iOS单元测试

> 本文侧重讲述如何在iOS程序的开发过程中使用单元测试。使用Xcode自带的OCUnit作为测试框架。

一、单元测试概述

单元测试作为敏捷开发实践的组成之一，其目的是提高软件开发的效率，维持代码的健康性。其目标是证明软件能够正常运行，而不是发现bug（发现bug这一目的与开发成本是正相关的，虽然发现bug是保证软件质量的一种手段，但是很显然这与降低软件开发成本这一目的背道而驰）。它是对软件质量的一种保证，例如重构之后我们需要保证软件产品的正常运行。

很多人认为编写单元测试没有用是认为单元测试并不能保证一定能减少bug发生的几率，而由于编写单元测试一定会花费一定的时间与精力，因而必然的会增加成本。客观的说，造成这种原因很大的程度上是程序员的水平不够高。我认为使用使用单元测试带来巨大好处的必要条件如下所示：

```
程序员本身的编程水平--是否有较多的代码经验，是否熟练掌握重构
程序员对项目的认知--是否能正确理解软件或模块的需求
项目质量--是否稳定，是否长期多版本，是否需要应对较多变化
```

如果程序员水平较高，对需求理解较为清晰，项目需要面对较多的变化，那么毫无疑问单元测试对于软件非常有益。假如软件功能简单且开发周期短，不需要进行复杂的维护工作，那么单元测试的意义并不大。

优秀的单元测试实践的好处：

```
好的单元测试就是一份好的文档，并且比文档更能为程序员所接受，它直接描述了测试员对受测代码的结果所持的预期。
当代码由别人维护时（或自己进行重构时），通过单元测试的约束，才能保证在加入新功能或修改旧功能时代码的正确性。
由于单元测试的自动化执行，保证了在整个开发流程中代码都会被测试，这非常符合X
```

P思想。
保证在面对软件功能的变化时，程序员可以较为放心的进行代码重构，而不必担心是否破坏了原有功能。
好的单元测试可以降低bug数量，而对于项目管理来说，修改bug这个过程是无法制定计划的，可以使软件的开发流程更容易掌控。
可以由老程序员编写描述某个类行为的测试，以此指导新程序员对类的编码。
……

好处还有很多，但最重要的一点就是保证了软件质量的同时，由于减少bug和应对变化造成的回归bug的产生等，提高了劳动生产率。而且，在敏捷流程中，使用单元测试是必须掌握的手段，否则就没发保证重构的正确性，从而造成代码无法面对变化。

二、iOS的单元测试概述

刚接触客户端编程时，我在很长一段时间内都想不通对于客户端程序如何编写单元测试。单元测试本质上说白了就是用一些断言来判定结果，而这种方式是如何应用到具有复杂交互的界面测试上来的呢?

我们要做的就是将客户端代码转化为易于测试的代码。什么样的代码易于测试呢？它至少是这样的：

- 1、被测方法需要产生可测量的结果。
- 2、类之间的关系应该是松耦合的。

其中第一条是必要条件。使用断言这种形式指明了测试的方法最终要造成某些可以度量的结果。因而，我们需要尽量的将展示和业务逻辑分离开来。展示的代码是没法测试的，例如有的方法只是播放动画。而业务逻辑最终都会造成一些数据的改变，这是容易测试的。

大略的讲，作为一个iOS程序员来说，首先要了解一个叫做MVC的模式。这个模式定义了Cocoa Touch框架的总体结构。在iOS程序中，我们也需要按照这种模式进行界面代码的编写。这样设计出来的类具有较好的结构，且比较适合于做单元测试。

然后一定要懂得不停重构代码，这样我们才能使代码不停地改善，不停地变得更加适合单元测试。

有一些框架可以帮助大家更好的测试，分别

是`OCUnit`、`GTM`、`GHUnit`、`CATCH`、`OCMock`，但目前对我来说，OCUnit足够用了。作为苹果官方提供的测试框架，它最大的优点就是简单易用。

