**Pull Bears技术文档**

－**关于APP（iOS）单元测试的建议**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 更新日期 | 更新版本 | 更新人 | 更新内容 | 审批人 |
| 2015/8/11 | 1.0.0 | 胡家驹 | 新建 |  |
|  |  |  |  |  |

# 文档说明

## 1.背景：

**a，为什么要单元测试？**

传统的移动应用测试通常是系统集成测试，测试人员进行黑盒手工测试或者UI自动化测试。系统集成测试通常是开发将产品完成交付给测试之后才能进行测试，bug发现得较晚，而且仅仅测试人员进行测试，难以保证软件质量。分层测试包含单元测试、集成测试、验收测试等。开发人员在项目早期就介入单元测试，在提高开发人员的质量意识的同时，bug能够快速反馈，降低了测试风险

**b，为什么开发不愿意做单元测试？**

编写测试是一项投资，开发需要花时间编写和维护它们。我们可以证明这种投资有回报的唯一方法就是我们期望节省时间。

如何让单元测试节省时间？答案是要注意测试的颗粒度，不宜过细也不宜过粗；注意测试的思路，对代码的行为进行测试。

## 2．编写目的：

本文档编写的目的是为app（使用Objective-C的iOS）开发人员在开发过程中编写单元测试而编写。针对APP的测试我简单区分为两层：单元测试和自动化测试，其中本文仅讨论单元测试，自动化测试请参见其他文档。

## 3．适用范围：

本文档描述的内容适用于app（使用Objective-C的iOS）开发人员，也适用于使用swift作为开发语言的iOS应用开发人员及相关人员。

# 文档内容

目录

文档说明 2

1.背景： 2

2．编写目的： 2

3．适用范围： 3

文档内容 3

前言： 5

iOS单元测试： 6

什么时候开始写单元测试？ 7

单元测试框架选型： 7

XC Test框架： 11

单元测试用例： 13

单元测试命名规范： 15

XCTest断言解释：（OCHamrest断言扩展请查阅资料此不再赘述） 15

## 前言：

**什么是单元测试？**

打个比方，单元测试就像一把尺子，当测量的对象是一个曲面时，也许可以花费大力气去将它抽象成平面，但我更提倡量身定做一把弯曲的尺子去适应这个曲面。无论怎样，单元测试是为了生产代码而写，它应当足够的自由奔放，去适应各种各样的生产代码。

单元测试的目的显而易见，控制数据使用和操作的过程，检验模块和接口能否达到预期的效果。

**为什么要单元测试？**

随着项目规模越来越大，逻辑和数据将会变的更复杂；模块负责人的变更，新员工的入职，由于对逻辑和数据结构的不熟悉，为了避免牵一发而动全身的风险，导致重复代码堆积。而单元测试更像一个框架、脚手架，让开发更顺畅，代码质量更健壮。

**单元测试开发驱动**

**a，TDD 测试驱动开发（Test Driven Development）**

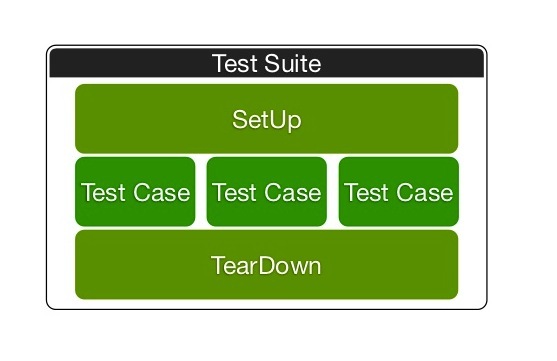
测试驱动开发（Test Driven Development,英文缩写TDD）是极限编程的一个重要组成部分，它的基本思想就是在开发功能代码之前，先编写测试代码。也就是说在明确要开发某个功能后，首先思考如何对这个功能进行测试，并完成测试代码的编写，然后编写相关的代码满足这些测试用例。然后循环进行添加其他功能，直到完成全部功能的开发。代码整洁可用(clean code that works) 是测试驱动开发所追求的目的

