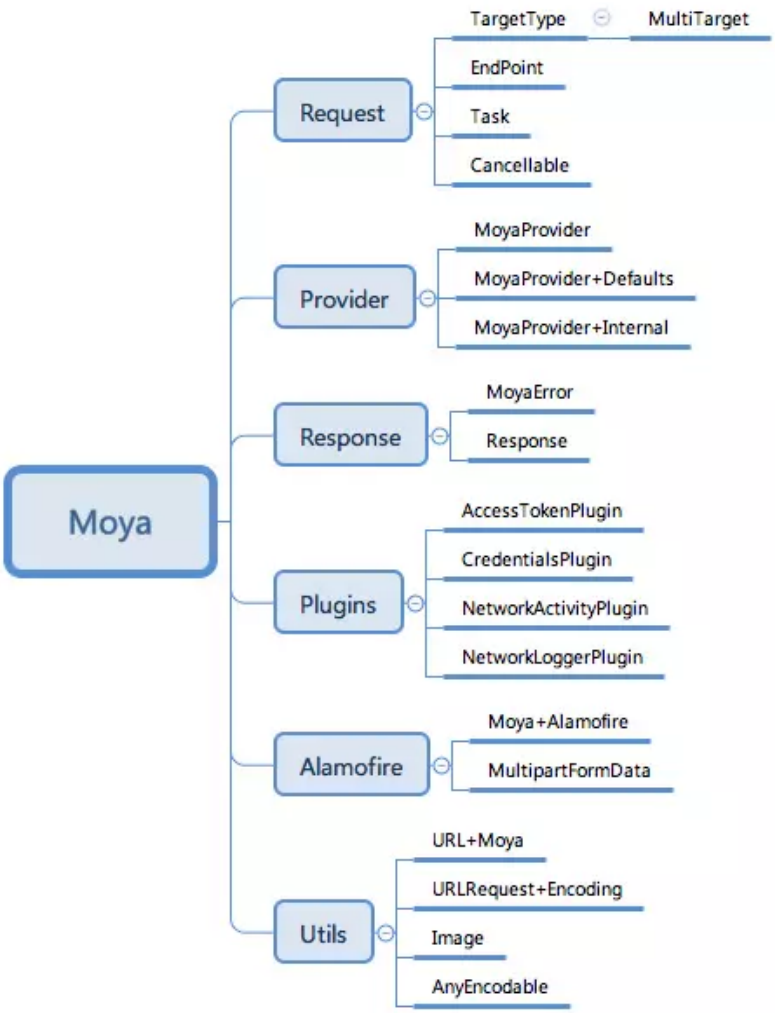
面向协议编程（POP）

在阅读Moya源码之前，如果对POP有一定了解，那么理解其Moya会事半功倍的效果。在Objective-C也有协议，一般是让对象遵守协议，然后实现协议规定的方法，通过这种方式来对类实现扩展。POP其实就是把这种思路进一步强化。很多时候事物具备多样化的特质，这些特质是无法单纯从一个类中继承而来的。为了解决这个痛点，C++有了多继承，即一个

处的属性和方法。比如父类进行了修改，那么很难避免影响到子类。C++的多继承还会带来 菱形缺陷 ，什么是 菱形缺陷 ？本节的下方我会放两个链接，方便大家查阅。而Swift则引入了面向协议编程，通过协议来规定事物的实现。通过遵守不同的协议，来对一个类或者结构体或者枚举进行定制，它只需要实现协议所规定的属性或方法即可，有点类似于搭建积木，取每一块有需求的模块，进行组合拼接，相对于OOP，其耦合性更低，也为代码的维护和拓展提供更多的可能性。关于POP思想大致是这样，下面是王巍关于POP的两篇文章，值得读一番。 面向协议编程与 Cocoa 的邂逅 (上) 面向协议编程与 Cocoa 的邂逅 (下)

Moya的模块组成

由于Moya是使用POP来设计的一个网络抽象层，因此他整体的逻辑结构并没有明显的继承关系。Moya的核心代码，可以分成以下几个模块



Provider

provider是一个提供网络请求服务的提供者。通过一些初始化配置之后，在外部可以直接用provider来发起request。

Request

在使用Moya进行网络请求时，第一步需要进行配置，来生成一个Request。首先按照官方文档，创建一个枚举，遵守TargetType协议，并实现协议所规定的属性。为什么要创建枚举来遵守协议，而不像Objective-C那样创建类来遵守协议呢？其实使用类或者结构体也是可以的，这里猜测使用枚举的原因是因为swift的枚举功能比Objective-C强大许多，枚举结合switch语句，使得API管理起来比较方便。 Request的生成过程如下图

