[译]Swift 网络单元测试完全手册

> - 原文地址：The complete guide to Network Unit Testing in Swift
> - 原文作者：S.T.Huang
> - 译文出自：掘金翻译计划
> - 本文永久链接：github.com/xitu/gold-m...
> - 译者：swants
> - 校对者：pthtc ZhiyuanSun

不得不承认，对于 iOS 开发写测试并不是很普遍(至少和后端写测试程度相比)。我过去是个独立开发者而且最初也没经过原生“测试驱动”的开发培训，因此我花费了大量的时间来学习如何编写测试用例，如何写出可测试的代码。这也是我写这篇文章的初衷，我想把自己用 Swift 写测试时摸索到的心得分享给大家，希望我的见解能够帮助大家节省学习时间，少走些弯路。

在这篇文章，我们将会讨论着手写测试的入门知识：**依赖注入**。

想象一下，你此时正在写测试。 如果你的测试对象(被测系统)是和真实世界相连的，比如 Networking 和 CoreData，编写测试代码将会非常复杂。原则上讲，我们不希望我们的测试代码被客观世界的事物所影响。被测系统不应依赖于其他的复杂系统，这样我们才能够保证在时间恒定和环境恒定条件下迅速完成测试。况且，保证我们的测试代码不会“污染”生产环境也是十分重要的。“污染”意味着什么？意味着我们的测试代码将一些测试对象写进了数据库，提交了些测试数据到生产服务器等等。而避免这些情况的发生就是 **依赖注入** 存在的意义。

让我们从一个例子开始。 假设你拿到个应该联网并且在生产环境下才能被执行的类，联网部分就被称作该类的 **依赖**。如之前所言，当我们执行测试时这个类的联网部分必须能够被模拟的，或者假的环境所替换。换句话说，该类的依赖必须支持“可注入”，依赖注入使我们的系统更加灵活。我们能够为生产代码“注入”真实的网络环境；与此同时，也能够“注入”模拟的网络环境来让我们在不访问互联网的条件下运行测试代码。

TL;DR

> 译者注：TL;DR 是 Too long;Don't read 的缩写。在这里的意思是篇幅较长，不想深入研究，请直接看文章总结。

在这篇文章，我们将会讨论：

1. 如何使用 **依赖注入** 技术设计一个对象
2. 在 Swift 中如何使用协议设计一个模拟对象
3. 如何测试对象使用的数据及如何测试对象的行为

依赖注入

开始动手吧! 现在我们打算实现一个叫做 **HttpClient** 的类。这个 HttpClient 应该满足以下要求：

1. HttpClient 跟初始的网络组件对于同一 URL 应提交同样的 request。
2. HttpClient 应能够提交 request。

所以我们对 HttpClient 的初次实现是这样的：

```
class HttpClient {
    typealias completeClosure = ( _ data: Data?, _ error: Error?)->Void
    func get( url: URL, callback: @escaping completeClosure ) {
        let request = NSMutableURLRequest(url: url)
        request.httpMethod = "GET"
        let task = URLSession.shared.dataTask(with: request) { (data, response, error) in
            callback(data, error)
        }
        task.resume()
    }
}
```

HttpClient 看起来可以提交一个“GET”请求，并通过“callback”闭包将返回值回传。

```
HttpClient().get(url: url) { (success, response) in // Return data }
```

HttpClient 的用法。

这就是问题所在：我们怎么对它测试？我们如何确保这些代码达到上述的两点要求？凭直觉，我们可以给 HttpClient 传入一个 URL，运行代码，然后在闭包里观察得到的结果。但是这些操作意味着我们在运行 HttpClient 时必须每次都连接互联网。更糟糕的是如果你测试的 URL 是连接生产服务器：你的测试在一定程度上会影响服务器性能，而且你提交的测试数据将会被提交到真实的世界。就像我们之前描述的，我们必须让 HttpClient “可测试”。

我们来看下 URLSession。URLSession 是 HttpClient 的一种‘环境’，是 HttpClient 连接互联网的入口。还记得我们刚讨论的“可测试”代码吗？ 我们需要将互联网部分变得可替换，于是我们修改了 HttpClient 的实现：

```
class HttpClient {
    typealias completeClosure = ( _ data: Data?, _ error: Error?)->Void
    private let session: URLSession
    init(session: URLSessionProtocol) {
        self.session = session
    }
    func get( url: URL, callback: @escaping completeClosure ) {
        let request = NSMutableURLRequest(url: url)
        request.httpMethod = "GET"
        let task = session.dataTask(with: request) { (data, response, error) in
            callback(data, error)
        }
        task.resume()
    }
}
```

我们将

```
let task = URLSession.shared.dataTask()
```

修改成了

```
let task = session.dataTask()
```

我们增加了新的变量：**session**，并添加了对应的 **init** 方法。之后每当我们创建 HttpClient 对象时，就必须初始化 **session**。也就是说，我们已经将 session “注入”到了我们创建的 HttpClient 对象中。现在我们就能够在运行生产代码时注入 ‘URLSession.shared’，而运行测试代码时注入一个模拟的 session。Bingo!