**b，BDD 行为驱动开发（Behaviour Driven Development）**

行为驱动开发是测试驱动开发的进化，但关注的核心是设计。行为驱动开发中，定义系统的行为是主要工作，而对系统行为的描述则变成了测试标准。在行为驱动开发中，我们需要使用通用语言来定义系统行为。而通用语言，实际上是一个最小化的词汇表。我们使用这些词汇来书写故事。选入词汇表的词汇必须具有准确无误的表达能力和一致的含义。

## iOS单元测试：

**一个完成的测试类组成如下：**



### 什么时候开始写单元测试？

关于测试的资料，网上有很多。基本都有高深的理论和方法来指导怎么写单元测试。咱们从简单出发，简化考虑。 无非就3种时候会去想写测试：

1. 代码完成以后
2. 开始写代码之前
3. 修复了一个bug以后

分析：  
第一种是在完成了代码之后，我要测试一下我写的这些方法可靠不可靠。那这时候可以写测试。

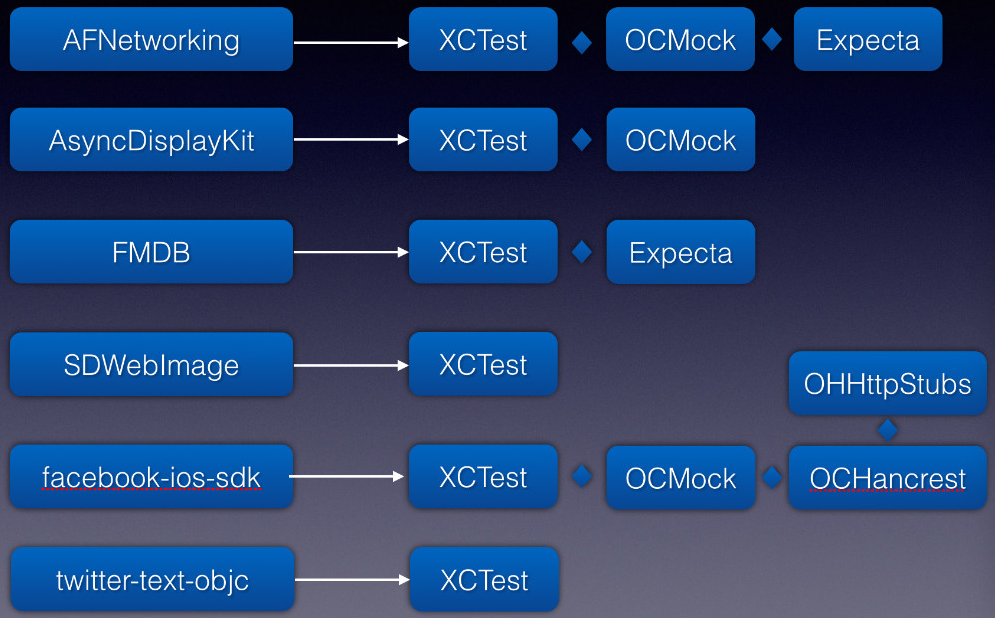
第二种有一个著名的方法论TDD（上边简单有提到）。主要思想就是在写代码之前，就全部设计好借口。函数名字什么的。然后在写能通过测试的方法。

第三种就是发现了bug，我修复了这个bug。为了确保修复是成功的。那就写个测试吧。

### 单元测试框架选型：

**XCTest Or GHUnit**

写测试用例总得有个框架吧，现在比较流行的就属Apple自带的XCTest和第三方的GHUnit。我们来看看他两之间的区别。 XCTest：与Xcode深度集成。而且可以享受Apple后续对XCTest升级的福利。 GHUnit：集成度不如XCTest，安装麻烦。但是有自己的GUI界面。 貌似都各有所长，那么我们来看看Github上的一些知名的开源库都用的是什么测试框架吧。



