iOS 单元测试

&UI 测试

2017-01-04

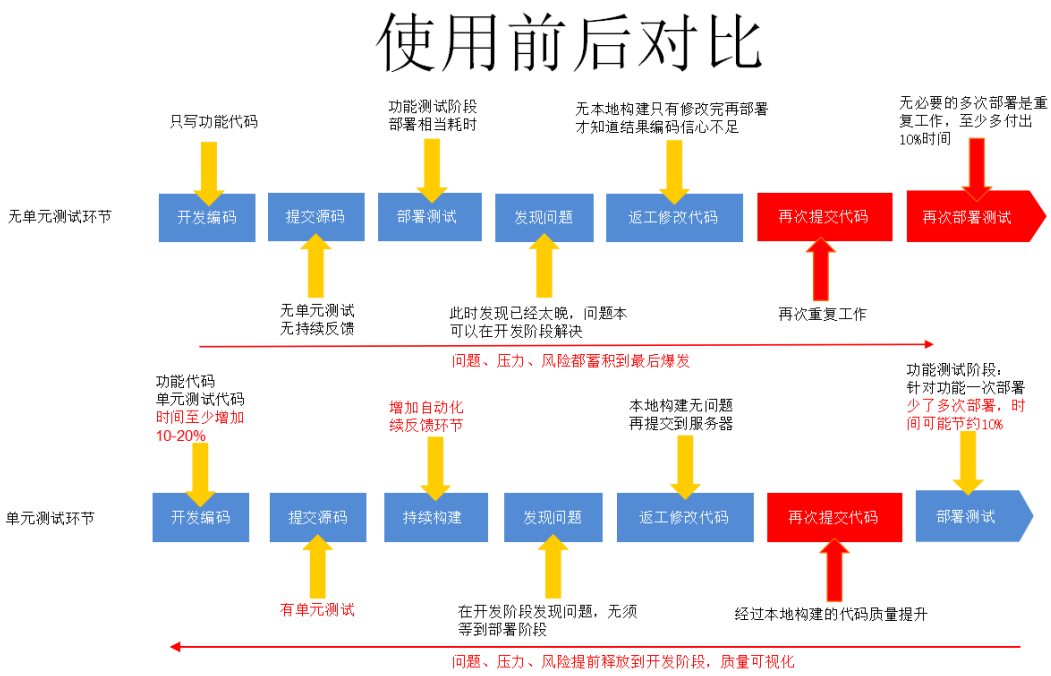
目录

1、单元测试解释

单元测试是开发者编写的一小段代码，用于检验被测代码中的一个很明确的功能是否正确。通常而言，一个单元测试是用于判断某个特定条件（或者场景）下某个特定函数的行为。

执行单元测试，是为了证明某段代码的行为确实和开发者所期望的一致。因此，我们所要测试的是规模很小的、非常独立的功能片段。通过对所有单独部分的行为建立起信心。然后，才能开始测试整个系统

2、单元测试好处



2.1、单元测试的好处：

1. 单元测试使工作完成的更轻松
2. 经过单元测试的代码，质量能够得到保证
3. 单元测试发现的问题很容易定位。
4. 修改代码犯的错，经过单元测试易发现
5. 单元测试可以在早期就发现性能问题
6. 单元测试使你的设计更好
7. 大大减少花在调试上的时间

2.2 不做单元测试的坏处：

1. 代码会暗藏很多缺陷,健壮性不强
2. 系统测试发现的缺陷比较难以定位

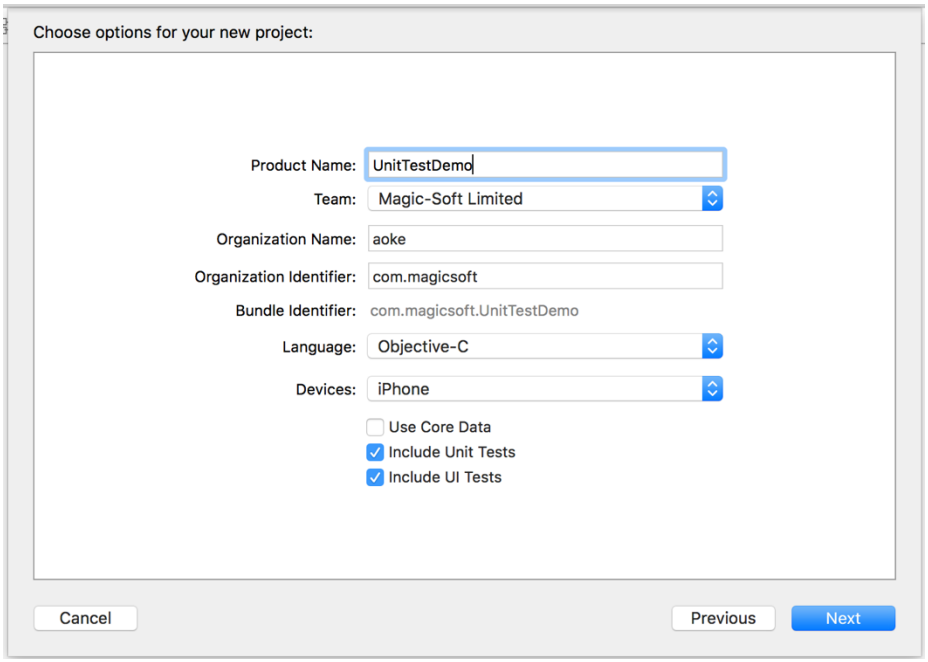
3. 为了修复缺陷而修改代码，很可能会不小心犯错，但是又不能及时发现这些新错误。
4. 性能问题很难定位，性能优化的时间很难控制。

3、前期准备

3.1、创建项目

源码地址：**https://github.com/ly92/UnitTestDemo**

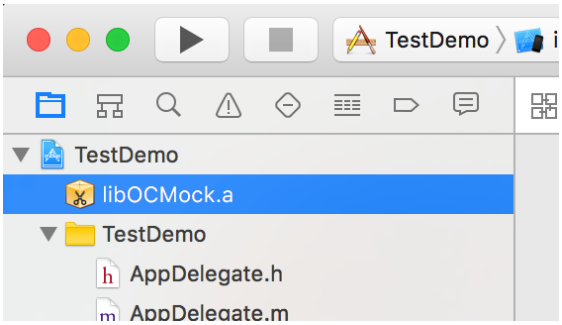
如同普通创建项目一样，在输入项目名以及其他信息后点击选择“Include Unit Tests”和“Include UI Tests”，前者标示单元测试，后者表示 UI 测试。如下所示：



