Android Studio单元测试技术

|  |  |  |
| --- | --- | --- |
| 版本号 | 描述 | 日期(更新日期)/更新人 |
| 1.0 | Android Studio单元测试技 | 2018/08/27 刘志保 |
|  |  |  |
| Github地址 | https://github.com/MMLoveMeMM/[AngryPandaUnitText](https://github.com/MMLoveMeMM/AngryPandaUnitText)https://github.com/MMLoveMeMM/ [AngryPandaUIUnitText](https://github.com/MMLoveMeMM/AngryPandaUIUnitText) 其中UI单元测试参考学习,我上面没怎么写:  https://github.com/geniusmart/LoveUT | |

Android Studio提供了两种单元测试,一种是Java单元测试,一种是App[Android]程序单元测试.下面分别介绍一下操作流程:

Android单元测试

在Java单元测试的基础上,增加一个获取Context(这个需要在AS开发平台进行)

增加UI等测试需要增加依赖包robolectric[这个依赖库限制在SDK<23的版本]

Android逻辑单元测试

Android UI单元测试

Java单元测试

手工建立单元测试目录即可和gradle脚本,配置gradle环境变量,在cmd终端即可进行

注意事项:

<a> : robolectric这个依赖库限制在SDK < 23的版本,这个有点坑,现在Android 9.0都出来了,让开发有点懵逼.

<b> : 上面有一个没有提出,就是mockie这个架构,不然分分让你的单元测试没有输入数据信息.这个架构可以应用在所有的单元测试中.

单元测试所在开发的流程如下:

所有的单元测试通过

Java单元测试

Android单元测试

开发项目程序功能代码

配置gradle命令行环境

构建工程开发环境

构建Java开发环境

构建Android开发环境[AS]

统计单元测试覆盖率

Gradle打包

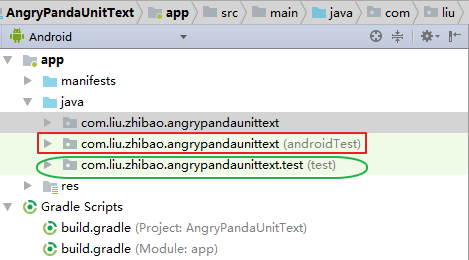
首先熟悉预备知识:

<0> : gradle/gradlew 的基本使用;

<1> : 单元测试判断参考 : <http://www.tutorialspoint.com/junit/junit_using_assertion.htm>,熟悉里面几个比较的方法,不适用断言也可以,以后测试的时候方便比对实际值和预期值.单元测试周期: https://www.tutorialspoint.com/junit/junit\_execution\_procedure.htm

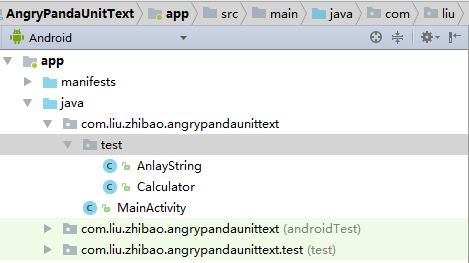
<2> : 单元测试环境:

新建一个Android工程.



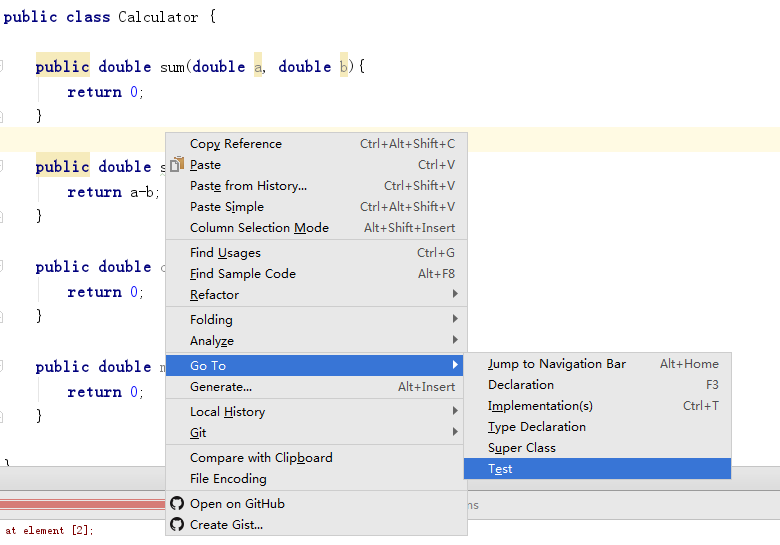
红色的为android应用单元测试,绿色的为普通java单元测试.最上面那个是正常的软件代码.

新建Calculator.java的类

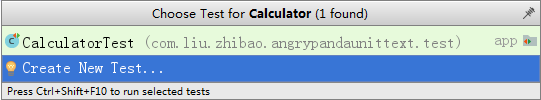


并且写好使用的程序,被单元测试的方法必须是public.

