单元测试

- @SprintBootTest
- @MockBean
- @Import
- @Runwith(SpringRunner.class)
- @AutoConfigureMockMvc
- @Transactional
- @TestPropertySource
- @Mock
  - @Mock, @Mockbean, @InjectMocks 的区别
- @Spy
  - @Mock 和@Spy
- @InjectMocks
- @BeforeClass
- @before
- @afterclass
- @After
- @Test
- Mockito.mock()
- Mockito.mockStatic()
- ReflectionTestUtils.setField()

应谨慎使用模拟，并且仅在必要时才使用。过度使用模拟会导致测试过于复杂并降低测试效率。在测试中使用模拟和真实依赖关系之间取得平衡以确保有意义和可靠的测试结果很重要。

模拟不应该取代实际实现的真实测试。重要的是要在具有模拟的单元测试和具有真实依赖关系的集成测试之间取得平衡，以确保全面的测试覆盖率。

@SprintBootTest

用来标记一个测试类，它告诉 Spring Boot 启动一个完整的应用程序上下文，而不仅仅是一个单一的测试类或测试方法。

这个完整的应用程序上下文将包含所有的 Spring Bean、配置和依赖项，这样我们就可以像在实际的应用程序中一样运行我们的测试用例。

由于@SpringBootTest 注解会启动一个完整的应用程序上下文，因此比较慢。因此，建议将@SpringBootTest 注解仅用于集成测试，而不是单元测试。

由于@SpringBootTest 注解启动的应用程序上下文可能包含大量的 Spring Bean 和依赖项，因此它可能会与其他测试用例产生意外的干扰。

因此，建议在测试类中使用@DirtiesContext 注解，以便在每个测试方法之间重新加载应用程序上下文。

Spring Boot Test 介绍_@springboottest_卡布奇诺-海晨的博客-CSDN 博客

Spring Boot Test 支持的测试种类，大致可以分为如下三类：

- 单元测试：一般面向方法，编写一般业务代码时，测试成本较大。涉及到的注解有@Test。
- 切片测试：一般面向难于测试的边界功能，介于单元测试和功能测试之间。涉及到的注解有@RunWith @WebMvcTest 等。
- 功能测试：一般面向某个完整的业务功能，同时也可以使用切面测试中的 mock 能力，推荐使用。涉及到的注解有@RunWith @SpringBootTest 等。

功能测试过程中的几个关键要素及支撑方式如下：

- 测试运行环境：通过@RunWith 和 @SpringBootTest 启动 spring 容器。
- mock 能力：Mockito 提供了强大 mock 功能。
- 断言能力：AssertJ、Hamcrest、JsonPath 提供了强大的断言能力。

@SpringBootTest 时并没有像@ContextConfiguration 一样显示指定 locations 或 classes 属性，原因在于@SpringBootTest 注解会自动检索程序的配置文件。

检索顺序是从当前包开始，逐级向上查找被@SpringBootApplication 或 @SpringBootConfiguration 注解的类。

properties="test.prop=hello"

args = "--test.prop=test code222"

优先级：args > properties

classes = MyConfiguration.class 指定要加载的配置类

webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT

- MOCK：根据当前设置确认是否启动 web 环境，例如使用了 Servlet 的 API 就启动 web 环境，属于适配性的配置

- DEFINED_PORT：使用自定义的端口作为 web 服务器端口
- RANDOM_PORT：使用随机端口作为 web 服务器端口
- NONE：不启动 web 环境

@MockBean

创建一个虚拟对象，代替那些不易构造或者不易获取的对象。

```
@MockBean
private UserService mockUserService;


@Test
public void testFindByUserNameByMockService() {
    User user = new User();
    user.setId(new IDGenerator(0, 0).nextId());
    user.setUserName("tt");
    when(mockUserService.saveUser("tt")).thenReturn(user);
    User user2 = mockUserService.saveUser("tt");
    Assert.assertTrue("数据一致", "tt".equals(user2.getUserName()));
}
```

@Import

```
@Import(MsgConfig.class)
```

在启动测试环境时，使用@Import 注解导入测试环境专用的配置类。

@Runwith(SpringRunner.class)

表明 Test 测试类需要使用注入的类，比如@Autowired 注入的类，有了 @RunWith(SpringRunner.class)，这些类才能实例化到 Spring 容器中，自动注入才能生效。

