# 参考

Java编程技巧之单元测试用例编写流程： https://developer.aliyun.com/article/783992

如何写出有效的单元测试： 　　　　　　　 https://developer.aliyun.com/article/1002205

5个编写技巧，有效提高单元测试实践：　　https://developer.aliyun.com/article/1081898

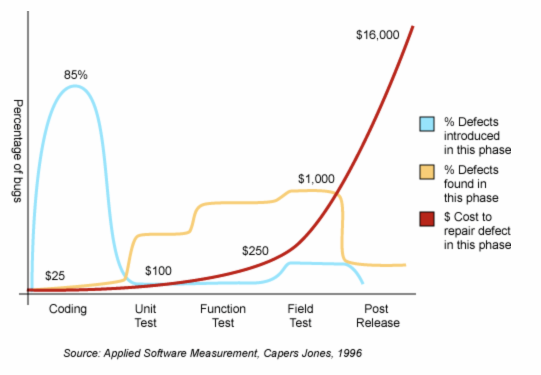
https://www.baeldung.com/junit

实战（基于Mockito）：

https://www.bilibili.com/video/BV15S4y1F7Xr/?spm\_id\_from=333.337.search-card.all.click

# 为何要写单元测试

* 减少BUG，释放资源



* 为代码重构保驾护航
* 既是编写单测也是CodeReview
* 便于调试与验证
* 驱动设计与重构

# 单元测试理论

## FIRST原则

FAST(快速原则)：单元测试应该是可以快速运行的，在各种测试方法中，单元测试的运行速度是最快的，通常应该在几分钟内运行完毕

Independent(独立原则): 单元测试应该是可以独立运行的，单元测试用例互相无强依赖，无对外部资源的强依赖

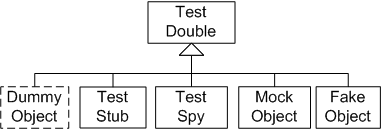
Repeatable(可重复原则): 单元测试应该可以稳定重复的运行，并且每次运行的结果都是相同的

Self Validating(自我验证原则): 单元测试应该是用例自动进行验证的，不能依赖人工验证(比如println打印结果，但是需要人工确认）

Timely(及时原则）: 单元测试必须及时的进行编写，更新和维护，以保证用例可以随着业务代码的变化动态的保障质量

## 测试替身

<http://xunitpatterns.com/Test%20Double.html>



* Dummy：一种什么也不做的实现方式。接口中的每个方法什么也不做，如果方法有返回值，返回的值尽量接近null或者0。
* Stub：Dummy的一种，Stub的函数并不返回null或0，而是返回能推动函数沿预定路径被测试的值。
* Spy：Stub的一种，它返回测试所需的特定值，推动系统沿着我们期望的路径前行。
* Mock：Spy的一种，它返回测试所需的特定值，推动系统沿着我们期望的路径前行，而且还会记住对它所做的事。不过，Mock还知道我们的预期，基于这些预期，判断测试是否通过；换而言之，Mock中写明了测试断言。
* Fake：Fake是一种模拟器，它实现基础业务规则，这样测试就能要求该Fake按需要的路径执行。

# 单元测试方法规范

## 测试方法的命名

建议采用should\_{预期结果}\_when\_{被测方法}\_given\_{给定场景}

@Test

public void should\_returnFalse\_when\_deleteContent\_given\_invokeFailed() {

...

}

反例

@Test

public void testDeleteContent() {

...

}

## 单测方法实现分层

单测方法的实现如果分层清晰，能让代码便于理解，一目了然，同时也能提高后续的CR的效率

建议采用given-when-then的三段落结构

@Test

public void should\_returnFalse\_when\_deleteContent\_given\_invokeFailed() {

// given

Result<Boolean> deleteDocResult = new Result<>();

deleteDocResult.setEntity(Boolean.FALSE);

when(docManageService.deleteContentDoc(anyLong())).thenReturn(deleteDocResult);

when(docManageService.queryContentDoc(anyLong())).thenReturn(new DocEntity());

// when

Long contentId = 123L;

Boolean result = contentService.deleteContent(contentId);

