JUnit Mockito unit testing

deepwaterooo

2021 年 12 月 21 日

目录

1 Mockito (AndroidUT)

- https://blog.csdn.net/qq_17766199/article/details/78450007
- 添加 Mockito 的依赖：

```
//mockito
testCompile "org.mockito:mockito-core:2.8.9"
androidTestCompile "org.mockito:mockito-android:2.8.9"
```

1.1 基本步骤

1.1.1 初始化注入

- 首先我们在 setUp 函数中进行初始化：

```
private ArrayList mockList;
@Before
public void setUp() throws Exception {
    //MockitoAnnotations.initMocks(this);
    //mock creation
    mockList = mock(ArrayList.class);
}
```

- 当然，你也这样这样进行注入：

```
@Mock
private ArrayList mockList;
@Before
public void setUp() throws Exception {
    MockitoAnnotations.initMocks(this); // 在 base class 中或者初始化的地方配置
}
```

- initMocks(this) 后，就可以通过 @Mock 注解直接使用 mock 对象。
- 使用 JUnit4 的 rule 来配置：

```
@Rule
public MockitoRule rule = MockitoJUnit.rule().strictness(Strictness.STRICT_STUBS);
```

- 以上 2 种方式可以达到同样的效果。
- 下面列几种常用又比较典型的用法
- 一个最普通的例子

```
  @Test
    public void sampleTest1() throws Exception {
        //使用 mock 对象执行方法
        mockList.add("one");
        mockList.clear();

        //检验方法是否调用
        verify(mockList).add("one");
        verify(mockList).clear();
    }
```

- 我们可以直接调用 mock 对象的方法，比如 ArrayList.add() 或者 ArrayList.clear()，然后我们通过 verify 函数进行校验。

1.1.2 直接 **mock** 接口对象

- 正常来讲我们想要一个接口类型的对象，首先我们需要先实例化一个对象并实现，其对应的抽象方法，但是有了 mock，我们可以直接 mock 出一个接口对象：

```
@Test
public void sampleTest2() throws Exception {
    //我们可以直接 mock 一个借口，即使我们并未声明它
    MVPContract.Presenter mockPresenter = mock(MVPContract.Presenter.class);
    when(mockPresenter.getUserName()).thenReturn("qingmei2"); //我们定义，当 mockPresenter 调用 getUserName() 时，返回 qingme
    String userName = mockPresenter.getUserName();
```

```
    verify(mockPresenter).getUserName(); //校验 是否 mockPresenter 调用了 getUserName() 方法
    Assert.assertEquals("qingmei2", userName); //断言 userName 为 qingmei2

//        verify(mockPresenter).getPassword();  //校验 是否 mockPresenter 调用了 getPassword() 方法
    String password = mockPresenter.getPassword();  //因为未定义返回值，默认返回 null
    verify(mockPresenter).getPassword();
    Assert.assertEquals(password, null);
}
```

1.1.3 参数匹配器

```
@Test
public void argumentMatchersTest3() throws Exception {
    when(mockList.get(anyInt())).thenReturn(" 不管请求第几个参数 我都返回这句");
    System.out.println(mockList.get(0));
    System.out.println(mockList.get(39));

    //当 mockList 调用 addAll() 方法时，「匹配器」如果传入的参数 list size==2，返回 true；
    when(mockList.addAll(argThat(getListMatcher()))).thenReturn(true);

    //根据 API 文档，我们也可以使用 lambda 表达式：「匹配器」如果传入的参数 list size==3，返回 true；
//        when(mockList.addAll(argThat(list -> list.size() == 3))).thenReturn(true);
    //我们不要使用太严格的参数 Matcher，也许下面会更好
//        when(mockList.addAll(argThat(notNull())));
    boolean b1 = mockList.addAll(Arrays.asList("one", "two"));
    boolean b2 = mockList.addAll(Arrays.asList("one", "two", "three"));
    verify(mockList).addAll(argThat(getListMatcher()));
    Assert.assertTrue(b1);
    Assert.assertTrue(!b2);
}
private ListOfTwoElements getListMatcher() {
    return new ListOfTwoElements();
}
/**
 * 匹配器，用来测试 list 是否有且仅存在两个元素
 */
class ListOfTwoElements implements ArgumentMatcher<List> {
    public boolean matches(List list) {
        return list.size() == 2;
    }

    public String toString() {
        //printed in verification errors
        return "[list of 2 elements]";
    }
}
```

