支付业务产品研发中心

# 单元测试编写规范

（版本 V1.2.3）

# 版本信息

| 版本 | 时间 | 状态 | 简要描述 | 部门 | 更改人 | 批准人 |
|---|---|---|---|---|---|---|
| V1.0 | 2015/06/10 | N | 创建 | 出入款产品开发部 | 徐必涛 | 孙捷 |
| V1.1 | 2015/09/12 | M | 简化示例 | 出入款产品开发部 | 徐必涛 | 孙捷 |
| V1.2 | 2015/09/17 | A | 补充识别被测单元小节 | 出入款产品开发部 | 徐必涛 | 孙捷 |
| V1.2.1 | 2015/09/30 | M | 更新BeanUtil版本 | 出入款产品开发部 | 徐必涛 | 孙捷 |
| V1.2.2 | 2015/10/20 | M | 更新BeanUtil Jar包引用，修正Mock静态方法相关描述 | 出入款产品开发部 | 徐必涛 | 孙捷 |
| V1.2.3 | 2015/11/4 | A | 新增BeanUtil对泛型的转换 | 出入款产品开发部 | 徐必涛 | 孙捷 |

注：状态可以为N-新建、A-增加、M-更改、D-删除。

# 目录

# 第 1 章　概 述

## 1.1　文档目的

针对支付业务产品研发中心的研发需求，为提高单元测试的质量、实现单元测试自动化而制定本文档，用于指导开发和设计人员单元测试编写方面的工作。同时也为单元测试评审提供统一评审依据。

## 1.2　适用范围

本文档适用于以下应用场景：

- *本文档适用于支付业务产品研发中心测试先行（TDD）和测试后行所编写的单元测试；*
- *本文档涉及规范在编写单元测试时必须遵守。*

## 1.3　规范定义原则

本文档内容定义及更新应遵循以下原则：

- *简单性原则*

  *文档只定义单元测试编写中应遵循的规则和建议，不作为技术提高资料，对一些规范的理解，可以查询相关的技术资料；*

- *指导性原则*

  *文档中尽量给出建议的编写示例供开发人员；*

# 第 2 章　单元测试规范

## 2.1　单元测试的定义

针对一个被测试的单元，通过给定相关输入，**验证**其**输出**及**过程**。

对于包含与第三方协作者交互的被测试单元，不仅需要验证预期的输出，还要验证其交互过程。

## 2.2　识别被测试单元

被测试单元：即 Unit Under Test，简称 **UUT**.通常是一个**业务类**中的一个 public 方法。

在一个项目中，有众多的 Java 类，但并不需要为每个 Java 类编写单元测试。通常情况下，只为**具有一定业务行为**的类编写单元测试。诸如 DTO、Result、工具类等，不必在做业务开发时为其编写单元测试。

值得注意的是，Façade 层和 Controller 在不存在业务逻辑仅仅作为请求的出入口（推荐）的情况下，也可以不写单元测试。但是在许多项目中，此两层往往包含着许多业务逻辑，在识别被测试单元时，需要具体问题具体分析。

**注：由于网关层系统通常缺乏业务逻辑，不用编写单元测试。**

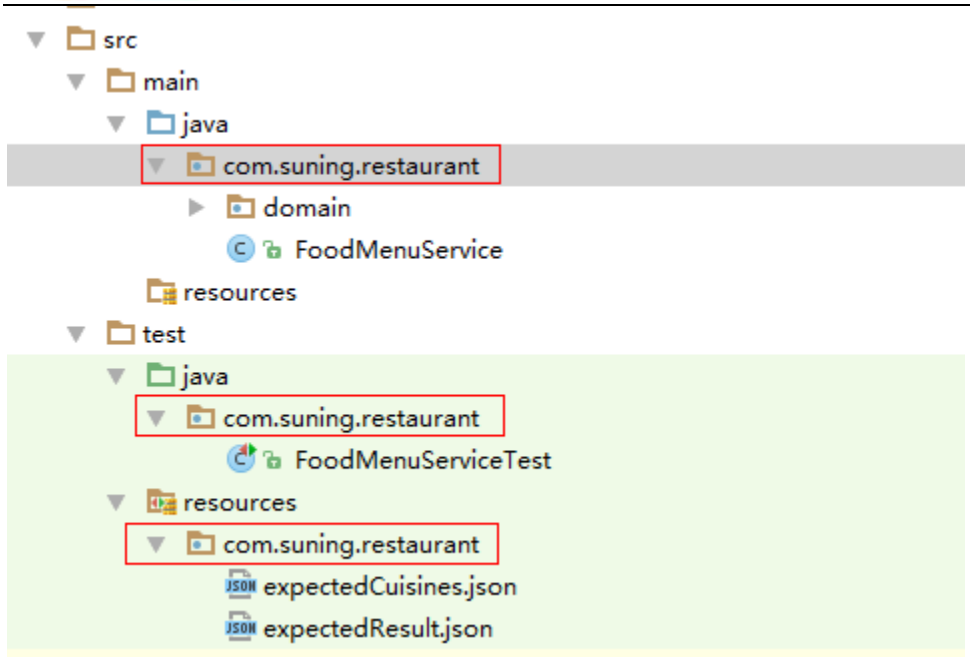
## 2.3　单元测试的包组织管理

创建测试类时，应该可以轻松实现如下目标：

- *找到一个项目的所有相关测试；*
- *找到一个类的所有相关测试；*
- *找到一个方法的所有相关测试。*

为了实现以上目标，创建单元测试时，应遵循以下约定：

将测试映射到被测试单元。比如，我们在 com.suning.restaurant 下有一个 FoodMenuService 类，那么在创建该类的测试类 FoodMenuServicTest 时，在 test 目录下必须建立与 FoodMenuService 相同的目录结构，test resource 目录中同样适用。如下图所示：