// then

verify(docManageService, times(1)).queryContentDoc(contentId);

verify(docManageService, times(1)).deleteContentDoc(contentId);

Assert.assertFalse(result);

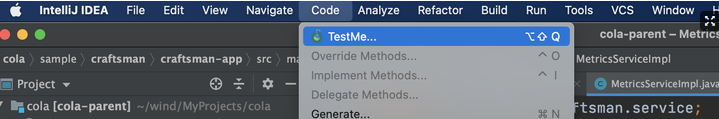
}

# 实战：单测生成插件

IDEA有两款比较好用的单测自动生成插件TestMe与Diffblue，这里主要介绍TestMe，如果大家有比较好的插件也可以推荐。

1. 安装：在IDEA设置中的Plguins插件里搜索TestMe，下载安装即可。

2. 使用：在code按钮找到入口，或者直接使用快捷键option+shift+Q



生成代码效果



# 实战：JUnit

JUnit是目前Java语言应用最为广泛的单元测试框架，用于编写和运行可重复的自动化测试，它包含以下特性：

• 用于测试期望结果的断言（Assertion）

• 用于共享共同测试数据的测试工具

• 用于方便的组织和运行测试的测试套件

• 图形和文本的测试运行器

## Pom引入

Springboot项目

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

Springmvc项目

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

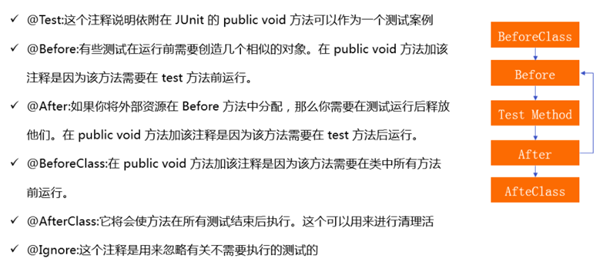
</dependency>

## 断言

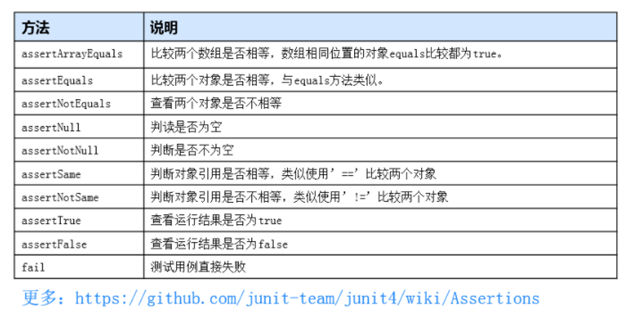
### 执行顺序

在类上面指定用例执行顺序：

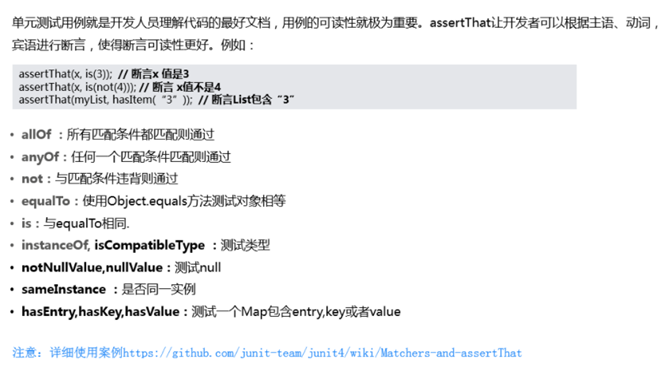
* @FixMethodOrder，取值JVM(代码定义顺序）
* NAME\_ASCEDING(方法字母顺序)
* DEFAULT(不可预期顺序）



### Assert



### AssertThat



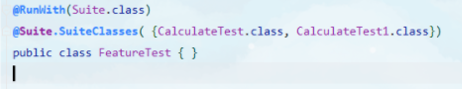
### 异常

　　@Test( expected = NullPointerException.class)

### 参数化测试 @RunWith

@RunWith(Parameterized.class)