- 对于一个 Mock 的对象，有时我们需要进行校验，但是基础的 API 并不能满足我们校验的需要，我们可以自定义 Matcher，比如案例中，我们自定义一个 Matcher, 只有容器中两个元素时，才会校验通过。

1.1.4 验证方法的调用次数

```
 /**
   * 我们也可以测试方法调用的次数
   * https://static.javadoc.io/org.mockito/mockito-core/2.8.9/org/mockito/Mockito.html#exact_verification
   *
   * @throws Exception
   */
@Test
public void simpleTest4() throws Exception {
    mockList.add("once");

    mockList.add("twice");
    mockList.add("twice");

    mockList.add("three times");
```

```
    mockList.add("three times");
    mockList.add("three times");

    verify(mockList).add("once");  //验证 mockList.add("once") 调用了一次 - times(1) is used by default
    verify(mockList, times(1)).add("once");//验证 mockList.add("once") 调用了一次

    //调用多次校验
    verify(mockList, times(2)).add("twice");
    verify(mockList, times(3)).add("three times");

    //从未调用校验
    verify(mockList, never()).add("four times");

    //至少、至多调用校验
    verify(mockList, atLeastOnce()).add("three times");
    verify(mockList, atMost(5)).add("three times");
//        verify(mockList, atLeast(2)).add("five times"); //这行代码不会通过
}
```

1.1.5 抛出异常

```
/**
  * 异常抛出测试
  * https://static.javadoc.io/org.mockito/mockito-core/2.8.9/org/mockito/Mockito.html#stubbing_with_exceptions
  */
@Test
public void throwTest5() {
    doThrow(new NullPointerException("throwTest5. 抛出空指针异常")).when(mockList).clear();
    doThrow(new IllegalArgumentException(" 你的参数似乎有点问题")).when(mockList).add(anyInt());

    mockList.add("string");//这个不会抛出异常
    mockList.add(12);//抛出了异常，因为参数是 Int
    mockList.clear();
}
```

- 如案例所示，当 mockList 对象执行 clear 方法时，抛出空指针异常，当其执行 add 方法，且传入的参数类型为 int 时，抛出非法参数异常。

1.1.6 校验方法执行顺序

```
/**
  * 验证执行执行顺序
  * https://static.javadoc.io/org.mockito/mockito-core/2.8.9/org/mockito/Mockito.html#in_order_verification
  *
  * @throws Exception
  */
@Test
public void orderTest6() throws Exception {
    List singleMock = mock(List.class);

    singleMock.add("first add");
    singleMock.add("second add");

    InOrder inOrder = inOrder(singleMock);

    //inOrder 保证了方法的顺序执行
    inOrder.verify(singleMock).add("first add");
    inOrder.verify(singleMock).add("second add");

    List firstMock = mock(List.class);
    List secondMock = mock(List.class);

    firstMock.add("first add");
    secondMock.add("second add");

    InOrder inOrder1 = inOrder(firstMock, secondMock);

    //下列代码会确认是否 firstmock 优先 secondMock 执行 add 方法
    inOrder1.verify(firstMock).add("first add");
```

```
    inOrder1.verify(secondMock).add("second add");
}
```

