单元测试难？来试试这些套路

作者：阿里技术 2020-11-04 16:34:45

测试不应该是一门很高大尚的技术，应该是我们技术人的基本功。但现在好像慢慢地，单元测试已经脱离了基本功的范畴。笔者曾经在不同团队中推过单元测试，要求过覆盖率，但发现实施下去很难。后来在不停地刻意练习后，发现阻碍写UT的只是笔者的心魔，并不是时间和项目的问题。在经过一些项目的实践后，也是有了一些自己的理解和实践，希望和大家分享一下，和大家探讨下如何克服“单元测试”的心魔。

内功

前人们在单元测试方面的研究很多，有很多的方法论，我们可以拿来即用。我简单介绍两个方法论，一个概念。希望大家可以查阅更多的资料，凝聚自己的内功心法。

TDD

Test Driven Development，也被认为是Test Driven Design，我们这里按第一种定义来聊。TDD一改以往的破坏性测试的思维方式，测试在先、编码在后，更符合“缺陷预防”的思想。简单来说，TDD的流程是“红-绿-重构”三个步骤的循环往复。

- 红：测试先行，现在还没有任何实现，跑UT的时候肯定不过，测试状态是红灯。编译失败也属于“红”的一种情况。
- 绿：当我们用最快，最简单的方式先实现，然后跑一遍UT，测试会通过，变成“绿”的状态。
- 重构：看一下系统中有没有要重构的点，重构完，一定要保证测试是“绿”的。

业界有很多TDD的呼声，也有TDD已死的文章。方法本来没有对错，只有优劣，我们要辩证地来看。只能说TDD不是一个银弹，不能解决所有问题。以笔者自己的经验，TDD比较适用于输入输出很明确的CASE，很多时候我们在摸索一种新的模式的时候，可能并不太适用。

如果你和前端已经商议好了接口的出参、入参，可以尝试一下TDD，一种新的思路，新的思想。

BDD

严格来说BDD是TDD衍生出来的一个小分支。但也可以用于一些不同维度的东西。概念大家自行寻找资料。这里讲一下BDD的一点实践经验。直接上代码：

复制

```
@RunWith(SpringBootRunner.class)
@DelegateTo(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes = {Application.class})
public class ApiServiceTest {

    @Autowired
    ApiService apiService;

    @Test
    public void testMobileRegister() {
        AlispResult<Map<String, Object>> result = apiService.mobileRegister();
        System.out.println("result = " + result);
        Assert.assertNotNull(result);
        Assert.assertEquals(54,result.getAlispCode().longValue());

        AlispResult<Map<String, Object>> result2 = apiService.mobileRegister();
        System.out.println("result2 = " + result2);
        Assert.assertNotNull(result2);
        Assert.assertEquals(9,result2.getAlispCode().longValue());

        AlispResult<Map<String, Object>> result3 = apiService.mobileRegister();
        System.out.println("result3 = " + result3);
        Assert.assertNotNull(result3);
        Assert.assertEquals(200,result3.getAlispCode().longValue());
    }

    @Test
    public void should_return_mobile_is_not_correct_when_register_given_a_invalid_phone_number() {
```

```
29.         AlispResult<Map<String, Object>> result = apiService.mobileRegister();
30.         Assert.assertNotNull(result);
31.         Assert.assertFalse(result.isSuccess());
32.     }
33. }
```

