单元测试

针对类中某一个方法进行验证是否正确的过程，单元指`独立的粒子`。

1. 使用单元测试：
    1. 提高开发效率
    2. 为了测试改善代码设计
    3. 单元测试用例需要对某一段业务逻辑验证，深度了解业务流程
2. Android测试分为三方面：
    1. 单元测试（Junit4|Mockito|PowerMockito|Roboletric）
    2. UI测试（Espresso、UI Automator）
    3. 压力测试（Monkey）
3. 单元测试测试目标：列出想要测试覆盖的正常、异常情况，进行测试验证；性能测试，如某算法耗时

Junit4

Junit4是Java标准测试库，利用`Java5.Annotation`特性简化测试用例的编写

junit运行在**JVM**上，所以只能测试纯java，若要测试依赖android库的代码，可以用mockito**隔离依赖**

1. Junit4框架使用到的重要注解

```
@Test // 指明这是一个测试方法，可接受两个参数，param1：预期错误expected，param2：超时时间timeout
    eg: @Test(expected = IndexOutOfBoundsException.class) | @Test(timeout = 1000)
@Before // 在所有测试方法之前执行，用于准备测试环境（初始化类|读输入流等），一个测试类中每个@Test方法执行都会触发一次调用
@After  // 在所有测试方法之后执行，用于清理测试环境数据，一个测试类中每个@Test方法执行都会触发一次调用
@BeforeClass    // 在该类的所有测试方法和@Before方法之前执行(修饰的方法必须是静态的)，用于做耗时的初始化工作(连接DB)
@AfterClass // 在该类的所有测试方法和@After方法之后执行(修饰的方法必须是静态的)，用于清理数据(断开DB)
@Ignore // 忽略当前测试方法，常用于测试方法尚未准备就绪或耗时过多
```

2. 主要测试方法——断言

```
assertEquals(expected, actual); // 判断2个值是否相等，相等则通过测试
assertEquals(expected, actual, tolerance);  // tolereance：偏差容忍值
// 上述每个方法都有一个重载方法可加一个String参数，表示如果验证失败则将该String作为失败的结果报告
assertThat(expected, Matchers.greaterThan(1));
```

3. 单元测试代码存储位置

```
app/src
    - androidTest/java  仪器化单元测试、UI测试
    - main/java 业务代码
    - test/java 本地单元测试
```

Robolectric

实现一套JVM能运行的Android代码，然后在unit test运行的时候去截取android相关的代码调用，然后转到自己实现的代码去执行这个调用的过程

参考文档：[单元测试框架Robolectric的学习使用]

1. 测试类配置
    1. @config配置

```
@Config(sdk = 30)   // 会根据manifest文件配置的targetSdkVersion选择运行测试代码的SDK版本，指定sdk来运行测试用例
@Config(application = MyApplication.class)  // 根据manifest文件配置的Application配置去实例化一个Application类，测试用例中重新指定
@Config(manifest = "some/build/path/AndroidManifest.xml",
        assetDir = "some/build/path/assetDir",
        resourceDir = "some/build/path/resourceDir")    // 配置manifest、resource和assets路径
 @Config(qualifiers = "zh-rCN") // 加载特定的资源文件
```

2. Shadow

    通过一套测试API扩展了Android framework，这些API提供了额外的可配置性，并提供了对测试有用的Android组件的内部状态和历史的访问性。这种访问性就是通过Shadow类（影子类）来实现

    可以使用`Shadows.shadowOf()`方法访问测试API

    通过`ShadowActivity、ShadowDialog、ShadowToast、ShadowApplication`这些Shadow类来模拟Android系统的真实行为，当这些Android系统类被创建的时候，Robolectric会查找对应的Shadow类并创建一个Shadow类对象与原始类对象关联。每当系统类的方法被调用的时候，Robolectric会保证Shadow对应的方法会调用。每个Shadow对象都可以修改或扩展Android操作系统中相应类的行为。因此我们可以用Shadow类的相关方法对Android相关的对象进行测试。

3. 验证生命周期

    可以使用`ActivityController|DialogController`等Controller类让对应`Activity|Dialog`执行相应的生命周期方法，模拟各种生命周期的变化，但要遵循组件的生命周期规律

4. 验证DelayedRunnable

    通过`ShadowLooper.runUiThreadTasksIncludingDelayedTasks()|ShadowLooper.runMainLooperOneTask()`方法使所有UI线程上的延时任务即刻发生

```
@Override
public void onClick(View v) {
    switch(v.getId()) {
        case R.id.btn_login:
            mLoginBtn.postDelayed(new Runnable() {
                @Override
                public void run() {
                    mLoginBtn.setText("test");
                }
            }, 500);
    }
}

@Test
public void testPostRunnable() {
```

```
    MainActivity mainActivity =
Robolectric.setUpActivity(MainActivity.class);
    Assert.assertNotNUll(mainActivity);
    Button btn = mainActivity.findViewById(R.id.btn_login);
    btn.performClick();
    ShadowLooper.runUiThreadTasksIncludingDelayedTasks();
    // ShadowLooper.runMainLooperOneTask(); 使UI线程上所有延时任务即刻发生
    Assert.assertEquals("test", btn.getText());
}
```

5. 隔离Application

用`RuntimeEnvironment.application`的实际上会去创建应用真实的Application类，而一般在实际的Application类的`oncreate()`方法中我们会去初始化第三方的库，这可能导致运行测试方法报错。解决方法是为测试类配置一个空的Application类，通过`@Config`指定：

```
public class RobolectricApp extends Application {
    @Override
    public void onCreate() {
        super.onCreate();
    }
}
// Test
@Config(application = RobolectricApp.class)
```

调用`ShadowApplication.getInstance().getApplicationContext()`得到的是和`RuntimeEnvironment.application`是相同的实例

6. 测试网络异步请求

是需要通过`Robolectric.getForegroundThreadScheduler()`获取`Scheduler`来驱动主线程去轮询消息队列，这样使得我们最终可以在测试方法中调用网络请求之后进行断言操作，否则断言结果一定不通过，因为测试方法是同步的，跑完测试方法的时候，网络请求还没有结束。

```
@Test
public void testGetMusicChannelsSuccess() throws Exception {
    MusicChannelActivity activity =
Robolectric.setupActivity(MusicChannelActivity.class);
    //等待接口请求完毕 再往下执行
    waitAfterAsyncRequest();
    TextView resultTextView = activity.findViewById(R.id.tv_result);
    Assert.assertEquals("请求成功", resultTextView.getText().toString());
}

public void waitAfterAsyncRequest() throws Exception {
    //获取主线程的消息队列的调度者，通过它可以知道消息队列的情况并驱动主线程主动轮询消息队
列
    Scheduler scheduler = Robolectric.getForegroundThreadScheduler();
    //因为网络请求是在异步线程中进行，过一段时间请求完毕才会通知主线程
    //所以在这里进行等待，直到消息队列里存在消息
    while (!scheduler.areAnyRunnable()) {
        Thread.sleep(500);
    }
    //轮询消息队列，这样就会在主线程进行通知
    scheduler.runOneTask();
}
```

```
@Test
public void testGetMusicChannelsSuccess() throws Exception {
    MusicChannelActivity activity =
Robolectric.setupActivity(MusicChannelActivity.class);
    MusicChannelPresenter presenter = new MusicChannelPresenter(activity);
    presenter.requestMusicChannels();
    //等待接口请求完毕 再往下执行
    waitAfterAsyncRequest();
    TextView resultTextView = activity.findViewById(R.id.tv_result);
    Assert.assertEquals("请求成功", resultTextView.getText().toString());
    // TODO: 2019/4/29 presenter中增加public方法获取响应结果的实体类对象进行验证
}
```

7. 实践建议
    1. 不要在Robolectric单元测试中去模拟layout的inflate操作，你应该直接测试你的Activity与布局的**交互操作**，以验证是否设置了正确的点击事件回调，而不是去mock `LayoutInflater` 或者模拟View类。
    2. 非静态方法使用`Whitebox.invokeMethod(context, method, params)`，静态方法使用`ClassName.method()`

Mockito

`Mockito` 的底层原理是使用 `cglib` 动态生成一个 **代理类对象**，拥有对象的所有方法和属性，会覆盖整个对象，该代理在**没有配置/指定行为[stub]**的情况下，只能返回默认值。`mock` 实例默认的会给所有的方法添加基本实现：返回 `null` 或空集合，或者 `0` 等基本类型的值。这取决于方法返回类型，如 `int` 会返回 `0`，布尔值返回 `false`，对于非基本数据类型会返回 `null`。

> By default, for all methods that return a value, a mock will return either null, a primitive/primitive wrapper value, or an empty collection, as appropriate. For example 0 for an int/Integer and false for a boolean/Boolean.

1. 使用
    1. `mock` mock一个接口或类
    2. `stub` 打桩功能，使方法调用返回期望的值，把所需的测试数据塞进对象中，适用于基于状态的（state-based）测试，关注的是输入和输出。
        - 对于 static 和 final 方法， Mockito 无法对其 when(...).thenReturn(...) 操作。
        - 当我们连续两次为同一个方法使用 stub 的时候，他只会只用最新的一次。

        ```
        Mockito.when(studentService.findAll()).thenReturn(students);    // `when thenReturn`会真实调用函数,再把结果改变
        Mockito.doReturn(students).when(studentService.findAll());  // `doReturn when`不会真的去调用函数，直接把结果改变 （如果when中是私有方法，依旧会去执行这个方法，不会直接跳过）
        ```

        对于同一个方法，如果我们想让其在 **多次调用** 中分别 **返回不同** 的数值，那么就可以使用**存根连续调用**

```
when(mock.someMethod("some arg")).thenThrow(new
RuntimeException()).thenReturn("foo");
mock.someMethod("some arg");    // First call: throws runtime exception
System.out.println(mock.someMethod("some arg"));    // Second call:
prints "foo"
System.out.println(mock.someMethod("some arg"));    // Any consecutive
call: prints "foo" as well (last stubbing wins).
```

3. `verify` 验证方法是否被调用、被调用次数，测试不关心返回结果，而是侧重方法有否被正确的参数调用过，这时候就应该使用验证方法了，**行为测试**

```
verify(mock).someMethod(anyInt(), anyString(), eq("third argument"));
// 自定义错误校验输出信息
// Will print a custom message on verification failure
verify(mock, description("This will print on failure")).someMethod();
// Will work with any verification mode
verify(mock, times(2).description("someMethod should be called
twice")).someMethod();
```

   - never() 没有被调用，相当于 times(0)
   - times(n) 被调用n次
   - atLeast(N) 至少被调用 N 次
   - atLeastOnce() 相当于 atLeast(1)
   - atMost(N) 最多被调用 N 次

4. `argument matches` 在`stubbing|verify`时模拟传入的参数。要stubbing或者verify的方法有参数，但是不关心输入的具体内容。只是完成打桩或者验证。这时候就可以用Mockito提供的argument matchers 机制。

   - 如果使用了参数匹配器，那么所有的参数都需要提供一个参数匹配器。

```
Mockito.when(studentService.findById(anyLong())).thenReturn(student)
;   // anyLong(): 打桩时传入任意Long型参数
// Below is incorrect - exception will be thrown because third
argument is given without an argument matcher.
verify(mock).someMethod(anyInt(), anyString(), "third argument");
```

2. `mock & spy`

   1. 区别

      1. `mock` 对象对于未指定处理规则的调用会按方法返回值类型返回该类型的默认值（如int、long则返回0，boolean则返回false，对象则返回null，void则什么都不做）
      2. `mock` 对该对象私有方法的调用无法进行模拟，会调用`真实方法`
      3. `spy` 对该对象所有方法的调用在未指定处理规则时都直接调用`真实方法`
      4. 如果使用`mock`进行对象的局部`mock`，通过 `doCallRealMethod() | thenCallRealMethod()` 方法可调用真实方法。

```
Foo mock = mock(Foo.class);
when(mock.someMethod()).thenCallRealMethod();
```

   2. `spy`：修改某个真实对象的`某些`方法的行为特征，而不改变他的基本行为特征

`spy` 并不是 **真实对象** 的 **代理**。相反的，它对传递过来的 **真实对象** 进行 **克隆**。所以，对 **真实对象** 的任何操作，`spy` 对象并不会感知到。同理，对 `spy` 对象的任何操作，也不会影响到 **真实对象**。

3. 监视真实对象

   mock对象没有`stub`的话，结果就会返回 **空类型**。而如果使用 **特务对象**（`spy`），那么对于没有存根的行为，它会调用原来对象的方法。可以把 `spy` 想象成局部`mock`。

   - 如果我们定制了一个方法A后，再下一个测试方法中又想调用真实方法，那么只需在方法A被调用前，调用`Mockito.reset(spyObject);`就行了。
   - 由于`spy`是局部`mock`，所以有时候使用`when(Object)`时，无法做到存根作用。此时，就可以考虑使用`doReturn() | Answer() | Throw()`这类方法进行存根

```
List list = new LinkedList();
List spy = spy(list);

// Impossible: real method is called so spy.get(0) throws
IndexOutOfBoundsException (the list is yet empty)
when(spy.get(0)).thenReturn("foo");
// You have to use doReturn() for stubbing
doReturn("foo").when(spy).get(0);

// Optionally, you can stub out some methods:
when(spy.size()).thenReturn(100);
// Use the spy calls *real* methods
spy.add("one");
spy.add("two");
// Prints "one" - the first element of a list
System.out.println(spy.get(0));
// Size() method was stubbed - 100 is printed
System.out.println(spy.size());
// Optionally, you can verify
verify(spy).add("one");
verify(spy).add("two");
```

3. 模拟static类

   可替代PowerMock的@PrepareForTest()和PowerMockito.mockStatic(ClassName.class);，解决PowerMock|Mockito版本不兼容导致的问题

```
MockedStatic<ClassName> mocked = Mockito.mockStatic(ClassName.class);
mocked.when(() -> ClassName.someMethod(params).thenReturn(xxx); //
ClassName.someMethod()含参数时的调用语法
mocked.when(ClassName::noArgMethod).thenReturn(xxx);    //
ClassName.noArgMethod()无参数时的调用语法
mocked.close(); // MockedStatic使用完毕后需要close()，否则报错：Static mocking is
already registered in the current thread
```

PowerMock

PowerMock对比Mockito的优势在于可以mock静态类、私有方法。（Mockito-incline:3.4后也支持mock静态类、私有方法）

1. 使用

   1. mock静态方法

```
// 1.@PrepareForTest(Class.class)，指定要测试的静态类
@PrepareForTest(Static.class)
// 2.使用PowerMockito.mockStatic()对静态类进行mock
PowerMockito.mockStatic(Static.class);
// 3.后续使用PowerMockito.when().thenReturn()进行stub
Mockito.when(Static.firstStaticMethod(param)).thenReturn(value);
```

2. verify

```
// 验证静态类方法行为     注：对于每一个方法的验证，都需要去调用
verifyStatic(Static.class)方法。
PowerMockito.verifyStatic(Static.class);     // 1.调用
PowerMockito.verifyStatic(Static.class) 开始验证行为
PowerMockito.verifyStatic(Static.class, Mockito.times(1));
Static.staticMethod(param); // 2.调用 Static.class 的静态方法进行验证
// 验证private方法行为
PowerMockito.verifyPrivate(tested).invoke("privateMethodName",
arguments);
```

3. doThrow

```
// 非private方法
PowerMockito.doThrow(new ArrayStoreException("Mock
error")).when(StaticService.class);
StaticService.executeMethod();
// 对final类方法执行相同操作
PowerMockito.doThrow(new ArrayStoreException("Mock
error")).when(myFinalMock).myFinalMethod();
// private方法
PowerMockito.when(tested, "methodToExpect",
argument).thenReturn(myReturnValue);
```

4. 局部mock

通过PowerMockito.spy()对方法进行局部mock

```
// 有时，你并不能使用默认的 when(..)方法为spies打桩
List list = new LinkedList();
List spy = spy(list);
//Impossible: real method is called so spy.get(0) throws
IndexOutOfBoundsException (the list is yet empty)
when(spy.get(0)).thenReturn("foo");
//You have to use doReturn() for stubbing
doReturn("foo").when(spy).get(0);
```

5. mock私有方法

```
MockDemo spy = PowerMockito.spy(new MockDemo());
PowerMockito.doReturn(new ArrayList<>()).when(spy, "privateMethod",
Mockito.any());
```

6. mock单例类

```
PowerMockito.mockStatic(SchoolManageProxy.class);
// Powermock mock出单例类
SchoolManageProxy mockSchoolManageProxy =
PowerMockito.mock(SchoolManageProxy.class);
// 给单例类的getInstance方法打桩

PowerMockito.when(SchoolManageProxy.getInstance()).thenReturn(mockSchool
ManageProxy);
// 对mock类queryPerson的方法打桩
when(mockSchoolManageProxy.queryPerson(anyList())).thenReturn(Collection
s.emptyList());
```

7. 跳过方法执行

```
PowerMockito.suppress(PowerMockito.method(StudentService.class,
"highlightResult"));
```

8. stub构造函数

```
PowerMockito.whenNew(File.class).withArguments(direcPath).thenReturn(moc
kDirectory);
verifyNew(File.class).withArguments(direcPath);
```

9. Stubbing with callbacks

```
when(mock.someMethod(anyString())).thenAnswer(new Answer() {
     Object answer(InvocationOnMock invocation) {
         Object[] args = invocation.getArguments();
         Object mock = invocation.getMock();
         return "called with arguments: " + args;
     }
 });

 //the following prints "called with arguments: foo"
 System.out.println(mock.someMethod("foo"));
```

- 利用 `InvocationOnMock` 提供的方法可以获取 mock 方法的调用信息。下面是它提供的方法：
  getArguments() 调用后会以 Object 数组的方式返回 mock 方法调用的参数。
  getMethod() 返回 java.lang.reflect.Method 对象
  getMock() 返回 mock 对象
  callRealMethod() 真实方法调用，如果 mock 的是接口它将会抛出异常