

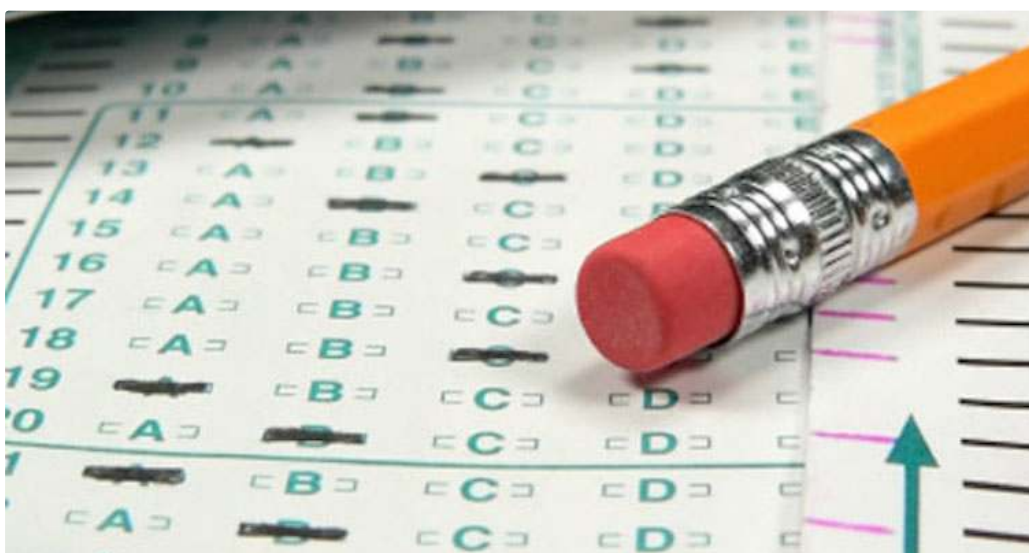


OneV's Den

上善若水，人淡如菊

2014-05-09 • 能工巧匠集

Kiwi 使用进阶 Mock, Stub, 参数捕获和异步测试



Kiwi 是 iOS 的一个行为驱动开发 (Behavior Driven Development, BDD) 的测试框架，我们在[上一篇入门介绍](#)中简单了解了一些 iOS 中测试的概念以及 Kiwi 的基本用法。其实 Kiwi 的强大远不止如此，它不仅包含了基本的期望和断言，也集成了一些相对高级的测试方法。在本篇中我们将在之前的基础上，来看看 Kiwi 的这些相对高级的用法，包括模拟对象 (mock)，桩程序 (stub)，参数捕获和异步测试等内容。这些方法都是在测试中会经常用到的，用来减少我们测试的难度的手段，特别是在耦合复杂的情况下的测试以及对于 UI 事件的测试。

Stub 和 Mock 的基本概念

如果您曾经有过为代码编写测试的经验，您一定会知道其中不易。我们编写生产代码让它能够工作其实并不难，项目中编码方面的工作难点往往在于框架搭建以及随着项目发展如何保持代码优雅可读可维护。而测试相比起业务代码的编写一般来说会更难一些，很多时候你会发现有些代码是“无法测试”的，因为代码之间存在较高的耦合程度，因此绕不开对于其他类的依赖，来对某个类单独测试其正确性。我们不能依赖于一个没有经过测试的类来对另一个需要测试的类进行测试，如果这么做了，我们便无法确定测试的结果是否正是按我们的需要得到的（不能排除测试成功，但是其实是因为未测试的依赖类恰好失败了而恰巧得到的正确结果的可能性）。

Stub

解决的方法之一是我们用一种最简单的语言来“描述”那些依赖类的行为，而避免对它们进行具体实现，这样就能最大限度地避免出错。比如我们有一个复杂的算法通过输入的温度和湿度来预测明天的天气，现在我们在存储类中暴露了一个方法，它接受输入的温度和湿度，通过之前复杂算法的计算后将结果写入到数据库中。相关的代码大概是下面这个样子，假设我们有个 `WeatherRecorder` 类来做这件事：

```
//WeatherRecorder.m
-(void) writeResultToDatabaseWithTemperature:(NSInteger)temperature
                                     humidity:(NSInteger)humidity
{
    id result = [self.weatherForecaster resultWithTemperature:temperature humidity:humidity];
}
```