第一个UT是以方法维度，把所有场景放到一个方法来测试。

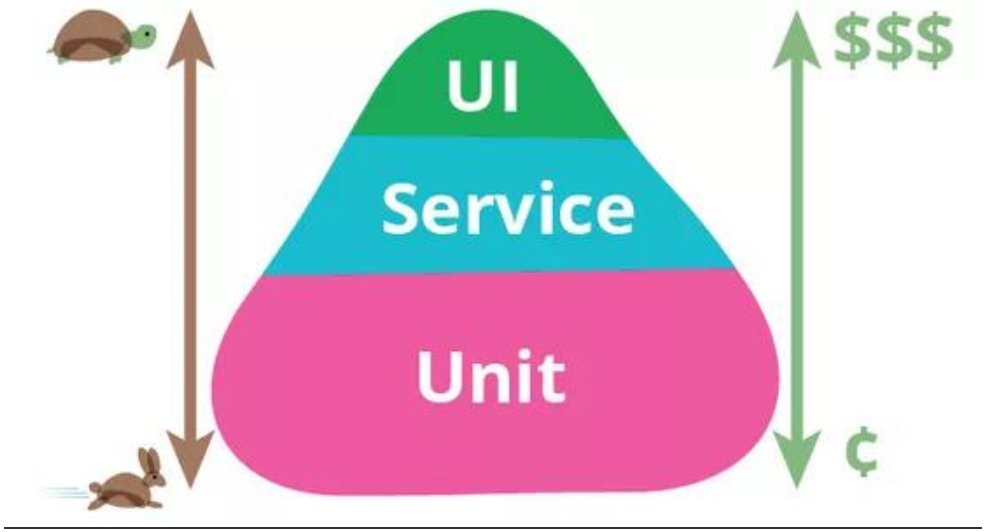
第二个UT是以case为角度，针对每个case单独的测试。

其实TDD里面有一个概念是隔离性，单元测试之间应该隔离开，不要互相干扰。另外，从命名上，第二种也更好一点。我个人还是比较推荐以下命名方式的：

- should：返回值，应该产生的结果
- when：哪个方法
- given：哪个场景

另外BDD或者TDD中也有Task的概念，写代码之前先准备好case。大家可以看一些BDD的文章，自己体会。如果对这个感兴趣，可以在评论区探讨。

测试金字塔



上图来自martin fowler博客的TestPyramid[1]一文，也可以读一下《Practical Test Pyramid》[2]。特别棒的文章，希望大家可以去读一读。

上面的金字塔的意思是，从Unit到Service，再到UI，速度越来越慢，成本也越来越高。

我们可以从服务端的角度把这三层稍微改一下：

- 契约测试：测试服务与服务之间的契约，接口保证。代价最高，测试速度最慢。
- 集成测试(Integration)：集成当前spring容器、中间件等，对服务内的接口，或者其他依赖于环境的方法的测试。

```
// 加载spring环境
@RunWith(SpringBootRunner.class)
@DelegateTo(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes = {Application.class})
public class ApiServiceTest {

@Autowired
ApiService apiService;
//do some test
}
```

复制

单元测试(Unit Test)：纯函数，方法的测试，不依赖于spring容器，也不依赖于其他的环境。

```java
@RunWith(JUnit4.class)
@Slf4j
public class LotteryServiceTest {
    private final static int WORKER_THREAD_COUNT = 30;
    private final static int WORKER_THREAD_ITERATION = 1000;
    private final static int WORKER_THREAD_ITERATION_DELAY = 4000;

    private final static int RETRY_TIMES = 3;
    private final static double UNLOCK_WEIGHT_PERCENT_BY_TIME = 0.8;

    @Test
    public void should_close_correct_when_lottery_given_a_activity_contains_all_resource() throws InterruptedException {...}
```