- 有时候我们需要校验方法执行顺序的先后，如案例所示，inOrder 对象会判断方法执行顺序，如果顺序不对，该测试案例 failed。

1.1.7 确保 mock 对象从未进行过交互：verifyZeroInteractions && verifyNoMoreInteractions

```
/**
  * 确保 mock 对象从未进行过交互
  * https://static.javadoc.io/org.mockito/mockito-core/2.8.9/org/mockito/Mockito.html#never_verification
  *
  * @throws Exception
  */
@Test
public void noInteractedTest7() throws Exception {
    List firstMock = mock(List.class);
    List secondMock = mock(List.class);
    List thirdMock = mock(List.class);

    firstMock.add("one");

    verify(firstMock).add("one");

    verify(firstMock, never()).add("two");

    firstMock.add(thirdMock);
    // 确保交互 (interaction) 操作不会执行在 mock 对象上
//        verifyZeroInteractions(firstMock); //test failed, 因为 firstMock 和其他 mock 对象有交互
    verifyZeroInteractions(secondMock, thirdMock);   // 0 交互： test pass
}
```

- 另一个小例子

```
 @Test
    public void testMock6() {
        List list = mock(List.class);
        // 验证 mock 对象没有产生任何交互，也即没有任何方法调用
        verifyZeroInteractions(list);

        List list2 = mock(List.class);
        list2.add("one");
        list2.add("two");
        verify(list2).add("one");
        // 验证 mock 对象是否有被调用过但没被验证的方法。这里会测试不通过，list2.add("two") 方法没有被验证过
        verifyNoMoreInteractions(list2); //
    }
```

- 可能是因为水平有限，笔者很少用到这个 API（好吧除了学习案例中用过其他基本没怎么用过），不过还是敲一遍，保证有个基础的印象。

1.1.8 简化 mock 对象的创建

```
/**
  * 简化 mock 对象的创建，请注意，一旦使用 @Mock 注解，一定要在测试方法调用之前调用 (比如 @Before 注解的 setUp 方法)
  * MockitoAnnotations.initMocks(testClass);
  */
@Mock
List mockedList;
@Mock
User mockedUser;
@Test
public void initMockTest8() throws Exception {
    mockedList.add("123");
```

```
    mockedUser.setLogin("qingmei2");
}
```

- 注释写的很明白了，不赘述

1.1.9 重置 mocks

```
@Test
public void testMock12() {
    List list = mock(List.class);
    when(list.size()).thenReturn(100);
    // 打印出"100"
    System.out.println(list.size());
    // 充值 mock, 之前的交互和 stub 将全部失效
    reset(list);
    // 打印出"0"
    System.out.println(list.size());
}
```

1.1.10 使用 @InjectMocks 在 Mockito 中进行依赖注入

- 我们也可以使用 @InjectMocks 注解来创建对象，它会根据类型来注入对象里面的成员方法和变量。假定我们有 ArticleManager 类:

```
public class ArticleManager {
    private ArticleDatabase database;
    private User user;
    public ArticleManager(User user, ArticleDatabase database) {
        super();
        this.user = user;
        this.database = database;
    }
    public void initialize() {
        database.addListener(new ArticleListener());
    }
}
```

- 这个类可以通过 Mockito 构建，它的依赖可以通过模拟对象来实现，如下面的代码片段所示:

```
@RunWith(MockitoJUnitRunner.class)
public class ArticleManagerTest  {
       @Mock User user;
       @Mock ArticleDatabase database;
       @Mock ArticleCalculator calculator;
       @Spy private UserProvider userProvider = new ConsumerUserProvider();

       @InjectMocks private ArticleManager manager; //

       @Test public void shouldDoSomething() {
           // calls addListener with an instance of ArticleListener
           manager.initialize();
           // validate that addListener was called
           verify(database).addListener(any(ArticleListener.class));
       }
}
```

- 创建一个实例 ArticleManager 并将其注入到它中
- 更多的详情可以查看:
  - http://docs.mockito.googlecode.com/hg/1.9.5/org/mockito/InjectMocks.html