## 2.4 单元测试的命名

**测试类的命名规则**：被测试类的类名+Test 后缀。

**测试用例的命名**：由于一个被测试单元可能对应着不同的场景，因此测试用例应根据场景命名。比如某测试用例的场景为“当参数不合法时抛出异常”，则可以命名为“throwExceptionWhenParamsIsInvalid”.

**变量命名**：测试一个工作单元时，预期结果命名为 expectedResult，实际结果命名 actualResult.凡是我们预定的期望的结果均以 expected 作为前缀。

## 2.5 测试用例的结构规范

每个测试用例应用注释注明预设条件、调用 UUT、断言三个部分，每部分中间空一行，如下所示：

```
@Test
public void 根据学号获取学生姓名和分数() {
   JSONObject expectedResult =
JSONObject.parseObject("{'name':'Jack','score':90,'message':'','code
':'0000'}");
   String studentNo = "0001";

   when(studentRepository.findByNo(studentNo)).thenReturn("Jack");
   when(scoreRepository.queryScoreByNo(studentNo)).thenReturn(90);
```

```
    JSONObject actualResult = student.loadResult(studentNo);

    assertEquals(expectedResult, actualResult);
    verify(studentRepository).findByNo(studentNo);
    verify(scoreRepository).queryScoreByNo(studentNo);
}
```

## 2.6 单元测试覆盖率

测试用例应包含测试用例评审时概括的所有场景。被测试工作单元的测试覆盖率应不低于 95%，技术经理在代码评审时检查。

## 2.7 将集成测试与单元测试分离

需要集成测试时，在单元测试同级目录下新建一个“integrationTest”目录，当前被测试类的集成测试均放到该目录下。由于集成测试总是依赖于不确定的第三方，容易失败，且运行较为耗时，因此在提交到版本库前，需要将集成测试添加@Ignore 注解。集成测试不参与自动化测试，需要运行时手动移除注解。如下所示：

```
@Ignore
@Test
public void DescribeYourSceneInEnglish() {

}
```

## 2.8 测试用例的修改与删除

在业务逻辑发生变化或者 API 发生变化时，应首先通过测试用例定位到问题。在业务代码中完成当次调整后，应修复测试用例，以保证测试可以运行通过并与业务代码保持实时同步。

除非业务需要，否则不可以随意删除测试用例。技术经理应在封版或每个迭代结束前检查业务代码的测试覆盖率。

代码发现 BUG 时，应首先编写一个测试来暴露该 BUG.

## 2.9 提交单元测试到版本库

所有单元测试都必须提交到版本库，提交前必须确保所有单元测试均运行通过，未通过的单元测试不得提交到版本库。

## 2.10 优秀单元测试的特性

编写单元测试时，应对照以下特性检查所编写的单元测试是否符合要求。

- ***运行速度快（FAST）****单个测试运行时间应控制在 0.5s 内，最长不可以超过 1s.所有单元测试运行总时间不超过 3s.*
- ***相互隔离，即每个测试相互独立运行（ISOLATED）***

  *单元测试之间彼此隔离，任何一个单元测试的失败不可以影响其他单元测试的正常运行。*
- ***不需要进行外部配置（CONFIGURATION-FREE）***

  *运行单元测试仅需要依赖于测试框架和业务代码，不可以依赖任何额外的配置。在具备相同测试框架的机器上，都应该可以运行通过。单元测试运行时严禁连接数据库，启动 Spring 框架。*
- ***产生稳定的通过或失败结果（CONSISTENT）***

  *所有单元测试可以无限次自由运行，且每次运行的结构是稳定的。*

## 2.11 单元测试质量管理

**详细设计阶段**

根据概要设计文档和产品设计文档，编写测试用例。在详细设计文档中，按照模块划分测试用例。详细设计文档评审时，邀请测试人员和产品经理参与。用例格式如下：

| 模块 | 场景 | 条件 | 预期结果 | 版本 |
|---|---|---|---|---|
| 充值 | 资金发起一笔充值 | 数据合法且此前订单不存在 | 充值订单创建成功 | V1.0 |

注：关于场景的划分，请参考第六章。

**编码阶段**

编码阶段采用测试先行方式。项目组人员一人以上时，交叉编写测试用例。交叉编写有助于了解彼此业务，同时也相当于一次详细设计，可以帮助发现彼此设计中存在的问题，改善架构设计。
测试用例全部编写完毕，开始编写业务代码。每个测试用例通过后，可直接进行代码评审。

**代码评审会阶段**

项目组负责人及技术经理在代码评审时抽查或直接检查测试用例编写情况，也可根据持续集成反馈的结果（参考第七章），抽查测试用例编写质量等问题。

以上三个阶段采用喷泉式迭代，以不断完善测试用例的质量。

# 附录一 Mockito 的使用

## 1.1 引入依赖

```
<mockito-all.version>2.0.2-beta</mockito-all.version>

<dependency>
   <groupId>org.mockito</groupId>
   <artifactId>mockito-all</artifactId>
   <version>${mockito-all.version}</version>
   <scope>test</scope>
</dependency>
```