我们现在写测试，一般是单元测试和集成测试两层。针对具体场景，选择适合自己的测试粒度。

招数

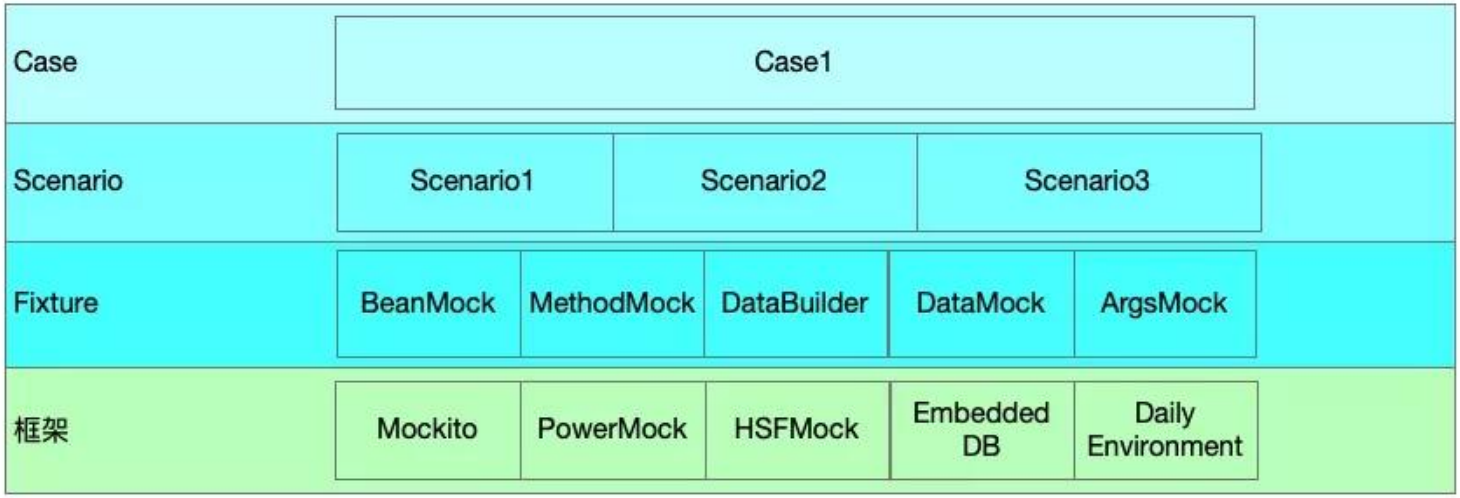
其实写单元测试是有一些招数的，下面会介绍笔者很喜欢的一种单元测试代码组织结构，也会介绍一些常用的招数，以及使用场景。

常见问题

- 一个类里面测试太多怎么办?
- 不知道别人mock了哪些数据怎么办?
- 测试结构太复杂?
- 测试莫名奇妙起不来?

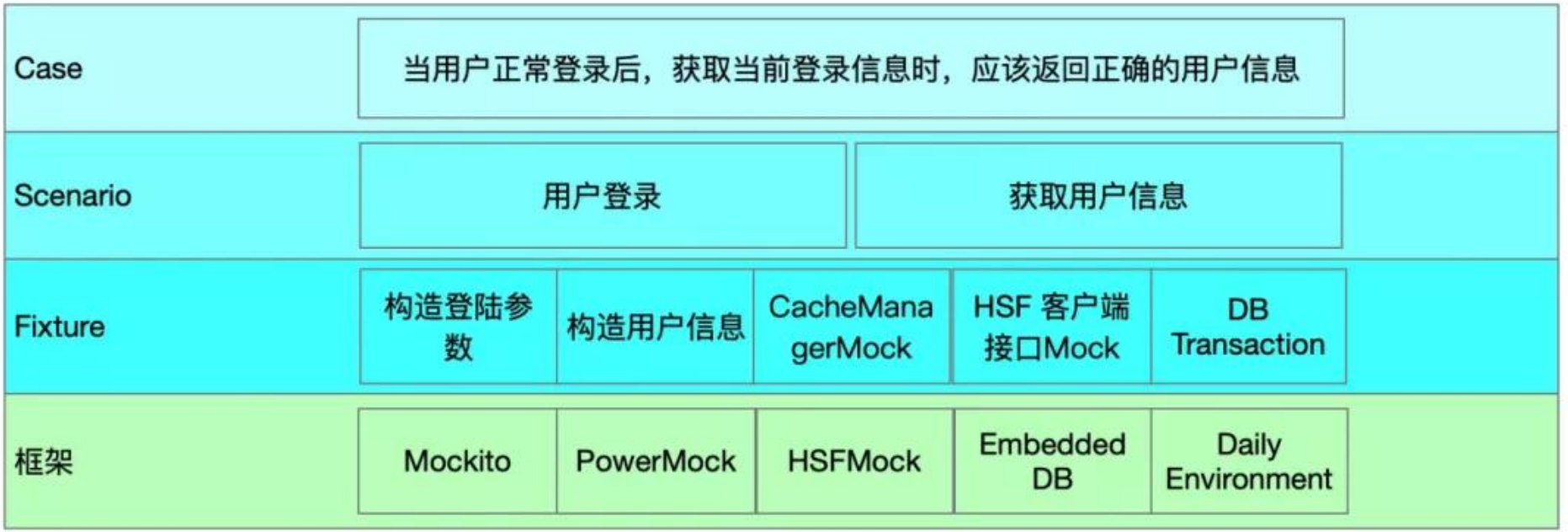
Fixture-Scenario-Case

FSC(Fixture-Scenario-Case)是一种组织测试代码的方法，目标是尽量将一些MOCK信息在不同的测试中共享。其结构如下：



- 通过组合Fixture(固定设施)，来构造一个Scenario(场景)。
- 通过组合Scenario(场景)+ Fixture(固定设施)，构造一个case(用例)。