三、单元测试实践

下面是一些我所理解的单元测试中比较好的实践。

顾名思义，单元测试面向的对象是单元，这个专有名词源自编译器领域的术语“编译单元”。在面向过程中，指的是函数，而在面向对象中，指的通常就是“类”。因而，每个功能类都应该提供对应的单元测试。

实践1 每个功能类都应提供单元测试，且每一个测试类，只依赖于其要测试的受测类。使用伪造对象可以避免对其他类的依赖。

解释 保证一个测试类只关注一个被测类，当测试不通过时，就能迅速的定位到是谁发生了错误，而不会受到其他类的干扰。

简单的数据类等可以不提供，但是要保证该测试的都要覆盖到。并不存在一种合适的度量指标可以量化地判断某种单元测试方案是否成功。常用的标准（代码覆盖率和成功执行的测试用例数）都可以在受测软件的质量不变的情况下人为的修改（作假）。当然，在无法确保程序员素质的情况下，作为没有办法的办法，使用这种标准也是可以的（或者无奈的说，必须的）。单元测试需要程序员自己把关，关注哪些功能确实需要测试覆盖。这也就是前面所说的一些程序员不相信单元测试可以提高生产率的理由--它更多的依赖于程序员的素质，这是没有保证的。但同样的，由于敏捷是一种以人为本的思想实践，因而这种行为似乎又是一种必然。

实践1.1 使用伪造类避免对其他类的依赖

解释 避免依赖的一种手段。
例如，某个被测的方法声明是这样的：

```
-(void)xxxx:(Person *)person;
```

如果测试时传入Person的话，就造成了测试类依赖于两个类。当由于person中的错误引发测试不通过时，就不能迅速的定位到受测类中是否有问题。遇到这种情况，就可以使用伪造类。假如方法中只使用了person的一个属性name，那么可以将方法名重构为

```
-(void)xxxx:(id)person; (此处id有待商榷，只是这样做最简单)
```

然后在单元测试的target中添加只包含name属性的fakePerson来作为伪造类。这样，一旦发生错误就可以迅速的推测出错误的来源。

实践1.2 使用伪造环境避免其他环境的干扰。

解释 适合于异步的方法测试。

很经常遇到的一种情况是测试有网络环境的代码。由于异步的存在，这会造成测试代码不好写。一种简单的解决方法是，我们假定网络一定是通畅的，则我们测试的代码将分为两部分，即拼装发送功能和接收解析功能。假如发送和接收功能各自都能通过测试，那么我们大约可以确定这个异步方法的正确性。另一种方法是使用GHUnit，它支持异步代码的测试。

实践2 测试用例（方法）名应该是自解释的且是独立的。

解释 基本功。

如果被测试类的名称是XXX，那么测试类可以命名为XXXTests。而对于其中要测试的功能，命名应该是自解释的。这可以在发现错误时尽快的定位问题所在。例如，如果某个属性obj应该是非空的，那么我们可以将其命名为：

```
-(void)testObjNotNil{}
```

每个方法目标应该是单一的，大多数情况下每个方法内都只有一个断言语句；方法不应该依赖于其他方法的结果作为输入，保证原子性。

实践3 断言语句需要解释测试者的意图。

解释 基本功

每种单元测试框架都提供了很多断言语句，从根本上来说它们都是一样的。但是测试者需要根据自己的目的选择适当的语句，这样才可以让别人阅读测试代码时理解用例设计的目的。例如对于STAssertNil和STAssertNotNil等等。

实践4 判断某个意图有没有达到的很好的方法是检测方法影响的数据有没有合理的变化。

解释 基本功

由于单元测试是使用断言语句来做判断的，因而最容易做的就是判断数据的变化。这也就限定了单元测试能测试的方法范围，即引起数据变化的方

法。对于一些纯展示的方法，例如播放一段特效，这种方法是无法靠单元测试来进行约束的。测试数据的特性包括取值范围（int、float等），排列顺序（NSArray等），类型等等。