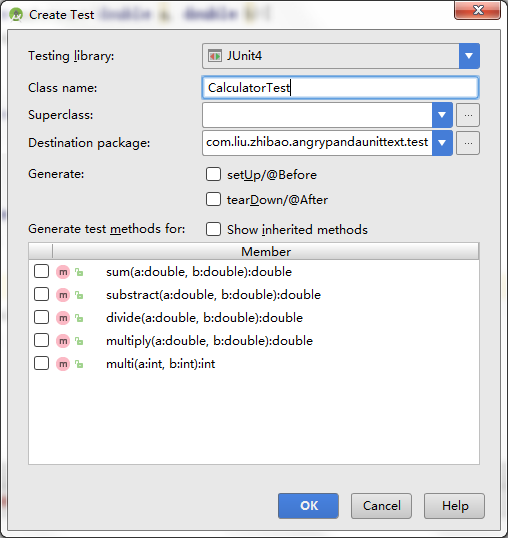
程序写完后,在文件里面右击选择goto->test,如下图:



然后弹出:

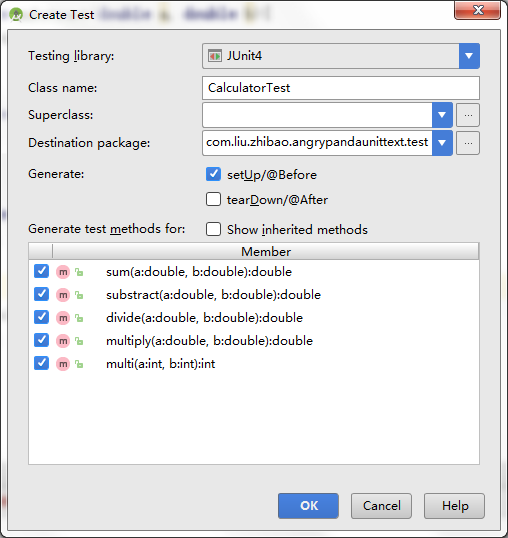


这个地方,直接选择Create New Test …这一项,如果前面已经新建了也没关系,它只会在原有的基础上增加新的,后面有勾选那些需要进行新增测试的.然后弹出下框:

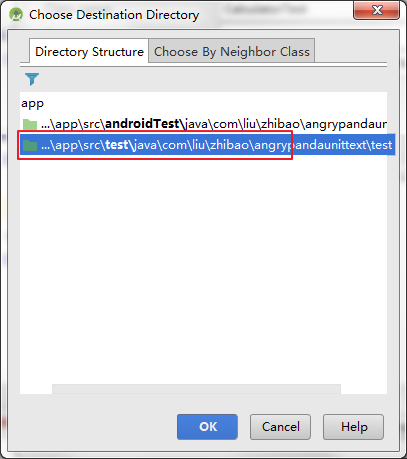


这里面勾选如下,所有的方法都需要单元测试,当然实际上根据自己的需要:

其中Generate那一项勾选setUp/@Before



点击OK,下面选择红色框的,这里先做java程序的单元测试:



点击OK.然后:



开始自己的单元测试:

**package** com.liu.zhibao.angrypandaunittext.test;  
  
**import** org.junit.Before;  
**import** org.junit.Test;  
  
**import static** org.junit.Assert.\*;  
  
*/\*\*  
 \* Created by zhibao.Liu on 2018/8/27.  
 \*  
 \** ***@version*** *:  
 \** ***@date*** *: 2018/8/27  
 \** ***@des*** *:  
 \** ***@see{@link}*** *\*/***public class** CalculatorTest {  
 @Test  
 **public void** multi() **throws** Exception {  
 }  
  
 **private** Calculator **mCalculator**;  
  
 @Before  
 **public void** setUp() **throws** Exception {  
 **mCalculator** = **new** Calculator();  
 }  
  
 @Test  
 **public void** sum() **throws** Exception {  
 */\*  
 \* 第一个参数为期望的值;  
 \* 后面调用的被测试方法  
 \* \*/  
 assertEquals*(6d, **mCalculator**.sum(1d, 5d), 0);  
 }  
  
 @Test  
 **public void** substract() **throws** Exception {  
 *assertEquals*(4d, **mCalculator**.substract(5d, 1d), 0);  
 }  
  
 @Test  
 **public void** divide() **throws** Exception {  
 }  
  
 @Test  
 **public void** multiply() **throws** Exception {  
 }  
  
}

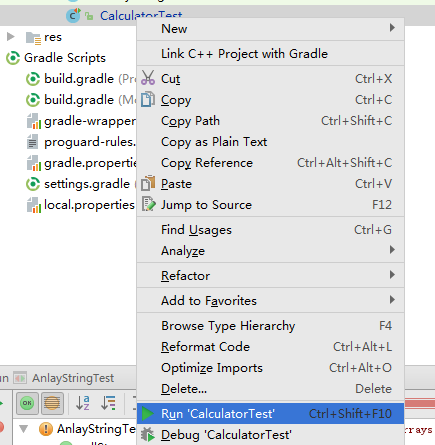
其中:



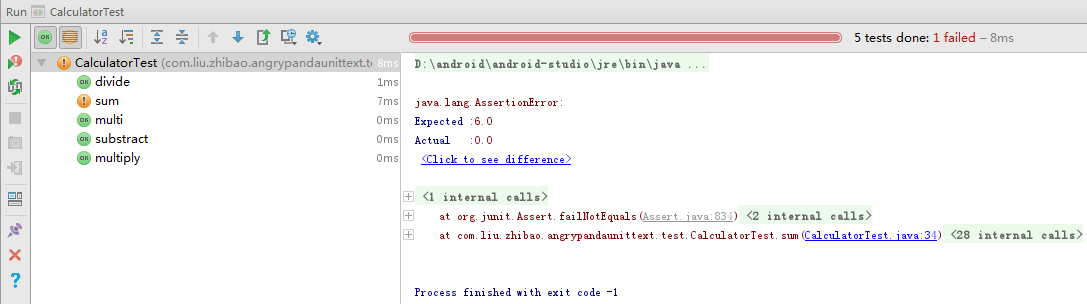
@Ignore 所修饰的测试方法会被测试运行器忽略

|  |  |
| --- | --- |
| @Test(timeout = 3000) | 测试方法执行时间是否在预期时间内,超过时间将强制终止方法 |
| @Test(expected=ArithmeticException/NullPointerException  .class) | 处理运行时异常 |
|  |  |

写完单元测试后,运行:注意:单元测试也可以断点debug的.



运行后看结果:



左侧表示方法通过单元测试,符合”预期”;

 : 表示方法未能通过单元测试,不符合”预期”;

右侧: 信息即sum方法没有通过单元测试的详细信息.

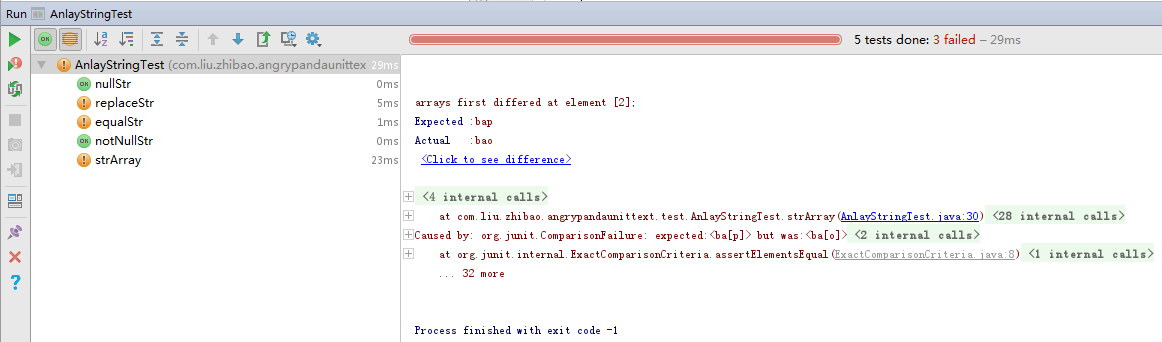
下面再写一个类AnlayString.java的类,测试一下<1>里面的各个API

**package** com.liu.zhibao.angrypandaunittext.test;  
  
**import** android.text.TextUtils;  
  
*/\*\*  
 \* Created by zhibao.Liu on 2018/8/28.  
 \*  
 \** ***@version*** *:  
 \** ***@date*** *: 2018/8/28  
 \** ***@des*** *:  
 \** ***@see{@link}*** *\*/***public class** AnlayString {  
  
 **public** String replaceStr(String info){  
 String ret=**null**;  
 **if**(info.contains(**"a"**)){  
 ret = info.replace(**"a"**,**"c"**);  
 }  
 **return** ret;  
 }  
  
 **public** String nullStr(){  
 **return null**;  
 }  
  
 **public** String notNullStr(){  
 **return "hello"**;  
 }  
  
 **public boolean** equalStr(String data){  
 **if**(data.equalsIgnoreCase(**"hello"**)){  
 **return true**;  
 }  
 **return false**;  
 }  
  
 **public** String[] strArray(){  
 String[] info={  
 **"liu"**,**"zhi"**,**"bao"** };  
 **return** info;  
 }  
}

然后根据上面的步骤,写出测试单元:

**package** com.liu.zhibao.angrypandaunittext.test;  
  
**import** org.junit.Before;  
**import** org.junit.Test;  
  
**import static** org.junit.Assert.\*;  
**import static** org.junit.Assert.*assertFalse*;  
  
*/\*\*  
 \* Created by zhibao.Liu on 2018/8/28.  
 \*  
 \** ***@version*** *:  
 \** ***@date*** *: 2018/8/28  
 \** ***@des*** *:  
 \** ***@see{@link}*** *\*/***public class** AnlayStringTest {  
  
 AnlayString **anlayString**;  
 @Before  
 **public void** setUp() **throws** Exception {  
 **anlayString** = **new** AnlayString();  
 }  
  
 @Test  
 **public void** strArray() **throws** Exception {  
 String[] strs={  
 **"liu"**,**"zhi"**,**"bap"** };  
 *assertArrayEquals*(strs,**anlayString**.strArray());  
 }  
  
 @Test  
 **public void** equalStr() **throws** Exception {  
 *// assertFalse(anlayString.equalStr("helloa")); // 不相等,false,即设置的条件是成立的  
 assertTrue*(**anlayString**.equalStr(**"helloa"**));  
 }  
  
 @Test  
 **public void** replaceStr() **throws** Exception {  
  
 *// assertEquals("cbcdd", anlayString.replaceStr("abcdd")); // 比较值即可  
 assertSame*(**"cbcdd"**, **anlayString**.replaceStr(**"abcdd"**)); *// 比较Object,说白了就是看地址是不是一样的* }  
  
 @Test  
 **public void** nullStr() **throws** Exception {  
 *assertNull*(**anlayString**.nullStr());  
 }  
  
 @Test  
 **public void** notNullStr() **throws** Exception {  
 *assertNotNull*(**anlayString**.notNullStr());  
 }  
  
}

运行后结果如下:



正常的java代码逻辑测试可以根据自己的设定灵活变通,整理来说还是比较简单的,也是最常见的.