(虽然这个例子设计得不太好, 因为服务层架构不对, 但是其实) 在实际项目中是可能会有不少类似的代码。对于这样的方法和相应的 `WeatherRecorder` 应该如何测试呢? 这个方法依赖了 `weatherForecaster` 的计算方法, 而我们这里关心的更多的是 `write` 这个方法的正确性 (算法的测试应该被分开写在对应的测试中), 对于计算的细节和结果我们其实并不关心。但是这个方法本身和算法耦合在了一起, 我们当然可以说直接给若干组输入, 运行这个方法然后检测数据库中的结果是否与我们预期的一致, 但是这其实做了假设, 那就是: 在测试中我们自己的计算结果和预报计算方法的结果是一致的。这个假设可能在一开始是成立的, 但是你无法知道在之后的开发中这个算法会不会改变, 会变成怎样。也许之后有修正模型出现, 结果和现在大相径庭, 这时就会出现 `write` 数据库的测试居然因为预报的算法变更而失败。这不仅使得测试涵盖了它不应该包括的内容, 违背了测试的单性, 也凭添了不少麻烦。

一个完美的解决的方案是, 我们人为地来指定计算的结果, 然后测试数据库的写入操作。人为地让一个对象对某个方法返回我们事先规定好的值, 这就叫做 `stub`。

在 Kiwi 中写一个 `stub` 非常简单, 比如我们有一个 `Person` 类的实例, 我们想要 `stub` 让它返回一个固定的名字, 可以这么写:

```
Person *person = [Person somePerson];
[person stub:@selector(name) andReturn:@"Tom"];
```

在这个 `stub` 下, 如下测试将会通过, 而不论 `person` 到底具体是谁:

```
NSString *testName = [person name];
[ testName should] equal:@"Tom"];
```

另外, 对于我们之前天气预报例子中的带有参数的方法, 我们可以使用 Kiwi `stub` 的带参数版本来进行替换, 比如:

```
[weatherForecaster stub:@selector(resultWithTemperature:humidity:)
                    andReturn:someResult
                    withArguments:theValue(23),theValue(50)];
```

这时我们再给 `weatherForecaster` 发送参数为温度 `23` 和湿度 `50` 的消息时, 方法会直接将 `someResult` 返回给我们, 这样我们就可以不再依赖于天气预报算法的具体实现, 也不用担心算法变更会破坏测试, 而对数据库写入进行稳定的测试了。

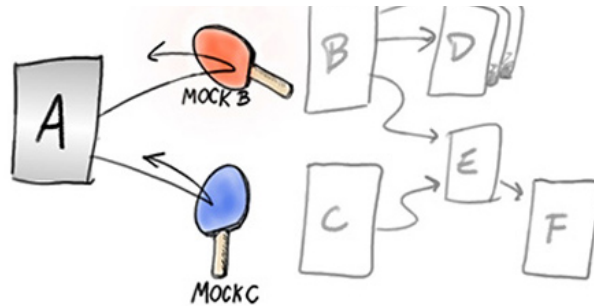
对于 Kiwi 的 `stub`, 需要注意的是它不是永久有效的, 在每个 `it` block 的结尾 `stub` 都会被清空, 超出范围的方法调用将不会被 `stub` 截取到。

Mock

`mock` 是一个非常容易和 `stub` 混淆的概念。简单来说, 我们可以将 `mock` 看做是一种更全面和更智能的 `stub`。

首先解释全面, 我们需要明确, `mock` 其实就是一个对象, 它是对现有类的行为一种模拟 (或是对现有接口实现的模拟)。在 `objc` 的 OOP 中, 类或者接口就是指导对象行为的蓝图, 而 `mock` 则遵循这些蓝图并模拟它们的实例对象。从这方面来说, `mock` 与 `stub` 最大的区别在于 `stub` 只是简单的方法替换, 而不涉及新的对象, 被 `stub` 的对象可以是业务代码中真正的对象。而 `mock` 行为本身产生新的 (不可能在业务代码中出现的) 对象, 并遵循类的定义相应某些方法。

其次是更智能。基础上来说, 和 `stub` 很相似, 我们可以为创造的 `mock` 定义在某种输入和方法调用下的输出, 更进一步, 我们还可以为 `mock` 设定期望 (准确来说, 是我们一定会为 `mock` 设定期望, 这也是 `mock` 最常见的用例)。即, 我们可以为一个 `mock` 指定这样的期望: “这个 `mock` 应该收到以 `X` 为参数的 `Y` 方法, 并规定它的返回为 `Z`”。其中”应该收到以 `X` 为参数的 `Y` 方法”这个期望会在测试与其不符合时让你的测试失败, 而“返回 `Z`”这个描述行为更接近于一种 `stub` 的定义。XCTest 框架想要实现这样的测试例可以说要费九牛之力, 但是这在 Kiwi 里却十分自然。