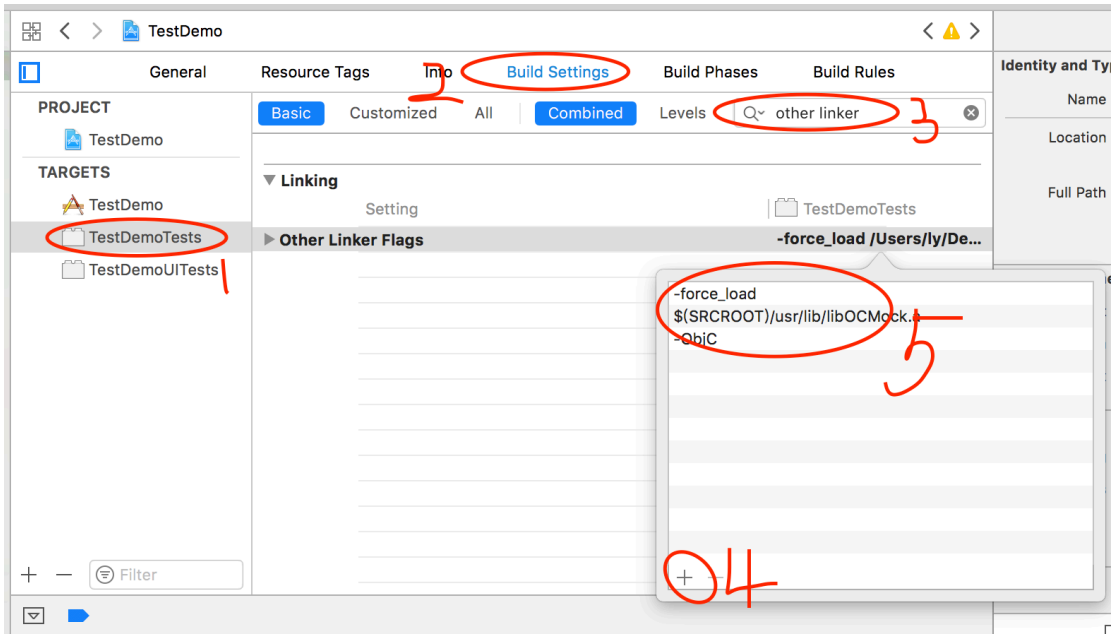
3.2、引入 OCMock

可以前往 OCMock：https://github.com/erikdoe/ocmock 的官方 GitHub 上下载 demo 以及三方库文件，不准备使用 OCMock 的可以忽略。

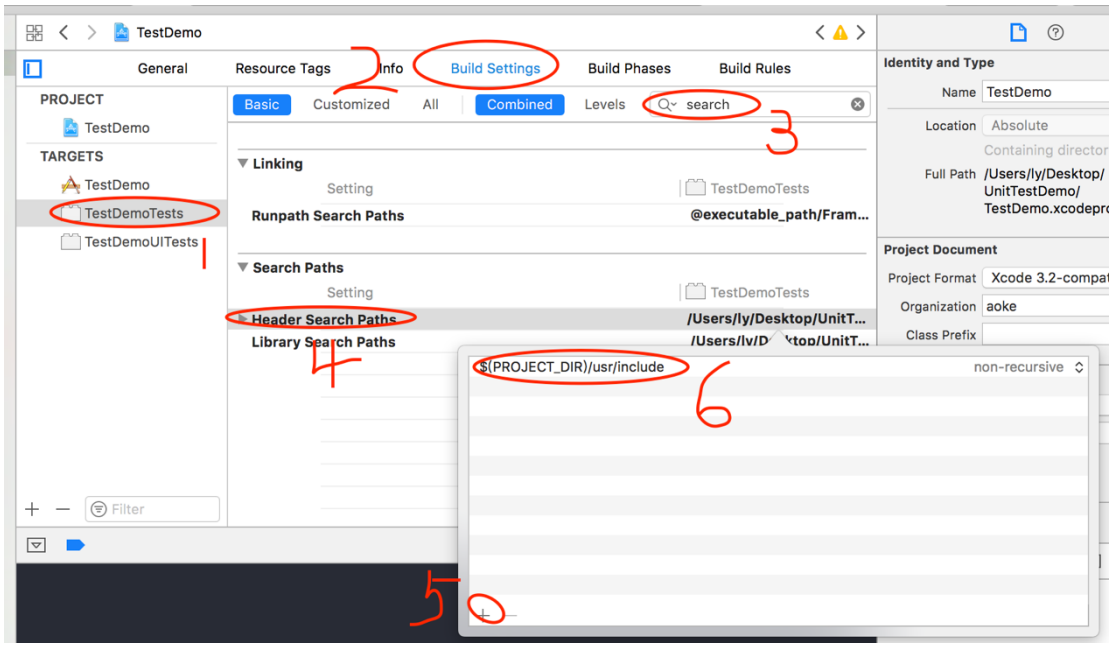
1．下载静态库的包，并引入到工程里的对应的 TestDemo 的 target 里



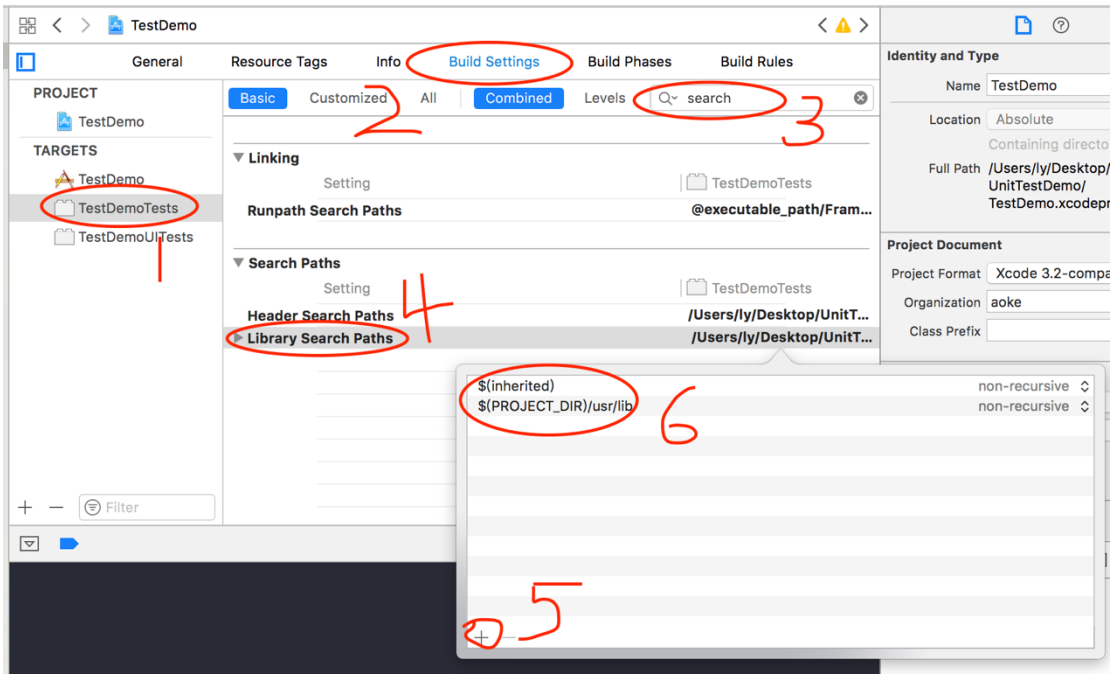
2.配置 TARGETS：TestDemoTests Other linker flags，中间是静态库的绝对路径。$(SRCROOT)/usr/lib/libOCMock.a