下面是Android 单元测试,参考下面url链接, Robolectric使用教程:进行UI测试和网络测试

<1> : https://blog.csdn.net/shensky711/article/details/53561172?locationNum=1&fps=1.

<2> : https://www.jianshu.com/p/a01628c3ea16

<3> : https://www.cnblogs.com/slgkaifa/p/7354609.html

<4> : https://www.jianshu.com/p/9d988a2f8ff7

Github :

https://github.com/JieYuShi/Luffy

另外google默认推荐的Espresso架构,参考:

https://blog.csdn.net/chen19960724/article/details/52540885

个人感觉Android UI等方面的单元测试没太大必要,浪费时间,界面复杂,单元测试就非常复杂,而且界面都是由很强的交互性的,单元测试很难测试出来.还不如交给monkey;

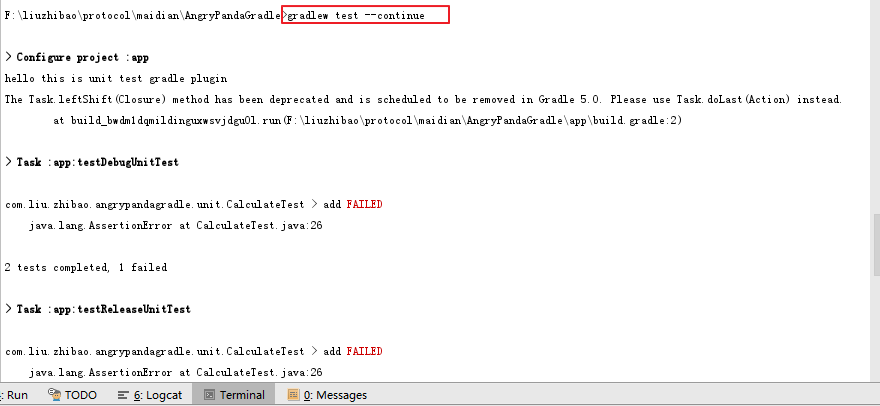
**下面介绍利用命令行进行单元测试验证.**

<1> : 命令行 gradlew test --continue

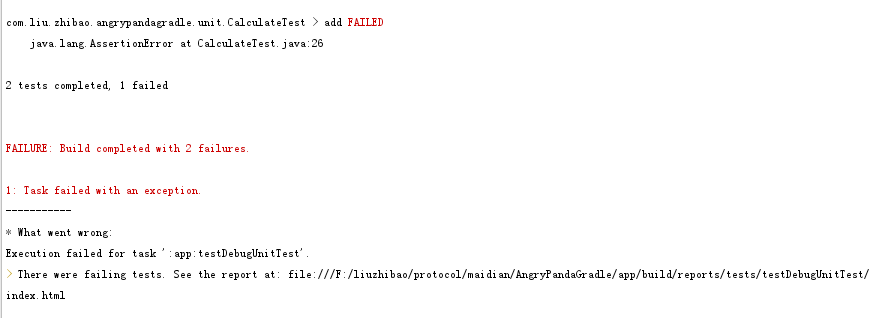
下面是验证的程序:

@Test  
**public void** add() **throws** Exception {  
 *assertEquals*(4,**mCalculate**.Add(2,5),0);  
}

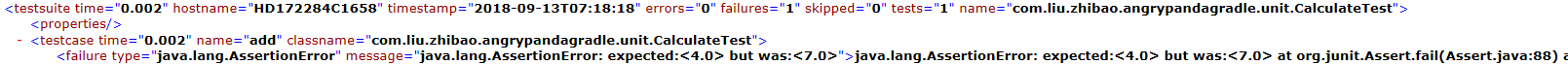
在终端中执行命令行:



结论:



然后根据上面的结论,看看它生成index.html网页:



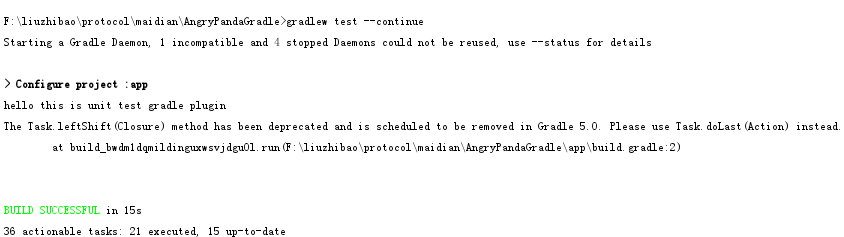
提示CalculateTest单元测试中add方法实际结果和预期结果不符.