还是举上面的天气预报的例子。我们在 stub 时将 `weatherForecaster` 的方法替换处理了。细心的读者可能会有疑惑，问这个 `weatherForecaster` 是怎么来的。因为这个对象其实只是 `WeatherRecorder` 中一个属性，而且很有可能在测试时我们并不能拥有一个恰好合适的 `weatherForecaster`。`WeatherRecorder` 是不需要将 `weatherForecaster` 暴露到头文件中的，VC 是不需要知道它的实现细节的），而我们在上面的 stub 的前提是我们在测试代码中拿到这个 `weatherForecaster`，很多时候只能修改代码将其暴露，但是这并不是好的实践，很多时候也并不现实。现在有了 mock 后，我们就可以自创一个虚拟的 `weatherForecaster`，并为其设定期望的调用来确保我们输入温度和湿度确实经过了计算然后存入了数据库中了。mock 所使用的期望和普通对象的调用期望类似：

```
id weatherForecasterMock = [WeatherForecaster mock];
[[weatherForecasterMock should] receive:@selector(resultWithTemperature:humidity:)
                                andReturn:someResult
                                withArguments:theValue(23),theValue(50)];
```

然后，对于要测试的 `weatherRecorder` 实例，用 stub 将 `-weatherForecaster` 的返回换为我们的 mock：

```
[weatherRecorder stub:@selector(weatherForecaster) andReturn:weatherForecasterMock];
```

这样一来，在 `-writeResultToDatabaseWithTemperature:humidity:` 中我们就可以使用一个 mock 的 `weatherForecaster` 来完成工作，并检验是否确实进行了预报了。类似的组合用法在 mock/stub 测试中是比较常见的，在本文最后的例子中我们会再次见到类似的用法。

参数捕获

有时候我们会对 mock 对象的输入参数感兴趣，比如期望某个参数符合一定要求，但是对于 mock 而言一般我们是通过调用别的方法来验证 mock 是否被调用的，所以很可能无法拿到传给 mock 对象的参数。这种情况下我们就可以使用参数捕获来获取输入的参数。比如对于上面的 `weatherForecasterMock`，如果我们想捕获温度参数，可以在调用测试前使用

```
KWCaptureSpy *spy = [weatherForecasterMock captureArgument:@selector(resultWithTemperature:humidity:) atIndex:0];
```

来加一个参数捕获。这样，当我们在测试中使用 stub 将 `weatherForecaster` 替换为我们的 mock 后，再进行如下调用

```
[weatherRecorder writeResultToDatabaseWithTemperature:23 humidity:50]
```

后，我们可以通过访问 `spy.argument` 来拿到实际输入 `resultWithTemperature:humidity:` 的第一个参数。

在这个例子中似乎不太有用，因为我们输入给 `-writeResultToDatabaseWithTemperature:humidity:` 的参数和 `-resultWithTemperature:humidity:` 的是一样的。但是在某些情况下确实会很有效果，我们会在之后看到一个实际的使用例。

异步测试

异步测试是为了对后台线程的结果进行期望检验时所需要的，Kiwi 可以对某个对象的未来的状况书写期望，并进行检验。通过将要检验的对象加上 `expectFutureValue`，然后使用 `shouldEventually` 即可。就像这样：

```
[[expectFutureValue(theValue(myBool)) shouldEventually] beYes];
```

比如在 REST 网络测试中，我们可能大部分情况下会选择用一组 mock 来替代服务器的返回进行验证，但是也不排除会有直接访问服务器进行测试的情况。在这种情况下我们就可以使用延时来进行异步测试。这里直接照抄一个官方 Wiki 的例子进行说明：

```
context(@"Fetching service data", ^{
    it(@"should receive data within one second", ^{

        __block NSString *fetchedData = nil;

        [[LRResty client] get:@"http://www.example.com" withBlock:^(LRRestyResponse* r) {
            NSLog(@"That's it! %@", [r asString]);
            fetchedData = [r asString];
        }];
        [[expectFutureValue(fetchedData) shouldEventually] beNotNil];
    });
});
```

这个测试保证了返回的 `LRRestyResponse` 对象可以转为一个字符串并且不是 `nil`。

其实没什么神奇的，就是生成了一个延时的验证，在一定时间间隔后再对观测的对象进行检查。这个时间间隔默认是 1 秒，如果你需要其他的时间间隔的话，可以使用 `shouldEventuallyBeforeTimingOutAfter` 版本：