可以看到清一色的被XCTest刷屏了。也的确，GHUnit的GUI界面对我们来说没有什么特别大的意义。而XCTest血统纯正，背后站着东家Apple。而对于我们的选择也应该是XCTest，应该既然Github上又这么多XCTest的case例子可以参考，对我们的帮助肯定不言而喻。

**OCMock Or OCMockito**

这两个都是用来mock对象，stub方法的。[OCMock](https://github.com/erikdoe/ocmock" \t "_self) 和 [OCMockito](https://github.com/jonreid/OCMockito" \t "_self)个人感觉功能区别不大。他们之间的区别在于使用OCMock的库比OCMockito的库多，而且文档和教程更加丰富。大家可以打开OCMock官网看一下。所以个人选择我选了OCMock作为我们的测试mock工具。

**Expecta Or OCHamcrest**

[Expecta](https://github.com/specta/expecta" \t "_self)和[OCHamcrest](https://github.com/hamcrest/OCHamcrest" \t "_self)这两个都是断言的扩展框架。但是相比而言OCHamcrest更加成熟、稳定，可扩展性好，可自定义断言格式，更加灵活，但是它也有一些局限性，譬如：OCHamcrest不支持superClass验证，我们必须自己去比较，返回一个bool值，然后去判断bool是否为真。很麻烦。更多的可以自行查阅相关资料。所以推荐使用OCHamcrest框架。

**BDD Or Not**

BDD的全称是Behavior Driven Development。也就是行为驱动开发。BDD确实让人眼前一亮。他能将测试语言写成类似自然语言。BDD的理念是你不是在写代码，而是在讲故事。而整个故事是由Given…When…Then组成。我们可以来看看BDD框架Kiwi的一段测试代码:

describe(@"Team", ^{

    context(@"when newly created", ^{

        it(@"has a name", ^{

            id team = [Team team];

            [[team.name should] equal:@"Black Hawks"];

        });

        it(@"has 11 players", ^{

            id team = [Team team];

            [[[team should] have:11] players];

        });

    });

});

这个测试用例就是在说Given a Team,When newly created,it should have a name, and should have 11 players。 的确很清晰，基本不需要注释就能知道在干嘛了。 既然BDD这么好，那么我们比较下BDD框架Kiwi和XCTest + OCMock组合的优劣吧。为什么是XCTest + OCMock而不是XCTest，因为Kiwi自带mock功能，而XCTest没有mock功能。

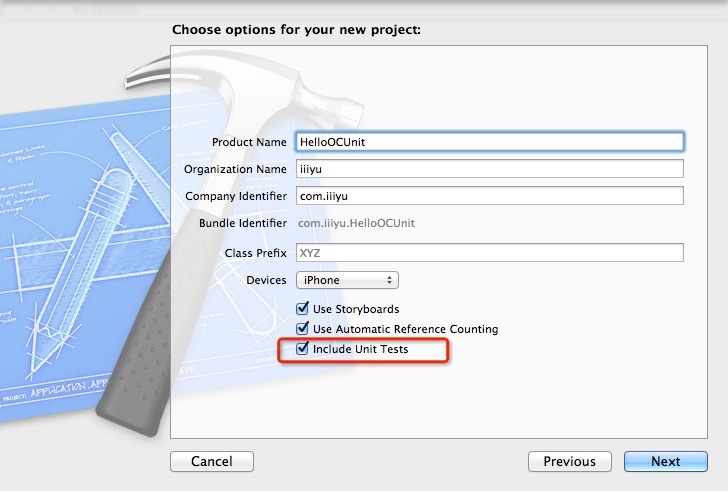


可以看出Kiwi还是蛮诱人的。但是Kiwi的mock功能api远没有OCMock设计的好，尤其是OCMock3推出后，所以可能你会想把Kiwi和OCMock一起用，但是这两个库存在不兼容性。