我们根据上图，结合代码来分析其Request的生成过程。 根据创建了一个遵守TargetType协议的名为Myservice的枚举，我们完成了如下几个变量的设置。

```
baseURL
path
method
sampleData
task
headers
```

提供了这些网络请求的“基本材料”之后，就可以进一步配置去生成所需要的请求。看上图的第一个箭头，通过了一个EndpointClosure生成了endPoint。endPoit是一个对象，把网络请求所需的一些属性和方法进行了包装，在EndPoint类中有如下属性：

```
public typealias SampleResponseClosure = () -> EndpointSampleResponse

open let url: String
open let sampleResponseClosure: SampleResponseClosure
open let method: Moya.Method
open let task: Task
open let httpHeaderFields: [String: String]?
```

可以很直观地看出来，EndPoint这几个属性可以和上面通过TargetTpye配置的变量对应起

```
/// Closure that defines the endpoints for the provider.
    public typealias EndpointClosure = (Target) -> Endpoint<Target>

    open let endpointClosure: EndpointClosure
```

声明了一个闭包，参数为Target，它是一个泛型，然后返回一个EndPoint。endPoint是一个类，它对请求的参数和动作进行了包装，下面会对它进行详细说明，先继续看endpointClosure做了什么。

```
endpointClosure: @escaping EndpointClosure = MoyaProvider.defaultEndpointMapping
```

在MoyaProvider的初始化方法里，调用其扩展的类方法 `defaultEndpointMapping` 输入Target作为参数，返回了一个endPoint对象。

```
    public final class func defaultEndpointMapping(for target: Target) -> Endpoint<Targe
        return Endpoint(
            url: URL(target: target).absoluteString,
            sampleResponseClosure: { .networkResponse(200, target.sampleData) },
            method: target.method,
            task: target.task,
            httpHeaderFields: target.headers
        )
    }
```

Target就是一开始进行配置的枚举，通过点语法取出Target的变量，完成endPoint的初始化。这里可能对于url和sampleResponseClosure会感到一些疑惑。url初始化，可以进入 `URL+Moya.swift` 查看，它对NSURL类进行构造器的扩展，让其具备根据Moya的TargetType来进行初始化的能力。

```
    /// Initialize URL from Moya's `TargetType`.
     init<T: TargetType>(target: T) {
        // When a TargetType's path is empty, URL.appendingPathComponent may introduce t
        if target.path.isEmpty {
            self = target.baseURL
        } else {
            self = target.baseURL.appendingPathComponent(target.path)
        }
    }
```

sampleResponseClosure是一个和网络请求返回假数据相关的闭包，这里可以先忽略，不

```
endpointClosure: @escaping EndpointClosure = MoyaProvider.defaultEndpointMapping
```

使用@escaping把endpointClosure声明为逃逸闭包，我们可以把

```
EndpointClosure = MoyaProvider.defaultEndpointMapping
```

转换为

```
(Target) -> Endpoint<Target> = func defaultEndpointMapping(for target: Target) -> Endpoi
```

再进一步转换，等号左边的可以写成一个常规的闭包表达式

```
{(Target)->Endpoint<Target> in
        return Endpoint(
            url: URL(target: target).absoluteString,
            sampleResponseClosure: { .networkResponse(200, target.sampleData) },
            method: target.method,
            task: target.task,
            httpHeaderFields: target.headers
        )
}
```

即endpointClosure这个闭包，传入了Target作为参数，该闭包可以返回一个endPoint对象，如何获取到闭包返回的endPoint对象？MoyaProvider提供了这么一个方法

```
    /// Returns an `Endpoint` based on the token, method, and parameters by invoking the
    open func endpoint(_ token: Target) -> Endpoint<Target> {
        return endpointClosure(token)
    }
```

以上就是关于TargetType通过endpointClosure转化为endPoint的过程。

下一步就是把利用requestClosure，传入endPoint，然后生成request。request生成过程和endPoint很相似。

在MoyaProvider中声明：

```
  /// Closure that decides if and what request should be performed
    public typealias RequestResultClosure = (Result<URLRequest, MoyaError>) -> Void
    open let requestClosure: RequestClosure
```

```
requestClosure: @escaping RequestClosure = MoyaProvider.defaultRequestMa
```