Mockito 的最新版本可以查看
http://mvnrepository.com/artifact/org.mockito/mockito-all

## 1.2 如何打桩

**说明** 在被测试代码中，如果需要和第三方对象发生交互时，为了避免第三方代码影响当前测试，需要通过打桩以屏蔽其影响。

**示例**

```
//我们不仅可以 mock 一个接口，还可以 mock 一个具体的类。
LinkedList mockedList = mock(LinkedList.class);

//打桩之返回值
when(mockedList.get(0)).thenReturn("first");
//打桩之抛出异常
when(mockedList.get(1)).thenThrow(new RuntimeException());

//虽然 mockedList 是 mock 出来的，没有初始化，但是由于已经设置桩，下面这段语句将在控制台上输出指定内容
System.out.println(mockedList.get(0));
```

```
//根据桩中的设置，下面语句执行时将会抛出异常
System.out.println(mockedList.get(1));

//下面这段语句将会在控制台输出 null，因为 get(999)没有在桩中设置。
System.out.println(mockedList.get(999));

verify(mockedList).get(0);
```

**示例解析**

a) 默认情况下，针对有返回值的方法，Mock 都可以设定一个相同类型的返回值；
b) 值得注意的是，桩是可以被重写（override）的。比如，我们有时可能会在@Before 中已经打桩，但是具体测试用例中的桩与其稍稍有些不同，这是就需要重写。但是，此法虽然可行，但要注意这可能是编码中的坏味道；
c) 一旦对象的行为被打桩，那么无论执行多少次，它都将按照设定返回相应的值。

## 1.3 打桩方法

**说明** 在上面介绍了用 when 来打桩以返回预期的值或抛出预期的值，本节将介绍打桩方法家族中的另外几个方法：doReturn()|doThrow()|doNothing().

**示例**

```
doThrow(new RuntimeException()).when(mockedList).clear();

//following throws RuntimeException:
mockedList.clear();
```

**示例解析**

我们可以使用 doReturn()|doThrow()|doNothing()|doCallRealMethod() 来替代 when()中的相关方法。但是，为了统一和规范，我们推荐使用 when().当被打桩的方法为 void 类型时，可以考虑使用本节介绍的方法。

## 1.4 参数匹配器

**说明** 目标代码在执行过程中,mock 会通过参数比较其是否被设置桩，如果设置，将会按照预定的桩的行为执行。在参数比较的过程中，就需要用到参数匹配器。

**示例**

```
//此桩中使用 anyInt()参数匹配器，表示 mockedList.get()接收到任何 int 类型参数时，都将按照桩中设定执行。
when(mockedList.get(anyInt())).thenReturn("element");

//argThat 允许创建我们自己的匹配器
when(mockedList.contains(argThat(isValid()))).thenReturn("element");

//下面这条语句将打印出"element"
System.out.println(mockedList.get(999));

//验证时的参数匹配器要和打桩时的一致，此处均为 anyInt()
verify(mockedList).get(anyInt());
```

**示例解析**

值得注意的是：如果我们使用了参数匹配器，那么所有的参数都需要使用参数匹配器，如下：

```
verify(mock).someMethod(anyInt(), anyString(), eq("third argument"));
//上面的用法是正确的 - eq() 也是个匹配器

verify(mock).someMethod(anyInt(), anyString(), "third argument");
//这个是错误的，因为第三个参数没有使用匹配器
```

### *1.4.1 ArgumentCaptor*

**说明** 某些场景中，不光要对方法的返回值和调用进行验证，同时需要验证一系列交互后所传入方法的参数。那么我们可以用参数捕获器来捕获传入方法的参数进行验证，看它是否符合我们的要求。适用于没有 `equal()`且对象内属性层次较深的引用类型的参数匹配比较。

**示例**

```
@Test
public void paymentSuccess() throws FileNotFoundException {
    Result paymentResult = new Result();
    PaymentDto paymentDto = BeanUtil.loadFromFile("buildPaymentData.json", PaymentDto.class, this.getClass());
    BatchOrderDto batchOrderDto = this.buildBatchOrderDto();

    ArgumentCaptor<PaymentDto> argumentCaptor = ArgumentCaptor.forClass(PaymentDto.class);
    when(batchOrderManageService.updateBatchOrder(refEq(batchOrderDto))).thenReturn(1);
```

```
when(psClientService.invokePayment(argumentCaptor.capture())).thenReturn(paymentResult
);

    paymentHandleServiceImpl.payment(batchOrderDto);

    verify(batchOrderManageService).updateBatchOrder(refEq(batchOrderDto));
    verify(psClientService).invokePayment(argumentCaptor.capture());
    assertTrue(compare(paymentDto, argumentCaptor.getValue()));
}
```

**示例解析**

psClientService.invokePayment()的参数在代码运行过程中产生，且 PaymentDto 属性中含有另一个对象。此时就应该使用 ArgumentCaptor 匹配器。ArgumentCaptor.getValue()可捕获运行中的实际参数，取出后可以去预期的作比较。compare()是 BeanUtil 中的一个工具方法，用于比较两个对象是否相等，具体用法可参见第五章。

### *1.4.2 refEq*

**说明** refEq 适用于没有 equal()的引用类型的参数匹配比较。

**示例**

```
when(mockedList.get(refEq(order))).thenReturn("element");
```

**示例解析**

参数 order 是引用类型，有多个字段且没有 equals().refEq()会类似于 EqualsBuilder.reflectionEquals(this, other, exlucdeFields)那样通过反射来比较。**注意：refEq 只能进行浅层次的比较，当对象中包含另一对象时，请使用 ArgumentCaptor().**