一个例子：测试 ViewController

举个实际一点例子吧，我们来看看平时觉得难以测试的 `UIViewController` 的部分，包括一个 `tableView` 和对应的 `dataSource` 和 `delegate` 的测试方法。我们使用了 objc.io 第一期中的 **Lighter View Controllers** 和 **Clean table view code** 中的代码来实现一个简单可测试的 VC 结构，然后使用 Kiwi 替换完成了 **Testing View Controllers** 一文中的所有测试模块。这里篇幅有限，实现的具体细节就不在复述了，有兴趣的同学可以看看 objc.io 的这三篇文章，或者也可以在 **objc 中国** 上找到它们的译文：**更轻量的 View Controllers**，**整洁的 Table View 代码**以及**测试 View Controllers**。

我们在这里结合 Kiwi 的方法对重写的测试部分进行一些说明。**objc.io 原来的项目**使用的是 **OCMock** 实现的解耦测试，而为了进行说明，我用 Kiwi 简单重写了测试部分的代码，这个项目也可以在 **Github** 上找到。

对于 `ArchiveReading` 的测试都是 Kiwi 最基本的内容，在**上一篇文章**中已经详细介绍了；对于 `PhotoCell` 的测试形式上比较新颖，其实是一个对 `xib` 的测试，保证了 `xib` 的初始化和 `outlet` 连接的正确性，但是测试内容也比较基本。剩下的是对于 `tableView` 的 `dataSource` 和 `viewController` 的测试，我们来具体看看。

Data Source 的测试

首先是 `ArrayDataSourceSpec`，得益于将 `array` 的 `dataSource` 进行抽象和封装，我们可以单独对其进行测试。基本思路是我们希望在一个 `tableView` 设置好数据源后，`tableView` 可以正确地数据源获取组织 UI 所需要的信息，基本上来说，也就是能够得到“有多少行”以及“每行的 cell 是什么”这两个问题的答案。到这里，有写过 iOS 的开发者应该都明白我们要测试的是什么了。没错，就是 `-tableView:numberOfRowsInSection:` 以及 `-tableView:cellForRowAtIndexPath:` 这两个接口的实现。

测试用例关键代码如下：

```
TableViewCellConfigureBlock block = ^(UITableViewCell *a, id b){
    configuredCell = a;
    configuredObject = b;
};
ArrayDataSource *dataSource = [[ArrayDataSource alloc] initWithItems:@[@"a", @"b"] cellIdentifier:@"foo"];

id mockTableView = [UITableView mock];
UITableViewCell *cell = [UITableViewCell alloc] init];

it(@"should be 2 items", ^{
    NSInteger count = [dataSource tableView:mockTableView numberOfRowsInSection:0];
```

```

__block id result = nil;
NSIndexPath *indexPath = [NSIndexPath indexPathForRow:0 inSection:0];

it(@"should receive cell request", ^{
    [[mockTableView should] receive:@selector(dequeueReusableCellWithIdentifier:forIndexPath:) andReturn:
    result = [dataSource tableView:mockTableView cellForRowAtIndexPathIndexPath:indexPath];
});

```

为了简要说明，我改变了 repo 中的代码组织结构，不过意思是一样的。我们要测试的是 `ArrayDataSource` 类，因此我们生成一个实例对象。在测试中我们不希望测试依赖于 `UITableView`，因此我们 mock 了一个对象代替之。接下来向 `dataSource` 发送询问元素个数的方法，这里应该毫无疑问返回数组中的元素数量。接下来我们给 `mockTableView` 设定了一个期望，当将向这个 mock 的 `tableView` 请求 `dequeue indexPath` 为 (0,0) 的 cell 时，将直接返回我们预先生成的一个 cell，并进行接下来的处理。完成设定后，我们调用要测试的方法 `[dataSource tableView:mockTableView cellForRowAtIndexPathIndexPath:indexPath]`。`dataSource` 在接到这个方法后，向 `mockTableView` 请求一个 cell（这个方法已经被 mock），接下来通过之前定义的 block 来对 cell 进行配置，最后返回并赋值给 `result`。于是，我们就得到了一个可以进行期望断言的 `result`，它应该和我们之前做的 cell 是同一个对象，并且经过了正确的配置。至此这个 `dataSource` 测试完毕。

您当然还可以扩展这个 `dataSource` 并且为其添加对应的测试，但是对于这两个 `required` 方法的测试已经揭示了测试 Data Source 的基本方法。

ViewController 的测试

ViewController 一般被认为是最难测试甚至不可测试的部分。而通过 `objc.io` 的抽离方式可以使 MVC 更加清晰，也让 ViewController 的代码简洁不少。保持良好的 MVC 结构，尽可能精简 ViewController，对其的测试还是有可能及有意义的。在 `PhotosViewControllerSpec` 里做了对 ViewController 的一个简单测试。我们模拟了 `tableView` 中对一个 cell 的点击，然后检查 `navigationController` 的 `push` 操作是否确实被调用，以及被 `push` 的对象是否是我们想要的下一个 ViewController。

要测试的是 `PhotosViewController` 的实例，因此我们生成一个。对于它的 `UINavigationController`，因为其没有导航栈中，也这不是我们要测试的对象（保持测试的单一性），所以用一个 mock 对象来代替。然后为其设定 `-pushViewController:animated:` 需要被调用的期望。然后再用输入参数捕获将被 `push` 的对象抓出来，进行判断。关键部分代码如下：

```

UINavigationController *mockNavController = [UINavigationController mock];
[photosViewController stub:@selector(navigationController) andReturn:mockNavController];

[[mockNavController should] receive:@selector(pushViewController:animated:)];
KWCaptureSpy *spy = [mockNavController captureArgument:@selector(pushViewController:animated:) atIndex:0];
[photosViewController tableView:photosViewController.tableView didSelectRowAtIndexPath:[NSIndexPath indexPathWithIndex:0]];

id obj = spy.argument;
PhotoViewController *vc = obj;
[[vc should] beKindOfClass:[PhotoViewController class]];
[[vc.photo shouldNot] beNil];

```

在这里我们用 `stub` 替换了 `photosViewController` 的 `navigationController`，这个替换进去的 `UINavigationController` 的 mock 被期望响应 `-pushViewController:animated:`。于是在点击 `tableView` 的 cell 时，我们期望 `push` 一个新的 `PhotoViewController` 实例，这一点可以通过捕获 `push` 消息的参数来达成。大体的步骤和原理与之前天气预报的例子最终版本很相似，在此就不再详细展开了。

关于 mock 还有一点需要补充的是，使用 `+mock` 方法生成的 mock 对象对于期望收到的方法是严格判定的，就是说它能且只能响应那些你添加了期望或者 `stub` 的方法。比如只为一个 mock 设定了 `should receive selector(a)` 这样的期望，那么对这个 mock 发送一个消息 `b` 的话，将会抛出异常（当然，如果你没有向其发送消息 `a` 的话，测试会失败）。如果你的 mock 还需要相应其他方法的话，可以使用 `+nullMock` 方法来生成一个可以接受任意预定消息而不会抛出异常的空 mock。

总结

区中可以说是直线上升，个论是 Apple 官方的维护或者是开发者的重视程度的提高。良好的代码习惯和良好的测试应该是相辅相成，良性循环的。

最后对几个常见问题做一些总结：

我应不应该测试，要怎么做

应该，即使在你的工作中没有要求。测试会让你的生活更美好轻松。你是愿意花 10 分钟完成对你代码的自动化测试，还是在 QA 找过来以后花一整天去找 bug，并且同时制造更多的 bug？即使你现在还完全不了解也不会编写测试，你也可以从最简单的 model 测试开始，抽离并封装逻辑部分，然后用 XCTest 做简单断言。先建立测试的概念，然后有意编写可测试代码，最终掌握测试方法。

需要使用 TDD 么

建议使用。虽然看上去这有点疯狂，虽然一开始会有不适应，但是这确实是对思维检验的非常好的时机。在实际很爽地去一通乱写之前先做好整体设计，TDD 确实可以帮助提高项目结构和质量。开始的时候建议小粒度进行，可能开发效率会有一段低谷时期（但是相较于代码质量的提高，这点付出还是很值得的），熟悉之后可以加大步伐，并且积累一套适合自己的测试风格，这时候你就会发现开发效率像坐火箭一般提升了。

需要使用 BDD 么

可以考虑在有一定积累后使用。因为有些情况下 XCTest 确实略显苍白，有些测试实现起来也很繁芜。BDD 在一定程度上可以将测试的目的理得更清晰，当然，前提是你需要明确知道想测试的是什么，以及尽量保证测试的单一性和无耦合。另外虽然这两篇文章介绍的是 Kiwi，但是实际上我们在 objc 的 BDD 时还有不少其他选择，比如 **specta** 或者 **cedar** 都是很好的框架。我个人喜欢 Kiwi 纯粹是因为和 Kiwi 作者和维护社区里的几位大大的个人关系，大家如果要实践 BDD，在选择的时候也可以进行一些对比，选择合适自己的。