进入查看defaultRequestMapping方法

```
public final class func defaultRequestMapping(for endpoint: Endpoint<Target>, closur
    do {
        let urlRequest = try endpoint.urlRequest()
        closure(.success(urlRequest))
    } catch MoyaError.requestMapping(let url) {
        closure(.failure(MoyaError.requestMapping(url)))
    } catch MoyaError.parameterEncoding(let error) {
        closure(.failure(MoyaError.parameterEncoding(error)))
    } catch {
        closure(.failure(MoyaError.underlying(error, nil)))
    }
}
```

和endpointClosure类似，我们经过转换，可以得到requestClosure的表达式为

```
{(endpoint:Endpoint<Target>, closure:RequestResultClosure) in
    do {
        let urlRequest = try endpoint.urlRequest()
        closure(.success(urlRequest))
    } catch MoyaError.requestMapping(let url) {
        closure(.failure(MoyaError.requestMapping(url)))
    } catch MoyaError.parameterEncoding(let error) {
        closure(.failure(MoyaError.parameterEncoding(error)))
    } catch {
        closure(.failure(MoyaError.underlying(error, nil)))
    }

}
```

整体上使用do-catch语句来初始化一个urlRequest，根据不同结果向闭包传入不同的参数。一开始使用try来调用endpoint.urlRequest()，如果抛出错误，会切换到catch语句中去。endpoint.urlRequest()这个方法比较长，这里就不放出来，感兴趣可自行到Moya核心代码里的Endpoint.swift里查看。它其实做的事情很简单，就是根据前面说到的endpoint的那些属性来初始化一个NSURLRequest的对象。

以上就是上方图中所画的，根据TargetType最终生成Request的过程。很多人会感到疑惑为什么搞得这么麻烦，直接一步到位，传一些必要参数生成Request不就完了？为什么还要

```
    /// Convenience method for creating a new `Endpoint` with the same properties as the
    open func adding(newHTTPHeaderFields: [String: String]) -> Endpoint<Target>

        /// Convenience method for creating a new `Endpoint` with the same properties as
    open func replacing(task: Task) -> Endpoint<Target>
```

借用这些方法，在endpointClosure中可以给一些网络请求添加请求头，替换请求参数，让这些请求配置更加灵活。

我们看完了整个Request生成过程，那么通过requestClosure生成的的Request是如何被外部拿到的呢？这就是我们下一步要探讨的，Provider发送请求实现过程。在下一节里将会看到如何使用这个Request。

Provider发送请求

我们再来看一下官方文档里说明的Moya的基本使用步骤

1. 创建枚举，遵守TargetType协议，实现规定的属性。
2. 初始化 `provider = MoyaProvider<Myservice>()`
3. 调用provider.request，在闭包里处理请求结果。

其中第一步我们在上方已经说明完了，MoyaProvider的初始化我们只说明了一小部分。在此不准备一口气初始化方法中剩余的部分讲完，这又会涉及很多东西，同时理解起来会比较麻烦。在后面的代码解读中，如果有涉及到相关属性，再回到初始化方法中一个一个突破。

```
    open func request(_ target: Target,
                      callbackQueue: DispatchQueue? = .none,
                      progress: ProgressBlock? = .none,
                      completion: @escaping Completion) -> Cancellable {

        let callbackQueue = callbackQueue ?? self.callbackQueue
        return requestNormal(target, callbackQueue: callbackQueue, progress: progress, c
    }
```

直接从这里可能看不出什么，再追溯到 `requestNormal` 中去 这个方法内容比较长，其中一些插件相关的代码，和测试桩的代码，暂且跳过不做说明，暂时不懂他们并不会成为理解provider.request的阻碍，它们属于可选内容，而不是必须的。

```
        let endpoint = self.endpoint(target)
```

生成了endPoint对象，这个很好理解，前面已经做过说明。 查看 `performNetworking` 闭包

```
        if cancellableToken.isCancelled {
                self.cancelCompletion(pluginsWithCompletion, target: target)
                return
            }
```