### *1.4.3 eq*

**说明** eq()匹配器将比较运行时的参数与打桩时的参数是否相等，比较时它将调用两个参数的equals()。

**示例**

```
when(mockedList.get(eq("e"))).thenReturn("element");
```

**示例解析**
当参数对象都具有 equals 时，用 eq()较为合适。但是，在我们的项目中，dto 经常没有 equal 方法，就不适合使用 eq()，而是用 refEq().

### 1.4.4 *any*

**说明** 只匹配参数类型，不匹配参数值。

**示例**

```
when(mockedList.get(anyInt())).thenReturn("element");
```

**示例解析**
此处使用了 anyInt()匹配器，意即只要是 int 类型的参数，无论值如何，都认为是匹配成功的。类似的用法还有 anyBoolean, anyByte, anyChar, anyCollection, anyCollectionOf, anyDouble, anyFloat, anyInt, anyList, anyListOf, anyLong, anyMap, anyMapOf, anyObject, anySet, anySetOf, anyShort, anyString, anyVararg.
在 Mockito 中，这些基本类型都有自己的 any()，但是在我们的项目中，参数常常是自定义的某个 DTO 等。此时，可以使用 any(DTO.class)，比如 anyInt()也可以写成 any(Integer.class)，效果等同。

## 1.5 验证对象行为

**说明** 对于一个 mock 出来的对象，在代码执行结束后，需要验证其相关行为是否发生。被 mock 的对象一旦创建，Mock 框架将会记录其所有行为，我们可以有针对性地选择一些进行验证。

**示例**

```
List mockedList = mock(List.class);
```

```
mockedList.add("one");
mockedList.clear();

verify(mockedList).add("one");
verify(mockedList).clear();
```

**示例解析** 当前示例中，我们 Mock 一个实例 mockedList，后两句的 verify 验证 mockedList 是否执行了 add 和 clear 两个行为。verify 用于验证对象的行为，关于其具体用法，可参见 verify.

## 1.6 验证行为发生的次数

**说明** 一个对象的某个行为在一段执行逻辑中，有可能会发生多次，在验证时需要对其执行次数进行验证。

**示例**

```
mockedList.add("once");

mockedList.add("twice");
mockedList.add("twice");

mockedList.add("three times");
mockedList.add("three times");
mockedList.add("three times");

//下面这两种使用方法效果是一样的。默认是 times(1)
verify(mockedList).add("once");
verify(mockedList, times(1)).add("once");

//次数验证
verify(mockedList, times(2)).add("twice");
verify(mockedList, times(3)).add("three times");

//使用 never(). never() 是 times(0)的别名，两者效果等同
verify(mockedList, never()).add("never happened");

//使用 atLeast()/atMost()
verify(mockedList, atLeastOnce()).add("three times");
verify(mockedList, atLeast(2)).add("five times");
```

```
verify(mockedList, atMost(5)).add("three times");
```

**示例解析**

`times(1)`是默认的，所以使用时可以省略。

```
//passes when someMethod() is called within given time span
verify(mock, timeout(100)).someMethod();
//above is an alias to:
verify(mock, timeout(100).times(1)).someMethod();

//passes when someMethod() is called *exactly* 2 times within given
time span
verify(mock, timeout(100).times(2)).someMethod();

//passes when someMethod() is called *at least* 2 times within given
time span
verify(mock, timeout(100).atLeast(2)).someMethod();

//verifies someMethod() within given time span using given
verification mode
//useful only if you have your own custom verification modes.
verify(mock, new Timeout(100,
yourOwnVerificationMode)).someMethod();
```

## 1.7 验证行为超时

**说明** 在并发编程或者某种情况下，需要对对象的某个行为需要限时。如未能在指定时间完成，则视为超时失败。

**示例**

```
//超过 100 毫秒则验证失败
verify(mock, timeout(100)).someMethod();
//上面的验证方法和下面的同效:
verify(mock, timeout(100).times(1)).someMethod();

//两次调用在 100 毫秒内完成
verify(mock, timeout(100).times(2)).someMethod();

//100 毫秒内至少完成两次
verify(mock, timeout(100).atLeast(2)).someMethod();
```

**示例解析**
验证行为超时的特性运用的不多。

## 1.8 测试用例超时

**说明** 通过 timeout 属性限制测试的运行时间。

**示例**

```
@Test(timeout = 10)
public void shouldDoSomething() {
    manager.initiateArticle();
    verify(database).addListener(any(ArticleListener.class));
}
```

**示例解析**
若超过 10 毫秒该测试用例仍未运行结束，则视为测试失败。

## 1.9 测试预期异常

**说明** 测试代码中期望的异常抛出。

**示例**

```
@Rule
public ExpectedException thrown= ExpectedException.none();
@Test
public void vote_throws_IllegalArgumentException_for_zero_age() {
    Student student = new Student();
    //期望的异常类型为 IllegalArgumentException
    thrown.expect(IllegalArgumentException.class);
    //期望的异常信息：age should be +ve
   thrown.expectMessage("age should be +ve");
   student.vote(0);
}
```

**示例解析**
执行 student.vote(0)时，如果确实有期望的异常，则测试通过。否则，测试失败。

## 1.10 用@Mock 简化 mock 对象的创建

**说明** 在此前的例子，mock 一个对象时，用的是 mock(List.class)等，本节将描述如何简化创建，这将使我们的代码更简洁、更易读。

**示例**

```
//等同于 ArticleCalculator calculator=mock(ArticleCalculator.clss);
 @Mock
 private ArticleCalculator calculator;
 @Mock
 private ArticleDatabase database;
 @Mock
 private UserProvider userProvider;

 private ArticleManager manager;
```

**示例解析**
需要注意的是，使用该方法创建 mock 对象时，需要在测试用例执行前执行以下代码。通常，这句代码可以放在测试基类或者@Before 中。

```
MockitoAnnotations.initMocks(testClass);
```