实践5 运用重构的手段使方法变得易于被测试。

解释 单元测试是保障重构安全的手段，重构也可以使代码易于被测试。

什么样的代码是容易进行单元测试的？最简单的一点就是，每个被测方法都应该是功能单一的。当然，这也是代码规范中应该做到的。方法的功能单一，则测试方法的断言也会比较好确定。如果你发现某个方法很难进行测试，则就应该对这个方法进行拆分重构。

实践5.1 面向抽象设计类之间的关系。

解释 利于伪造类的实现。

类之间通讯如果依赖于抽象（接口），则可以较容易的使用伪造类。参照实践1.1。

实践6 运用自上而下的方式构建类。

解释 自上而下的方式可以使类的功能明确，类的构成将会清晰紧凑，不会出现一些废方法。

先确定类需要负担的责任，以此来确定类具有的公有方法以及属性。通过重构将公有方法中的代码转化为私有方法，以使方法尽量短小紧凑。

实践6.1 应对所有暴露的属性和方法提供测试，私有方法则不必。

解释 如果运用自上而下的方式构建类，则理论上私有方法应该都是公有方法重构而得到的。实际上测试公有方法时这些私有方法都应该被测试到了。而且，由于私有方法相对公有方法来说发生变动的可能性很大，会造成不必要的修改测试代码的成本。

回调方法不属于私有方法，也需要进行测试。

实践6.2 回调方法的测试方法是直接调用。

解释 基本功

由于回调方法一般是异步和不可触发的（按正常流程），例如网络事件的返回和按下按钮的触发事件。因而，测试的时候要直接调用来对其流程进行检测。例如某个按钮的touch up inside事件：

```
-(void)buttonPressed:(id)sender;
```

可以根据方法中用到的方法、属性伪造一个FakeButton按钮作为参数传递进行测试。

实践6.2 测试私有的方式，KVC、子类化和类别。

解释 基本功。

遇到需要通过验证私有数据才能编写的测试时，可以考虑使用KVC和子类化。子类继承于被测类，只包含于单元测试target，其作用就是在不该变受测类的情况下，使受测类具有某些易于被测的能力。

实践7：变化需要新测试的支持。

解释：保证测试的覆盖度。

就像敏捷中提到的“改变需要抽象”一样，在测试中改变需要新的测试。当然，度依然由程序员自己掌控。

四、一般流程

使用OCUnit最大的好处就是流程非常的简单，简单到让你觉得非常愉悦。由于有XCode的支持，添加测试变得异常简单。只要在新建工程时勾选“Include Unit Tests”，就会自动的加入一个示例。然后再需要添加新的单元测试时，新建一个“Objective-C test case class”就可以了。

测试文件中，只要知道setUp是初始化的地方，tearDown是结束清理的地方，而且它们在每个用例方法执行时都会重新执行--这保证了测试用例的原子性。然后知道每个测试用例都是以test作为前缀的，并且无返回值。然后在方法中编写断言语句就可以了。输入STAssertxxxxx就可以看到它们的联想提示。编写完成后，执行菜单Product->Test，单元测试就完成了！

五、测试驱动（TDD）

敏捷当中提到了TDD这种开发方式。TDD的主旨是使开发者对其编写的代码更有信心，使开发者修改代码时心里更加踏实。对于其总结，还是引用原文比较妥当：“测试驱动开发的妙处即在于，它以需求为引领，通过测试的形式，来指导开发者进行软件的设计与架构，并编写出最为精炼的代码，使得测试用例运行通过。经过适当的重构之后，测试用例与产品代码可达到较为健康的状态。”也就是上面提到的，通过自上而下的形式设计

类，通过单元测试来不停地审视和重构类，从而达到代码的健康。

如果在代码写完之后在编写单元测试，那么就体现不出这种模式的好处了。这就好像写完代码再补文档一样，没有什么意义。测试应该在代码开始之前，或者在代码编写中不停地进行编写更新，这样才能使代码不停进步。这也正是TDD的意思。

六、总结

单元测试的代码如此简单，但是想写好单元测试却并不是一件简单的事情。它需要程序员比较深的功底。由于个人水平所限，有一些东西说的比较啰嗦。把复杂问题简单化是本事，任重而道远。希望大家可以在日常开发中运用好这种简洁高效的技术。