如果取消请求，则调用取消完成的回调，并return,不在执行闭包内下面的语句。 在这个闭包里传入了参数 `(requestResult: Result<URLRequest, MoyaError>)` ， 这里用到了Result,想 解，可自行研究，这里简单说一下Result是干什么的。Result使用枚举方式，提供一些运行处理的结果,如下，很容易能看懂它所表达的意思。

```
    switch requestResult {
            case .success(let urlRequest):
                request = urlRequest
            case .failure(let error):
                pluginsWithCompletion(.failure(error))
                return
            }
```

如果请求成功，会拿到URLRequest，如果失败，会使用插件去处理失败回调。

```
            // Allow plugins to modify request
            let preparedRequest = self.plugins.reduce(request) { $1.prepare($0, target:
```

使用插件对请求进行完善

```
            cancellableToken.innerCancellable = self.performRequest(target, request: pre
```

这里的 `self.performRequest` 就是进行实际的网络请求，内部代码比较多，但是思路很简单，使用Alamofire的SessionManager来发送请求。 配置完成后就可以调用 `requestClosure(endpoint, performNetworking)` ， 执行这个闭包获取到上方所说的Request，来执行具体的网络请求了。

Response

在使用Alamofire发送请求时，定义了闭包来处理请求的响应。Response这个类对于请求结果，提供了一些加工方法，比如data转json,图片转换等。

Plugins

```
/// Called to modify a request before sending
    func prepare(_ request: URLRequest, target: TargetType) -> URLRequest

    /// Called immediately before a request is sent over the network (or stubbed).
    func willSend(_ request: RequestType, target: TargetType)

    /// Called after a response has been received, but before the MoyaProvider has invok
    func didReceive(_ result: Result<Moya.Response, MoyaError>, target: TargetType)

    /// Called to modify a result before completion
    func process(_ result: Result<Moya.Response, MoyaError>, target: TargetType) -> Resu
```

- `prepare` 可以在请求之前对request进行修改。
- `willSend` 在请求发送之前的一瞬间调用，这个可以用来添加请求时转圈圈的Toast
- `didReceive` 在接收到请求响应时，且MoyaProvider的completion handler之前调用。
- `process` 在completion handler之前调用，用来修改请求结果 可以通过以下图来直观地理解插件调用时机

使用插件的方式，让代码仅保持着主干逻辑，使用者根据业务需求自行加入插件来配置自己的网络业务层，这样做更加灵活，低耦合。Moya提供了4种插件

- AccessTokenPlugin OAuth的Token验证

- NetworkActivityPlugin 网络请求状态
- NetworkLoggerPlugin 网络日志 可以根据需求编写自己的插件，选取NetworkActivityPlugin来查看插件内部构成。

```
public final class NetworkActivityPlugin: PluginType {

    public typealias NetworkActivityClosure = (_ change: NetworkActivityChangeType, _ ta
    let networkActivityClosure: NetworkActivityClosure

    public init(networkActivityClosure: @escaping NetworkActivityClosure) {
        self.networkActivityClosure = networkActivityClosure
    }

    public func willSend(_ request: RequestType, target: TargetType) {
        networkActivityClosure(.began, target)
    }

    public func didReceive(_ result: Result<Moya.Response, MoyaError>, target: TargetTyp
        networkActivityClosure(.ended, target)
    }
}
```

插件内部结构很简单，除了自行定义的一些变量外，就是遵守 `PluginType` 协议后，去实现协议规定的方法，在特定方法内做自己需要做的事。因为 `PluginType` 它已经有一个协议扩展，把方法的默认实现都完成了，在具体插件内不一定需要实现所有的协议方法，仅根据需要实现特定方法即可。 写好插件之后，使用起来也比较简答，MoyaProvider的初始化方法中，有个形参 `plugins: [PluginType] = []` ，把网络请求中需要用到的插件加入数组中。

总结

Moya可以说是非常Swift式的一个框架，最大的优点是使用面向协议的思想，让使用者能以搭积木的方式配置自己的网络抽象层。提供了插件机制，在保持主干网络请求逻辑的前提下，让开发者根据自身业务需求，定制自己的插件，在合适的位置加入到网络请求的过程中。

设计 Swift iOS Objective-C

相关文章

让你的代码自动格式化

小顾Bruce 19 4

奇舞周刊第 245 期：普通用户与你之间的差距

奇舞周刊 1

苹果官方对PWA支持步伐奇快， iOS 11.3 和 macOS 10.13.4 将默认支持Servic...

BrilliantOpenWeb 55 8

iOS多线程：『pthread、NSThread』详尽总结

WalkingBoy 5

评论

说说你的看法