@AutoConfigureMockMvc

@AutoConfigureMockMvc 开启虚拟 MVC 调用。

```
@RunWith(SpringRunner.class)
@SpringBootTest
```

```
@AutoConfigureMockMvc
public class TxwBootStarterApplicationTests {
   @Autowired
   MockMvc mockMvc;

   @Test
   public void whenIndexSuccess() throws Exception {
      mockMvc.perform(MockMvcRequestBuilders.get("/txw-boot-starter/hello")
                  .contentType(MediaType.APPLICATION_JSON_UTF8))
            .andExpect(MockMvcResultMatchers.status().isOk());
   }

   // 状态码匹配
   @Test
   public void whenIndexSuccess2() throws Exception {
      // 创建虚拟请求
      MockHttpServletRequestBuilder mockHttpServletRequestBuilder =
MockMvcRequestBuilders.get("/txw-boot-
starter/hello").contentType(MediaType.APPLICATION_JSON_UTF8);
      // 执行请求，获取调用结果
      ResultActions resultActions =
mockMvc.perform(mockHttpServletRequestBuilder);
      // 设置预期值，准备与调用结果比较
      StatusResultMatchers statusResultMatchers =
MockMvcResultMatchers.status();
      // 预期值为 ok，返回码 200
      ResultMatcher resultMatcher = statusResultMatchers.isOk();
      // 把调用结果与预期值匹配，若匹配不上则测试结果失败
      resultActions.andExpect(resultMatcher);
   }

   // 响应体匹配（非 JSON 格式）
   @Test
   public void whenIndexSuccess3() throws Exception {
      // 创建虚拟请求
      MockHttpServletRequestBuilder mockHttpServletRequestBuilder =
MockMvcRequestBuilders.get("/txw-boot-
starter/hello").contentType(MediaType.APPLICATION_JSON_UTF8);
      // 执行请求，获取调用结果
      ResultActions resultActions =
mockMvc.perform(mockHttpServletRequestBuilder);
```

```
        // 设置预期值，准备与调用结果比较
        ContentResultMatchers contentResultMatchers =
MockMvcResultMatchers.content();
        // 定义预期值
        ResultMatcher resultMatcher = contentResultMatchers.string("test");
        // 把调用结果与预期值匹配，若匹配不上则测试结果失败
        resultActions.andExpect(resultMatcher);
    }

    // 响应体匹配（JSON 格式）
    @Test
    public void whenIndexSuccess4() throws Exception {
        // 创建虚拟请求
        MockHttpServletRequestBuilder mockHttpServletRequestBuilder =
MockMvcRequestBuilders.get("/txw-boot-
starter/hello").contentType(MediaType.APPLICATION_JSON_UTF8);
        // 执行请求，获取调用结果
        ResultActions resultActions =
mockMvc.perform(mockHttpServletRequestBuilder);
        // 设置预期值，准备与调用结果比较
        ContentResultMatchers contentResultMatchers =
MockMvcResultMatchers.content();
        // 定义预期值
        ResultMatcher resultMatcher =
contentResultMatchers.json("{\"code\":200,\"msg\":\"操作失败
\",\"data\":\"hahahah\"}");
        // 把调用结果与预期值匹配，若匹配不上则测试结果失败
        resultActions.andExpect(resultMatcher);
    }

    // 响应头匹配
    @Test
    public void whenIndexSuccess5() throws Exception {
        // 创建虚拟请求
        MockHttpServletRequestBuilder mockHttpServletRequestBuilder =
MockMvcRequestBuilders.get("/txw-boot-
starter/hello").contentType(MediaType.APPLICATION_JSON_UTF8);
        // 执行请求，获取调用结果
        ResultActions resultActions =
mockMvc.perform(mockHttpServletRequestBuilder);
        // 设置预期值，准备与调用结果比较
```

```
        HeaderResultMatchers headerResultMatchers =
MockMvcResultMatchers.header();
        // 定义预期值
        ResultMatcher resultMatcher = headerResultMatchers.string("Content-
Type", "application/json");
        // 把调用结果与预期值匹配，若匹配不上则测试结果失败
        resultActions.andExpect(resultMatcher);
    }
}
```

```
@Autowired

MockMvc  mvc;
```

定义发起虚拟调用的对象 MockMVC，通过自动装配的形式初始化对象。

主机号和端口不需要写，只写访问路径。因为前面的服务器 IP 地址和端口使用的是当前虚拟的 web 环境。

MockMvcRequestBuilders 可以发送多种请求（get，post，delete，put…）。

@Transactional

问题：如果测试时测试用例产生了事务提交，就会在测试过程中对数据库数据产生影响，进而产生垃圾数据。

如果注解@Transactional 出现的位置存在注解@SpringBootTest，SpringBoot 就会认为这是一个测试程序，无需提交事务，所以也就可以避免事务的提交。