如果不使用 MockitoAnnotations.initMocks(testClass)的话，也可以使用 MockitoRule 方法：

```
public class ExampleTest {

    @Rule
    public MockitoRule rule = MockitoJUnit.rule();

    @Mock
    private List list;

    @Test
    public void shouldDoSomething() {
        list.add(100);
    }
}
```

## 1.11 用@spy 模拟真实对象的部分行为

**说明** 在某些情况下，我们需要使用一个真实对象。但是，我们同时需要自定义该对象的部分行为，此时用@spy 就可以帮我们达到这个目的。

**示例**

```
//通过 spy 创建一个可以模拟部分行为的对象,spy()接收的参数是一个真实对象
List list = spy(new LinkedList());
list.add(1);

when(list.get(0)).thenThrow(new Exception());

//如果 get(0)正常执行的话，将会返回 1.但是由于对象经过 spy 并打桩后，它将抛出异常
list.get(0);

//与 spy 相反的是，也可以让被 mock 的对象执行真实的行为
Foo mock = mock(Foo.class);
//Be sure the real implementation is 'safe'.
//If real implementation throws exceptions or depends on specific state of the object then you're in trouble.
when(mock.someMethod()).thenCallRealMethod();
```

**示例解析**
使用 thenCallRealMethod 时，需要注意真实的实现部分是安全的，否则将会带来麻烦。

**注意　Mock 和 spy 用法的区别在于：当测试用例中需要使用某个对象的真实方法更多些时，请使用 spy，反之请使用 Mock.**

## 1.12 用@InjectMocks 完成依赖注入

**说明** 类中需要第三方协作者时，通常会用到 get 和 set 方法注入。通过 spring 框架也可以同@Autowird 等方式完成自动注入。在单元测试中，没有启动 spring 框架，此时就需要通过 InjectMocks 完成依赖注入。InjectMocks 会将带有@Spy 和@Mock 注解的对象尝试注入到被测试的目标类中。

**示例**

```
public class ArticleManagerTest extends SampleBaseTestCase {

    @Mock
```

```
    private ArticleCalculator calculator;
    @Mock(name = "database")
    private ArticleDatabase dbMock;
     @Spy
     private UserProvider userProvider = new ConsumerUserProvider();

     @InjectMocks
     private ArticleManager manager;

     @Test public void shouldDoSomething() {
         manager.initiateArticle();
         verify(database).addListener(any(ArticleListener.class));
     }
}
```

**示例解析**

InjectMocks 在注入依赖的对象前，会首先创建一个目标对象的实例。所以，manager 有@InjectMocks 后，不需要再手动创建实例。

**注意：在前面的实例中，我们创建 mock 对象时没有使用@Mock.在工作中，请全部使用@Mock,@Spy,@InjectMocks.**

## 1.13 模拟静态方法

**说明** 与普通方法不同的是，静态方法必须通过其类直接调用。如此，Mockito 中的方法将不再适用于静态方法。此时需要使用 PowerMock.

PowerMock 是一个扩展了其它如 EasyMock 等 mock 框架的、功能更加强大的框架。PowerMock 使用一个自定义类加载器和字节码操作来模拟静态方法，构造函数，final 类和方法，私有方法，去除静态初始化器等等。通过使用自定义的类加载器，简化采用的 IDE 或持续集成服务器不需要做任何改变。熟悉 PowerMock 支持的 mock 框架的开发人员会发现 PowerMock 很容易使用，因为对于静态方法和构造器来说，整个的期望 API 是一样的。PowerMock 旨在用少量的方法和注解扩展现有的 API 来实现额外的功能。目前 PowerMock 支持 EasyMock 和 Mockito。

**引入依赖**

```
<powermock-api-mockito.version>1.6.2</powermock-api-mockito.version>
<powermock-module-junit4.version>1.6.2</powermock-module-junit4.version>
<powermock-core.version>1.6.2</powermock-core.version>

<dependency>
  <groupId>org.powermock</groupId>
  <artifactId>powermock-api-mockito</artifactId>
  <version>${powermock-api-mockito.version}</version>
  <scope>test</scope>
```

```
</dependency>
<dependency>
  <groupId>org.powermock</groupId>
  <artifactId>powermock-module-junit4</artifactId>
  <version>${powermock-module-junit4.version}</version>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>org.powermock</groupId>
  <artifactId>powermock-core</artifactId>
  <version>${powermock-core.version}</version>
  <scope>test</scope>
</dependency>
```

**示例**

```
//带模拟的静态类
public class FileHelper {
    public static boolean isExist() {
        return false;
    }
}


@RunWith(PowerMockRunner.class)
public class TestClassUnderTest {
    @Test
    @PrepareForTest(FileHelper.class)
    public void testCallStaticMethod() {
        ClassUnderTest underTest = new ClassUnderTest();
        PowerMockito.mockStatic(FileHelper.class);
        PowerMockito.when(FileHelper.isExist()).thenReturn(true);

        Assert.assertTrue(underTest.callStaticMethod());
        verifyStatic();
        FileHelper.isExist()
    }
}
```

**示例解析**

PowerMockRunner 在模拟静态时配置较多，使用时需要注意。另外，静态方法断言时，必须要有 verifyStatic(),然后需要断言的方法紧随其后，如上。

## 1.14 Mock 使用时的注意事项

若单个测试用例中（限领域对象）出现了三个及以上的 Mock 对象时，表明被测试的单元与第三方发生了较多的交互，可能违反了单一职责原则。此时，你可能需要暂停测试用例的编写，检查代码设计是否存在设计问题，以保证业务代码和测试用例的可读性。