3.在 Header Search Paths 中增加 $(PROJECT)/usr/include,里面包含 OCMock 的文件。



4. Libray Search Paths 包含 $(PROJECT)/usr/lib，静态库的相对路径



4、单元测试

4.1、系统方法解释

TestDemoTests.m 是创建项目时选择单元测试自动生成的文件。

```
/**
 *  每个test方法执行之前调用，在此方法中可以定义一些全局属性，类似controller中的viewdidload方法
 */
- (void)setUp {
[super setUp];
//定义
    self.VC = [[ViewController alloc] init];
}

/**
 *  每个test方法执行之后调用,释放测试用例的资源代码，这个方法会每个测试用例执行后调用
 */
- (void)tearDown {
    //结束后释放
    self.VC = nil;

    [super tearDown];
}
/**
```

```
 *   测试用例的例子，注意测试用例一定要test开头
 */
- (void)testExample {
    //测试view是否加载出来
    XCTAssertNotNil(self.VC.view,@"view未成功加载出来");
}

- (void)testPerformanceExample {
    //主要测试代码性能
    [self measureBlock:^{
        //检测在此block中代码的性能
    }];
}
```

4.2、函数测试

在ViewController.h中定义函数并在ViewController.m实现：

```
- (int)getNum;
- (int)getNum{
    return 100;
}
```

在里面TestDemoTests.m里面定义函数，必须以test开头，如果返回值不为100则测试失败

```
//必须以test开头的函数
- (void)testMyFuc{
    int result = self.VC.getNum;
    XCTAssertEqual(result, 100,@"测试普通函数不通过");
}
```

4.3、测试图片处理

```
//测试图片处理大小的性能，以及处理成功与否
- (void)testImageResize{
    UIImage *image = [UIImage imageNamed:@"icon1.jpeg"];
    [self measureBlock:^{ //测试处理图片代码的性能
        // Put the code you want to measure the time of here.
        UIImage *resizedImage = [self imageWithImage:image
scaledToSize:CGSizeMake(100, 100)];
        XCTAssertNotNil(resizedImage, @"缩放后图片不应为nil");
        CGFloat resizedWidth = resizedImage.size.width;
        CGFloat resizedHeight = resizedImage.size.height;
        XCTAssertTrue(resizedWidth == 100 && resizedHeight == 100, @"缩放后尺
寸");
    }];
}
```

```
- (UIImage *)imageWithImage:(UIImage *)image scaledToSize:(CGSize)newSize {
    UIGraphicsBeginImageContext(newSize);
    [image drawInRect:CGRectMake(0, 0, newSize.width, newSize.height)];
    UIImage *newImage = UIGraphicsGetImageFromCurrentImageContext();
    UIGraphicsEndImageContext();
    return newImage;
}
```

4.4、异步测试

异步测试，有三种方式（expectationWithDescription，expectationForPredicate和expectationForNotification）

4.4.1、expectationWithDescription

```
// 测试接口(异步测试)使用expectationWithDescription
- (void)testAsynchronousURLConnection {
    [self measureBlock:^{
        NSLog(@"testAsynchronousURLConnection");
        //预先定义
        XCTestExpectation *expectation = [self expectationWithDescription:@"GET Baidu"];
        //测试地址
        NSURL *url = [NSURL URLWithString:@"https://www.baidu.com/"];

        NSURLSession *session = [NSURLSession sharedSession];
        NSURLSessionDataTask *task = [session dataTaskWithURL:url completionHandler:^(NSData * _Nullable data, NSURLResponse * _Nullable response, NSError * _Nullable error) {
            //            NSLog(@"data : %@", data);
            // XCTestExpectation条件已满足，接下来的测试代码可以开始执行了。
            [expectation fulfill];
            XCTAssertNotNil(data, @"返回数据不应非nil");
            XCTAssertNil(error, @"error应该为nil");
            if (nil != response) {
                NSHTTPURLResponse *httpResponse = (NSHTTPURLResponse *)response;
                XCTAssertEqual(httpResponse.statusCode, 200, @"HTTPResponse的状态码应该是200");
                XCTAssertEqual(httpResponse.URL.absoluteString, url.absoluteString, @"HTTPResponse的URL应该与请求的URL一致");
                //                XCTAssertEqual(httpResponse.MIMEType, @"text/html", @"HTTPResponse的内容应该是text/html");
            } else {
```

```
                XCTFail(@"返回内容不是NSHTTPURLResponse类型");
            }
        }];
        [task resume];

        // 超时后执行
        [self waitForExpectationsWithTimeout:10.0 handler:^(NSError * _Nullable
error) {
            [task cancel];
        }];
    }];
}
```

4.4.2、expectationForPredicate

```
//异步测试，使用expectationForPredicate,设置一个期望，在规定时间内满足期
望则测试通过
- (void)testAsynExampleWithExpectationForPredicate {

    XCTAssertNil(self.imageView.image);

    self.imageView.image = [UIImage imageNamed:@"icon2"];

    //设置一个期望
    NSPredicate *predicate = [NSPredicate predicateWithFormat:@"image != nil"];
    //若在规定时间内满足期望，则测试成功
    [self expectationForPredicate:predicate
              evaluatedWithObject:self.imageView
                          handler:nil];

    // 超时后执行
    [self waitForExpectationsWithTimeout:10.0 handler:^(NSError * _Nullable
error) {

    }];
}
```

4.4.3、expectationForNotification

```
//异步测试，使用expectationForNotification,该方法监听一个通知,如果在规定时
间内正确收到通知则测试通过
- (void)testAsynExampleWithExpectationForNotification {

    //监听通知，在规定时间内受到通知，则测试通过
```

```
    [self expectationForNotification:@"监听通知的名称测试" object:nil
handler:^BOOL(NSNotification * _Nonnull notification) {
        NSLog(@"请求成功");
        //做后续处理
        return YES;
    }];

    //下面2个地址可以查看测试通过与不通过的区别
    //测试通过
    NSURL *url = [NSURL URLWithString:@"https://www.baidu.com/"];
    //测试失败
//      NSURL *url = [NSURL URLWithString:@"www.baidu.com/"];

    NSURLSession *session = [NSURLSession sharedSession];
    NSURLSessionDataTask *task = [session dataTaskWithURL:url
completionHandler:^(NSData * _Nullable data, NSURLResponse * _Nullable
response, NSError * _Nullable error) {

        if (data && !error && response) {
            //发送通知
            [[NSNotificationCenter defaultCenter]postNotificationName:@"监听
通知的名称测试" object:nil];
        }
    }];
    [task resume];
    //设置延迟多少秒后，如果没有满足测试条件就报错
    [self waitForExpectationsWithTimeout:10.0 handler:^(NSError * _Nullable
error) {
        [task cancel];
    }];
}
```

4.5、测试私有属性和私有方法

在ViewController中定义一个私有属性和一个私有方法

```
@interface ViewController ()
@property (strong, nonatomic) IBOutlet UITableView *tv;
@property (nonatomic, copy) NSString *privateString;
@end
//私有方法
- (NSString *)privateFuc{
    return @"123456";
}
```

在TestDemoTests中声明ViewController的分类

```
//测试ViewController的私有方法-通过分类的方式
@interface ViewController (TestDemoTests)
- (NSString *)privateFuc;
@property (nonatomic, copy) NSString *privateString;
@end
```