1.1.11 方法连续调用测试

```
/**
  * 方法连续调用的测试
  * https://static.javadoc.io/org.mockito/mockito-core/2.8.9/org/mockito/Mockito.html#stubbing_consecutive_calls
  */
@Test
public void continueMethodTest9() throws Exception {
    when(mockedUser.getName())
            .thenReturn("qingmei2")
            .thenThrow(new RuntimeException(" 方法调用第二次抛出异常"))
            .thenReturn("qingemi2 第三次调用");

    //另外一种方式
    when(mockedUser.getName()).thenReturn("qingmei2 1", "qingmei2 2", "qingmei2 3");
    String name1 = mockedUser.getName();
    try {
        String name2 = mockedUser.getName();
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
    String name3 = mockedUser.getName();
    System.out.println(name1);
    System.out.println(name3);
}
```

- 有用，但不重要，学习一下加深印象。

1.1.12 为回调方法做测试

```
/**
  * 为回调方法做测试
  * https://static.javadoc.io/org.mockito/mockito-core/2.8.9/org/mockito/Mockito.html#answer_stubs
  */
@Test
public void callBackTest() throws Exception {
    when(mockList.add(anyString())).thenAnswer(new Answer<Boolean>() {
        @Override
        public Boolean answer(InvocationOnMock invocation) throws Throwable {
            Object[] args = invocation.getArguments();
            Object mock = invocation.getMock();
            return false;
        }
    });
    System.out.println(mockList.add(" 第 1 次返回 false"));
    //lambda 表达式
    when(mockList.add(anyString())).then(invocation -> true);
    System.out.println(mockList.add(" 第 2 次返回 true"));

    when(mockList.add(anyString())).thenReturn(false);
    System.out.println(mockList.add(" 第 3 次返回 false"));
}
```

- 在 Mockito 的官方文档中，这样写道：
  - 在最初的 Mockito 里也没有这个具有争议性的特性。我们建议使用 thenReturn() 或 thenThrow() 来打桩。这两种方法足够用于测试或者测试驱动开发。
  - 实际上笔者日常开发中也不怎么用到这个特性。

1.1.13 拦截方法返回值（常用）

```
/**
  * doReturn()、doThrow()、doAnswer()、doNothing()、doCallRealMethod() 系列方法的运用
  * https://static.javadoc.io/org.mockito/mockito-core/2.8.9/org/mockito/Mockito.html#do_family_methods_stubs
  */
@Test
```

```
public void returnTest() throws Exception {
    //返回值为 null 的函数，可以通过这种方式进行测试

    doAnswer(invocation -> {
        System.out.println(" 测试无返回值的函数");
        return null;
    }).when(mockList).clear();

    doThrow(new RuntimeException(" 测试无返回值的函数-> 抛出异常"))
            .when(mockList).add(eq(1), anyString());

    doNothing().when(mockList).add(eq(2), anyString());

//        doReturn("123456").when(mockList).add(eq(3), anyString());    //不能把空返回值的函数与 doReturn 关联

    mockList.clear();
    mockList.add(2, "123");
    mockList.add(3, "123");
    mockList.add(4, "123");
    mockList.add(5, "123");

    //但是请记住这些 add 实际上什么都没有做，mock 对象中仍然什么都没有
    System.out.print(mockList.get(4));
}
```

- 我们不禁这样想，这些方法和 when(mock.do()).thenReturn(foo) 这样的方法有什么区别，或者说，这些方法有必要吗？
- 答案是肯定的，因为在接下来介绍的新特性 Spy 中，该方法起到了至关重要的作用。
- 可以说，以上方法绝对是不可代替的。
- Mockito 框架不支持 mock 匿名类、final 类、static 方法、private 方法。
- 而 PowerMock 框架解决了这些问题。关于 PowerMock，下一篇会讲到