### 测试套



## Junit 5 vs Junit 4

https://www.baeldung.com/junit-5-migration

注解变化

@Before -> @BeforeEach

@After -> @AfterEach

@BeforeClass -> @BeforeAll

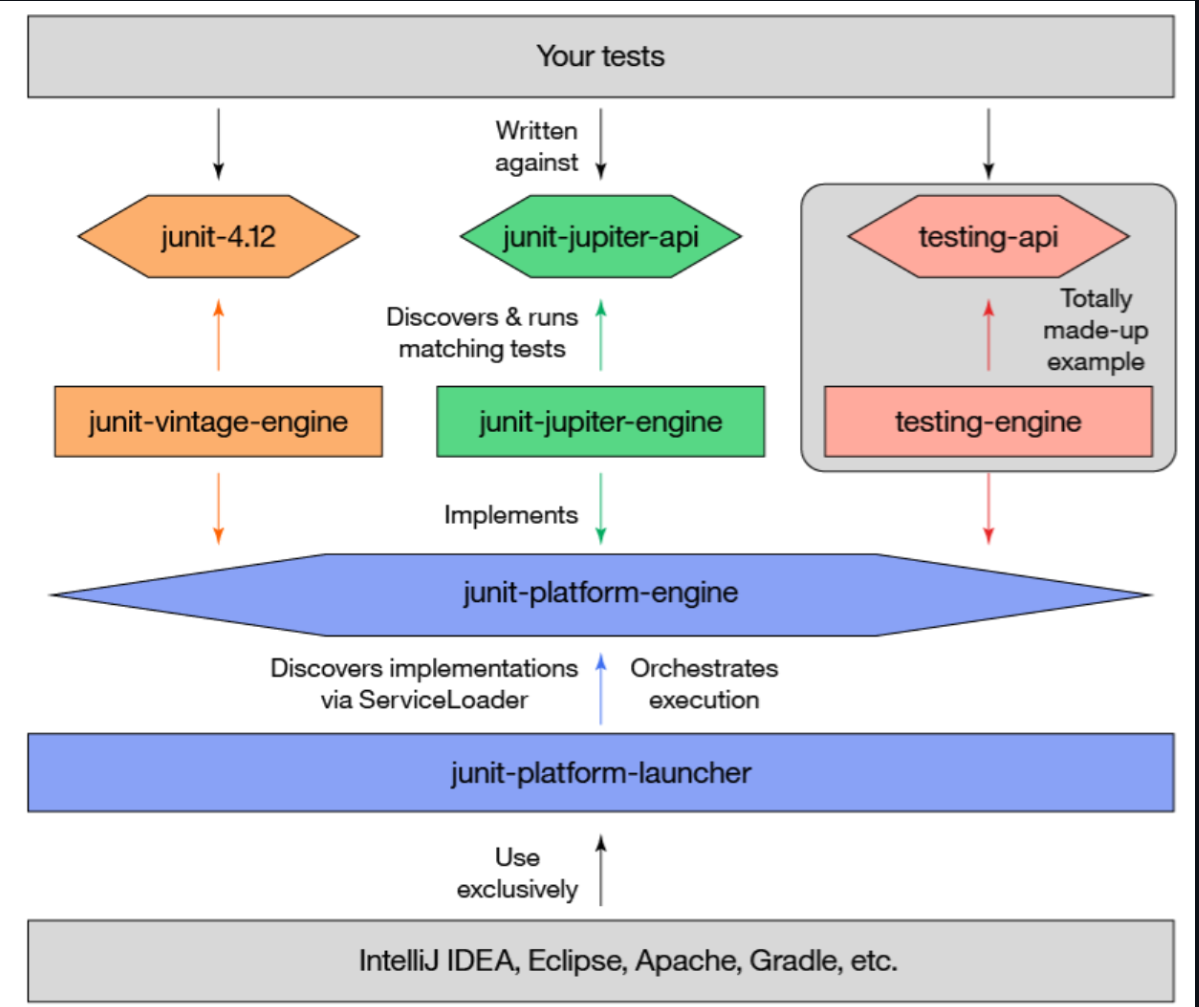
@AfterClass -> @AfterAll

@Ingore -> @Disabled

@RunWith -> @ExtendWith

与以前版本的JUnit不同，JUnit 5由三个不同子项目中的几个不同模块组成

* JUnit Platform 是基于JVM的运行测试的基础框架在，它定义了开发运行在这个测试框架上的TestEngine API。此外该平台提供了一个控制台启动器，可以从命令行启动平台，可以为Gradle和 Maven构建插件，同时提供基于JUnit 4的Runner。
* JUnit Jupiter 是在JUnit 5中编写测试和扩展的新编程模型和扩展模型的组合.Jupiter子项目提供了一个TestEngine在平台上运行基于Jupiter的测试。
* JUnit Vintage 提供了一个TestEngine在平台上运行基于JUnit 3和JUnit 4的测试



