unit test 简介

Last edited by **caoquanli** 1 month ago

UnitTest概述

Basic concept

单元测试(单体测试)是测试策略中最基础的测试，是一种白盒测试。被测对象一般指可执行的最小代码单元（方法、类或是组件）。

通过针对代码创建和运行单元测试，可以轻松验证各个单元的逻辑是否正确。每次build后运行单元测试可以帮助快速捕获错误并轻松修复代码。在打包时单元测试的代码并不会被编译进入release apk中。

> 单元测试中，程序员关注的是开发的单元或小的组件，测试的目的是确保单元能够独立地正确工作。
>
> --摘自软件测试基础教程

补充... 根据软件生命周期阶段可以划分成

软件生命周期阶段	测试技术
编码	单元测试
集成	集成测试
系统集成	系统测试
维护	回归测试xinteng

为什么要写单元测试

在敏捷开发中，项目版本快速迭代，往往需要除了黑盒测试以外更加可靠的质量保证。
有时自己模块代码很早就写好了，其他模块借口还未完成，在最后集成联调的时候自己模块出现bug，可能连代码最基本的逻辑都跑不通。 使用单元测试可以提前暴露自己模块代码的问题，从而保证自己代码逻辑的准确性和可靠性。

Android UnitTest的分类

Local UnitTest

仅在本地(自己)的计算机上运行的单元测试。 这些测试通过编译后可以在Java虚拟机(JVM)上运行。Local UnitTest的好处在于最大限度的缩短了执行的时间，不依赖Android框架。
使用Mockito框架可以将被测单元与其他依赖项隔离开。Mockito框架的主要思想是创建模拟对象，模拟对象以受控方式模仿行为或真实对象，并使用它来测试其他对象的行为。例如模拟时间、温度、网络错误等。

instrumentation UnitTest

在Andorid设备或模拟器上运行的单元测试。 通常我们使用Mock的方式不能很好解决对Android的API的依赖的这个问题，而使用这种测试方式可以依赖Android的API，使用Android提供的Instrumentation系统，将单元测试代码运行在模拟器或者是真机上，但很多情况来说，我们还是会需要和Mockito一起使用的。

- 适用Instrumentation UnitTest的情况总结如下:
  - 测试时需要Android api支持
  - 测试时需要使用Android中的组件
  - 测试时需要访问Android中特定环境元素(ex. Context)

MVP	一般使用框架
Presenter	JUnit4 + Mockito
View	Espresso + AndroidJUnitTest

MVP	一般使用框架
Model	AndroidJUnitTest
Model	Junit4 + Mockito

Android Junit4

- JUnit4
  - @Before:被这个注解的方法会阻塞 @Test 注解方法，在 @Test 注解方法前执行，如果存在多个 @Before 方法，执行顺序随机。这个注解的方法主要用来执行一些测试的设置。
  - @After 被这个注解标志的方法，会在所有的 @Test 方法之后执行，主要用来释放资源的操作。
  - @Test 被这个注解标志的方法，就是你的测试方法，一个单元测试类可以有多个测试方法。
  - @Test(timeout=) 带参数的 @Test 注解标签，如果方法在参数时间内没有成功完成，则返回失败。
  - @BeforeClass 被这个标签标注的方法，会在Class第一次加载调用，所以你可以在里面执行一个 static 方法，比如一些 static 方法是很费资源的，例如 JDBC 操作。
  - @AfterClass 对应着 @BeforeClass ，你可以进行一些 static 操作之后的资源释放。
- Assert方法的使用
  - Junit 提供了一系列的 Assert 重载方法，提供你把预期结果（Object，long，Arrays 等类型)和实际结果进行比较。同时还有一个 AssertThat()方法，提供了一个失败 Message 的输出。

Android UnitTest的配置

Local UnitTest

在Android Studio中，需要将Local Unittest的测试源文件存储在 **module-name*/src/test/java/*（项目创建时就会存在）。
类似的有 *ivi/packages/apps/**home**/app/src/test/java/*。

同时需要在app中的build.gradle中配置dependencies

```
dependencies {
    // Required -- JUnit 4 framework
    testImplementation 'junit:junit:4.12'
    // Optional -- Mockito framework
    testImplementation 'org.mockito:mockito-core:1.10.19'
}
```

*推荐配合使用Mockito框架来进行Local Unittest。

Instrumentation UnitTest

在Android Studio中，需要将instrumentation Unittest的测试源文件存储在 **module-name*/src/androidTest/java/*（项目创建时就会存在）。
类似的有 *ivi/packages/apps/**home**/app/androidTest/test/java/*。

