Mockito 框架

动机

- 如果被测类不依赖其他类，如 `Money` 类
  - 可以使用 `JUnit` 和 `AAA` 模式对其进行测试
  - 通过**直接输入**和**直接输出**
- 如果被测类依赖了其他类
  - 连同其他类一块测试
  - 只测它自己，不测其他类：把它与其他类进行隔离
- Mockito 框架使用测试替身（`Test Double`）来帮助被测试组件的隔离
  - 版本：3.3.6
  - 网址：https://site.mockito.org
  - 文档：https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html
- `Test Double`: 出于单元测试的目的，用于替代真实组件的对象

安装

```
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-junit-jupiter</artifactId>
  <version>3.1.0</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-api</artifactId>
  <version>5.6.1</version>
  <scope>test</scope>
</dependency>
```

基本使用

- 创建 `Test Double`

```
public interface Car {
  boolean needsFuel();    // 是否需要加油
  double getEngineTemperature();      // 获取发动机温度
  void driveTo(String destination);  // 驶往某地
}
```

- 想象 `SUT` 使用了类型为 `Car` 的 `DOC`，出于测试的目的，需要创建 `Car` 类型的 `Test Double`

```
Car ferrari = Mockito.mock(Car.class);  // 也可以创建类的test double
```

- `Test Double` 默认值

```
assertFalse(ferrari.needsFuel());
assertEquals(0.0, ferrari.getEngineTemperature())
```

- 根据测试需求改变 `Test Double` 的行为

```
// 修改返回值
when(ferrari.needsFuel()).thenReturn(true);  // 让 ferrari.needsFuel() 返回 true
assertTrue(ferrari.needsFuel());

// 抛出异常
when(ferrari.needsFuel()).thenThrow(new RuntimeException());
assertThrows(RuntimeException.class, () -> ferrari.needsFuel());
```

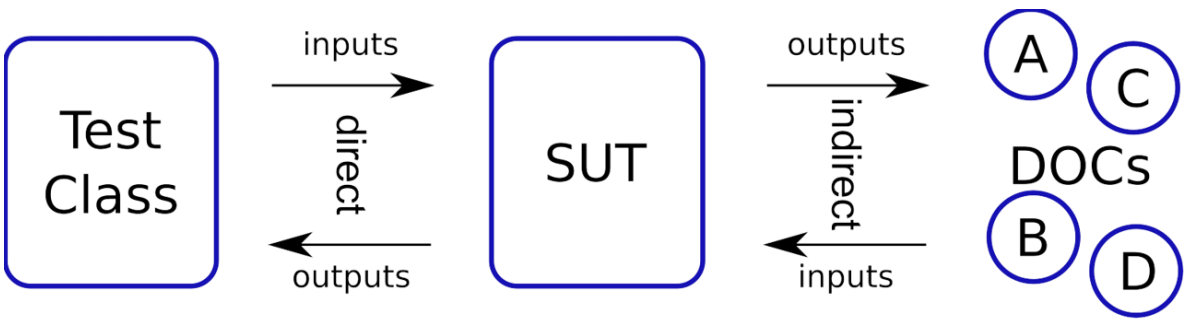
- 验证 `Test Double` 的调用情况

```
ferrari.needFuel();
verify(ferrari).needsFuel();

ferrari.driveTo("家");
verify(ferrari).driveTo("家");

verify(ferrari).getEngineTemperature(); // 失败，因为之前没有调用此方法
verify(ferrari).driveTo("学校"); // 失败，因为参数不正确
```

间接输入和间接输出



```
public class FinancialService {

  .... // definition of fields and other methods omitted

  public BigDecimal calculateBonus(long clientId, BigDecimal payment) {
    Short clientType = clientDAO.getClientType(clientId);
    BigDecimal bonus = calculator.calculateBonus(clientType, payment);
    clientDAO.saveBonusHistory(clientId, bonus);
    return bonus;
  }
}
```

- `SUT`（System Under Test）：被测组件
- `DOC`（Depended On Component）：`SUT` 所依赖的组件
- **直接输入**：`SUT` 方法的参数
  - `calculateBonus` 方法中的 `clientId` 和 `payment` 参数
- **直接输出**：在调用 `SUT` 的方法后，`SUT` 返回的值
  - 由 `cacluateBonus` 方法返回的 `bonus` 值
- **间接输入**：调用 `DOC` 的方法后，返回到 `SUT` 的值，或抛出的异常
  - 由 `calculator` 返回的 `bonus`
  - `clientDAO` 返回的 `clientType` 值
  - 为什么需要关注间接输入?
    - 间接输入会影响 `SUT` 的执行
    - `SUT` 中的许多执行路径都是为了处理间接输入
- **间接输出**：由 `SUT` 传递给 `DOC` 的方法参数
  - 传递给 `clientDAO.saveBonusHistory()` 的 `clientId` 和 `bonus` 参数
  - 传递给 `calculator.calculateBonus()` 的 `clientType` 和 `payment` 参数
  - 为什么需要关心间接输出?
    - 考虑没有返回值的 `DOC`，此时无法获取其返回值（间接输入）
    - 办法：验证 `DOC` 调用情况