测试用例的编写与重构紧密结合。代码的质量直接影响测试用例的编写，而代码质量的提高依赖于设计和重构。如果发现测试用例编写比较困难，你可能需要重新审视代码设计是否合理。

# 附录二 断言

## 2.1 assertEquals

**说明** 比较两个对象的原始值。如果对象是值类型时，将调用对象的 `equals()`.

**示例**

```
assertEqual(expectedResult,actualResult);
```

## 2.2 assertTrue

**说明** 验证结果是否为 `true`.

**示例**

```
assertTrue(actualResult);
```

## 2.3 assertFalse

**说明** 验证结果是否为 `false`.

**示例**

```
assertFalse(actualResult);
```

## 2.4 assertNotNull

**说明** 验证一个结果是否不为 `null`.

**示例**

```
assertNotNull(actualResult)
```

## 2.5 assertNull

**说明** 验证一个结果是否为 null.

**示例**

```
assertNull(actualResult)
```

## 2.6 assertSame

**说明** 比较两个对象的引用地址是否相同。

**示例**

```
assertSame(expectedResult,actualResult);
```

## 2.7 assertNotSame

**说明** 比较两个对象的引用地址是否不相同。

**示例**

```
assertNotSame(expectedResult,actualResult);
```

# 附录三　BeanUtil 的使用

为方便编写测试用例，部门重新组织了 epps-puma 组件，并在该组件中引入 BeanUtil 工具类。

## 3.1 引入依赖

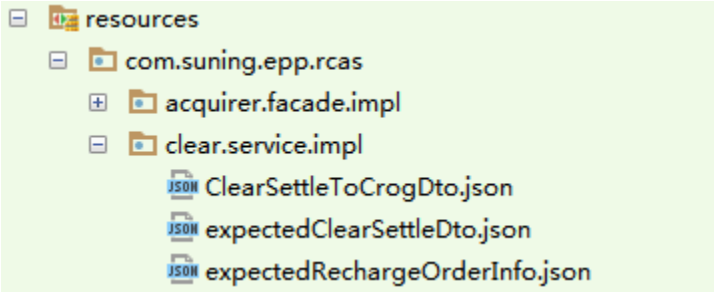
```
<dependency>
  <groupId>com.suning.epp</groupId>
  <artifactId>epps-puma</artifactId>
  <version>0.0.9</version>
</dependency>
```

## 3.2 从 JSON 文件中构建 POJO 对象

**说明** 编写测试时，如果目标代码中包含了计算、转换等数据操作时，需要传入真实的对象。如果不借助工具，手动初始化相关对象不仅繁琐，还会降低代码可读性。BeanUtil.fromFile() 正是为了解决此问题而生。

**示例**

**a）在 test resources 下建立与被测试类相同层级的目录结构，添加相关 JSON 文件**




**b)JSON 文件内容如下：**

```json
{
  "merchantSerialNo": "324232",
  "aqPayOrderId": "1",
  "serialOrderId": "2",
  "clearOrderNo": "323222",
  "acquireType": "05",
  "productCode": "00050000184",
  "currency": "CNY",
  "payerNo": "10002541",
  "payerType": "CORPORAT",
  "payerMerchantNo": "32320193",
  "payeeNo": "10002541",
  "payeeType": "CORPORAT",
  "payeeMerchantNo": "32320193",
  "payTypeGroup": [
    {
      "待定": "222"
    }
  ],
  "payAmount": "100",
  "poundageType": "1",
  "poundageAmount": "0",
  "capDataType": "st2corg"
}
```

**c)调用时用法如下：**

```
RechargeOrderInfo rechargeOrderInfo =
BeanUtil.loadFromFile("expectedRechargeOrderInfo.json",
RechargeOrderInfo.class, this.getClass());
```

**示例解析**

该方法适合构建字段较多的 POJO，使用时注意文件名的命名要能准确反映出其构建的对象信息。

**注意**：fromFile 仅支持普通 DTO 的转换，如果需要转换一个 List 对象，则需要使用 fromArrayFile 方法，用法如下：

```
List<TradeOrderDmo> TradeOrderDmos =
fromArrayFile("fileName",TradeOrderDmo.class);
```

**当目标类型为泛型时，使用 toGeneric().**比如

```
SDKGenericResult<ValidRuleResponseDto> expectRuleResult =
BeanUtil.toGeneric("ruleResult_accountPayMode", new
TypeReference<SDKGenericResult<ValidRuleResponseDto>>(){});
```

## 3.3 从简易字符串构建 POJO 对象

**说明** BeanUtil.loadFromFile()可以快速构建复杂的 POJO 对象。当 POJO 对象较为简单时，可以不必建立文件。BeanUtil.build()可以较为方便地创建出简单的 POJO 对象。

**示例**

```
ClearSettleCallbackResult settleCallbackResult =
BeanUtil.build("acquireOrderNo:1,clearOrderNo:323222",
ClearSettleCallbackResult.class);
```

**示例解析**

注意：fromFile 仅支持普通 DTO 的转换，如果需要转换一个 List 对象，则需要使用 fromArrayFile 方法，用法如下：

```
List<TradeOrderDmo> TradeOrderDmos =
fromArrayFile("fileName",TradeOrderDmo.class);
```

BeanUtil.build()还不够强大，功能仍较为单一。后续的工作若有需要，可适当拓展。

## 3.4 从 POJO 快速构建 JSON 文件

**说明** BeanUtil.printJSONString 用于通过 POJO 快速构建 JSON 字符串.

**示例**

```
@Test
public void test() throws Exception {
   BeanUtil.printJSONString(TradeOrderDmo.class);
}
```

**示例解析**

TradeOrderDmo.class 传入 BeanUtil.printJSONString 后，将在控制台打印出如下 JSON 字符串：

```
{"acqId":"","acqType":"","amount":0,"createTime":"2015-10-23
10:56:22","csOrderId":"","csSerialId":"","csTime":"2015-10-23
10:56:22","csrOrderId":"","csrSerialId":"","csrTime":"2015-10-23
```

10:56:22","delStatus":"","goodsType":"","id":0,"inOutFlag":0,"lastUpdateTime":"2015-10-23
10:56:22","merOrderNo":"","orderName":"","orderStatus":"","otherBankCardno":"","otherBankName":"","otherBankUserName":"","otherCurrency":"","otherUserAlias":"","otherUserName":"","otherUserNo":"","otherUserType":"","outOrderNo":"","ownerCurrency":"","ownerUserAlias":"","ownerUserName":"","ownerUserNo":"","ownerUserType":"","payDetail":"","poundage":0,"poundageType":"","prOrderId":"","prSerialId":"","prTime":"2015-10-23
10:56:22","productCode":"","psOrderId":"","psSerialId":"","psTime":"2015-10-23
10:56:22","refundAmount":0,"refundDetail":"","sysVersion":"","ticketRefundTime":"2015-10-23 10:56:22","tradeMerNo":"","withdrawTime":"2015-10-23 10:56:22"}

这些字符串可以直接粘贴到 JSON 文件中使用。

## 3.5 对象比较

**说明** BeanUtil.compare () 用于比较两个对象是否相等。

**示例**

```
@Test
public void test() throws Exception {
    assertEquals(3, 3);

    Map dataObject1 = MapUtil.newMap("name", "jack", "age", 20,"score",80);
    Map dataObject2 = MapUtil.newMap("name", "lily", "age", 20,"score",85);
    assertTrue(compare(dataObject1,dataObject2));
}
```

**示例解析**

compare()通过将对象转换成 JSON 后进行比较，可以进行深层次的比较。在示例的测试中，由于两个对象中的 name 和 score 属性值不一样，所以两者不相等。在项目中，经常有两个待比较的对象的个别属性无法比较，比如其中一个属性是在运行时复制 new Date().为了排除类似的不可比较的属性，compare 提供了第三个参数 excludeFields，用法如下：

```
@Test
public void test() throws Exception {
    assertEquals(3, 3);

    Map dataObject1 = MapUtil.newMap("name", "jack", "age", 20,"score",80);
    Map dataObject2 = MapUtil.newMap("name", "lily", "age", 20,"score",85);
    assertTrue(compare(dataObject1,dataObject2,"name","score"));
}
```

## 3.6 用 MapUtil 快速构建 Map

**说明** MapUtil.*newMap()* 用于在编写测试用例时快速构建 Map.

**示例**

```
@Test
public void test() throws Exception {
    assertEquals(3, 3);

    Map dataObject1 = MapUtil.newMap("name", "jack", "age", 20,"score",80);
    Map dataObject2 = MapUtil.newMap("name", "lily", "age", 20,"score",85);
    assertTrue(compare(dataObject1,dataObject2));
}
```

**示例解析**

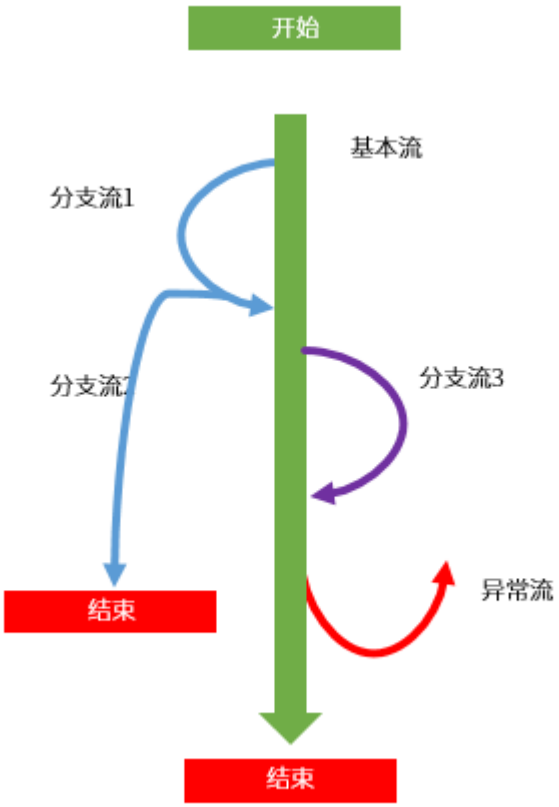
MapUtil.newMap()接收的参数中，基数位参数为键，偶数为参数为值。

# 附录四 用例场景划分

## 4.1 划分原则

**总原则：先基本后分支，先正常后异常。**

用例场景要通过描述流经用例的路径来确定，这个流经过程要从用例开始到结束遍历其中所有基本流和分支流。由此会产生很多组场景，如下图所示：




**基本流**：经过测试用例最简单的路径。

**分支流**：一个分支流可能从基本流开始，在某个特定条件下执行，然后重新加入基本流中（如分支流 1 和 3）；也可能起源于另一个分支流（如分支流 2），或者终止用例而不再重新加入到某个流（如分支流 2 和异常流）。

**先基本后分支**

基本流是业务的基本逻辑，也是业务逻辑的主线。先从基本流入手，一方面有助于抓住业务主线，另一方面在不考虑分支的情况下，先编写基本分支的测试用例，有助于测试用例的逐步深入，保持清晰的层次逻辑。