下面是一个FSC的示例：



- Case：当用户正常登录后，获取当前登录信息时，应该返回正确的用户信息。这是一个简单的用户登录的case，这个case里面总共有两个动作、场景，一个是用户正常登录，一个是获取用户信息，演化为两个scenario。
- Scenario：用户正常登录，肯定需要登录参数，如：手机号、验证码等，另外隐含着数据库中应该有一个对应的用户，如果登录时需要与第三方系统进行交互，还需要对第三方系统进行mock或者stub。获取用户信息时，肯定需要上一阶段颁发的凭证信息，另外该凭证可能是存储于一些缓存系统的，所以还需要对中间件进行mock或者stub。
- Fixture
- 利用Builder模式构造请求参数。
- 利用DataFile来存储构造用户的信息，例如DB transaction进行数据的存储和隔离。
- 利用Mockito进行三方系统、中间件的Mock。

当这样组织测试时，如果另外一个Case中需要用户登录，则可以直接复用用户登录的Scenario。也可以通过复用Fixture来减少数据的Mock。下面我们来详细解释看一下每一层如何实现，show the code。

Case

case是用例的意思，在这里用例是场景和一些固定设施的组合。这里要注意的是，尽量不要直接修改接口的数据，一个场景所依赖的环境应该是另一个场景的输出。当然有些特定场景下，还是需要直接改数据的，这里不是禁止，而是建议。

复制

```
public class GetUserInfoCase extends BaseTest {
    private String accessToken;

    @Autowired
    private UserFixture userFixture;

    /**
     * 通用场景的mock
     */
    @Before
    public void setUp() {
        //三方系统mock
        userFixture.whenFetchUserInfoThenReturn("1", new UserVO());

        //依赖的其他场景
        accessToken = new SimpleLoginScenario()
                .mobile("1234567890")
                .code("aaa")
                .login()
                .getAccessToken();
    }

    /**
     * BDD的三段式
     */
    @Test
```

```
    public void should_return_user_info_when_user_login_given_a_effective_access_token() {
        Response userInfoResponse = new GetUserInfoScenario()
                .accessToken(accessToken)
                .getUserInfo();

        assertThat(userInfoResponse.jsonPath().getString("id"), equals("1"));
    }
}
```

Scenario

JUNIT的用法就不说了，相信大家都了解，这里提两个框架REST Assured和Mock MVC。这两个框架都可以用来做接口测试，Mock MVC是spring原生的，可以指定加载的Resource，一定程度上可以提升UT速度，但是和spring是耦合在一起的。REST Assured是脱离Spring的，可以理解为利用http进行接口的测试，耦合性更低，使用灵活。两者各有千秋，笔者比较推荐REST Assured。我们看一下，一个REST Assured打造的Scenario怎么写，怎么用?

复制

```
@Data
public class SimpleLoginScenario {
    // 请求参数
    private String mobile;
    private String code;

    // 登录结果
    private String accessToken;

    public SimpleLoginScenario mobile(String mobile) {
        this.mobile = mobile;
        return this;
    }

```

```
    public SimpleLoginScenario code(String code) {
        this.code = code;
        return this;
    }

    //登录，并且保存AccessToken，这里返回自身，是因为有可能返回参数是多个。
    public SimpleLoginScenario login() {
        Response response = loginWithResponse();
        this.accessToken = response.jsonPath().getString("accessToken");
        return this;
    }

    //利用RestAssured进行登录，这个方法可以是public，也可以通过参数传递一些验证方法
    private Response loginWithResponse() {
        return RestAssured.get(API_PATH, ImmutableMap.of("mobile", mobile, "code", code))
                .thenReturn();
    }

}
```

Fixture

固定设施部分，主要是用来提供一些固定的组件和数据。尽量的让这部分东西有复用性，如果没复用性，尽量和测试放在一起，不要干扰他人。

(1)方法

(a)Mock

mockito挺通用的，而且spring也提供了@MockBean，可以直接将Mock一个bean放入spring的容器中。然后可以利用mockito提供的方法对方法进行模拟或者验证。代码示例：

```
public class MockitoTest {
  @MockBean(classes = CacheImpl.class)
  private Cache cache;

  @Test
  public void should_return_success() {
      // 固定参数，固定返回值
      Mockito.when(cache.get("KEY")).thenReturn("VALUE");

      // 动态参数，固定返回值
      Mockito.when(cache.get(Mockito.anyString())).thenReturn("VALUE");

      // 动态参数，固定返回值
      Mockito.when(cache.get(Mockito.anyString())).then((invocation) -> {
          String key = (String) invocation.getArguments()[0];
          return "VALUE";
      });

      // 固定参数，异常
      Mockito.when(cache.get("KEY")).thenThrow(new RuntimeException("ERROR"));

      // 验证调用次数
      Mockito.verify(cache.get("KEY"), Mockito.times(1));
  }
}
```

复制

(b)stub

stub是打桩，关于打桩和mock的区别，请自行百度，这里只是想展示一下，在spring的环境下，覆盖原有bean达到stub的效果。

```
//使用spring的@Primary来替换一个bean，如果不同的测试需要的bean不同，推荐使用@Configuration + @Import的方式，动态加载Bean
@Primary
@Component("cache")
public class CacheStub implements Cache {

  @Override
  public String get(String key) {
      return null;
  }

  @Override
  public int setex(String key, Integer ttl, String element) {
      return 0;
  }

  @Override
  public int incr(String key, Integer ttl) {
      return 0;
  }

  @Override
  public int del(String key) {
      return 0;
  }
}
```

复制

(c)嵌入式DB

这里简单介绍几种嵌入式DB，可以自行选择使用。

对比项	H2	Embedded - Mysql 、Embedded - PostgresSQL
启动速度	特快	快
语法兼容性	差	优
启动依赖	少	多，例如mysql依赖于openssl的特定版本

(d)直连DB + Transaction

- 除了使用嵌入式的DB，也可以直连环境，但不推荐，因为环境上的数据是多变的，如果测试出现问题，排查的复杂度会增加。这里其实想强调下@Transactional。因为Mock的数据最好做到隔离，比如一个接口的操作是批量删除数据，有可能会把一个其他测试依赖的数据删除掉，这样问题一旦出现很难排查，因为单独跑每个测试都是通过的，但是一起跑就会出问题。这里推荐两种做法：
- 使用@Transactional在一些测试的类上，这样在跑完测试后，数据不会commit，会回滚。但如果测试中对事物的传播有特殊要求，可能不适用。

通用的trancateAll和initSQL通过在每个测试前跑清除数据、mock数据的脚本，来达到每个测试对应一个隔离环境，这样数据间就不会产生干扰。

(e)PowerMock

PowerMock是用来创建一些静态方法的Mock的，如果你的代码中会调用一些静态方法，但是静态方法依赖于一些其他复杂的逻辑或者资源。可以使用这个包。

```
PowerMockito.mockStatic(C.class);
PowerMockito.when(C.isTrue()).thenReturn(true);
```

复制

注意：

- PowerMock不仅仅是用来mock静态方法的。
- 不建议mock静态方法，因为静态方法的使用场景都是些纯函数，大部分的纯函数不需要mock。部分静态方法依赖于一些环境和数据，针对这些方法，需要考虑下到底是要mock其依赖的数据和方法，还是真的要mock这个函数，因为一旦mock了这个函数，意味着隐藏了细节。

(2)数据

(a)Builder模式

数据最简单的mock方式就是Builder，然后自己手填各种参数，但有些对象有几十个字段，而你的一个测试只需要改其中的两个字段，你该怎么办?Copy、Paste?

```
@Builder
@Data
public class UserVO {
  private String name;
  private int age;
  private Date birthday;
}

public class UserVOFixture {
    // 注意：这里是个Supplier，并不是一个静态的实例，这样可以保证每个使用方，维护自己的实例
  public static Supplier<UserVO.UserVOBuilder> DEFAULT_BUILDER = () ->  UserVO.builder().name("test").age(11).birthday(new Date());
}
```

(b)数据文件

有时候通过builder构造对象的时候，字段太多，并且数据的来源是前端或者其他服务提供的json。这个时候可以将这个数据存储到文件中，利用一些工具方法，将数据读取成制定的文件。这也是数据mock的常用手段。我这里是以json为例，其实sql等数据也可以这样。

数据文件的优点：可承载的数据量大、编辑方便。

```
public class UserVOFixture {

  public static UserVO readUser(String filename) {
      return readJsonFromResource(filename, UserVO.class);
```

```
    }

    public static <T> T readJsonFromResource(String filename, Class<T> clazz) {
        try {
            String jsonString = StreamUtils.copyToString(new ClassPathResource(filename).getInputStream(), Charset.defaultCharset());
            return JSON.parseObject(jsonString, clazz);
        } catch (IOException e) {
            return null;
        }
    }
}
```

使用场景

在笔者的实践中, 目前主要把FSC是用在接口测试上，也就是测试金字塔的Integration Test部分，放在这个层次，有几个原因：

- FSC本身会给测试带来复杂度，而UnitTest应该简单，如果UnitTest本身都很复杂了，项目带来难以估量的测试成本。
- Fixture其实可以在任何场景中使用，因为是底层的复用。

缺陷

- 增加了代码复杂度。
- 通过IDE工具无法直接定位的测试文件，折衷的方案是case的命名符合ResouceTest的命名。

校场

从简单到复杂

上面我们介绍了测试金字塔，越靠上层，复杂度越高。所以刚接触单元测试的同学，可以从“单元测试”的层次开始练习，可以练习Builder，Fixture怎么写，方法怎么Mock。如果你感觉这些都到了拿来即用的阶段，那就可以往上层写，考虑下怎么给项目增加一些通用的基础设施，来减少测试的整体复杂度。

刻意练习：3F原则

刻意练习，简而言之，就是刻意的练习，它突出的是有目的的练习。刻意练习也有它的一整套过程，在这个过程里，你需要遵守它的3F法则：

- 第一，Focus(保持专注)。
- 第二，Feedback(注重反馈，收集信息)。
- 第三，Fix it(纠正错误，并且进行修改)。

UT本身是一项技术，是需要我们打磨、练习的，最好的练习方式，就是刻意练习，如果有决心，一个周末在家刻意练习，为项目中的部分场景加上UT，相信收获会很丰富。

打造自己的测试环境

自己要不断的摸索，什么样的组织方式，什么样的工具方法是适合自己项目的。软件工程中没有银弹，没有最好，只有合适。

常见问题

- 应不应该连日常环境进行测试?
- 个人不建议直接连日常环境进行测试，如果两个人同时在跑测试，那么很有可能测试环境的数据会处于混乱状态。而且UT尽可能不要依赖过多的外部环境，依赖越多越复杂。测试还是简单点好。
- 一个类里面测试太多怎么办?
- 考虑按测试的case区分，也可按测试的方法区分，也可以按正常、异常场景区分。
- 不知道别人mock了哪些数据怎么办?
- 尽量让大家Mock数据的命名规范，通过Fixutre的复用，来减少新写测试的成本。
- 测试结构太复杂?
- 考虑是不是自己应用的代码组织就有问题?
- 测试莫名奇妙起不来?
- 需要详细了解JUNIT、Spring、PandoraBoot等是如何进行测试环境的mock的，是不是测试间的数据冲突等。详细的我们会在方法篇持续更新，遇到问题解决问题。

心魔

单元测试这件事，实施的时候还是有很多阻力的，笔者原来给自己也找过很多理由，无论是用来说服领导的，还是说服自己的。下面是笔者对于这些理由的一些思考，希望能和大家有一些共鸣。

不会写

虽然很不愿意承认这个事，但最后还是承认了自己是真的不会写单元测试。刚接触单元测试的时候，看了看junit的文档，心想单元测试，不就是个“Assert”吗，有啥不会的，这东西好学。后来实施过程中发现，单元测试不仅仅是“Assert”，还需要准备环境，Mock数据，复现场景，验证。着实是个麻烦事。

后来反思，为什么单元测试麻烦?一开始学习ORM框架的时候不麻烦吗?一开始学Spring不麻烦吗?后来熟悉了Bean的生命周期、BeanFactory、BeanProcessor等，Spring已经不是个麻烦事了。仔细想想，自己对单元测试的理解仅仅是：“一个Mock加一个Assert”。仅仅学了几个框架，看了几篇文章，还做不到把单元测试这件事真正落地。

在落地单元测试的时候，有一些常见的问题：

场景太复杂，需要的数据太多，怎么处理?

