Spock Test

为什么使用Spock Test（Quick Start）

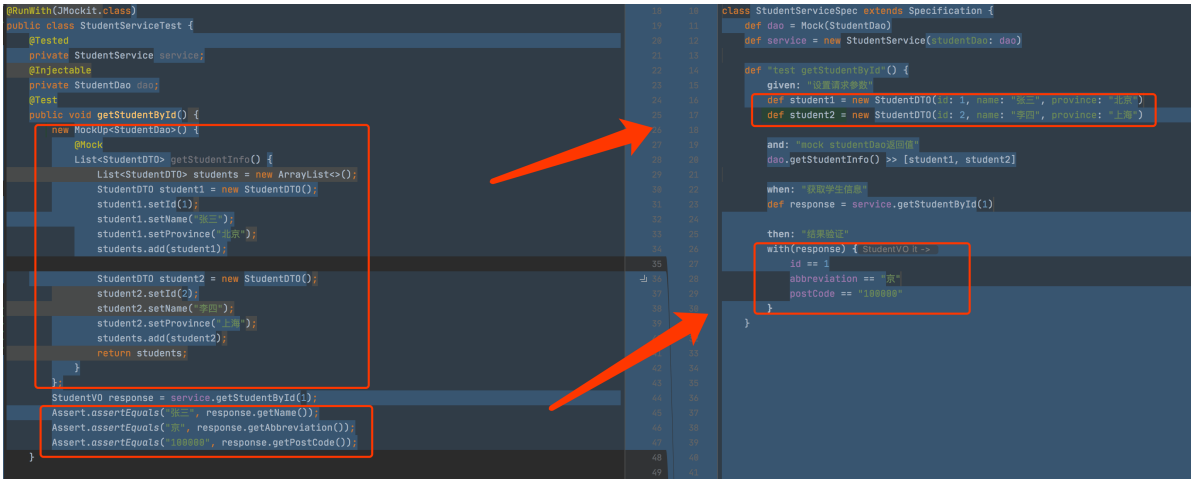
JMock或Mockito虽然提供了mock功能，可以把接口等依赖屏蔽掉，但不提供对静态类静态方法的mock，PowerMock或Jmockit虽然提供静态类和方法的mock，但它们之间需要整合(junit+mockito+powermock)，语法繁琐，而且这些工具并没有告诉你“**单元测试代码到底应该怎么写？**”

工具多了也会导致不同的人写出的**单元测试代码五花八门，风格迥异。。。**

Spock通过提供规范描述，定义多种标签(given、when、then、where等)去描述代码“应该做什么”，输入条件是什么，输出是否符合预期，从语义层面规范代码的编写。

Spock自带Mock功能，使用简单方便(也支持扩展其他mock框架，比如power mock)，再加上groovy动态语言的强大语法，能写出简洁高效的测试代码，同时更方便直观的验证业务代码行为流转，增强我们对代码执行逻辑的可控性。

```
public StudentVO getStudentById(int id) {
        List<StudentDTO> students = studentDao.getStudentInfo();
        StudentDTO studentDTO = students.stream().filter(u -> u.getId() ==
id).findFirst().orElse(null);
        StudentVO studentVO = new StudentVO();
        if (studentDTO == null) {
            return studentVO;
        }
        studentVO.setId(studentDTO.getId());
        studentVO.setName(studentDTO.getName());
        studentVO.setSex(studentDTO.getSex());
        studentVO.setAge(studentDTO.getAge());
        // 邮编
        if ("上海".equals(studentDTO.getProvince())) {
            studentVO.setAbbreviation("沪");
            studentVO.setPostCode("200000");
        }
        if ("北京".equals(studentDTO.getProvince())) {
            studentVO.setAbbreviation("京");
            studentVO.setPostCode("100000");
        }
        return studentVO;
    }
```



Mock的用法

几个关键字用来区分不同单测代码的作用：

- given: 输入条件(前置参数)
- when: 执行行为(mock接口、真实调用)
- then: 输出条件(验证结果)
- and: 衔接上个标签，补充的作用
- 每个标签后面的双引号里可以添加描述，说明这块代码的作用(非强制)，如"when: "调用获取用户信息方法""