这时 HttpClient 的用法就变成了：HttpClient(session: SomeURLSession()).get(url: url) { (success, response) in // Return data }

给此时的 HttpClient 写测试代码就会变得非常简单。因此我们开始布置我们的测试环境：

```
class HttpClientTests: XCTestCase {
    var httpClient: HttpClient!
    let session = MockURLSession()
    override func setUp() {
        super.setUp()
        httpClient = HttpClient(session: session)
    }
    override func tearDown() {
        super.tearDown()
    }
}
```

这是个规范的 XCTestCase 设置。**httpClient** 变量就是被测系统，**session** 变量是我们将为 httpClient 注入的环境。因为我们要在测试环境运行代码，所以我们将 MockURLSession 对象传给 **session**。这时我们将模拟的 **session** 注入到了 httpClient，使得 httpClient 在 URLSession.shared 被替换成 MockURLSession 的情况下运行。

测试数据

现在让我们注意下第一点要求：

1. HttpClient 和初始的网络组件对于同一 URL 应提交同样的 request 。

我们想达到的效果是确保该 request 的 url 和我们传入“get”方法的 url 完全一致。

以下是我们的测试用例：

```
func test_get_request_withURL() {
    guard let url = URL(string: "https://mockurl") else {
        fatalError("URL can't be empty")
    }
    httpClient.get(url: url) { (success, response) in
        // Return data
    }
    // Assert
}
```

这个测试用例可表示为：

- **Precondition**: Given a url “https://mockurl”
- **When**: Submit a http GET request

- **Assert**: The submitted url should be equal to “https://mockurl”

我们还需要写断言部分。

但是我们怎么知道 HttpClient 的“get”方法确实提交了正确的 url 呢？让我们再看眼依赖：URLSession。通常，“get”方法会用拿到的 url 创建一个 request，并把 request 传给 URLSession 来完成提交：

```
let task = session.dataTask(with: request) { (data, response, error) in
    callback(data, error)
}
task.resume()
```

接下来，在测试环境中 request 将会传给 MockURLSession，所以我们只要 hack 进我们自己的 MockURLSession 就可以查看 request 是否被正确创建了。

下面是 MockURLSession 的粗略实现：

```
class MockURLSession {
    private (set) var lastURL: URL?
    func dataTask(with request: NSURLRequest, completionHandler: @escaping DataTaskResult) -> URL
        lastURL = request.url
        completionHandler(nextData, successHttpURLResponse(request: request), nextError)
        return // dataTask, will be impletmented later
    }
}
```

MockURLSession 的作用和 URLSession 一样，URLSession 和 MockURLSession 有同样的 dataTask() 方法和相同的回调闭包类型。虽然 URLSession 比 MockURLSession 的 dataTask() 做了更多的工作，但它们的接口是类似的。正是由于它们的接口相似，我们才能不需要修改“get”方法太多代码就可以用 MockURLSession 替换掉 URLSession。接着我们创建一个 **lastURL** 变量来跟踪“get”方法提交的最终 url 。简单点说，就是当测试的时候，我们创建一个注入 MockURLSession 的 HttpClient，然后观察 url 是否前后相同。

以下是测试用例的大概实现：

```
func test_get_request_withURL() {
    guard let url = URL(string: "https://mockurl") else {
        fatalError("URL can't be empty")
    }
    httpClient.get(url: url) { (success, response) in
        // Return data
    }
    XCTAssert(session.lastURL == url)
}
```

我们为 **lastURL** 和 **url** 添加断言，这样就会得知注入后的“get”方法是否正确创建了带有正确 url 的 request。

上面的代码仍有一处地方需要实现：`return // dataTask`。在 URLSession 中返回值必须是个 URLSessionDataTask 对象，但是 URLSessionDataTask 已经不能正常创建了，所以这个 URLSessionDataTask 对象也需要被模拟创建：

```
class MockURLSessionDataTask {
    func resume() { }
}
```

作为 URLSessionDataTask，模拟对象需要有相同的方法 resume()。这样才会把模拟对象当做 dataTask() 的返回值。

如果你跟着我一块敲代码，就会发现你的代码会被编译器报错：

```
class HttpClientTests: XCTestCase {
    var httpClient: HttpClient!
    let session = MockURLSession()
    override func setUp() {
        super.setUp()
        httpClient = HttpClient(session: session) // Doesn't compile } override
func tearDown() { super.tearDown() } }
```

这是因为 MockURLSession 和 URLSession 的接口不一样。所以当我们试着注入 MockURLSession 的时候会发现 MockURLSession 并不能被编译器识别。我们必须让模拟的对象和真实对象拥有相同的接口，所以我们引入了“协议”！

HttpClient 的依赖：

```
private let session: URLSession
```

我们希望不论 URLSession 还是 MockURLSession 都可以作为 session 对象，因此我们将 session 的 URLSession 类型改为 URLSessionProtocol 协议：

```
private let session: URLSessionProtocol
```

这样我们就能够注入 URLSession 或 MockURLSession 或者其它遵循这个协议的对象。

以下是协议的实现：

```
protocol URLSessionProtocol { typealias DataTaskResult = (Data?, URLResponse?, Error?) -> Void
    func dataTask(with request: NSURLRequest, completionHandler: @escaping DataTaskResult) -> URL
}
```