1.1.14 对复杂的 Mock 使用 Answers

- 通过 Answer 可以定义一个复杂的结果对象，虽然 thenReturn 每次都返回一个预定义的值，但是有了 Answer，你可以根据 stubbed 方法的参数来预估响应。如果你的 stubbed 方法调用其中一个参数的函数，或者返回第一个参数以允许方法链的进行，那么这会很有用。后者存在一种静态方法。注意，有不同的方式来配置 Answer：

```
import static org.mockito.AdditionalAnswers.returnsFirstArg;
@Test
public final void answerTest() {
    // with doAnswer():
    doAnswer(returnsFirstArg()).when(list).add(anyString());
    // with thenAnswer():
    when(list.add(anyString())).thenAnswer(returnsFirstArg());
    // with then() alias:
    when(list.add(anyString())).then(returnsFirstArg());
}
```

- 或者如果你需要对你的结果进行回调：

```
@Test
public final void callbackTest() {
    ApiService service = mock(ApiService.class);
    when(service.login(any(Callback.class))).thenAnswer(i -> {
        Callback callback = i.getArgument(0);
        callback.notify("Success");
        return null;
    });
}
```

- 甚至可以模拟像 DAO 这样的持久性服务，但是如果您的 Answers 过于复杂，您应该考虑创建一个虚拟类而不是 mock

```
List<User> userMap = new ArrayList<>();
UserDao dao = mock(UserDao.class);
when(dao.save(any(User.class))).thenAnswer(i -> {
    User user = i.getArgument(0);
    userMap.add(user.getId(), user);
    return null;
});
when(dao.find(any(Integer.class))).thenAnswer(i -> {
    int id = i.getArgument(0);
    return userMap.get(id);
});
```

1.1.15 Spy: 监控真实对象 (重要)

```
/**
  * 监控真实对象
  * https://static.javadoc.io/org.mockito/mockito-core/2.8.9/org/mockito/Mockito.html#spy
  */
@Test
public void spyTest() throws Exception {
   List list = new ArrayList();
   List spyList = spy(list);

   // 当 spyList 调用 size() 方法时，return100
   when(spyList.size()).thenReturn(100);

   spyList.add("one");
   spyList.add("two");

   System.out.println("spyList 第一个元素" + spyList.get(0));
   System.out.println("spyList.size = " + spyList.size());

   verify(spyList).add("one");
   verify(spyList).add("two");

   // 请注意！下面这行代码会报错！ java.lang.IndexOutOfBoundsException: Index: 10, Size: 2
   // 不可能 : 因为当调用 spy.get(0) 时会调用真实对象的 get(0) 函数，此时会发生异常，因为真实 List 对象是空的
//         when(spyList.get(10)).thenReturn("ten");

   // 应该这么使用

   doReturn("ten").when(spyList).get(9);
   doReturn("eleven").when(spyList).get(10);

   System.out.println("spyList 第 10 个元素" + spyList.get(9));
   System.out.println("spyList 第 11 个元素" + spyList.get(10));

   // Mockito 并不会为真实对象代理函数调用，实际上它会拷贝真实对象。因此如果你保留了真实对象并且与之交互
   // 不要期望从监控对象得到正确的结果。当你在监控对象上调用一个没有被 stub 的函数时并不会调用真实对象的对应函数，你不会在真实对象上

   // 因此结论就是 : 当你在监控一个真实对象时，你想在 stub 这个真实对象的函数，那么就是在自找麻烦。或者你根本不应该验证这些函数。
}
```

- Spy 绝对是一个好用的功能，我们不要滥用，但是需要用到对真实对象的测试操作，spy 绝对是一个很不错的选择。

1.1.16 捕获参数（重要）

```
/**
 * 为接下来的断言捕获参数 (API1.8+)
 * https:// static.javadoc.io/org.mockito/mockito-core/2.8.9/org/mockito/Mockito.html#captors
 */
@Test
public void captorTest() throws Exception {
```

```
    Student student = new Student();
    student.setName("qingmei2");

    ArgumentCaptor<Student> captor = ArgumentCaptor.forClass(Student.class);
    mockList.add(student);
    verify(mockList).add(captor.capture());

    Student value = captor.getValue();

    Assert.assertEquals(value.getName(),"qingmei2");
}

@Data
private class Student {
    private String name;
}
```