> 因为spock使用groovy作为单测开发语言，所以代码量上比使用java写的会少很多，比如given模块里通过构造函数的方式创建请求对象

```
given: "设置请求参数"
def user1 = new UserDTO(id:1, name:"张三", province: "上海")
def user2 = new UserDTO(id:2, name:"李四", province: "江苏")
```

- 实际上UserDTO.java 这个类并没有3个参数的构造函数，是groovy帮我们实现的，groovy默认会提供一个包含所有对象属性的构造函数
- 而且调用方式上可以指定属性名，类似于key:value的语法，非常人性化，方便我们在属性多的情况下构造对象，如果使用java写，可能就要调用很多setXXX()方法才能完成对象初始化的工作

```
and: "mock掉接口返回的用户信息"
userDao.getUserInfo() >> [user1, user2]
```

- Spock的mock用法，即当调用userDao.getUserInfo()方法时返回一个List，list的创建也很简单，中括号"[]"即表示list，groovy会根据方法的返回类型自动匹配是数组还是list，而list里的对象就是之前given块里构造的user对象
- 其中 ">>" 就是指定返回结果，类似mockito的when().thenReturn()语法，但更简洁一些

> 指定多个返回对象

```
userDao.getUserInfo() >>> [[user1,user2],[user3,user4],[user5,user6]]


userDao.getUserInfo() >> [user1,user2] >> [user3,user4] >> [user5,user6]
```

Where的用法

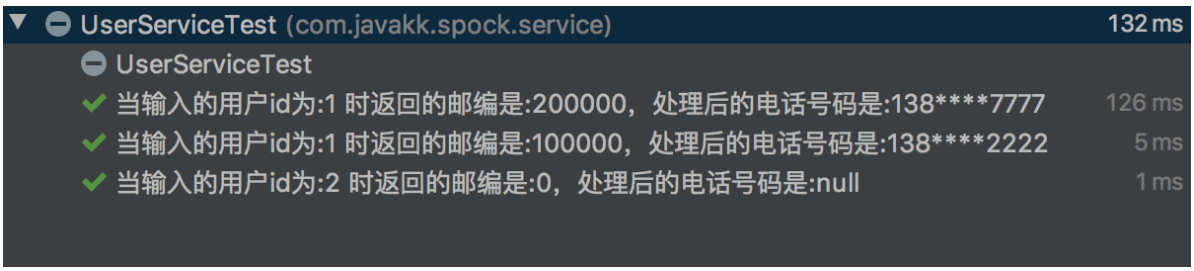
```
// 显示邮编
if("上海".equals(userDTO.getProvince())){
    userVO.setAbbreviation("沪");
    userVO.setPostCode(200000);
}
if("北京".equals(userDTO.getProvince())){
    userVO.setAbbreviation("京");
    userVO.setPostCode(100000);
}
// 手机号处理
if(null != userDTO.getTelephone() && !"".equals(userDTO.getTelephone())){

 userVO.setTelephone(userDTO.getTelephone().substring(0,3)+"****"+userDTO.getTel
ephone().substring(7));
}
```

```
// 显示邮编
if("上海".equals(userDTO.getProvince())){
    userVO.setAbbreviation("沪");
    userVO.setPostCode(200000);
}
if("北京".equals(userDTO.getProvince())){
    userVO.setAbbreviation("京");
    userVO.setPostCode(100000);
}
// 手机号处理
if(null != userDTO.getTelephone() && !"".equals(userDTO.getTelephone())){

userVO.setTelephone(userDTO.getTelephone().substring(0,3)+"****"+userDTO.getTele
phone().substring(7));
}
```

```
@Unroll
def "当输入的用户id为:#uid 时返回的邮编是:#postCodeResult，处理后的电话号码
是:#telephoneResult"() {
    given: "mock掉接口返回的用户信息"
    userDao.getUserInfo() >> users

    when: "调用获取用户信息方法"
    def response = userService.getUserById(uid)

    then: "验证返回结果是否符合预期值"
    with(response) {
        postCode == postCodeResult
        telephone == telephoneResult
    }

    where: "表格方式验证用户信息的分支场景"
    uid | users                          || postCodeResult | telephoneResult
    1   | getUser("上海", "13866667777") || 200000         | "138****7777"
    1   | getUser("北京", "13811112222") || 100000         | "138****2222"
    2   | getUser("南京", "13833334444") || 0              | null
}

def getUser(String province, String telephone){
    return [new UserDTO(id: 1, name: "张三", province: province, telephone:
telephone)]
}
```