# 实战：Mockito

单元测试Mock框架做的事就是模拟被测试类的依赖项，提供预期的行为和状态。

Mockito与PowerMock，两者都是作为单元测试模拟框架，模拟应用中复杂的依赖对象。Mockito基于动态代理的方式实现，PowerMock在Mockito基础上增加了类加载器以及字节码篡改技术，使其可以实现完成对private/static/final方法的Mock。

推荐使用Mockito来作为我们的单元测试Mock框架，原因有二：

1. 在版本3.4.0以后，Mockito支持静态方法的mock。并且作为SpringBootTest默认集成的Mock工具，所以建议大家使用高版本的Mockito，并通过它来完成静态方法的Mock

2. 不提倡使用PowerMock，并不是一味追求单测覆盖率，而是当我们需要使用到具备高级特性mock工具时，我们需要审视代码的合理性，并尝试进行优化重构，使其具备较好的可测性

## Pom引入

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>4.7.0</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-inline</artifactId>

<version>4.7.0</version>

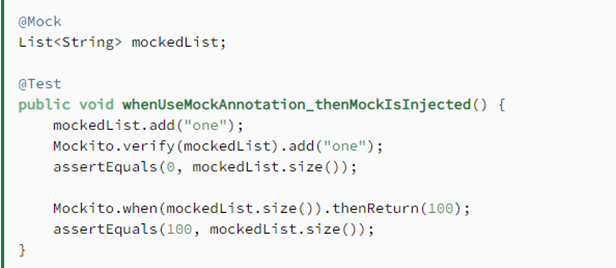
<scope>test</scope>

</dependency>

## 注解

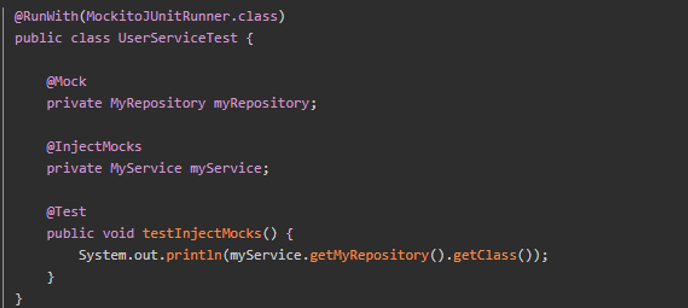
### @Mock

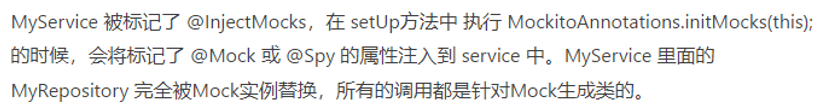
@Mock：在Mockito中用于创建mock对象，使用方法如下：



### @InjectMock

把这些类成员对象注入到被测试类的实例中。以便在调用被测试方法时，可能使用这些类成员对象，而不至于抛出空指针异常。





# 实战：用例编写过程

参考：Java编程技巧之单元测试用例编写流程

<https://developer.aliyun.com/article/783992#slide-25>

## 样例

### 典型代码

@Service

public class UserService {

*/\*\* 服务相关 \*/*

*/\*\* 用户DAO \*/*

@Autowired

private UserDAO userDAO;

*/\*\* 标识生成器 \*/*

@Autowired

private IdGenerator idGenerator;

*/\*\* 参数相关 \*/*

*/\*\* 可以修改 \*/*

@Value("${userService.canModify}")

private Boolean canModify;