- 我们将定义好的 ArgumentCaptor 参数捕获器放到我们需要去监控捕获的地方，如果真的执行了该方法，我们就能通过 captor.getValue() 中取到参数对象，如果没有执行该方法，那么取到的只能是 null 或者基本类型的默认值。
- 小结
  - 本文看起来是枯燥无味的，事实上也确实如此，但是如果想在开发中写出高覆盖率的单元测试，Mockito 强大的功能一定能让你学会之后爱不释手。
  - Mockito 当然也有一定的限制。例如，你不能 mock 静态方法和私有方法。

1.1.17 更多的注解

- 使用注解都需要预先进行配置，怎么配置见 6.2.7 说明
  - @Captor 替代 ArgumentCaptor
  - @Spy 替代 spy(Object)
  - @Mock 替代 mock(Class)
  - @InjectMocks 创建一个实例，其余用 @Mock（或 @Spy）注解创建的 mock 将被注入到用该实例中

1.2 容易概念混淆的几个点

1.2.1 @Mock 与 @Spy 的异同

- Mock 对象只能调用 stubbed 方法，不能调用其真实的方法。而 Spy 对象可以监视一个真实的对象，对 Spy 对象进行方法调用时，会调用真实的方法。
- 两者都可以 stubbing 对象的方法，让方法返回我们的期望值。
- 两者无论是否是真实的方法调用，都可进行 verify 验证。
- 对 final 类、匿名类、java 的基本数据类型是无法进行 mock 或者 spy 的。
- 注意 mockito 是不能 mock static 方法的。

1.2.2 @InjectMocks 与 @Mock 等的区别

- @Mock：创建一个 mock 对象。
- @InjectMocks：创建一个实例对象，然后将 @Mcok 或者 @Spy 注解创建的 mock 对象注入到该实例对象中。

- stackoverflow 上对这个有一个比较形象的解释：

```
@RunWith(MockitoJUnitRunner.class)
public class SomeManagerTest {
    @InjectMocks
    private SomeManager someManager;
    @Mock
    private SomeDependency someDependency; // 该 mock 对象会被注入到 someManager 对象中
    // 你不用向下面这样实例化一个 SomeManager 对象，@InjectMocks 会自动帮你实现
    // SomeManager someManager = new SomeManager();
    // SomeManager someManager = new SomeManager(someDependency);
}
```

1.2.3 when(...).thenReturn() 与 doReturn(...).when(...) 两种语法的异同

- 两者都是用来 stubbing 方法的，大部分情况下，两者可以表达同样的意思，与 Java 里的 do/while、while/do 语句类似。
- 对 void 方法不能使用 when/thenReturn 语法。
- 对 spy 对象要慎用 when/thenReturn，如：

```
List spyList = spy(new ArrayList());
// 下面代码会抛出 IndexOutOfBoundsException
when(spyList.get(0)).thenReturn("foo");
// 这里不会抛出异常
doReturn("foo").when(spyList).get(0);
System.out.println(spyList.get(0));
```

- 个人觉得讨论哪种语法好是没有意义的，推荐使用 doReturn/when 语法，不管是 mock 还是 spy 对象都适用。

1.3 怎样测试异步代码

- 异步无处不在，特别是网络请求，必须在子线程中执行。异步一般用来处理比较耗时的操作，除了网络请求外还有数据库操作、文件读写等等。一个典型的异步方法如下：

```
public class DataManager {
    public interface OnDataListener {
        public void onSuccess(List<String> dataList);
        public void onFail();
    }
    public void loadData(final OnDataListener listener) {
        new Thread(new Runnable() {
                @Override public void run() {
                    try {
                        Thread.sleep(1000);
                        List<String> dataList = new ArrayList<String>();
                        dataList.add("11");
                        dataList.add("22");
                        dataList.add("33");
                        if (listener != null)
                            listener.onSuccess(dataList);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                        if (listener != null)
                            listener.onFail();
                    }
                }
            }).start();
    }
}
```

- 上面代码里开启了一个异步线程，等待 1 秒之后在回调函数里成功返回数据。通常情况下，我们针对 loadData() 方法写如下单元测试：

```
@Test
public void testGetData() {
    final List<String> list = new ArrayList<String>();
    DataManager dataManager = new DataManager();
    dataManager.loadData(new DataManager.OnDataListener() {
            @Override
                public void onSuccess(List<String> dataList) {
                if(dataList != null) {
                    list.addAll(dataList);
                }
            }

            @Override
                public void onFail() {
            }
        });
    Assert.assertEquals(3, list.size());
}
```

- 执行这段测试代码，你会发现永远都不会通过。因为 loadData() 是一个异步方法，当我们在执行 Assert.assertEquals() 方法时，loadData() 异步方法里的代码还没执行，所以 list.size() 返回永远是 0。
- 这只是一个最简单的例子，我们代码里肯定充斥着各种各样的异步代码，那么对于这些异步该怎么测试呢？
- 要解决这个问题，主要有 2 个思路：一是等待异步操作完成，然后在进行 assert 断言；二是将异步操作变成同步操作。

1.3.1 1. 等待异步完成：使用 CountDownLatch

- 前面的例子，等待异步完成实际上就是等待 callback 函数执行完毕，使用 CountDownLatch 可以达到这个目标，不熟悉该类的可自行搜索学习。修改原来的测试用例代码如下：

```
@Test
public void testGetData() {
    final List<String> list = new ArrayList<String>();
    DataManager dataManager = new DataManager();
    final CountDownLatch latch = new CountDownLatch(1);
    dataManager.loadData(new DataManager.OnDataListener() {
            @Override
                public void onSuccess(List<String> dataList) {
                if (dataList != null)
                    list.addAll(dataList);
                // callback 方法执行完毕候，唤醒测试方法执行线程
                latch.countDown();
            }
            @Override
                public void onFail() {
            }
        });
    try {
        // 测试方法线程会在这里暂停，直到 loadData() 方法执行完毕，才会被唤醒继续执行
        latch.await();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    Assert.assertEquals(3, list.size());
}
```

- CountDownLatch 适用场景：

- 1. 方法里有 callback 函数调用的异步方法，如前面所介绍的这个例子。
- 2.RxJava 实现的异步，RxJava 里的 subscribe 方法实际上与 callback 类似，所以同样适用。

- CountDownLatch 同样有它的局限性，就是必须能够在测试代码里调用 countDown() 方法，这就要求被测的异步方法必须有类似 callback 的调用，也就是说异步方法的调用结果必须是通过 callback 调用通知出去的，如果我们采用其他通知方式，例如 EventBus、Broadcast 将结果通知出去，CountDownLatch 则不能实现这种异步方法的测试了。
- 实际上，可以使用 synchronized 的 wait/notify 机制实现同样的功能。我们将测试代码稍微改改如下：

```
@Test
public void testGetData() {
    final List<String> list = new ArrayList<String>();
    DataManager dataManager = new DataManager();
    final Object lock = new Object();
    dataManager.loadData(new DataManager.OnDataListener() {
            @Override
                public void onSuccess(List<String> dataList) {
                if (dataList != null)
                    list.addAll(dataList);
                synchronized (lock)
                    lock.notify();
            }
            @Override
                public void onFail() {
                }
        });
    try {
        synchronized (lock)
            lock.wait();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    Assert.assertEquals(3, list.size());
}
```

- CountDownLatch 与 wait/notify 相比而言，语义更简单，使用起来方便很多。

1.3.2 2. 将异步变成同步

- 下面介绍几种不同的异步实现。

1. 2.1 使用 RxJava

   - RxJava 现在已经被广泛运用于 Android 开发中了，特别是结合了 Rotrofit 框架之后，简直是异步网络请求的神器。RxJava 发展到现在最新的版本是 RxJava2，相比 RxJava1 做了很多改进，这里我们直接采用 RxJava2 来讲述，RxJava1 与之类似。对于前面的异步请求，我们采用 RxJava2 来改造之后，代码如下：

```
public Observable<List<String>> loadData() {
    return Observable.create(new ObservableOnSubscribe<List<String>>() {
            @Override public void subscribe(ObservableEmitter<List<String>> e) throws Exception {
                Thread.sleep(1000);
                List<String> dataList = new ArrayList<String>();
                dataList.add("11");
                dataList.add("22");
                dataList.add("33");
                e.onNext(dataList);
                e.onComplete();
            }
        }).subscribeOn(Schedulers.io()).observeOn(AndroidSchedulers.mainThread());
}
```

- RxJava2 都是通过 subscribeOn(Schedulers.io()).observeOn(AndroidSchedulers.mainThread(
来实现异步的，这段代码表示所有操作都在 IO 线程里执行，最后的结果是在主线程实现
回调的。这里要将异步变成同步的关键是改变 subscribeOn() 的执行线程，有 2 种方式
可以实现：
- 将 subscribeOn() 以及 observeOn() 的参数通过依赖注入的方式注入进来，正常运行时
跑在 IO 线程中，测试时跑在测试方法运行所在的线程中，这样就实现了异步变同步。
- 使用 RxJava2 提供的 RxJavaPlugins 工具类，让 Schedulers.io() 返回当前测试方法运行
所在的线程。

```
@Before
public void setup() {
    RxJavaPlugins.reset();
    // 设置 Schedulers.io() 返回的线程
    RxJavaPlugins.setIoSchedulerHandler(new Function<Scheduler, Scheduler>() {
            @Override
                public Scheduler apply(Scheduler scheduler) throws Exception {
                // 返回当前的工作线程，这样测试方法与之都是运行在同一个线程了，从而实现异步变同步。
                return Schedulers.trampoline();
            }
        });
}
@Test
public void testGetDataAsync() {
    final List<String> list = new ArrayList<String>();
    DataManager dataManager = new DataManager();
    dataManager.loadData().subscribe(new Consumer<List<String>>() {
            @Override
                public void accept(List<String> dataList) throws Exception {
                if(dataList != null) {
                    list.addAll(dataList);
                }
            }
        }, new Consumer<Throwable>() {
            @Override
                public void accept(Throwable throwable) throws Exception {

            }
        });
    Assert.assertEquals(3, list.size());
}
```

2. 2.2 new Thread() 方式做异步操作
    - 如果你的代码里还有直接 new Thread() 实现异步的方式，唯一的建议是赶紧去使用其他
的异步框架吧。
3. 2.3 使用 Executor
    - 如果我们使用 Executor 来实现异步，可以使用依赖注入的方式，在测试环境中将一个同
步的 Executor 注入进去。实现一个同步的 Executor 很简单。

```
Executor executor = new Executor() {
    @Override
    public void execute(Runnable command) {
        command.run();
    }
};
```

4. 2.4 AsyncTask
    - 现在已经不推荐使用 AsyncTask 了,如果一定要使用,建议使用 AsyncTask.executeOnExecutor(E
exec, Params... params) 方法，然后通过依赖注入的方式，在测试环境中将同步的 Ex-
ecutor 注入进去。

- 小结
  - 本文主要介绍了针对异步代码进行单元测试的 2 种方法：一是等待异步完成，二是将异步变成同步。前者需要写很多侵入性代码，通过加锁等机制来实现，并且必须符合 callback 机制。其他还有很多实现异步的方式，例如 IntentService、HandlerThread、Loader 等，综合比较下来，使用 RxJava2 来实现异步是一个不错的方案，它不仅功能强大，并且在单元测试中能毫无侵入性的将异步变成同步，在这里强烈推荐！