扩展阅读

- **Test-Driven iOS Development** 我的 iOS TDD 入门书
- **Kiwi 的 Wiki** 关于 Kiwi 你所需要知道的一切
- **Unit Testing - NSHipster**
- **Test Driving iOS Development with Kiwi** iBook的书，中国区不让卖书，所以可能需要非中国账号（日本账号这本书只要 5 美金）



最近的文章

近期随想和 WWDC 的计划

最近的博文总是写技术，本来其实是打算将这里建设成技术成长与人文关怀并重的博客的，但是现在看来思考不足。在刚被每周七天每天 18 小时的魔鬼般的封闭开发连续虐待了三周之后，我基本达到了看一眼代码就想吐的地步。每天让我坚持下来的动力可能只剩“过完这周就可以参加的 WWDC”这一件事情了。于是觉得，现在是时候可以写一点技术无关的博文来舒缓舒缓心情了。其实在封闭开发期间发生了不少事情，整理在一起看来，还是颇为值得思考的。首先是经历了一件很不幸的事情，我的一个非常优秀的大学同学，也是家内从高中开.....

2014-05-30 • 南箕北斗集 [继续阅读](#)


更早的文章

苹果应用描述中不能使用特殊字符的对应方法

该文章内容在iOS7中已经失效，请乖乖遵循苹果的规则写吧虽然很早Apple就说过从5月1日开始就不再允许UDID以及没有对iPhone5优化的应用上架，但是这次iTunes Connect的对于描述字符的限制还是让很多开发者措手不及。毕竟事先完全没有和大家打过招呼，Apple想要统一应用市场的风格和体验的心态可以理解，但是在开发者难得还有一点自由发挥的应用描述的地方突然作出这样的限制，确实不太厚道。相关的新闻报道可以参看这里但是，难道我们真的没法使用更漂亮的描述了么？答案是，有办法！解决办.....

2014-04-28 • 南箕北斗集 [继续阅读](#)

9条评论 OneV's Den

 登录 推荐 2  分享

最新发布



加入讨论...

通过以下方式登录

或注册一个 DISQUS 帐号 ?



DamianSheldon · 3年前

博文的内容部分丢失了，“如果你的 mock 还需要”应该还有下文的吧。

  · 回复 · 分享

onevcat 管理员 → DamianSheldon · 3年前

感谢提醒，已经补充

  · 回复 · 分享

amoblin · 3年前

博主的Ghost主题很不错，能分享一下吗？

  · 回复 · 分享

onevcat 管理员 → amoblin · 3年前

我有时间的话会整理一下然后开源的

  · 回复 · 分享

雷锋 → onevcat · 3年前

本来就是拿的别人的，还谈什么开源？原作者的链接：[http://www.daleanthony.com/...](http://www.daleanthony.com/)  · 回复 · 分享

onevcat 管理员 → 雷锋 · 3年前

基于 CC4.0 的演绎版本没什么不能谈开源的，遵守协议就可以~

  · 回复 · 分享

ChenYu Xiao · 3年前

喵神辛苦了啊。在各种天坑搞定之后。还更新了一下blog主题和后台。v587

  · 回复 · 分享

david · 3年前

不错，学习学习

  · 回复 · 分享

哈哈 · 3年前

感觉国内的公司很少有用到tdd的。。。

  · 回复 · 分享

在 ONEV'S DEN 上还有

面向协议编程与 Cocoa 的邂逅 (下)

20条评论 · 10个月前

Zerrun — 组合优于继承

Swift 并行编程现状和展望 - async/await 和参与者模式

17条评论 · 10个月前

Akring — Swift 的体验确实超过Objective-C，虽然之前一年一次的重新入门每次都让我放弃治疗，但是每次新开项目的时候我都会坚定的选Swift...

开发者所需要知道的 iOS 10 SDK 新特性

36条评论 · 1年前

mrboog — 每次大会开完后，都要来喵神的博客看看，哈哈

活久见的重构 - iOS 10 UserNotifications 框架解析

39条评论 · 1年前

sanSong zhang —

 订阅  在您的网站上使用 Disqus添加 Disqus添加  隐私