**先正常后异常**

先编写正常（正面）的测试用例，即在不发生异常情况下的业务逻辑。正面用例覆盖完全后，再编写异常（负面）的测试用例，用于测试程序对异常的处理情况。

正常的测试用例包括正常的基本流和分支流，程序中的异常通常包括：

- 参数校验错误；
- 数据不符合正常的业务要求；
- 边界错误；
- 数据库操作失败；
- 第三方系统调用异常。

## 4.2 划分举例

```
@Override
public OrderAcquirerResult acquireRecharge(RechargeOrderDto rechargeOrder) throws
Throwable {
    Logger.info("[充值收单]充值收单处理开始：订单数据{}", rechargeOrder);
    OrderAcquirerResult orderAcquirerResult = new OrderAcquirerResult();
    try {
        Long rechargeId = orderQueryManageService.acquireRecharge(rechargeOrder);
        orderAcquirerResult.setRechargeOrderId(String.valueOf(rechargeId));
        orderAcquirerResult.setStatus(OrderConstants.ORDER_NEW);
        sendPaymentEventService.sendPaymentEvent(rechargeId);
    } catch (Throwable e) {
        RechargeOrderInfo order =
orderQueryManageService.queryRechargeOrder(rechargeOrder.getTradeMerchantNo(),
                rechargeOrder.getMerchantSerialNo(), rechargeOrder.getOrderDate());
        if (order == null) {
            Logger.info("[充值收单]订单不存在");
            throw e;
        }
        Logger.info("[充值收单]订单已经存在");
        orderAcquirerResult.setRechargeOrderId(String.valueOf(order.getId()));
        orderAcquirerResult.setStatus(order.getStatus());
        if (theOrderStateIsPaymentFailOrClearSettleSuccess(order)) {
            Logger.info("[充值收单]插入通知补偿数据：订单数据{}", rechargeOrder);
            compensationPersistService.insertCompByKeyAndType(order.getId(),
                    OrderConstants.COMPENSATION_TYPE_NOTICE);
            sendNoticeService.sendNotice(order.getId());
```

```
        }
        orderAcquirerResult.setResponseMsg("订单受理成功。");

    }
    Logger.info("exit:充值收单处理结束 orderAcquirerResult = {}", orderAcquirerResult);
    return orderAcquirerResult;
}
```

这是充值收单中的一段业务代码。

其基本主线是：程序接收到充值请求后，在本地创建一个充值订单，订单状态为新建，创建完毕后发送信息到支付。

除了基本流外，这段代码也存在分支流和异常流。当订单已经存在时，数据库将保存失败，抛出异常。在异常流中，仍有分支存在：异常是由订单存在导致的主键冲突异常和订单不存在时的未知异常。如果订单存在，还需根据订单的状态考虑是否要插入通知补偿数据。

综上，本段业务共有 5 个场景：

| 场景编号 | 场景 | 条件 | 预期结果 | 版本 |
|---|---|---|---|---|
| 1 | 订单创建成功 | 此前订单不存在 | 数据库产生相应订单，调用支付正常，返回订单受理成功 | V1.0 |
| 2 | 订单创建失败-订单不存在 | 收单时出错 | 抛出未知异常 | V1.0 |
| 3 | 订单创建失败-订单已经存在且无需插入通知补偿数据 | 订单在数据库中已经存在，且订单状态非支付失败和清结算成功 | 返回订单受理成功 | V1.0 |
| 4 | 订单创建失败-订单已经存在且需要插入通知补偿数据（订单状态为支付失败） | 订单在数据库中已经存在，且订单状态为支付失败 | 返回订单受理成功 | V1.0 |
| 5 | 订单创建失败-订单已经 | 订单在数据库中 | 返回订单受理成 | V1.0 |

| | | | | |
|---|---|---|---|---|
| | 存在且需要插入通知补偿数据（订单状态为清结算成功） | 已经存在，且订单状态为清结算成功 | 功 | |

五个场景对应着五个测试用例。通常，在测试后行项目中，既有代码中的每个 `if` 中的一个条件即为一个场景。

# 附录五 测试自动化配置

Jenkins 是一个持续集成平台，利用 Jenkins 可以通过持续集成来实现测试自动化，并将测试结果以邮件形式发送至邮件配置的相关角色。

## 5.1 接入持续集成平台

1、选择一个分支（未封板）：


2.选择 **持续集成** 点击 **修改定制信息**


3.参考下图配置


分支基本信息
分支权限信息
持续集成
【Sonar代码质量分析设置】
每日三次：每8小时构建一次；每日一次：每天构建一次；每3小时一次；每3小时构建一次；每小时一次：每小时构建一次；未做打包配置项目暂不支持构建。
*定时计划：
每小时一次
自定义表达式：
H * * * *
扩展分析：
代码静态检查
单元测试
依赖jar包安全扫描
安全扫描检查
前端性能检查
蛙测
邮件通知触发器：
每次构建
邮件发送角色：
技术经理
产品经理
技术总监
产品总监
开发工程师
测试工程师
测试经理
系统管理员
产品顾问
EA架构师
界面架构师
运维工程师
产品负责人
技术负责人
安全扫描员
附加审批人员
发布专员
静态发布专员
版本CMO
版本团队
【在线代码评审设置】
ReviewBoard是一个开源的代码评审工具。持续交付平台集成了svn与ReviewBoard，使代码评审更方便！ 更多ReviewBoard资料见： ReviewBoard相关文档
使用ReviewBoard评审：
是
否
保存信息