综述：不推荐使用BDD，推荐使用XCTest＋OCMock＋OCHamcrest框架（有关OCMock、OCHamcrest等资料请自行查阅）。主要的原因是BDD框架hold不住业务的发展，BDD的讲故事理念在业务面前就是老太太的裹脚布，又臭又长！而且BDD需要一定的学习成本，不像XCTest这种类JUnit的对开发者更友好的代码。而且BDD的框架包装过深，可扩展性不高。还有就是BDD的框架普遍太年轻，bug相对较多，版本迭代太快。最最致命的是BDD的框架不能单个case单个跑，一跑所有的case全部跑一边这在平时写case 的时候是非常拖沓的！

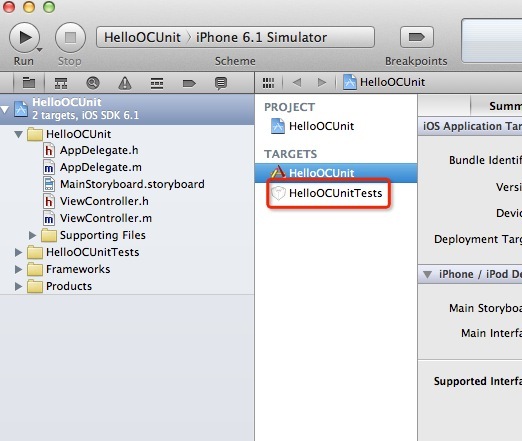
### XC Test框架：

XCTest是Xcode 5中自带的测试框架，它和Xcode 4及之前的SenTestKi一脉相承。幸运的是，从iOS5.0之后Apple便在Xcode中集成了OCUnit单元测试框架。不必安装第三方的其他库就可以使用。最简单的就是创建项目的时候你把单元测试的那个勾点上。xCode就会自动的为你加入一个单元测试的target。快捷键Command + U。就可以运行测试。

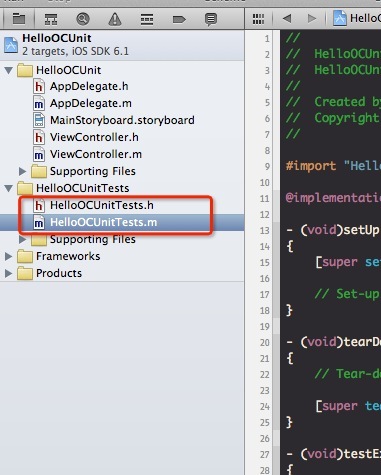


注意：新版Xcode（6）已经默认创建单元测试。

选上Unit Test之后在工程中就可以看到如下图：



并自动创建了一个测试类：



注意：在新版Xcode当中已经没有了头文件，只有.m文件。

### 单元测试用例：

**Given-When-Then分段**

每个case其实都可以分为三步走：

1.mock对象，准备测试数据。

2.调用目标API 。

3.验证输出和行为。

所以我们可以用如下方式将3步分别放入Given-When-Then三个分段中。

-(void)testSumFunction{

//given

int a = 3;int b = 4;

//when

int ret = [Utils sumA:a withB:b];

//then

XCTAssert(ret == 7,@"sum error");

}

测试一段代码执行时间的方法：

- (void)testPerformanceExample {

// This is an example of a performance test case.

[self measureBlock:^{

// Put the code you want to measure the time of here.

int sum ;

for (int i = 0; i < 10000; i++) {

sum += i;

}

}];

}

这样我们一眼扫过去就可以清晰的看出一个case大体上都在干什么。

### 单元测试命名规范：

**OCUnit的测试用例最常用的方法有三个**

1. - (void)setUp：每个test方法执行前调用

2. - (void)tearDown：每个test方法执行后调用

3. - (void)testXXX：命名为XXX的测试方法

1，针对一个类的方法测试建议新建一个对应的测试类

2，测试方法必须以test开头，无返回值、无参数。

3，其他注释及变量、常量等命名规则参考《OBJC编程规范》。

### XCTest断言解释：（OCHamrest断言扩展请查阅资料此不再赘述）

XCTFail(format…) 生成一个失败的测试；

XCTAssertNil(a1, format...)为空判断，a1为空时通过，反之不通过；

XCTAssertNotNil(a1, format…)不为空判断，a1不为空时通过，反之不通过；

XCTAssert(expression, format...)当expression求值为TRUE时通过；

XCTAssertTrue(expression, format...)当expression求值为TRUE时通过；

XCTAssertFalse(expression, format...)当expression求值为False时通过；

XCTAssertEqualObjects(a1, a2, format...)判断相等，[a1 isEqual:a2]值为TRUE时通过，其中一个不为空时，不通过；