然后在测试方法中直接调用即可

```
- (void)testExample {
    //测试私有方法
    XCTAssertEqualObjects(self.VC.privateFuc, @"123456",@"");
    //测试私有属性
    XCTAssertEqualObjects(self.VC.privateString, @"987654321",@"");
}
```

5、单元测试-OCMock

当我们写单元测试的时候，不可避免的要去尽可能少的实例化一些具体的组件来保持测试既短又快。而且保持单元的隔离。在现代的面向对象系统中，测试的组件很可能会有几个依赖的对象。我们用mock来替代实例化具体的依赖class。mock是在测试中的一个伪造的有预定义行为的具体对象的替身对象。被测试的组件不知道其中的差异！你的组件是在一个更大的系统中被设计的，你可以很有信心的用mock来测试你的组件。

5.1、准备

5.1.1、准备模型-PersonModel

在target：TestDemo中新加NSObject类型文件PersonModel

PersonModel.h

```
@interface PersonModel : NSObject
@property (nonatomic, copy) NSString *name;
@property (nonatomic, copy) NSString *gender;
- (NSString *)getPersonName;
- (NSString *)changeName:(NSString *)newName;
@end
```

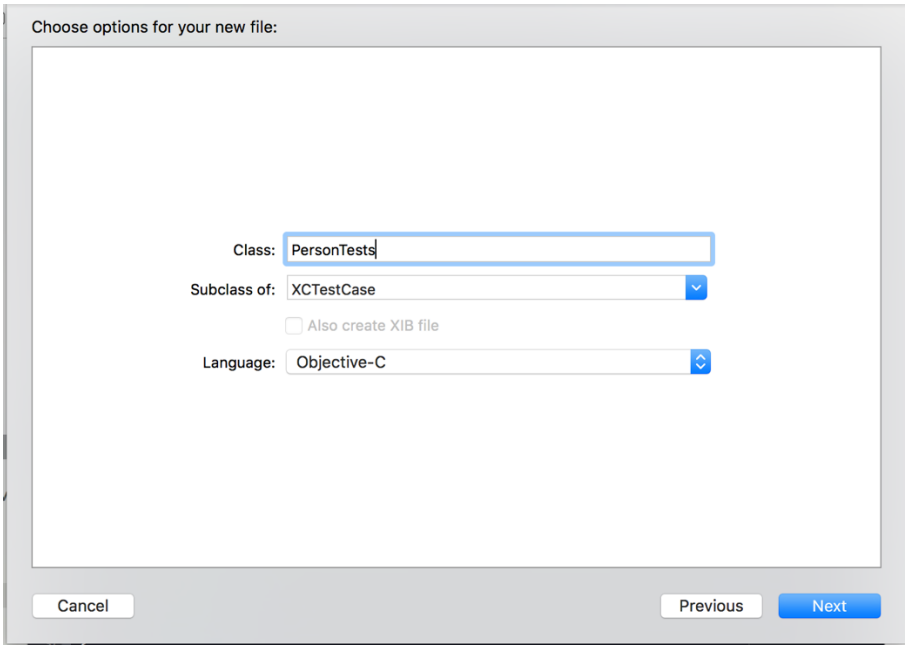
PersonModel.m

```
@implementation PersonModel
- (instancetype)init{
    if (self = [super init]){
        self.name = @"liyong";
        self.gender = @"男";
    }
    return self;
}
```

```
- (NSString *)getPersonName{
    PersonModel *person = [[PersonModel alloc] init];
    return person.name;
}
- (NSString *)changeName:(NSString *)newName{
    PersonModel *person = [[PersonModel alloc] init];
    person.name = newName;

    return person.name;
}
@end
```

5.1.2、新建单元测试文件

TestDemoTests.m 是创建项目时选择单元测试自动生成的文件，增加测试代码时避免不了需要新建文件，继承 XCTestCase 创建 PersonTests.m，得到的是 PersonTests.h 和 PersonTests.m 两个文件，只需将 PersonTests.h 的声明代码迁移到 PersonTests.m 中即可删除 PersonTests.h 文件。如果有需要可将系统自动生成的函数- (void)setUp 和- (void)tearDown 复制到 PersonTests.m 文件中。



5.2、测试没有参数的函数

```
//没有参数的方法
- (void)testGetName{
    PersonModel *person = [[PersonModel alloc] init];

    //创建一个mock对象
    id mockClass = OCMClassMock([PersonModel class]);
    //可以给这个mock对象的方法设置预设的参数和返回值
```

```
    OCMStub([mockClass getPersonName]).andReturn(@"liyong");

    //用这个预设的值和实际的值进行比较是否相等
    XCTAssertEqualObjects([mockClass getPersonName], [person getPersonName],
@"值相等");
}
```

5.3、测试有参数的函数

```
//有参数的方法
- (void)testCahngeName{

    PersonModel *person = [[PersonModel alloc] init];

    id mockClass = OCMClassMock([PersonModel class]);
    //[OCMArg any]是指任意参数,下面调用方法时传的参数必须与此处的参数
一样才会返回设定的值
    OCMStub([mockClass changeName:[OCMArg any]]).andReturn(@"wss");

    //验证getPersonName方法有没有被调用，如果没有调用则抛出异常
//      OCMVerify([mockClass getPersonName]);

    XCTAssertEqualObjects([mockClass changeName:[OCMArg any]], [person
changeName:@"wss"],@"值相等");
}
```

5.4、测试有参数的函数调用时传的参数

```
//检查参数
- (void)testArgument{
    id mockClass = OCMClassMock([PersonModel class]);
    //检查参数
    OCMStub([mockClass changeName:[OCMArg checkWithBlock:^BOOL(id obj) {
        //判断参数是否为NSString类型
        if ([obj isKindOfClass:[NSString class]]){
        }else{
            //提示错误
//              XCTAssertFalse(obj);

            obj = @"456";
        }
        NSLog(@"----------------%@",obj);
        return YES;
    }]]);
```

```
    [mockClass changeName:@"123"];
    [mockClass changeName:[OCMArg any]];
}
```

6、单元测试-table

继承XCTestCase创建TableTests.m文件

6.1、table 数据源函数返回行数

```
//测试table数据源函数返回行数
- (void)testControllerReturnsCorrectNumberOfRows
{
    XCTAssertEqual(3, [self.VC tableView:self.VC.tableView
numberOfRowsInSection:0],@"此处返回得到的行数错误");
}
```

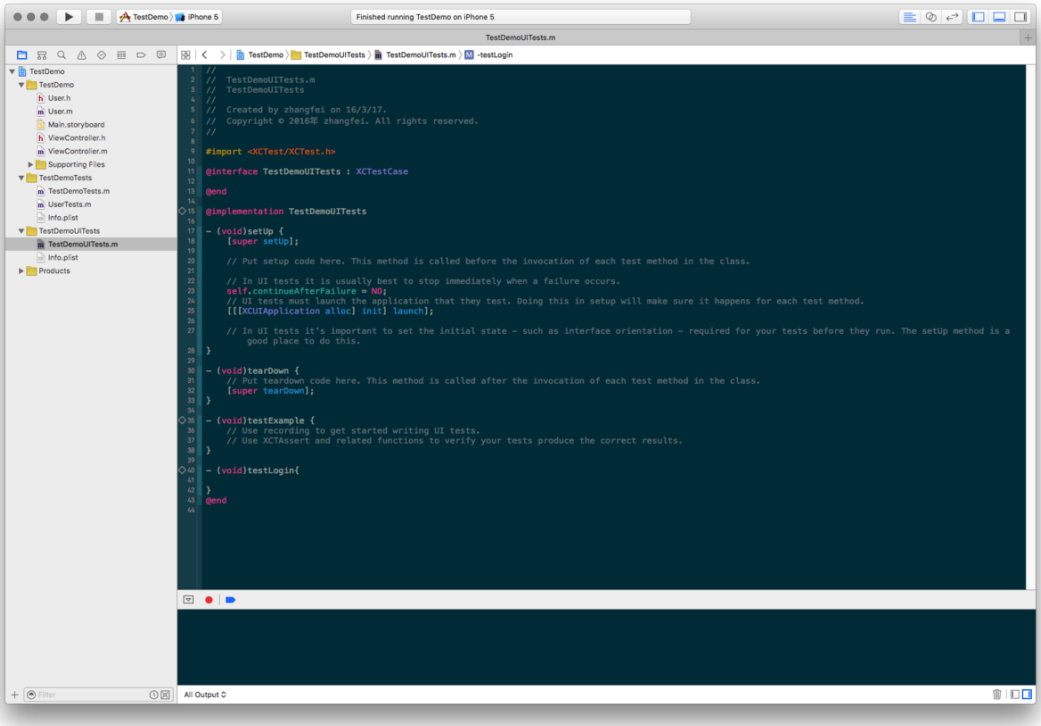
6.2、table 数据源函数返回 cell

```
//测试table数据源函数返回cell
- (void)testControllerSetsUpCellCorrectly
{
    id mockTable = OCMClassMock([UITableView class]);
    [[[mockTable expect] andReturn:nil]
dequeueReusableCellWithIdentifier:@"HappyNewYear"];

    NSIndexPath *indexPath = [NSIndexPath indexPathForRow:2 inSection:0];

    UITableViewCell *cell = [self.VC tableView:mockTable
cellForRowAtIndexPath:indexPath];

    XCTAssertNotNil(cell, @"此处应该返回一个cell");
    XCTAssertEqualObjects(@"x-2", cell.textLabel.text, @"返回的字符串错误");

    [mockTable verify];
}
```

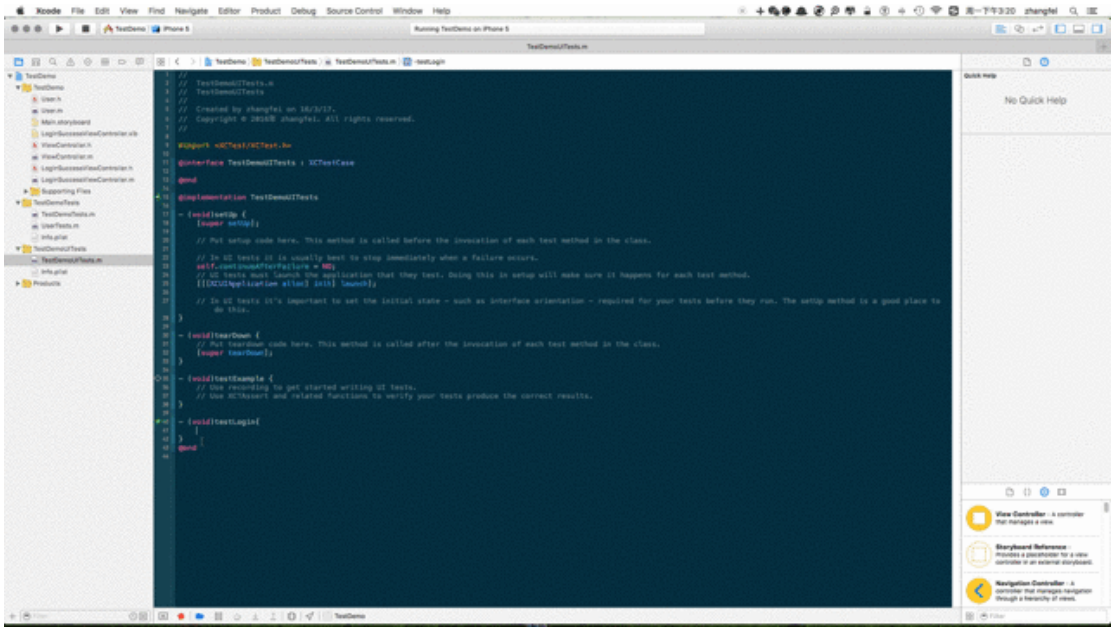
7、UI 测试

TestDemoUITests.m文件中写一个方法testLogin作为测试登录流程操作的UI测试方法。然后把光标放在方法体内，然后点击红色的那个录制按钮，如下：

7.1、测试登录-普通点击事件



下面这个是.gif拖拽到本地可查看动画



```
//检测登录
- (void)testLogin{

    //首先从tabbars找到“登录”然后点击
    [[[XCUIApplication alloc] init].tabBars.buttons[@"登录"] tap];
```

```
    //获取app
    XCUIApplication *app = [[XCUIApplication alloc] init];
    //在当前页面寻找与“accountTF”有关系的输入框，我测试时发现placeholder写为“accountTF”就可以寻找到
    XCUIElement *textField = app.textFields[@"accountTF"];
    [textField tap];//获取焦点成为第一响应者，否则会报“元素（此textField）未调起键盘”错误
    [textField typeText:@"liyong"];//为此textField键入字符串

    XCUIElement *textField2 = app.textFields[@"passwordTF"];
    [textField2 tap];
    [textField2 typeText:@"123456"];

    for (int i = 0; i < 2; i ++) {//n次点击登陆按钮
        [app.buttons[@"login"] tap];//login标示的button点击
    }

    //如果页面title为success则表示登录成功，也可用其他判断方式
    XCTAssertEqualObjects(app.navigationBars.element.identifier, @"success");
}
```

7.2、table 下拉上拉

```
//列表下拉以及上拉测试
- (void)testRefresh{
    //获取app
    XCUIApplication *app = [[XCUIApplication alloc] init];
    //点击tabbar中“列表”这个
    [app.tabBars.buttons[@"列表"] tap];
    //获取当前页面的tabble（此页面只有一个table，代码自动生成的）
    XCUIElement *table = [[[[[[[[app
childrenMatchingType:XCUIElementTypeWindow] elementBoundByIndex:0]
childrenMatchingType:XCUIElementTypeOther].element
childrenMatchingType:XCUIElementTypeOther].element
childrenMatchingType:XCUIElementTypeOther].element
childrenMatchingType:XCUIElementTypeOther].element
childrenMatchingType:XCUIElementTypeOther].element
childrenMatchingType:XCUIElementTypeOther].element
childrenMatchingType:XCUIElementTypeTable].element;

    //可通过循环上拉或者下拉无数次
    [table swipeDown];//下拉
    [table swipeUp];//上拉
}
```

7.3、tablecell 点击以及返回

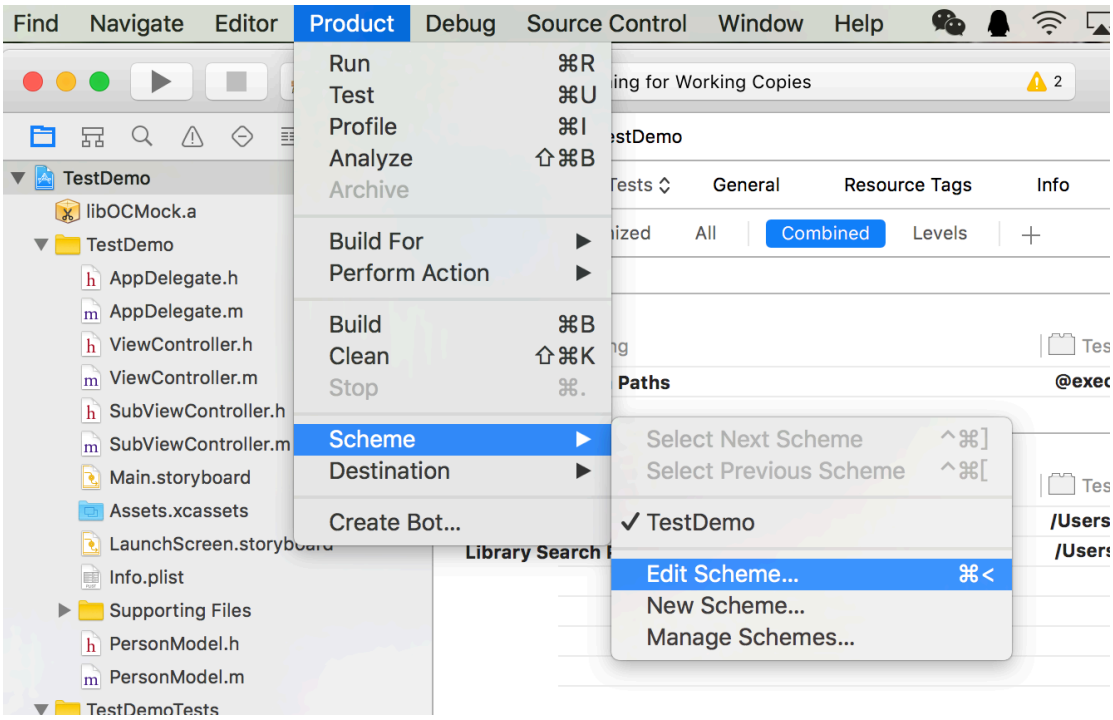
```
//tablecell点击以及返回
- (void)testCellClick{
    //获取app
    XCUIApplication *app = [[XCUIApplication alloc] init];
    //点击tabbar中“列表”这个
    [app.tabBars.buttons[@"列表"] tap];
    //在当前页面获取table的cell队列
    XCUIElementQuery *tablesQuery = app.tables;
    //点击了第一个cell，此cell有一个标示为“x-x”
    [[[tablesQuery childrenMatchingType:XCUIElementTypeCell]
elementBoundByIndex:0].staticTexts[@"x-x"] tap];
    //在“login”为title的页面中点击了导航栏中“table”按钮---login页面为点击cell
进入的页面，table是导航栏左侧按钮，点击返回列表页面
    XCUIElement *tableButton = app.navigationBars[@"login"].buttons[@"table"];
    [tableButton tap];//点击返回

    [[[tablesQuery childrenMatchingType:XCUIElementTypeCell]
elementBoundByIndex:7].staticTexts[@"x-x"] tap];
    [tableButton tap];
    [[[tablesQuery childrenMatchingType:XCUIElementTypeCell]
elementBoundByIndex:9].staticTexts[@"x-x"] tap];
    //点击login页面中“back”按钮返回，
    [app.buttons[@"back"] tap];
}

/**
     *在执行过程中如果只进行多次通过点击back来返回则可以使用
     *XCUIElement *backButton = app.buttons[@"back"];
     *后面直接用
     *[backButton tap];

     例如：

     XCUIElement *xXStaticText = [[app.tables
childrenMatchingType:XCUIElementTypeCell]
elementBoundByIndex:0].staticTexts[@"x-x"];
     [xXStaticText tap];
     XCUIElement *backButton = app.buttons[@"back"];
     [backButton tap];
     [xXStaticText tap];
     [backButton tap];
     [xXStaticText tap];
```

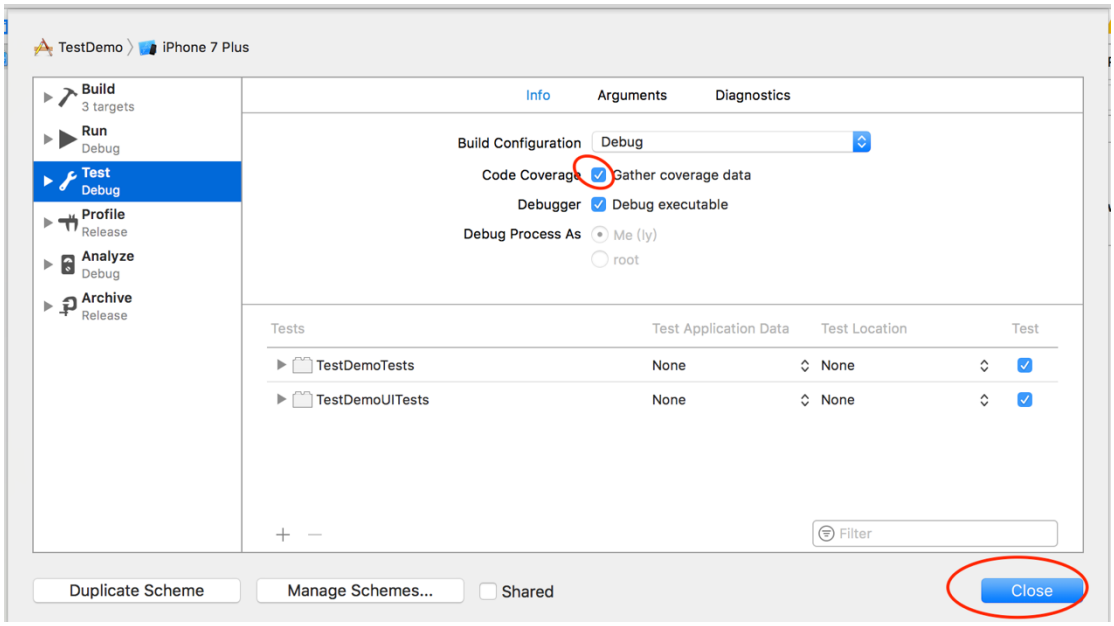
```
[backButton tap];
[xXStaticText tap];
[backButton tap];
[xXStaticText tap];
[backButton tap];
[xXStaticText tap];
[backButton tap];
*/
```

8、代码覆盖率

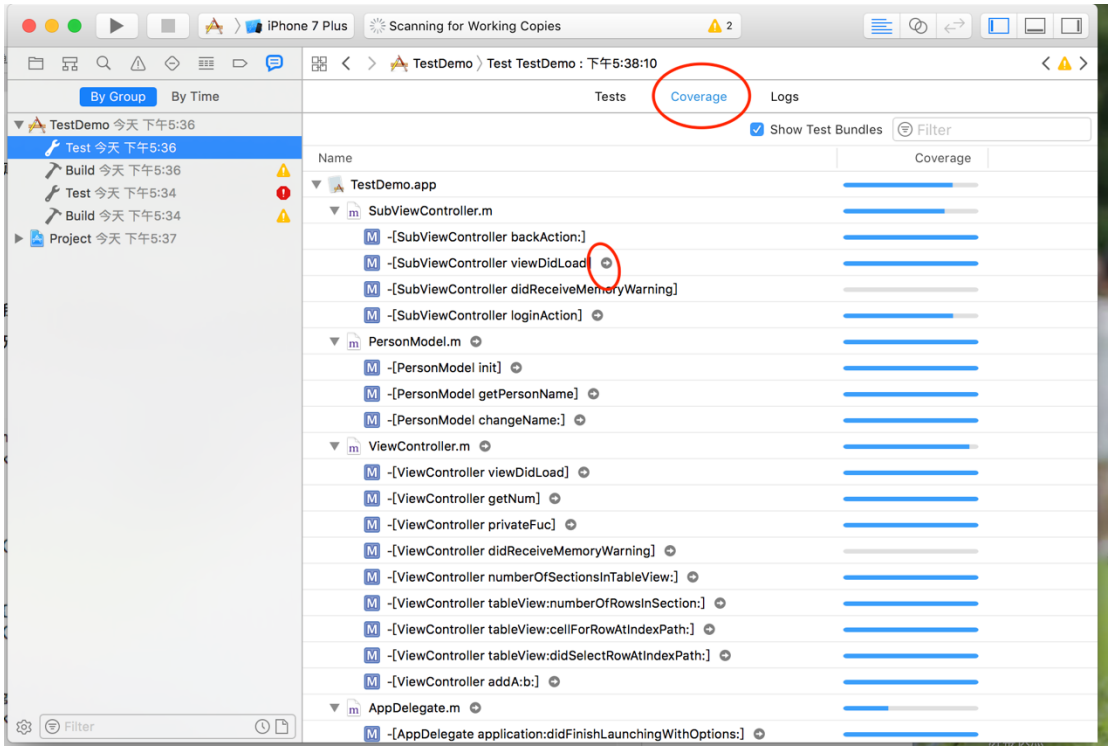
Code coverage 是一个计算你的单元测试覆盖率的工具。高水平的覆盖给你的单元测试带来信心，也表明你的应用被彻底的测试过了。你可能写了几千个单元测试，但如果覆盖率不高，那么你写的这套测试可能价值也不大。

在运行测试之前，我们必须先确认 code coverage 是否被打开了，写代码时，默认是关闭的。所以你需要编辑一下你的测试 scheme，把它打开。确保"Gather coverage data"是被选中的，然后点击关闭按钮，运行测试的 target. 我们希望刚刚创建的测试用例能够顺利通过。





测试通过后，你就能知道 checkWord 这个方法，至少有一条路径是对的。但你不知道的是，还多多少没有被测试到。这就是code coverage这个工具的好处。当你打开code coverage tab后，你可以清楚的看到测试的覆盖情况。他们按找 target, file, function 进行了自动分组。打开Xcode左边窗口的Report Navigator面板，选中你刚运行的测试。然后在tab中选中 Coverage。这会展示一个你的类、方法的列表，并展示每个的测试覆盖情况。双击方法的名字，Xcode会打开类的代码，并且看到code coverage的情况。



鼠标移动到方法的代码右侧时会展示代码的执行次数。

```
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}
- (IBAction)loginAction {

    NSString *account = self.accountTF.text;
    NSString *pwd = self.passwordTF.text;

    if ([account isEqualToString:@"liyong"] && [pwd isEqualToString:@"123456"]){

        self.navigationItem.title = @"success";
    }else{
        self.navigationItem.title = @"fail";
    }

}

/*
#pragma mark - Navigation
```

附件 1：

单元测试准则：

1. 保持单元测试小巧，快速
理论上，任何代码提交前都应该完整跑一遍所有测试套件. 保持测试代码执行迅捷能够缩短迭代开发周期.
2. 单元测试应该是全自动且无交互
测试套件通常是定期执行的，执行过程必须完全自动化才有意义. 需要人工检查输出结果的测试不是一个好的单元测试.
3. 让单元测试很容易跑起来
对开发环境进行配置，最好是敲条命令或是点个按钮就能把单个测试用例或测试套件跑起来.
4. 对测试进行评估
对执行的测试进行覆盖率分析，得到精确的代码执行覆盖率，并调查哪些代码未被执行.
5. 立即修正失败的测试
每个开发人员在提交前都应该保证新的测试用例执行成功，当有代码提交时，现有测试用例也都能跑通.
如果一个定期执行的测试用例执行失败，整个团队应该放下手上的工作优先解决这个问题.
6. 把测试维持在单元级别
单元测试即类 (Class) 的测试. 一个 "测试类" 应该只对应于一个 "被测类"，并且 "被测类" 的行为应该被隔离测试. 必须谨慎避免使用单元测试框架来测试整个程序的工作流，这样的测试既低效又难维护. 工作流测试 (译注: 指跨模块/类的数据流测试) 有它自己的地盘，但它绝不是单元测试，必须单独建立和执行.
7. 由简入繁
最简单的测试也远远胜过完全没有测试. 一个简单的 "测试类" 会促使建立 "被测类" 基本的测试骨架，可以对构建环境，单元测试环境，执行环境以及覆盖率分析工具等有效性进行检查，同时也可以证明 "被测类" 能够被整合和调用.
下面便是单元测试版的 Hello, world! :

```
void testDefaultConstruction()
{
Foo foo = new Foo();
assertNotNull(foo);
}
```

8. 保持测试的独立性
为了保证测试稳定可靠且便于维护，测试用例之间决不能有相互依赖，也不能依赖执行的先后次序.
9. Keep tests close to the class being tested
[译注: 有意翻译该规则，个人认为本条规则值得商榷，大部分 C++, Objective-C 和 Python 库均把测试代码从功能代码目录中独立出来，通常是创建一个和 src 目录同级的 tests 目录，被测模块/类名之前也常常 不加 Test 前缀. 这么做保

证功能代码和测试代码隔离，目录结构清晰，并且发布源码的时候更容易排除测试用例.]
If the class to test is Foo the test class should be called FooTest (not TestFoo) and kept in the same package (directory) as Foo. Keeping test classes in separate directory trees makes them harder to access and maintain.
Make sure the build environment is configured so that the test classes doesn't make its way into production libraries or executables.
10. 合理的命名测试用例
确保每个方法只测试 "被测类" 的一个明确特性，并相应的命名测试方法. 典型的命名俗定是 test[what]，比如 testSaveAs(), testAddListener(), testDeleteProperty() 等.
11. 只测公有接口
单元测试可以被定义为 通过类的公有 API 对类进行测试. 一些测试工具允许测试一个类的私有成员，但这种做法应该避免，它让测试变得繁琐而且更难维护. 如果有私有成员确实需要进行直接测试，可以考虑把它重构到工具类的公有方法中. 但要注意这么做是为了改善设计，而不是帮助测试.
12. 看成是黑盒
站在第三方使用者的角度，测试一个类是否满足规定的需求. 并设法让它出问题.
13. 看成是白盒
毕竟被测试类是程序员自写自测的，应该在最复杂的逻辑部分多花些精力测试.
14. 芝麻函数也要测试
通常建议所有重要的函数都应该被测试到，一些芝麻方法比如简单的 setter 和 getter 都可以忽略. 但是仍然有充分的理由支持测试芝麻函数:
芝麻 很难定义. 对于不同的人有不同的理解.
从黑盒测试的观点看，是无法知道哪些代码是芝麻级别的.
即便是再芝麻的函数，也可能包含错误，通常是 "复制粘贴" 代码的后果:
```
private double weight_;
private double x_, y_;
public void setWeight(int weight)
{
   weight = weight_;    // error
}
public double getX()
{
   return x_;
}
public double getY()
{
   return x_;    // error
}
```
因此建议测试所有方法. 毕竟芝麻用例也容易测试.
15. 先关注执行覆盖率

区别对待 执行覆盖率 和 实际测试覆盖率. 测试的最初目标应该是确保较高的执行覆盖率. 这样能保证代码在 少量 参数值输入时能执行成功. 一旦执行覆盖率就绪, 就应该开始改进测试覆盖率了. 注意, 实际的测试覆盖率很难衡量 (而且往往趋近于 0%).
思考以下公有方法:
void setLength(double length);
调用 setLength(1.0) 你可能会得到 100% 的执行覆盖率. 但要达到 100% 的实际测试覆盖率, 有多少个 double 浮点数这个方法就必须被调用多少次, 并且要一一验证行为的正确性. 这无疑是不可能的任务.
16. 覆盖边界值
确保参数边界值均被覆盖. 对于数字, 测试负数, 0, 正数, 最小值, 最大值, NaN (非数字), 无穷大等. 对于字符串, 测试空字符串, 单字符, 非 ASCII 字符串, 多字节字符串等. 对于集合类型, 测试空, 1, 第一个, 最后一个等. 对于日期, 测试 1 月 1 号, 2 月 29 号, 12 月 31 号等. 被测试的类本身也会暗示一些特定情况下的边界值. 要点是尽可能彻底的测试这些边界值, 因为它们都是主要 "疑犯".
17. 提供一个随机值生成器
当边界值都覆盖了, 另一个能进一步改善测试覆盖率的简单方法就是生成随机参数, 这样每次执行测试都会有不同的输入.
想要做到这点, 需要提供一个用来生成基本类型 (如: 浮点数, 整型, 字符串, 日期等) 随机值的工具类. 生成器应该覆盖各种类型的所有取值范围.
如果测试时间比较短, 可以考虑再裹上一层循环, 覆盖尽可能多的输入组合. 下面的例子是验证两次转换 little endian 和 big endian 字节序后是否返回原值. 由于测试过程很快, 可以让它跑上个一百万次.

```
void testByteSwapper()
{
   for (int i = 0; i < 1000000; i++) {
      double v0 = Random.getDouble();
      double v1 = ByteSwapper.swap(v0);
      double v2 = ByteSwapper.swap(v1);
      assertEquals(v0, v2);
   }
}
```

18. 每个特性只测一次
在测试模式下, 有时会情不自禁的滥用断言. 这种做法会导致维护更困难, 需要极力避免. 仅对测试方法名指示的特性进行明确测试.
因为对于一般性代码而言, 保证测试代码尽可能少是一个重要目标.
19. 使用显式断言
应该总是优先使用 assertEquals(a, b) 而不是 assertTrue(a == b), 因为前者会给出更有意义的测试失败信息. 在事先不确定输入值的情况下, 这条规则尤为重要, 比如之前使用随机参数值组合的例子.
20. 提供反向测试
反向测试是指刻意编写问题代码, 来验证鲁棒性和能否正确的处理错误.
假设如下方法的参数如果传进去的是负数, 会立马抛出异常:
void setLength(double length) throws IllegalArgumentExcepti

可以用下面的方法来测试这个特例是否被正确处理:

```
try {
   setLength(-1.0);
   fail();   // If we get here, something went wrong
}
catch (IllegalArgumentException exception) {
   // If we get here, all is fine
}
```

21. 代码设计时谨记测试

编写和维护单元测试的代价是很高的，减少代码中的公有接口和循环复杂度是降低成本，使高覆盖率测试代码更易于编写和维护的有效方法.

一些建议:

使类成员常量化，在构造函数中进行初始化. 减少 setter 方法的数量.

限制过度使用继承和公有虚函数.

通过使用友元类 (C++) 或包作用域 (Java) 来减少公有接口.

避免不必要的逻辑分支.

在逻辑分支中编写尽可能少的代码.

在公有和私有接口中尽量多用异常和断言验证参数参数的有效性.

限制使用快捷函数. 对于黑箱而言，所有方法都必须一视同仁的进行测试. 思考以下简短的例子:

```
public void scale(double x0, double y0, double scaleFactor)
{
   // scaling logic
}

public void scale(double x0, double y0)
{
   scale(x0, y0, 1.0);
}
```

删除后者可以简化测试，但用户代码的工作量也将略微增加.

22. 不要访问预设的外部资源

单元测试代码不应该假定外部的执行环境，以便在任何时候/任何地方都能执行. 为了向测试提供必需的资源，这些资源应该由测试本身提供.

比如一个解析某类型文件的类，可以把文件内容嵌入到测试代码里，在测试的时候写入到临时文件，测试结束再删除，而不是从预定的地址直接读取.

23. 权衡测试成本

不写单元测试的代价很高，但是写单元测试的代价同样很高. 要在这两者之间做适当的权衡，如果用执行覆盖率来衡量，业界标准通常在 80% 左右.

很典型的，读写外部资源的错误处理和异常处理就很难达到百分百的执行覆盖率. 模拟数据库在事务处理到一半时发生故障并不是办不到，但相对于进行大范围的代码审查，代价可能太大了.

24. 安排测试优先次序

单元测试是典型的自底向上过程，如果没有足够的资源测试一个系统的所有模块，就应该先把重点放在较底层的模块.

25. 测试代码要考虑错误处理

考虑下面的这个例子:

```
Handle handle = manager.getHandle();
assertNotNull(handle);
String handleName = handle.getName();
assertEquals(handleName, "handle-01");
```

如果第一个断言失败, 后续语句会导致代码崩溃, 剩下的测试都无法执行. 任何时候都要为测试失败做好准备, 避免单个失败的测试项中断整个测试套件的执行. 上面的例子可以重写成:

```
Handle handle = manager.getHandle();
assertNotNull(handle);
if (handle == null) return;
String handleName = handle.getName();
assertEquals(handleName, "handle-01");
```

26. 写测试用例重现 bug

每上报一个 bug, 都要写一个测试用例来重现这个 bug (即无法通过测试), 并用它作为成功修正代码的检验标准.

27. 了解局限

单元测试永远无法证明代码的正确性!!

一个跑失败的测试可能表明代码有错误, 但一个跑成功的测试什么也证明不了.

单元测试最有效的使用场合是在一个较低的层级验证并文档化需求, 以及 回归测试: 开发或重构代码时，不会破坏已有功能的正确性.

附件 2:

XCTest 测试-名词解释
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, 判断的是变量的地址，如果地址相同则返回 TRUE，否则返回 NO）；
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