测试替身分类

- 哑对象（Dummy Object）：帮助测试项目编译通过，不在具体测试里面起任何作用
- 测试桩（Test Stub）：用于替换 `SUT` 依赖的真实组件，向 `SUT` 提供间接输入
- 测试间谍（Test Spy）：为测试提供一种检查间接输出的方式，从而验证 `SUT` 的间接输出
- 仿制对象（Mock Object）：与 `Test Spy` 基本相同，用于验证间接输出
- 仿冒对象（Fake Object）：提供与 `DOC` 相同的功能，然后替换真实 `DOC` 功能的对象，以方便测试
  - 内存数据库 VS 真实数据库

案例说明

```
public class Messenger {
  private TemplateEngine templateEngine;
  private MailServer mailServer;

  public Messenger(MailServer mailServer, TemplateEngine templateEngine) {
    this.mailServer = mailServer;
```

```
    this.templateEngine = templateEngine;
  }

  public void sendMessage(Client client, Template template) { // 返回 void
    String msgContent = templateEngine.prepareMessage(template, client);
    mailServer.send(client.getEmail(), msgContent);
  }
}
```

- `Messenger`依赖四个类：`TemplateEngine`，`MailServer`，`Client`，`Template`
- 测试 `sendMessage` 方法的问题
  - 没有返回值，无法观察其直接输出
  - 不能获取 `client` 或 `template` 的状态进行验证
- 使用 `Dummy Object`

```
Template template = mock(Template.class);
messenger.sendMessage(client, template);  // 仅仅是为了填充参数，有时给个 null 值也够用
```

- 使用 `Test Stub`
  - 观察 `templateEngine` 对象，它的 `prepareMessage` 为 `SUT` 提供了间接输入
  - `dummy object` 只负责创建对象，但忽略了 `prepareMessage` 的调用
  - 使用 `Test Stub` 可以控制 `prepareMessage` 的返回值（间接输入）

```
TemplateEngine engine = mock(TemplateEngine.class);
Messenger messenger = new Messenger(mailServer, engine);
when(engine.prepareMessage(template, client)).thenReturn("Hello Mockito!");
messenger.sendMessage(client, template);
```

- 使用 `Test Spy`
  - 观察 `mailServer.send()` 方法，它是 `Messenger` 的间接输出
  - 使用 `Mockito.verify`

```
MailServer mailServer = mock(MailServer.class);
Messenger messenger = new Messenger(mailServer, templateEngine);
messenger.sendMessage(client, template);
verify(mailServer).send("some@email.com", "Hello Mockito!");
```

- 使用 `Mock Object`
  - 与 `Test Spy` 一样，都是为了验证间接输出
  - 与 `Test Spy` 区别很小：只是在测试代码的语法上有出入

- 综合

```
@Test
void test_send_email() {
  Template template = mock(Template.class);
  Client client = mock(Client.class);
  MailServer mailServer = mock(MailServer.class);
  TemplateEngine templateEngine = mock(TemplateEngine.class);
```

```
    Messenger messenger = new Messenger(mailServer, templateEngine);

    when(client.getEmail()).thenReturn("some@email.com");
    when(templateEngine.prepareMessage(template, client)).thenReturn("Hello
  Mockito");

    messenger.sendMessage(client, template);

    verify(mailServer).send("some@email.com", "Hello Mockito");
  }
```

- 思考：需要对所有的依赖都进行替换吗?
  - 对于数据库连接，文件，必须使用 `Test Double` ，否则将不再是单元测试
  - 如果 `DOC` 很简单，如 `javabean` ，或只包含简单行为，无需替换
  - 不要创建值对象的 `Test Double`
  - 不要创建不属于自己的类型，如第三方库
  - 只对业务逻辑进行单元测试

其他用法

```
// 验证 Test Double 被调用的次数
verify(mockedList, times(1)).add("once");
verify(mockedList, atMostOnce()).add("once");
verify(mockedList, atLeastOnce()).add("three times");
verify(mockedList, atLeast(2)).add("three times");
verify(mockedList, atMost(5)).add("three times");
verify(mockOne, never()).add("two");  // 一次都不发生

// 参数匹配器
when(mockedList.get(anyInt())).thenReturn("element");
when(mockedList.contains(argThat(isValid()))).thenReturn(true);
System.out.println(mockedList.get(999));
verify(mockedList).get(anyInt());
verify(mockedList).add(argThat(someString -> someString.length() > 5));

// 使用@Mock
@Mock Car ferrari;  // 等价于 Car ferrari = mock(Car.class);
```

作业：为 `UserService` 类编写单元测试

```
public class UserService {
  private UserDAO userDAO;
  private SecurityService securityService;

  public UserService(UserDAO dao, SecurityService security) {
    this.userDAO = dao; this.securityService = security;
  }

  public void assignPassword(User user) throws Exception {
    String passwordMd5 = securityService.md5(user.getPassword());
    user.setPassword(passwordMd5);
    userDAO.updateUser(user);
  }
}
```

- `DOC` 组件可根据需要自己创建（接口就够用了）
- 要求使用 `Test Stub` 和 `Test Spy` 分别验证间接输入和间接输出