<2> : 下面写两个单元测试:

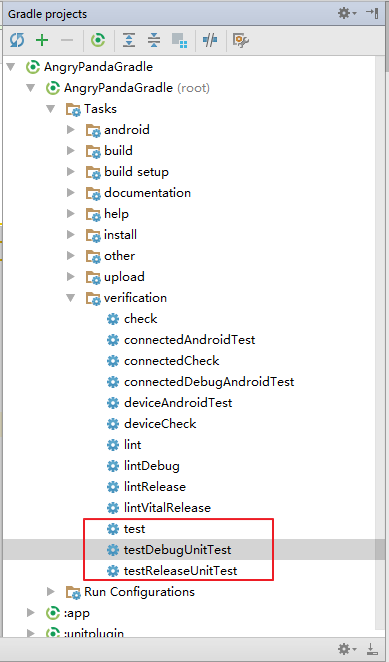


写好后,同样运行<1>中的命令行:

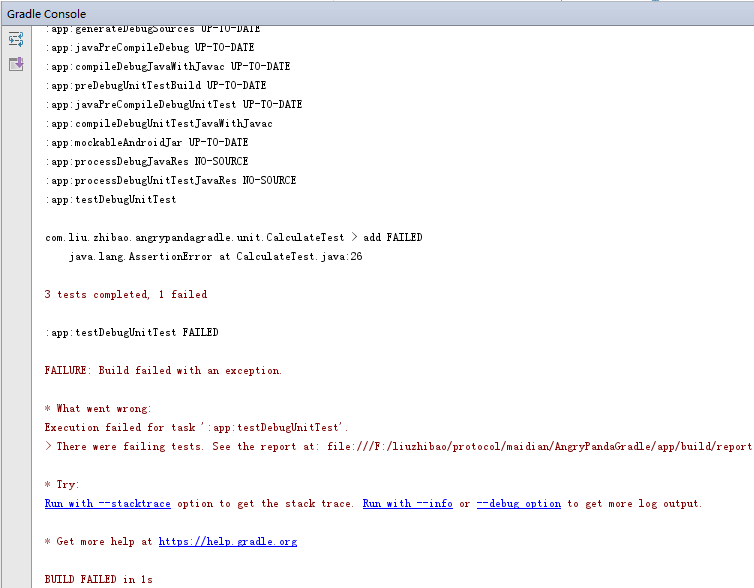
运行结果:



单元测试通过.上面的命令行也可以用下面的方法执行:

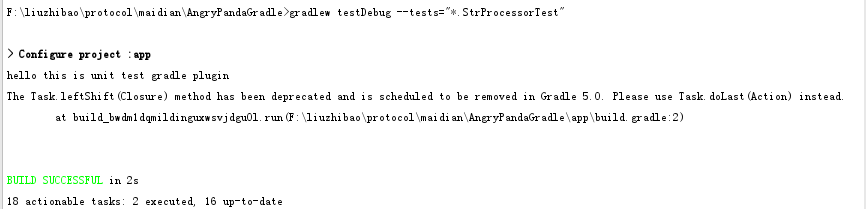


直接双击上面其中一个执行.执行结果:



这个结果在gradle console控制台.

<3> : 如果写了很多个单元测试类,但是实际上只需要测试其中一个类就可以,可以这样:

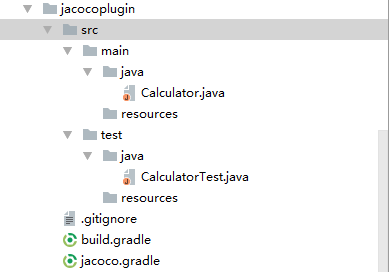


gradlew testDebug --tests="\*.StrProcessorTest"

或者 : gradlew testDebug --tests \*StrProcessorTest

这里面 试了很多次,不能够指定其中多个进行测试.

<4> : 这里做一个纯粹的扩展,如何单独新建一个java单元测试,利用build.gradle进行单元测试,在任意目录下新建一个jacocoplugin文件夹,然后新建一个build.gradle文件,目录结构如下:



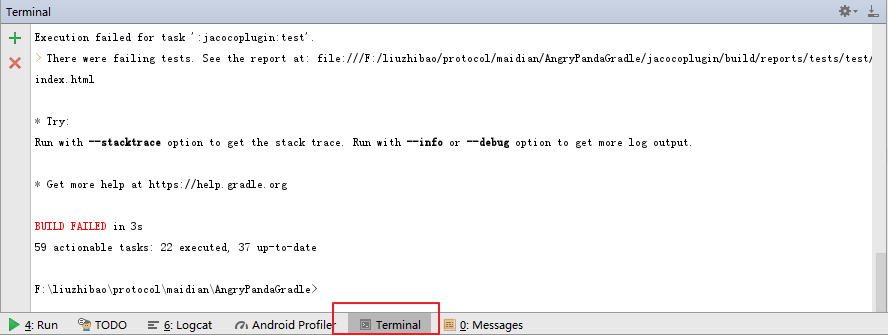
build.gradle文件内容:图中jacoco.gradle暂时不用管,当做是一个空文件

apply **plugin**: **'java'**repositories {  
 mavenCentral()  
}  
  
dependencies {  
 testCompile **'junit:junit:4.12'**}

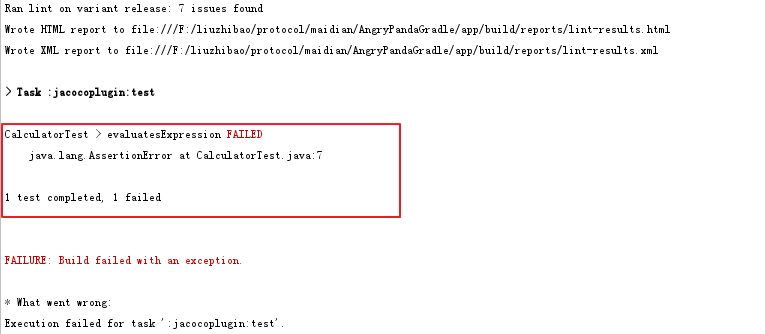
整体工程自动扫描到这个build.gradle文件.