*/\*\**

*\* 创建用户*

*\**

*\* @param userCreate 用户创建*

*\* @return 用户标识*

*\*/*

public Long createUser(UserVO userCreate) {

*// 获取用户标识*

Long userId = userDAO.getIdByName(userCreate.getName());

*// 根据存在处理*

*// 根据存在处理: 不存在则创建*

if (Objects.isNull(userId)) {

userId = idGenerator.next();

UserDO create = new UserDO();

create.setId(userId);

create.setName(userCreate.getName());

userDAO.create(create);

}

*// 根据存在处理: 已存在可修改*

else if (Boolean.TRUE.equals(canModify)) {

UserDO modify = new UserDO();

modify.setId(userId);

modify.setName(userCreate.getName());

userDAO.modify(modify);

}

*// 根据存在处理: 已存在禁修改*

else {

throw new UnsupportedOperationException("不支持修改");

}

*// 返回用户标识*

return userId;

}

}

### **测试用例文件**

**UserServiceTest.java：**

*/\*\**

*\* 用户服务测试类*

*\*/*

@RunWith(PowerMockRunner.class)

public class UserServiceTest {

*/\*\* 模拟依赖对象 \*/*

*/\*\* 用户DAO \*/*

@Mock

private UserDAO userDAO;

*/\*\* 标识生成器 \*/*

@Mock

private IdGenerator idGenerator;

*/\*\* 定义被测对象 \*/*

*/\*\* 用户服务 \*/*

@InjectMocks

private UserService userService;

*/\*\**

*\* 在测试之前*

*\*/*

@Before

public void beforeTest() {

*// 注入依赖对象*

Whitebox.setInternalState(userService, "canModify", Boolean.TRUE);

}

*/\*\**

*\* 测试: 创建用户-新*

*\*/*

@Test

public void testCreateUserWithNew() {

*// 模拟依赖方法*

*// 模拟依赖方法: userDAO.getByName* Mockito.doReturn(null).when(userDAO).getIdByName(Mockito.anyString());

*// 模拟依赖方法: idGenerator.next*

Long userId = 1L;

Mockito.doReturn(userId).when(idGenerator).next();

*// 调用被测方法*

String text = ResourceHelper.getResourceAsString(getClass(), "userCreateVO.json");

UserVO userCreate = JSON.parseObject(text, UserVO.class);

Assert.assertEquals("用户标识不一致", userId, userService.createUser(userCreate));

*// 验证依赖方法*

*// 验证依赖方法: userDAO.getByName*

Mockito.verify(userDAO).getIdByName(userCreate.getName());

*// 验证依赖方法: idGenerator.next*

Mockito.verify(idGenerator).next();

*// 验证依赖方法: userDAO.create*

ArgumentCaptor < UserDO> userCreateCaptor = ArgumentCaptor.forClass(UserDO.class);

Mockito.verify(userDAO).create(userCreateCaptor.capture());

text = ResourceHelper.getResourceAsString(getClass(), "userCreateDO.json");

Assert.assertEquals("用户创建不一致", text, JSON.toJSONString(userCreateCaptor.getValue()));

*// 验证依赖对象*

Mockito.verifyNoMoreInteractions(idGenerator, userDAO);

}

*/\*\**

*\* 测试: 创建用户-旧*

*\*/*

@Test

public void testCreateUserWithOld() {

*// 模拟依赖方法*

*// 模拟依赖方法: userDAO.getByName*

Long userId = 1L; Mockito.doReturn(userId).when(userDAO).getIdByName(Mockito.anyString());

*// 调用被测方法*

String text = ResourceHelper.getResourceAsString(getClass(), "userCreateVO.json");

UserVO userCreate = JSON.parseObject(text, UserVO.class);

Assert.assertEquals("用户标识不一致", userId, userService.createUser(userCreate));

*// 验证依赖方法*

*// 验证依赖方法: userDAO.getByName*

Mockito.verify(userDAO).getIdByName(userCreate.getName());

*// 验证依赖方法: userDAO.modify*

ArgumentCaptor < UserDO> userModifyCaptor = ArgumentCaptor.forClass(UserDO.class);

Mockito.verify(userDAO).modify(userModifyCaptor.capture());

text = ResourceHelper.getResourceAsString(getClass(), "userModifyDO.json");

Assert.assertEquals("用户修改不一致", text, JSON.toJSONString(userModifyCaptor.getValue()));

*// 验证依赖对象*

Mockito.verifyNoInteractions(idGenerator);

Mockito.verifyNoMoreInteractions(userDAO);

}

*/\*\**

*\* 测试: 创建用户-异常*

*\*/*

@Test

public void testCreateUserWithException() {

*// 注入依赖对象*

Whitebox.setInternalState(userService, "canModify", Boolean.FALSE);

*// 模拟依赖方法*

*// 模拟依赖方法: userDAO.getByName*

Long userId = 1L;

Mockito.doReturn(userId).when(userDAO).getIdByName(Mockito.anyString());

*// 调用被测方法*

String text = ResourceHelper.getResourceAsString(getClass(), "userCreateVO.json");

UserVO userCreate = JSON.parseObject(text, UserVO.class);

UnsupportedOperationException exception = Assert.assertThrows("返回异常不一致",

UnsupportedOperationException.class, () -> userService.createUser(userCreate));

Assert.assertEquals("异常消息不一致", "不支持修改", exception.getMessage());

}

}

### 模拟数据json

**userCreateVO.json**

{"name":"test"}

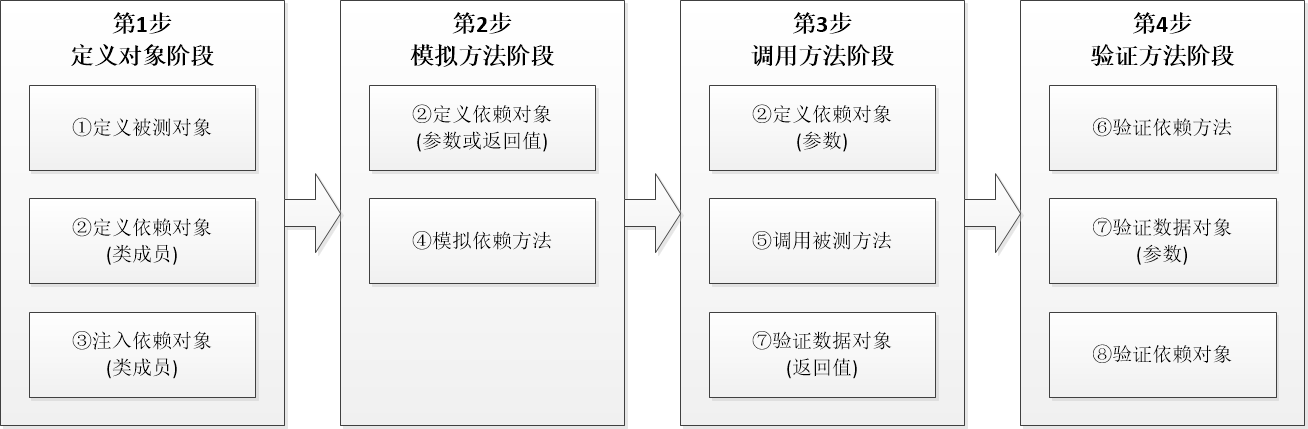
**userCreateDO.json：**

{"id":1,"name":"test"}

**userModifyDO.json：**

{"id":1,"name":"test"}

## 测试用例编写流程



单元测试用例编写流程

### 定义对象阶段

主要包括定义被测对象、模拟依赖对象（类成员）、注入依赖对象（类成员）3大部分。

#### 定义被测对象

在编写单元测试时，首先需要定义被测对象，或直接初始化、或通过Spy包装……其实，就是把被测试服务类进行实例化。

*/\*\* 定义被测对象 \*/*

*/\*\* 用户服务 \*/*

@InjectMocks

private UserService userService;

#### 模拟依赖对象（类成员）

在一个服务类中，我们定义了一些类成员对象——服务（Service）、数据访问对象（DAO）、参数（Value）等。在Spring框架中，这些类成员对象通过@Autowired、@Value等方式注入，它们可能涉及复杂的环境配置、依赖第三方接口服务……但是，在单元测试中，为了解除对这些类成员对象的依赖，我们需要对这些类成员对象进行模拟

*/\*\* 模拟依赖对象 \*/*

*/\*\* 用户DAO \*/*

@Mock

private UserDAO userDAO;

*/\*\* 标识生成器 \*/*

@Mock

private IdGenerator idGenerator;

#### 注入依赖对象（类成员）

当模拟完这些类成员对象后，我们需要把这些类成员对象注入到被测试类的实例中。以便在调用被测试方法时，可能使用这些类成员对象，而不至于抛出空指针异常。

*/\*\* 定义被测对象 \*/*

*/\*\* 用户服务 \*/*

@InjectMocks

private UserService userService;

*/\*\**

*\* 在测试之前*

*\*/*

@Before

public void beforeTest() {

*// 注入依赖对象*

Whitebox.setInternalState(userService, "canModify", Boolean.TRUE);

}

### 模拟方法阶段

主要包括模拟依赖对象（参数或返回值）、模拟依赖方法2大部分。

#### 模拟依赖对象（参数或返回值）

通常，在调用一个方法时，需要先指定方法的参数，然后获取到方法的返回值。所以，在模拟方法之前，需要先模拟该方法的参数和返回值。

Long userId = 1L;

#### 模拟依赖方法

在模拟完依赖的参数和返回值后，就可以利用Mockito和PowerMock的功能，进行依赖方法的模拟。如果依赖对象还有方法调用，还需要模拟这些依赖对象的方法。

*// 模拟依赖方法*

*// 模拟依赖方法: userDAO.getByName*

Mockito.doReturn(null).when(userDAO).getIdByName(Mockito.anyString());

*// 模拟依赖方法: idGenerator.next*

Mockito.doReturn(userId).when(idGenerator).next();

### 调用方法阶段

主要包括模拟依赖对象（参数）、调用被测方法、验证参数对象（返回值）3步

#### 模拟依赖对象（参数）

在调用被测方法之前，需要模拟被测方法的参数。如果这些参数还有方法调用，还需要模拟这些参数的方法。

String text = ResourceHelper.getResourceAsString(getClass(), "userCreateVO.json");

UserVO userCreate = JSON.parseObject(text, UserVO.class);

#### 调用被测方法

在准备好参数对象后，就可以调用被测试方法了。如果被测试方法有返回值，需要定义变量接收返回值；如果被测试方法要抛出异常，需要指定期望的异常。

userService.createUser(userCreate)

#### 验证数据对象（返回值）

在调用被测试方法后，如果被测试方法有返回值，需要验证这个返回值是否符合预期；如果被测试方法要抛出异常，需要验证这个异常是否满足要求。

Assert.assertEquals("用户标识不一致", userId, userService.createUser(userCreate));

### 验证方法阶段

主要包括验证依赖方法、验证数据对象（参数）、验证依赖对象3步。

#### 验证依赖方法

作为一个完整的测试用例，需要对每一个模拟的依赖方法调用进行验证。

*// 验证依赖方法*

*// 验证依赖方法: userDAO.getByName*

Mockito.verify(userDAO).getIdByName(userCreate.getName());

*// 验证依赖方法: idGenerator.next*

Mockito.verify(idGenerator).next();

*// 验证依赖方法: userDAO.create*

ArgumentCaptor < UserDO> userCreateCaptor = ArgumentCaptor.forClass(UserDO.class);

Mockito.verify(userDAO).create(userCreateCaptor.capture());

#### 验证数据对象（参数）

对应一些模拟的依赖方法，有些参数对象是被测试方法内部生成的。为了验证代码逻辑的正确性，就需要对这些参数对象进行验证，看这些参数对象值是否符合预期。

text = ResourceHelper.getResourceAsString(getClass(), "userCreateDO.json");

Assert.assertEquals("用户创建不一致", text, JSON.toJSONString(userCreateCaptor.getValue()));

#### 验证依赖对象

作为一个完整的测试用例，应该保证每一个模拟的依赖方法调用都进行了验证。正好，Mockito提供了一套方法，用于验证模拟对象所有方法调用都得到了验证。

*// 验证依赖对象*

Mockito.verifyNoMoreInteractions(idGenerator, userDAO);

。。。。