测试代码中我们只需要一个方法：`dataTask(NSURLRequest, DataTaskResult)`，因此在协议中我们也只需定义一个必须实现的方法。当我们需要模拟不属于我们的对象时这个技术通常很适用。

还记得 MockURLDataTask 吗？另一个不属于我们的对象，是的，我们要再创建个协议。

```
protocol URLSessionDataTaskProtocol { func resume() }
```

我们还需让真实的对象遵循这个协议。

```
extension URLSession: URLSessionProtocol {}
extension URLSessionDataTask: URLSessionDataTaskProtocol {}
```

URLSessionDataTask 有个同样的 resume() 协议方法，所以这项修改对于 URLSessionDataTask 是没有影响的。

问题是 URLSession 没有 dataTask() 方法来返回 URLSessionDataTaskProtocol 协议，因此我们需要拓展方法来遵循协议。

```
extension URLSession: URLSessionProtocol {
    func dataTask(with request: NSURLRequest, completionHandler: @escaping DataTaskResult) -> URL
        return dataTask(with: request, completionHandler: completionHandler) as URLSessionDataTas
    }
}
```

这个简单的方法只是将返回类型从 URLSessionDataTask 改成了 URLSessionDataTaskProtocol，不会影响到 dataTask() 的其它行为。

现在我们就能够补全 MockURLSession 缺失的部分了：

```
class MockURLSession {
    private (set) var lastURL: URL?
    func dataTask(with request: NSURLRequest, completionHandler: @escaping DataTaskResult) -> URL
        lastURL = request.url
        completionHandler(nextData, successHttpURLResponse(request: request), nextError)
        return // dataTask, will be impletmented later
    }
}
```

我们已经知道 // dataTask... 可以是一个 MockURLSessionDataTask：

```
class MockURLSession: URLSessionProtocol {
    var nextDataTask = MockURLSessionDataTask()
    private (set) var lastURL: URL?
    func dataTask(with request: NSURLRequest, completionHandler: @escaping DataTaskResult) -> URL
        lastURL = request.url
        completionHandler(nextData, successHttpURLResponse(request: request), nextError)
        return nextDataTask
    }
}
```

在测试环境中模拟对象就会充当 URLSession 的角色，并且 url 也能够被记录供断言判断。是不是有种万丈高楼平地起的感觉! 所有的代码都已经编译完成并且测试也顺利通过!

让我们继续。

测试行为

第二点要求是：

```
The HttpClient should submit the request
```

我们希望 HttpClient 的“get”方法将 request 如预期地提交。

和之前验证数据是否正确的测试不同，我们现在要测试的是方法是否被顺利调用。换句话说，我们想知道 URLSessionDataTask.resume() 方法是否被调用了。让我们继续使用刚才的老把戏： 我们创建一个新的 resumeWasCalled 变量来记录 resume() 方法是否被调用。

我们简单写一个测试：

```
func test_get_resume_called() {
    let dataTask = MockURLSessionDataTask()
    session.nextDataTask = dataTask
    guard let url = URL(string: "https://mockurl") else {
        fatalError("URL can't be empty")
    }
    httpClient.get(url: url) { (success, response) in
        // Return data
    }
    XCTAssert(dataTask.resumeWasCalled)
}
```

dataTask 变量是我们自己拥有的模拟对象，所以我们可以添加一个属性来监控 resume() 方法的行为：

```
class MockURLSessionDataTask: URLSessionDataTaskProtocol {
    private (set) var resumeWasCalled = false
    func resume() {
        resumeWasCalled = true
    }
}
```

如果 resume() 方法被调用了， `resumeWasCalled` 就会被设置成 `true` ! :) 很简单，对不对?

总结

通过这篇文章，我们学到：

1. 如何调整依赖注入来改变生产/测试环境。
2. 如何利用协议来创建模拟对象。
3. 如何检测传值的正确性。
4. 如何断言某个函数的行为。

刚起步时，你必须花费大量时间来写简单的测试，而且测试代码也是代码，所以你仍需要保持测试代码的简洁和良好的架构。但编写测试用例得到的好处也是弥足珍贵的，代码只有在恰当的测试后才能被扩展，测试帮你免于琐碎 bug 的困扰。所以让我们一起加油写好测试吧!

所有的示例代码都在 GitHub 上，代码是以 Playground 的形式展示的，我还在上面添加了个额外的测试。 你可以自由下载或 fork 这些代码，并且欢迎任何反馈!

感谢阅读我的文章 🐈

参考文献

1. Mocking Classes You Don't Own
2. Dependency Injection
3. Test-Driven iOS Development with Swift

感谢 Lisa Dziuba 和 Ahmed Sulaiman.

> 掘金翻译计划 是一个翻译优质互联网技术文章的社区，文章来源为 掘金 上的英文分享文章。内容覆盖 Android、iOS、前端、后端、区块链、产品、设计、人工智能等领域，想要查看更多优质译文请持续关注 掘金翻译计划、官方微博、知乎专栏。