[上面程序代码就不上了,放在github上面].

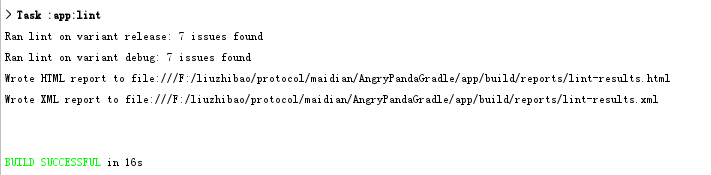
在终端执行: gradle build



执行后稍等片刻即有结果了.比如用一个通不过的测试单元:



如果测试单元通过:



另外:



# <5> : Mockito框架的使用,这个主要是模拟数据.

<a> : 首先环境匹配:

工程的build.gradle[project]修改如下:

allprojects {  
 repositories {  
 *// google()  
 // jcenter()* mavenCentral()  
 google()  
 }  
}

这个jcenter 仓库太恶心了,不好用[至少公司网络无法连接到该库].

然后在app的build.gradle中添加下面的依赖库:

*// 如果要使用Mockito，你需要添加此条依赖库*testCompile **'org.mockito:mockito-core:1.+'***// 如果你要使用Mockito 用于 Android instrumentation tests，那么需要你添加以下三条依赖库*androidTestCompile **'org.mockito:mockito-core:1.+'**androidTestCompile **"com.google.dexmaker:dexmaker:1.2"**androidTestCompile **"com.google.dexmaker:dexmaker-mockito:1.2"**

<b> : mockito框架学习:

[参考: <https://github.com/hehonghui/mockito-doc-zh>]

[参考: <https://blog.csdn.net/qq_17766199/article/details/78450007>]

[参考: https://www.imooc.com/article/51050]

<c> : 具体的使用就不介绍了,参考<b>中学习.

<6> : 下面在扯一下Android UI等相关的单元测试,这个地方一直没有更新是因为Roboletric依赖库无法下载.修改依赖库地址.

然后在github上面下载LoveUT-master.zip这个工程研究研究就可以了.

<7> : 单元测试覆盖率

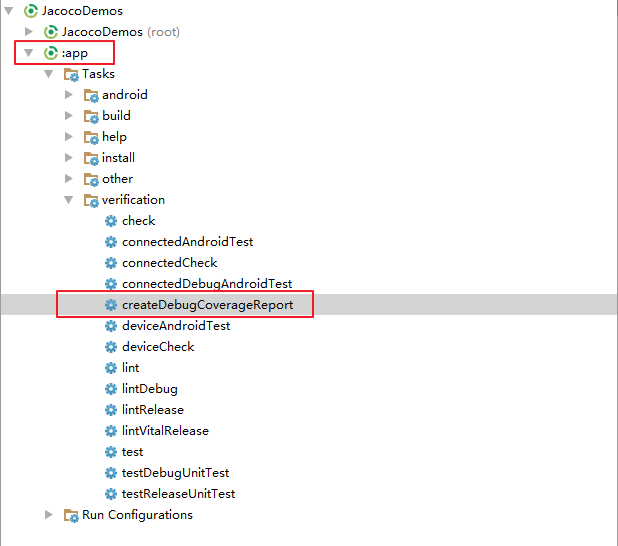
配置环境:build.gradle[app]

apply **plugin**: **'jacoco'**jacoco {  
 toolVersion = **"0.7.1.201405082137"**}

buildTypes {  
 …  
 debug {  
 testCoverageEnabled = **true** }  
}

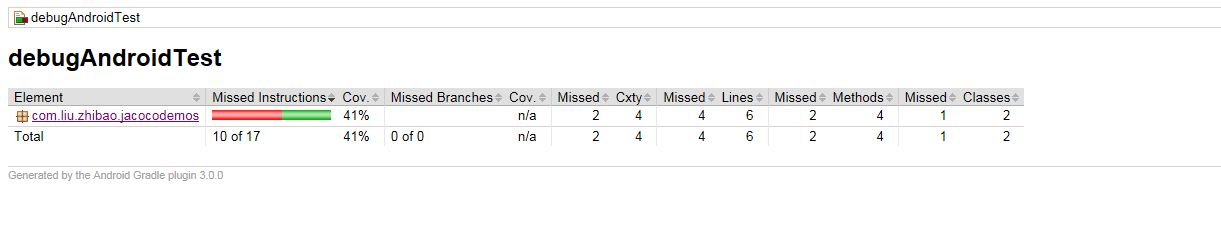
下载的仓库使用上面的配置.使用jcenter的仓库经常无法下载.