Dependencies

在开始之前应该下载Android Test，它提供的API允许快速构建和运行应用程序的检测测试代码。Android Test包括一个JUnit 4测试运行器（AndroidJUnitRunner）和用于功能UI测试的API（Espresso 和UI Automator）。
接着需要为项目配置Android测试依赖项，以使用测试运行器和测试支持库提供的规则API。为了简化测试开发，还应该包含 Hamcrest 库，它允许您使用Hamcrest匹配器API创建更灵活的断言。

在build.gradle中配置dependencies

```
dependencies {
    androidTestImplementation 'com.android.support:support-annotations:28.0.0'
    androidTestImplementation 'com.android.support.test:runner:1.0.2'
    androidTestImplementation 'com.android.support.test:rules:1.0.2'
    // Optional -- Hamcrest library
    androidTestImplementation 'org.hamcrest:hamcrest-library:1.3'
    // Optional -- UI testing with Espresso
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.2'
    // Optional -- UI testing with UI Automator
```

```
    androidTestImplementation 'com.android.support.test.uiautomator:uiautomator-v18:2.1.
3'
}
android {
    defaultConfig {
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
}
```

Makefile

每个新增的test module都必须拥有一个自己的makefile

snapshot here:

```
 LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := tests
LOCAL_SRC_FILES := $(call all-java-files-under, src)
LOCAL_JAVA_LIBRARIES := android.test.runner
LOCAL_STATIC_JAVA_LIBRARIES := ub-uiautomator junit legacy-android-test
LOCAL_PACKAGE_NAME := ShellTests
LOCAL_INSTRUMENTATION_FOR := Shell
LOCAL_COMPATIBILITY_SUITE := device-tests
LOCAL_CERTIFICATE := platform
include $(BUILD_PACKAGE)
```

```
 LOCAL_MODULE_TAGS := tests
```

声明Test Module

```
 LOCAL_CERTIFICATE := platform
```

build system使用平台证书对测试应用程序包进行签名。
这是因为为了使测试package能够在目标package上进行检测，这两个包必须使用相同的证书进行签名。

```
注：这是适用于platform的连续测试，最好不要和CTS测试模块一起使用。
在普通情况下，不需要设置这项：build system将使用默认的内置证书对其进行签名，通常称为dev-keys。
```

```
 LOCAL_JAVA_LIBRARIES := android.test.runner
```

build system在编译期间将android.test.runner（Java库）放在class路径上，而不是将用到的库静态地合并到当前包中。

```
 LOCAL_STATIC_JAVA_LIBRARIES := ub-uiautomator junit legacy-android-test
```

build system将命名模块的内容合并到当前模块的结果（apk）中。这意味着每个命名模块都需要生成一个.jar文件，其内容将用于在编译期间解析类路径引用，以及合并到生成的apk中。
平台源代码还包括其他有用的测试框架，例如ub-uiautomator，easymock，android-support-test等等。

```
android-support-test是Android测试支持库的预构建，其中包括AndroidJUnitRunner
```

```
 LOCAL_PACKAGE_NAME := avAndroidTest
```

为模块命名，并且生成同名apk文件。
例如在这种情况下，生成的测试apk被命名为avAndroidTest.apk。
此外，这还定义了模块的make目标名称，以便您可以使用它make [options] <LOCAL_PACKAGE_NAME>来构建测试模块及其所有依赖项。

```
 LOCAL_INSTRUMENTATION_FOR : = av
```

在执行检测测试期间，重新启动测试中的应用程序，并注入执行的检测代码。
测试可以引用被测试的应用程序的任何类及其实例。这意味着测试代码可能包含对被测应用程序定义的类的引用，因此在编译期间，build system需要正确解析此类引用。
此设置提供了受测试应用程序的模块名称，该名称应与LOCAL_PACKAGE_NAME应用程序的makefile中的模块名称相匹配。在编译时，build system将尝试查找指定模块的中间文件，并在Java编译器的class路径上使用它们。

Manifest file

就像常规应用程序一样，每个检测测试模块都需要一个清单文件。
如果将文件命名为AndroidManifest.xml并将其提供给Android.mk测试的module，则BUILD_PACKAGE核心makefile将自动包含该文件。

Snapshot here:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.android.shell.tests">
    <application>
        <uses-library android:name="android.test.runner" />
        <activity
            android:name="com.android.shell.ActionSendMultipleConsumerActivity"
            android:label="ActionSendMultipleConsumer"
            android:theme="@android:style/Theme.NoDisplay"
            android:noHistory="true"
            android:excludeFromRecents="true">
            <intent-filter>
                <action android:name="android.intent.action.SEND_MULTIPLE" />
                <category android:name="anmoduledroid.intent.category.DEFAULT" />
                <data android:mimeType="*/*" />
            </intent-filter>
        </activity>
    </application>
    <instrumentation android:name="android.support.test.runner.AndroidJUnitRunner"
        android:targetPackage="com.android.shell"
        android:label="Tests for Shell" />
</manifest>
```

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.android.shell.tests">
```

package属性是应用程序包名称：这是Android应用程序框架用于标识应用程序的唯一标识符。
系统中的每个用户只能安装一个具有该软件包名称的应用程序。

由于这是一个测试应用程序包，独立于测试中的应用程序包，因此必须使用不同的包名称：一个常见的约定是添加后缀.test。

此外，此package属性与ComponentName#getPackageName()返回的属性相同，也与用于与各种pm子命令进行交互相同adb shell。

我们可以使用一下命令来查看

```
pm list instrumentation
```

```
<uses-library android: name = “android.test.runner” />
```