可以直接使用JSON、SQL将现有数据修改后导入到系统中。这样的话可能需要mock的数据就不会那么多了，可以提炼一些工具类，直接从resource中读取数据文件，导入到数据库、或者提供给mock方法使用。

也可以构建一些Fixture，将自己系统中UT的数据固定下来，这样，如果前面一个同学已经mock过相关数据了，再新写UT的时候可以拿来即用。构建Fixture可以用工厂模式、构建者模式等来达到数据隔离的效果，避免相互干扰。

好多东西都是和中间件或者其他系统频繁交互，怎么写测试?

数据库层面可以使用内存型数据库“H2”、"Embedded Mysql"、“Embedded PostgreSql”等。

如果以上都不能解决问题，可以使用mockito直接mock相应的Bean。

单元测试的粒度问题，这个方法该不该写UT，另外一个方法为什么不需要写UT?

单元测试的粒度没有标准答案，笔者自己总结了一些写UT粒度方面的方法：

- 不熟悉单元测试写法，尽量写简单的单元测试，覆盖核心方法。
- 熟悉单元测试，业务复杂，覆盖正常、一般异常场景，另外对核心业务逻辑要有单独的测试。

测试如何复用?

测试应该是有组织、有结构的，就像我们写业务代码一样，会想着如何在代码层面复用、如何在功能层面复用、如何在业务维度复用。单元测试也应该有结构，可以尽量复用一些前人的经验。简单来说，测试的复用也分为三个维度：数据、场景、用例，好的代码结构应该尽量的能让测试复用，让增加UT不再是从头开始。

不想写

写测试有什么用?

很多人都写过单元测试的文章，罗列过很多单元测试的很多好处，这里就不赘述了。这里讲几个感触比较深的用处吧?

- DEBUG：阿里现在的基础设施是真的完善，中间件、各种监控、日志，只要系统埋点够好，遇到的很多问题都可以解决，即使有一些复杂问题，也可以local debug。但在一些特殊场景下，将数据MOCK好，利用UT来DEBUG，可能效率更高，大家可以试试。
- 测试如文档：我们现在开发有很多完善的文档，但文档这东西和代码上毕竟有一层映射关系，如果能快速了解业务，完善的测试，有时候也是个不错的选择，例如大家学习一些开源框架的时候，都会从测试开始看。
- 重构：当你想下定决心重构的时候，才发现项目中没有单元测试，什么心情?

价值不高

在面对复杂的接口时，常常需要Mock很多数据来支撑一个小的点，很多时候内心感觉没价值，因为一个if-else的变动，竟然需要准备N份数据，得不偿失。

后来反思，为什么一个if-else的变动，需要准备N份数据?如果这个接口一开始写的时候就有健全的UT，那一个if-else的变更还需要准备N份数据吗?大概率不需要了吧，有可能只需要改一个测试case就好了。所以说现在成本高，将来成本会更高，现在做了，做的好一点，后面可能成本就低了。

笔者观点：写单元测试，应该比写代码的成本更低。

懒

这个不用说吧，通用理由，大家都明白。路是人踩出来的，总要有人要先走。Why not you?

最后

如果大家对于单元测试有好的实践，或者对文章中的一些观点有些共鸣，大家可以在评论区留言，我们互相学习一下。大家也可以在评论区写出自己的场景，大家一起探讨如何针对特定场景来实践。

相关链接

[1]https://martinfowler.com/bliki/TestPyramid.html

[2]https://martinfowler.com/articles/practical-test-pyramid.html

戳这里，看该作者更多好文

责任编辑：武晓燕　　来源： 51CTO专栏

单元　测试　技术