Sync 后,

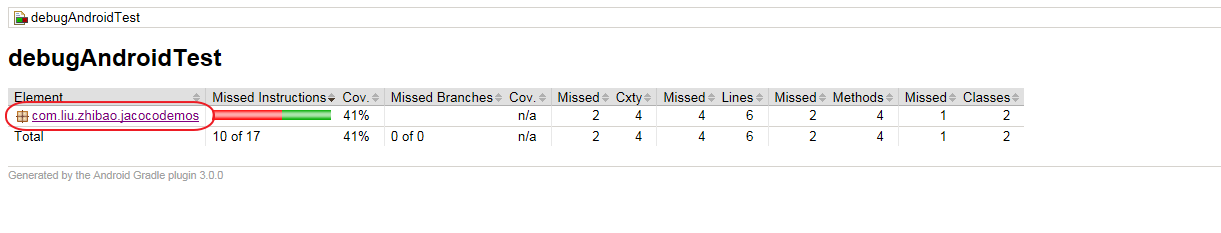


上面的createDebugCoverageReport相在上面配置前是没有的.写好测试单元以后,运行过测试单元没问题了,然后点击上面的菜单运行,就可以生成覆盖率的测试报告:

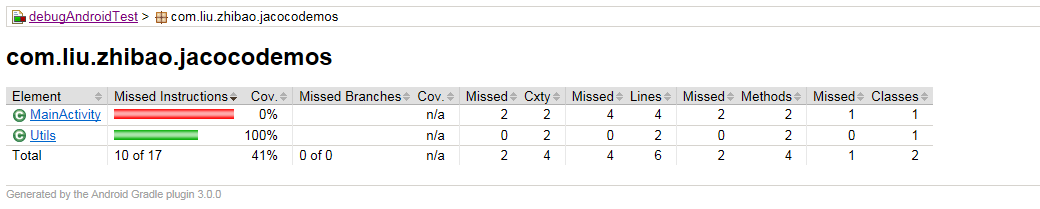
可以到app\build\reports\coverage\debug中的index.xml中查看结果



然后覆盖率是41%,也就是说还有59%没有覆盖到,然后点击报名[下面红色框],查看详情:

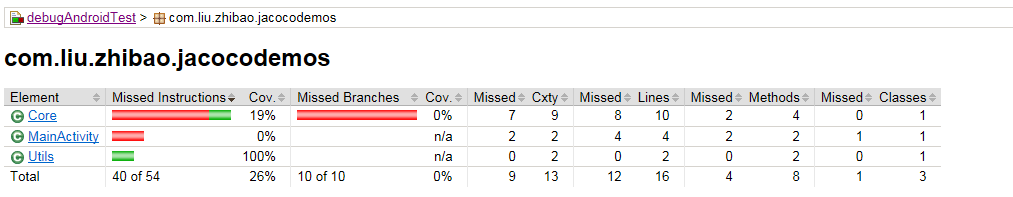


详情如下:

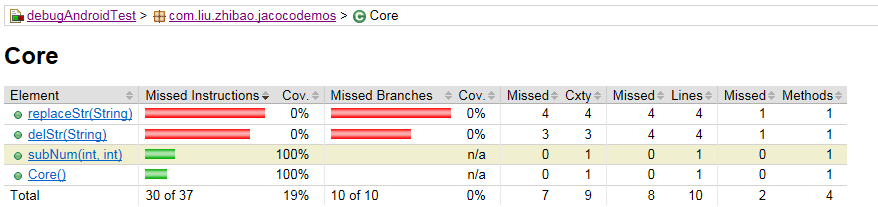


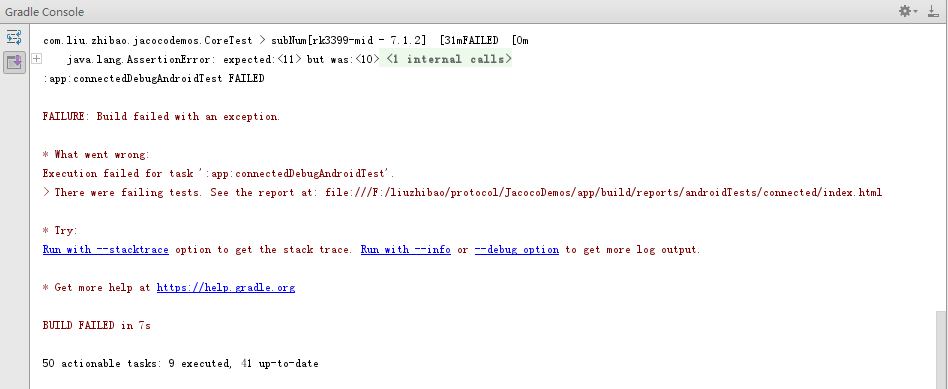
从上面可以看出,MainActivity类覆盖率为0,就是说这个类没有进行单元测试.Utils类单元测试覆盖为100%,即这个类完全进行了单元测试.

在增加一个类,看看一个类只是部分做了单元测试的详情:



可以看到Core类具体的单元测试情况:



覆盖率测试需要注意的是,单元测试一定要全部通过,比如下面如果单元测试没有通过,就会报下面的问题:  


上面的报告也可以通过命令行:

* gradlew jacocoTestReport

单元测试注意事项:

<a> : 方法的输入必须是模拟数据,没办法使用正常APP/Java运行时的内存数据,当然可以打桩,不能有依赖性,比如一个方法依赖于另外一个方法,保证最小单元的测试[原子单元],即需要被测试的A方法里面,不能调用B类里面的数据,这些因为B类里面再单元测试的时候是没有内存数据的,会造成测试失败[除非测试的就是B类为null的情况];

<b> : 一般方法要有返回值,即必须要有输入输出,如果没有输出[返回值],一般测试该方法的异常信息;

<c> : 单元测试不要通过打印的方式输出结果,通过肉眼去查看是否通过测试等,一定要使用断言,利用测试单元最终给出判定结果.