> 在Groovy测试中，`@Unroll` 是一个用于参数化测试的注解。参数化测试是一种在单个测试方法中运行多个测试案例的方法，每个案例都具有不同的输入参数和期望输出。使用 `@Unroll` 注解可以将参数化测试的案例展开，以便更清楚地了解每个案例的测试结果。

> @Unroll def "当输入的用户id为:#uid 时返回的邮编是:#postCodeResult，处理后的电话号码是:#telephoneResult"() {
>
> 即把请求参数值和返回结果值的字符串里动态替换掉，"#uid、#postCodeResult、#telephoneResult" 井号后面的变量是在方法内部定义的，前面加上#号，实现占位符的功能



异常处理

背景
有些方法需要抛出异常来中断或控制流程，比如参数校验的逻辑：不能为null，不符合指定的类型，list不能为空等验证，如果校验不通过则抛出checked异常，这个异常一般都是我们封装的业务异常信息，比如下面的业务代码：

```
/**
 * 校验请求参数user是否合法
 * @param user
 * @throws APIException
 */
public void validateUser(UserVO user) throws APIException {
    if(user == null){
        throw new APIException("10001", "user is null");
    }
    if(null == user.getName() || "".equals(user.getName())){
        throw new APIException("10002", "user name is null");
    }
    if(user.getAge() == 0){
        throw new APIException("10003", "user age is null");
    }
    if(null == user.getTelephone() || "".equals(user.getTelephone())){
        throw new APIException("10004", "user telephone is null");
    }
    if(null == user.getSex() || "".equals(user.getSex())){
        throw new APIException("10005", "user sex is null");
    }
    if(null == user.getUserOrders() || user.getUserOrders().size() <= 0){
        throw new APIException("10006", "user order is null");
    }
    for(OrderVO order : user.getUserOrders()) {
        if (null == order.getOrderNum() || "".equals(order.getOrderNum())) {
            throw new APIException("10007", "order number is null");
        }
        if (null == order.getAmount()) {
            throw new APIException("10008", "order amount is null");
        }
    }
}
```

```
//使用Mockit
import static org.mockito.Mockito.*;

// 在测试方法中
@Test
public void testValidateUserWithExceptions() throws APIException {
    // 创建一个模拟的UserVO对象
    UserVO mockUser = mock(UserVO.class);
    when(mockUser.getName()).thenReturn(null); // 触发 user name is null 异常
    when(mockUser.getAge()).thenReturn(0);     // 触发 user age is null 异常
    when(mockUser.getTelephone()).thenReturn("123456789"); // 设置用户电话
    when(mockUser.getSex()).thenReturn(null);  // 触发 user sex is null 异常
    when(mockUser.getUserOrders()).thenReturn(null); // 触发 user order is null 异
常

    // 创建一个模拟的 OrderVO 对象
    OrderVO mockOrder = mock(OrderVO.class);
    when(mockOrder.getOrderNum()).thenReturn(null); // 触发 order number is null
异常
    when(mockOrder.getAmount()).thenReturn(null);   // 触发 order amount is null
异常

    // 设置用户订单
    List<OrderVO> mockOrders = Arrays.asList(mockOrder);
    when(mockUser.getUserOrders()).thenReturn(mockOrders);

    // 创建一个待测试的类实例
    YourClassUnderTest yourClass = new YourClassUnderTest();
```

```
    // 调用 validateUser 方法，并期望抛出相应的异常
    assertThrows(APIException.class, () -> yourClass.validateUser(mockUser));

    // 验证模拟方法是否被正确调用
    verify(mockUser, times(1)).getName();
    verify(mockUser, times(1)).getAge();
    verify(mockUser, times(1)).getTelephone();
    verify(mockUser, times(1)).getSex();
    verify(mockUser, times(1)).getUserOrders();

    verify(mockOrder, times(1)).getOrderNum();
    verify(mockOrder, times(1)).getAmount();
}
```

```
package com.javakk.spock.controller

import com.javakk.spock.model.APIException
import com.javakk.spock.model.OrderVO
import com.javakk.spock.model.UserVO
import spock.lang.Specification
import spock.lang.Unroll


class UserControllerTest extends Specification {

    def userController = new UserController()

    @Unroll
    def "验证用户信息的合法性: #expectedMessage"() {
        when: "调用校验用户方法"
        userController.validateUser(user)

        then: "捕获异常并设置需要验证的异常值"
        def exception = thrown(expectedException)
        exception.errorCode == expectedErrCode
        exception.errorMessage == expectedMessage

        where: "表格方式验证用户信息的合法性"
        user            || expectedException | expectedErrCode | expectedMessage
        getUser(10001) || APIException      | "10001"         | "user is null"
        getUser(10002) || APIException      | "10002"         | "user name is
null"
        getUser(10003) || APIException      | "10003"         | "user age is
null"
        getUser(10004) || APIException      | "10004"         | "user telephone
is null"
        getUser(10005) || APIException      | "10005"         | "user sex is
null"
        getUser(10006) || APIException      | "10006"         | "user order is
null"
        getUser(10007) || APIException      | "10007"         | "order number is
null"
        getUser(10008) || APIException      | "10008"         | "order amount is
null"
    }

    def getUser(errCode) {
        def user = new UserVO()
        def condition1 = {
            user.name = "杜兰特"
        }
        def condition2 = {
            user.age = 20
        }
        def condition3 = {
            user.telephone = "15801833812"
        }
        def condition4 = {
            user.sex = "男"
        }
        def condition5 = {
            user.userOrders = [new OrderVO()]
```

```
        }
        def condition6 = {
            user.userOrders = [new OrderVO(orderNum: "123456")]
        }

        switch (errCode) {
            case 10001:
                user = null
                break
            case 10002:
                user = new UserVO()
                break
            case 10003:
                condition1()
                break
            case 10004:
                condition1()
                condition2()
                break
            case 10005:
                condition1()
                condition2()
                condition3()
                break
            case 10006:
                condition1()
                condition2()
                condition3()
                condition4()
                break
            case 10007:
                condition1()
                condition2()
                condition3()
                condition4()
                condition5()
                break
            case 10008:
                condition1()
                condition2()
                condition3()
                condition4()
                condition5()
                condition6()
                break
        }
        return user
    }
}
```