这是所有Instrumentation测试所必需的。
因为相关的类打包在一个单独的框架jar库文件中，所以在应用程序框架调用测试包时需要额外的类路径条目。

```
android:targetPackage="com.android.shell"
```

这会将目标package设置为com.android.shell.tests。
当通过am instrument命令调用instrumentation时，框架重新启动com.android.shell.tests进程，并将代码注入进程以进行测试。
这意味着测试代码可以访问在被测应用程序中运行的所有类实例。

AndroidTest.xml

如果要使用Tradefederation的话，需要写一个test configuration文件。 Snapshot here:

```
<configuration description="Runs sample instrumentation test.">
  <target_preparer class="com.android.tradefed.targetprep.TestFilePushSetup"/>
  <target_preparer class="com.android.tradefed.targetprep.TestAppInstallSetup">
    <option name="test-file-name" value="HelloWorldTests.apk"/>
  </target_preparer>
  <target_preparer class="com.android.tradefed.targetprep.PushFilePreparer"/>
  <target_preparer class="com.android.tradefed.targetprep.RunCommandTargetPreparer"/>
  <option name="test-suite-tag" value="apct"/>
  <option name="test-tag" value="SampleInstrumentationTest"/>
  <test class="com.android.tradefed.testtype.AndroidJUnitTest">
    <option name="package" value="android.test.example.helloworld"/>
    <option name="runner" value="android.support.test.runner.AndroidJUnitRunner"/>
  </test>
</configuration>
```

```
<target_preparer class="com.android.tradefed.targetprep.TestAppInstallSetup">
  <option name="test-file-name" value="HelloWorldTests.apk"/>
</target_preparer>
```

这告诉TradeFederation安装target_preparer，即HelloWorldTests.apk。一个AndroidTest.xml文件可以有很多target_preparer。

```
<test class="com.android.tradefed.testtype.AndroidJUnitTest">
  <option name="package" value="android.test.example.helloworld"/>
  <option name="runner" value="android.support.test.runner.AndroidJUnitRunner"/>
</test>
```

这指定了用于执行测试的TradeFederation测试类，并在要执行的设备上传递包，在本例中指定了JUnit的测试运行器框架。

UnitTest的运行

Local UnitTest

Android Studio方法

1. 确保项目中Gradle已经同步，若没有同步可以在Android Studio的工具栏中点击Sync Project
2. 按照以下步骤执行测试
   - (执行单个测试)打开Project窗口，点击一个测试并运。
   - (执行一个测试类中所有的方法)选择一个类或者方法在测试文件病选择执行。
   - (执行文件夹中所有测试)选择文件夹并执行。

Instrumentation UnitTest

Android Studio方法

同Local UnitTest一致

Command Line ADB方法

1. 在 *andoridTest文件夹下创建Android.mk和Androidmanifest.xml文件*，具体可参考 *ivi/packages/apps/**home**/app/androidTest/Andorid.mk & AndroidManifest.xml*
2. 生成.apk文件

```
EMMA_INSTRUMENT_STATIC=true mmma apkName
或
make moduleName -j
```

3. adb install 这个.apk文件

```
adb install -r ${out}
```

4. 使用各种选项运行测试
   a. apk中的所有测试

```
adb shell am instrument -w com.android.frameworks.coretests \
 /android.support.test.runner.AndroidJUnitRunner
```

   b. 特定Java包下的所有测试

```
adb shell am instrument -w -e package android.animation \
 com.android.frameworks.coretests \
 /android.support.test.runner.AndroidJUnitRunner
```

c. 特定类下所有的测试

```
adb shell am instrument -w -e class \
 android.animation.AnimatorSetEventsTest \
 com.android.frameworks.coretests \
 /android.support.test.runner.AndroidJUnitRunner
```

d. 一种特定的测试方法

```
adb shell am instrument -w -e class \
android.animation.AnimatorSetEventsTest #testCancel \
com.android.frameworks.coretests \
/android.support.test.runner.AndroidJUnitRunner
```

可通过adb shell pm list instrumentation 来查看对应instrumentation。

```
https://developer.android.com/studio/test/command-line
https://www.jianshu.com/p/5d6b1d2a5b71
```

UnitTest的结果

通过Android studio方法进行测试的结果会最直观的出现在AS下方。