XCTAssertNotEqualObjects(a1, a2, format...)判断不等，[a1 isEqual:a2]值为False时通过；

XCTAssertEqual(a1, a2, format...)判断相等（当a1和a2是 C语言标量、结构体或联合体时使用,实际测试发现NSString也可以）；

XCTAssertNotEqual(a1, a2, format...)判断不等（当a1和a2是 C语言标量、结构体或联合体时使用）；

XCTAssertEqualWithAccuracy(a1, a2, accuracy, format...)判断相等，（double或float类型）提供一个误差范围，当在误差范围（+/-accuracy）以内相等时通过测试；

XCTAssertNotEqualWithAccuracy(a1, a2, accuracy, format...) 判断不等，（double或float类型）提供一个误差范围，当在误差范围以内不等时通过测试；

XCTAssertThrows(expression, format...)异常测试，当expression发生异常时通过；反之不通过；（很变态） XCTAssertThrowsSpecific(expression, specificException, format...) 异常测试，当expression发生specificException异常时通过；反之发生其他异常或不发生异常均不通过；

XCTAssertThrowsSpecificNamed(expression, specificException, exception\_name, format...)异常测试，当expression发生具体异常、具体异常名称的异常时通过测试，反之不通过；

XCTAssertNoThrow(expression, format…)异常测试，当expression没有发生异常时通过测试；

XCTAssertNoThrowSpecific(expression, specificException, format...)异常测试，当expression没有发生具体异常、具体异常名称的异常时通过测试，反之不通过；

XCTAssertNoThrowSpecificNamed(expression, specificException, exception\_name, format...)异常测试，当expression没有发生具体异常、具体异常名称的异常时通过测试，反之不通过

特别注意下XCTAssertEqualObjects和XCTAssertEqual。

XCTAssertEqualObjects(a1, a2, format...)的判断条件是[a1 isEqual:a2]是否返回一个YES。

XCTAssertEqual(a1, a2, format...)的判断条件是a1 == a2是否返回一个YES。

对于后者，如果a1和a2都是基本数据类型变量，那么只有a1 == a2才会返回YES。例如下面代码中只有第二行可以通过测试：

// 1.比较基本数据类型变量

XCTAssertEqual(1, 2, @"a1 = a2 shoud be true"); // 无法通过测试

XCTAssertEqual(1, 1, @"a1 = a2 shoud be true"); // 通过测试

但是，如果a1和a2都是指针，那么只有a1和a2指向同一个对象才会返回YES。例如下面的代码中：

// 3.比较NSArray对象

NSArray \*array1 = @[@1];

NSArray \*array2 = @[@1];

NSArray \*array3 = array1;

XCTAssertEqual(array1, array2, @"a1 and a2 should point to the same object"); // 无法通过测试

XCTAssertEqual(array1, array3, @"a1 and a2 should point to the same object"); // 通过测试

再看一个神奇的事情：

// 2.比较NSString对象

NSString \*str1 = @"1";

NSString \*str2 = @"1";

NSString \*str3 = str1;

XCTAssertEqual(str1, str2, @"a1 and a2 should point to the same object"); // 通过测试

XCTAssertEqual(str1, str3, @"a1 and a2 should point to the same object"); // 通过测试

为什么呢？这是由于str1和str2指向同一常量，常量在内存的data段中地址是固定的，所以二者地址相同。

在单元测试中，要熟练使用这几个官方的断言，尽量做到简练、清晰、可读性强。