void方法测试

```
/**
 * 根据汇率计算金额
 * @param userVO
 */
public void setOrderAmountByExchange(UserVO userVO){
    if(null == userVO.getUserOrders() || userVO.getUserOrders().size() <= 0){
        return ;
    }
    for(OrderVO orderVO : userVO.getUserOrders()){
        BigDecimal amount = orderVO.getAmount();
        // 获取汇率(调用汇率接口)
        BigDecimal exchange =
moneyDAO.getExchangeByCountry(userVO.getCountry());
        amount = amount.multiply(exchange); // 根据汇率计算金额
        orderVO.setAmount(amount);
    }
```

```
}
```

```
class UserServiceTest extends Specification {
    def userService = new UserService()
    def moneyDAO = Mock(MoneyDAO)

    void setup() {
        userService.userDao = userDao
        userService.moneyDAO = moneyDAO
    }

    def "测试void方法"() {
        given: "设置请求参数"
        def userVO = new UserVO(name:"James", country: "美国")
        userVO.userOrders = [new OrderVO(orderNum: "1", amount: 10000), new
OrderVO(orderNum: "2", amount: 1000)]

        when: "调用设置订单金额的方法"
        userService.setOrderAmountByExchange(userVO)

        then: "验证调用获取最新汇率接口的行为是否符合预期: 一共调用2次, 第一次输出的汇率是
0.1413, 第二次是0.1421"
        2 * moneyDAO.getExchangeByCountry(_) >> 0.1413 >> 0.1421

        and: "验证根据汇率计算后的金额结果是否正确"
        with(userVO){
            userOrders[0].amount == 1413
            userOrders[1].amount == 142.1
        }
    }
}
```