<d> : …

下面提供阿里巴巴守则的约定:

1. 【强制】好的单元测试必须遵守 AIR 原则。  
说明： 单元测试在线上运行时，感觉像空气（ AIR） 一样并不存在，但在测试质量的保障上，却是非常关键的。好的单元测试宏观上来说，具有自动化、独立性、可重复执行的特点。  
⚫A： Automatic（自动化）  
⚫I： Independent（独立性）  
⚫R： Repeatable（可重复）  
2. 【强制】单元测试应该是全自动执行的，并且非交互式的。测试框架通常是定期执行的，执行过程必须完全自动化才有意义。输出结果需要人工检查的测试不是一个好的单元测试。单元测试中不准使用 System.out 来进行人肉验证，必须使用 assert 来验证。  
3. 【强制】保持单元测试的独立性。为了保证单元测试稳定可靠且便于维护，单元测试用例之间决不能互相调用，也不能依赖执行的先后次序。  
反例： method2 需要依赖 method1 的执行， 将执行结果做为 method2 的输入。  
4. 【强制】单元测试是可以重复执行的，不能受到外界环境的影响。  
说明： 单元测试通常会被放到持续集成中，每次有代码 check in 时单元测试都会被执行。如果单测对外部环境（网络、服务、中间件等） 有依赖，容易导致持续集成机制的不可用。  
正例： 为了不受外界环境影响，要求设计代码时就把 SUT 的依赖改成注入，在测试时用 spring这样的 DI 框架注入一个本地（内存）实现或者 Mock 实现。  
5. 【强制】对于单元测试，要保证测试粒度足够小，有助于精确定位问题。单测粒度至多是类级别，一般是方法级别。  
说明： 只有测试粒度小才能在出错时尽快定位到出错位置。单测不负责检查跨类或者跨系统的交互逻辑，那是集成测试的领域。  
6. 【强制】核心业务、核心应用、核心模块的增量代码确保单元测试通过。  
说明： 新增代码及时补充单元测试，如果新增代码影响了原有单元测试，请及时修正。  
7. 【强制】单元测试代码必须写在如下工程目录： src/test/java，不允许写在业务代码目录下。  
说明： 源码构建时会跳过此目录，而单元测试框架默认是扫描此目录。  
8. 【推荐】单元测试的基本目标：语句覆盖率达到 70%；核心模块的语句覆盖率和分支覆盖率都  
要达到 100%  
说明： 在工程规约的应用分层中提到的 DAO 层， Manager 层，可重用度高的 Service，都应该进行单元测试。9. 【推荐】编写单元测试代码遵守 BCDE 原则，以保证被测试模块的交付质量。  
⚫B： Border，边界值测试，包括循环边界、特殊取值、特殊时间点、数据顺序等。  
⚫C： Correct，正确的输入，并得到预期的结果。  
⚫D： Design，与设计文档相结合，来编写单元测试。  
⚫E： Error，强制错误信息输入（如：非法数据、异常流程、非业务允许输入等），并得到预期的结果。  
10. 【推荐】对于数据库相关的查询，更新，删除等操作，不能假设数据库里的数据是存在的，  
或者直接操作数据库把数据插入进去，请使用程序插入或者导入数据的方式来准备数据。  
反例： 删除某一行数据的单元测试，在数据库中， 先直接手动增加一行作为删除目标，但是这一行新增数据并不符合业务插入规则， 导致测试结果异常。  
11. 【推荐】和数据库相关的单元测试，可以设定自动回滚机制，不给数据库造成脏数据。或者对单元测试产生的数据有明确的前后缀标识。  
正例： 在 RDC 内部单元测试中，使用 RDC\_UNIT\_TEST\_的前缀标识数据。  
12. 【推荐】对于不可测的代码建议做必要的重构，使代码变得可测，避免为了达到测试要求而书写不规范测试代码。  
13. 【推荐】在设计评审阶段，开发人员需要和测试人员一起确定单元测试范围，单元测试最好覆盖所有测试用例（ UC）。  
14. 【推荐】单元测试作为一种质量保障手段，不建议项目发布后补充单元测试用例，建议在项目提测前完成单元测试。  
15. 【参考】为了更方便地进行单元测试，业务代码应避免以下情况：  
⚫构造方法中做的事情过多。  
⚫存在过多的全局变量和静态方法。  
⚫存在过多的外部依赖。  
⚫存在过多的条件语句。  
说明： 多层条件语句建议使用卫语句、策略模式、状态模式等方式重构。  
16. 【参考】不要对单元测试存在如下误解：  
⚫那是测试同学干的事情。本文是开发手册，凡是本文内容都是与开发同学强相关的。  
⚫单元测试代码是多余的。 汽车的整体功能与各单元部件的测试正常与否是强相关的。  
⚫单元测试代码不需要维护。一年半载后，那么单元测试几乎处于废弃状态。  
⚫单元测试与线上故障没有辩证关系。好的单元测试能够最大限度地规避线上故障

整个思路: