Android 单元测试

1.Building Local Unit Tests

1.1.简介

These tests run on the local JVM and do not hava access to functional Android

framework APIs.

1.2.set up your testing environment

In your top-level build.gradle file, add the following code:

dependencies {

// Required -- JUnit 4 framework

testCompile 'junit:junit:4.12'

// Optional -- Mockito framework

testCompile 'org.mockito:mockito-core:1.10.19'

}

1.3.Create a Local Unit Test Class

You can do as follows:

import org.junit.Test;

import java.util.regex.Pattern;

import static org.junit.Assert.assertFalse;

import static org.junit.Assert.assertTrue;

public class EmailValidatorTest {

@Test

public void emailValidator\_CorrectEmailSimple\_ReturnsTrue() {

assertThat(EmailValidator.isValidEmail("name@email.com"), is(true));

}

...

}

Tips:

1.如果你想让测试可读性更强，你可以使用Hamcrest matchers:https://github.com/hamcrest

1.4.Mock Android dependencies

You can do as follows:

import static org.hamcrest.MatcherAssert.assertThat;

import static org.hamcrest.CoreMatchers.\*;

import static org.mockito.Mockito.\*;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.mockito.Mock;

import org.mockito.runners.MockitoJUnitRunner;

import android.content.SharedPreferences;

@RunWith(MockitoJUnitRunner.class)

public class UnitTestSample {

private static final String FAKE\_STRING = "HELLO WORLD";

@Mock

Context mMockContext;

@Test

public void readStringFromContext\_LocalizedString() {

// Given a mocked Context injected into the object under test...

when(mMockContext.getString(R.string.hello\_word))

.thenReturn(FAKE\_STRING);

ClassUnderTest myObjectUnderTest = new ClassUnderTest(mMockContext);

// ...when the string is returned from the object under test...

String result = myObjectUnderTest.getHelloWorldString();

// ...then the result should be the expected one.

assertThat(result, is(FAKE\_STRING));

}

}

Tips:

1.如果android.jar中的Android API抛出的异常对你的测试来说是不确定的，你可以通过在最外层

build.gralde中加入以下代码来改变返回值比如null或者0：

android {

...

testOptions {

unitTests.returnDefaultValues = true

}

}

2.更多Mockito API网址：http://site.mockito.org/mockito/docs/current/org/mockito/Mockito.html

2.Building Instrumented Unit Tests

2.1.简介

和Local Unit Test相比，这个是运行在真机或模拟器上的测试

2.2.配置环境

dependencies {

androidTestCompile 'com.android.support:support-annotations:24.0.0'

androidTestCompile 'com.android.support.test:runner:0.5'

androidTestCompile 'com.android.support.test:rules:0.5'

// Optional -- Hamcrest library

androidTestCompile 'org.hamcrest:hamcrest-library:1.3'

// Optional -- UI testing with Espresso

androidTestCompile 'com.android.support.test.espresso:espresso-core:2.2.2'

// Optional -- UI testing with UI Automator

androidTestCompile 'com.android.support.test.uiautomator:uiautomator-v18:2.1.2'

}

Tips:

1.如果你的依赖中包含support-annotations的compile依赖以及espresso的androidTestCompile依赖，

build会因冲突而失败，你应该修改espresso-core代码如下：

androidTestCompile('com.android.support.test.espresso:espresso-core:2.2.2', {

exclude group: 'com.android.support', module: 'support-annotations'

})

2.为了使用JUnit 4 test class，确保指定AndroidJUnitRunner为默认的test instrumentation runner，

在app's module-levle中添加：

android {

defaultConfig {

testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"

}

}

3.由于版本问题，下面给出一个可用版本（来自google demo）：

https://github.com/googlesamples/android-testing/blob/ba14ef9e925fa17621bf86abe5336dcb9d53e466/runner/AndroidJunitRunnerSample/app/build.gradle#L36

以及一个测试可用版：

apply plugin: 'com.android.application'

android {

compileSdkVersion 23

buildToolsVersion "23.0.1"

defaultConfig {

applicationId "com.vgaw.changetest"

minSdkVersion 18

targetSdkVersion 23

versionCode 1

versionName "1.0"

testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"

}

buildTypes {

release {

minifyEnabled false

proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'

}

}

}

dependencies {

compile fileTree(include: ['\*.jar'], dir: 'libs')

compile 'com.android.support:appcompat-v7:23.0.1'

compile 'com.android.support:support-annotations:23.0.1'

testCompile 'junit:junit:4.12'

testCompile 'org.mockito:mockito-core:1.10.19'

androidTestCompile 'com.android.support.test:runner:0.4.1'

androidTestCompile 'com.android.support.test:rules:0.4.1'

androidTestCompile 'org.hamcrest:hamcrest-library:1.3'

androidTestCompile('com.android.support.test.espresso:espresso-core:2.2.1', {

exclude group: 'com.android.support', module: 'support-annotations'

})

androidTestCompile 'com.android.support.test.uiautomator:uiautomator-v18:2.1.2'

}

2.3.Create an Instrumented Unit Test Class

You can do as follows:

import android.os.Parcel;

import android.support.test.runner.AndroidJUnit4;

import android.util.Pair;

import org.junit.Test;

import org.junit.runner.RunWith;

import java.util.List;

import static org.hamcrest.Matchers.is;

import static org.junit.Assert.assertThat;

@RunWith(AndroidJUnit4.class)

@SmallTest

public class LogHistoryAndroidUnitTest {

public static final String TEST\_STRING = "This is a string";

public static final long TEST\_LONG = 12345678L;

private LogHistory mLogHistory;

@Before

public void createLogHistory() {

mLogHistory = new LogHistory();

}

@Test

public void logHistory\_ParcelableWriteRead() {

// Set up the Parcelable object to send and receive.

mLogHistory.addEntry(TEST\_STRING, TEST\_LONG);

// Write the data.

Parcel parcel = Parcel.obtain();

mLogHistory.writeToParcel(parcel, mLogHistory.describeContents());

// After you're done with writing, you need to reset the parcel for reading.

parcel.setDataPosition(0);

// Read the data.

LogHistory createdFromParcel = LogHistory.CREATOR.createFromParcel(parcel);

List<Pair<String, Long>> createdFromParcelData = createdFromParcel.getData();

// Verify that the received data is correct.

assertThat(createdFromParcelData.size(), is(1));

assertThat(createdFromParcelData.get(0).first, is(TEST\_STRING));

assertThat(createdFromParcelData.get(0).second, is(TEST\_LONG));

}

}

2.3.1.创建测试套件

可以将多个测试方法放到一个组里面，以使这些测试方法同时运行，测试组里面也可以包含测试组，

测试组代码所在包一般以.suite为后缀，代码如下：

import com.example.android.testing.mysample.CalculatorAddParameterizedTest;

import com.example.android.testing.mysample.CalculatorInstrumentationTest;

import org.junit.runner.RunWith;

import org.junit.runners.Suite;

// Runs all unit tests.

@RunWith(Suite.class)

@Suite.SuiteClasses({CalculatorInstrumentationTest.class,

CalculatorAddParameterizedTest.class})

public class UnitTestSuite {}

其中，Suite.SuiteClasses中的类为需要放到一个组里面的测试类；

2.4.Run your tests with Firebase Test Lab

相比于一般运行，你可以一次性得到不同配置（如方言、方向、屏幕大小等）的设备的测试结果，具体

请参考：https://developer.android.com/training/testing/unit-testing/instrumented-unit-tests.html#run，

应该要收费，每小时5美元；

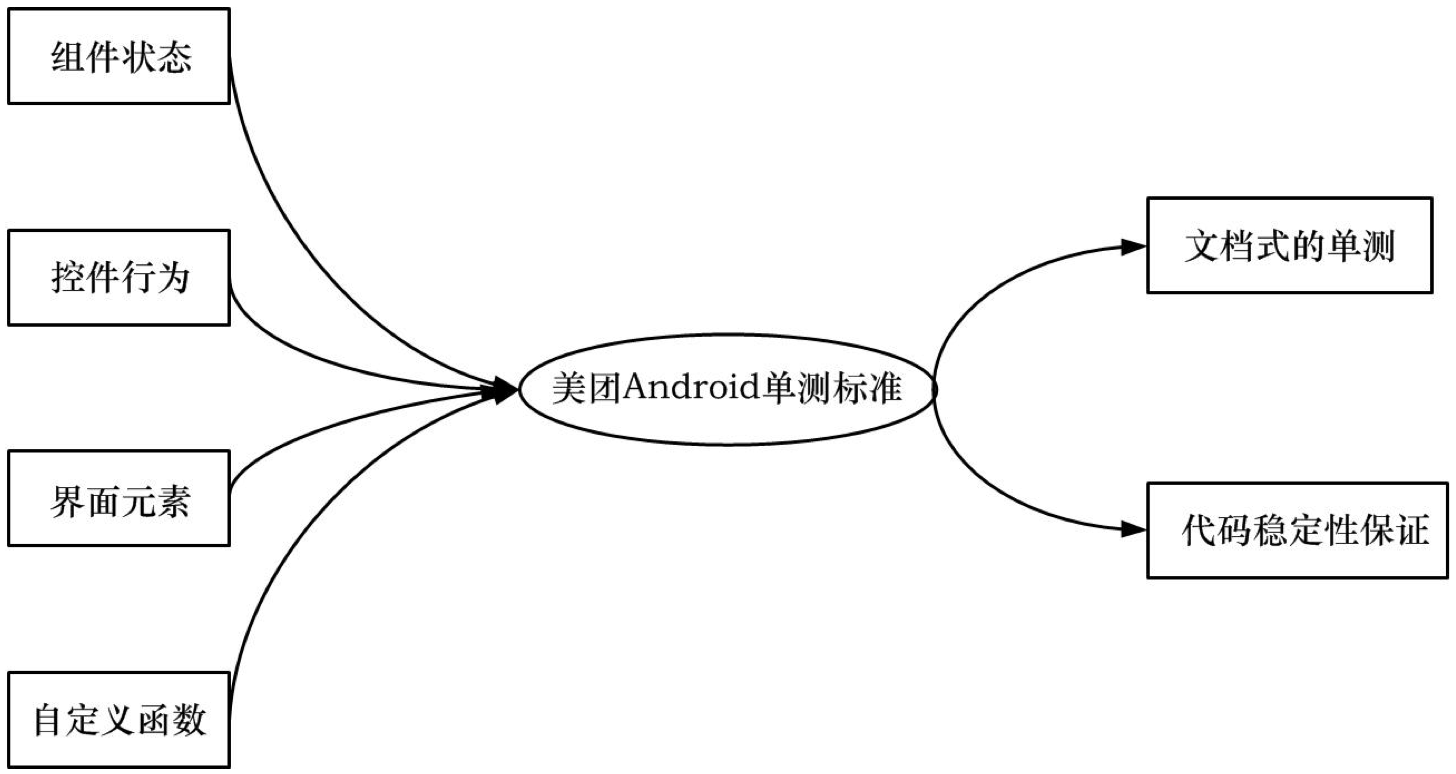
## 单元测试的范围

在Android项目中，单元测试的对象是组件状态、控件行为、界面元素和自定义函数。本文并不推荐每个函数进行你个一对一的测试，向onStart()、onDestroy()这些周期函数并不需要全部覆盖到。商业项目多采用Scrum模式，要求快速迭代，有时候未必有较多的时间写待援测试，不再要求逐个函数写单元测试。

本文单元测试的case多来源于一个简短的业务逻辑，单元测试case需要对这段业务逻辑进行验证。在验证过程中，开发人员可以深度了解业务流程，同时新人来了看一下项目单元测试就知道哪个逻辑跑了多少函数，需要注意哪些边界——是的，单元测试需要像我文档一样具备业务指导能力。

在大型项目中，遇到需要改动基类中代码的需求时，往往不能准确快速的指导改动后的范围影响，紧急时多采用创建子类覆盖父类函数的方法，但这不是长久之计，在足够覆盖率的单元测试支持下，跑一下单元测试就知道某个函数改动后的影响，可以放心的修改基类。

美团的Android单元测试编写流程如下图：



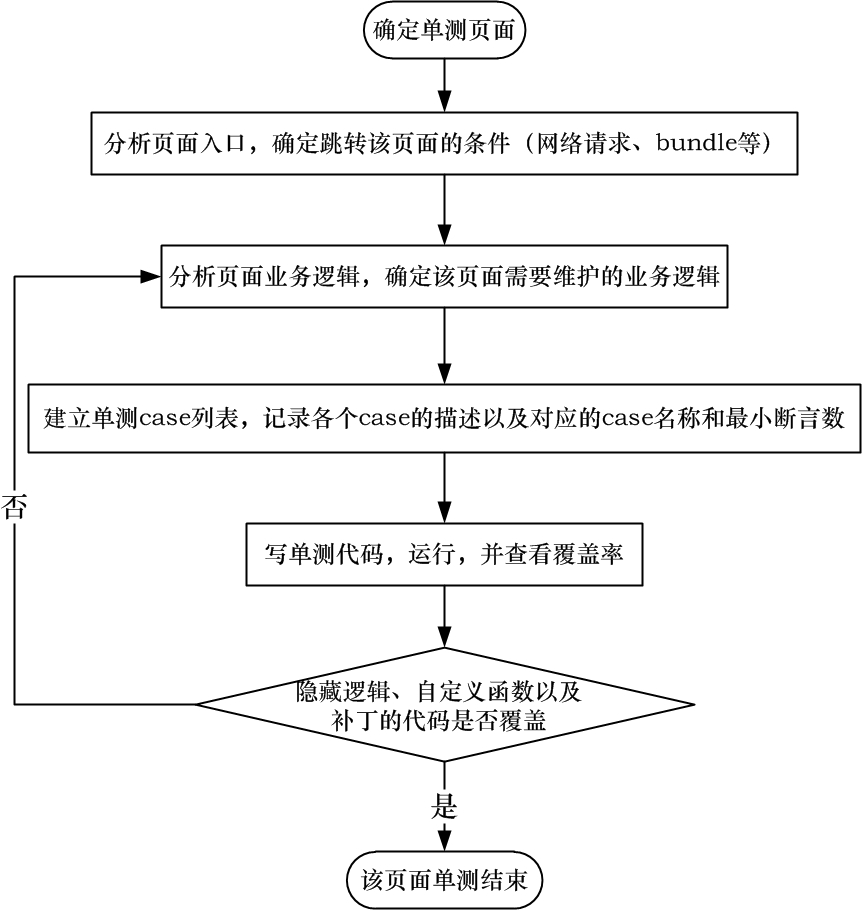
单元测试最终需要输出文档式的单元测试代码，为线上代码提供良好的代码稳定性保证。

## 单元测试的流程

实际项目中，单元测试对象与页面是一对一的，并不建议跨页面，这样的单元测试耦合度太大，维护困难。单元测试需要找到页面的入口，分析项目页面中的元素、业务逻辑，这里的逻辑不仅仅包括界面元素的展示以及控件组件的行为，还包括代码的处理逻辑。然后可以创建单元测试case列表（列表用于记录项目中的单元测试的范围，便于单元测试的管理以及新人了解业务流程），列表中记录单元测试对象的页面，对象中的case逻辑以及名称等。工程师可以根据这个列表开始写单元测试代码。

单元测试是工程师代码级别的质量保证工程，上述流程并不能完全覆盖重要的业务逻辑以及边界条件，因此，需要写完后，看覆盖率，找出单元测试中没有覆盖到的函数分支条件等，然后继续补充单元测试case列表，并在单元测试工程代码中补上case。

直到规划的页面中所有逻辑的重要分支、边界条件都被覆盖，该项目的单元测试结束。单元测试流程如下图：



上述分析页面入口所得到结果便是@Before标记的函数中的代码，之后的循环便是所有的case(@Test)标记的函数。

## Robolectric