@Transactional 注解，默认加上@Rollback(true)，表示不提交事务。

如果想提交事务，添加@RollBack(false)，即可正常提交事务。

@TestPropertySource

可以用来覆盖掉来自于系统环境变量、Java 系统属性、@PropertySource 的属性。

@TestPropertySource(properties=...)优先级高于
@TestPropertySource(locations=...)。

利用它我们可以很方便的在测试代码里微调、模拟配置（比如修改操作系统目录分隔符、数据源等）。

@Mock

创建一个 Mock 对象。

对函数的调用均使用 mock，不会调用真实方法，使用：

Mockito.when(testService.getById(Mockito.any())).thenReturn(testPo);

@Mock, @Mockbean, @InjectMocks 的区别

@Mock 可以生产一个空的类，这个类的方法体都是空的，方法的返回值（如果有的话）都是 null。用于模拟不属于 Spring 上下文的对象。从 org.mockito 包中导入。

对于 @Mock，模拟对象需要使用 @InjectMocks 注释或通过在测试设置中调用 MockitoAnnotations.initMocks(this) 手动注入到测试实例中。

@MockBean 可以生产一个空的类，并且用这个类替代 spring 容器中同类型的类。被 @autowired 标注的字段，只能被注入被@mockbean 标注的类（@mock 标注的不行）。在测试上下文设置期间由 Spring Boot 测试框架自动初始化。

从 org.springframework.boot.test.mock.mockito 包中导入的。使用@MockBean 时，Spring Boot 会自动将 Spring 上下文中的实际 bean 替换为模拟 bean，从而允许进行适当的依赖注入。

@InjectMocks 就是产生一个空的类，这个类内部的字段会被这个测试类中被@Mock 注解的类填充，而不会被@MockBean 注解的类填充。

要想把需要打桩的类注入到我们要测试的类里面,有下面两个思路

- @ExtendWith(MockitoExtension.class) + @InjectMocks +@Mock
- @SpringBootTest +@Autowired+@MockBean

[@Mock @MockBean @InjectMocks 之间的关系_mockbean 和 mock_程序员小董的博客-CSDN 博客]

@Spy

spy 对象，对函数的调用是真实调用，如果不想真实调用可以使用：

```
Mockito.doReturn(true).when(testService).save(Mockito.any());
```

@Mock 和@Spy

@Mock 是将目标对象整个模拟，所有方法默认都返回 null，并且原方法中的代码逻辑不会执行，被 Mock 出来的对象，想用哪个方法，哪个方法就需要打桩，否则返回 null。

@Spy 可实现对目标对象部分方法、特定入参条件时的打桩，没有被打桩的方法，将会真实调用。

[关于 Mock、Spy、@MockBean、@SpyBean 的笔记 - 简书 (jianshu.com)]

@InjectMocks

创建一个实例，其余用@Mock（或@Spy）注解创建的 mock 对象将被自动注入到该实例中。

在单元测试中，没有启动 Spring 框架，通过@InjectMocks 完成依赖注入。

```
@RunWith(SpringRunner.class)
public class UserServiceTest {
    @Mock
    UserDao userDao;

    @InjectMocks
    UserService userService;

    @Test
    public void testfindUserByUsername() {
        System.out.println(userService.findUserByUsername("").getClass());
    }
}
```

注意：InjectMocks 字段是无法注入其他 InjectMocks 字段。

@BeforeClass

被@BeforeClass 注解的方法会是：只被执行一次

运行 junit 测试类时第一个被执行的方法

这样的方法被用作执行计算代价很大的任务，如打开数据库连接。被@BeforeClass 注解的方法应该是静态的（即 static 类型的）。

@before

被@Before 注解的方法应是：junit 测试类中的任意一个测试方法执行前都会执行此方法

该类型的方法可以被用来为测试方法初始化所需的资源。

@afterclass

被@AfterClass 注解的方法应是：只被执行一次

运行 junit 测试类是最后一个被执行的方法

该类型的方法被用作执行类似关闭数据库连接的任务。被@AfterClass 注解的方法应该是静态的（即 static 类型的）。

@After

被@After 注解的方法应是：

junit 测试类中的任意一个测试方法执行后 都会执行此方法，即使被@Test 或 @Before 修饰的测试方法抛出异常

该类型的方法被用来关闭由@Before 注解修饰的测试方法打开的资源。

@Test

被@Test 注解的测试方法包含了真正的测试代码，并且会被 Junit 应用为要测试的方法。@Test 注解有两个可选的参数：

expected 表示此测试方法执行后应该抛出的异常，（值是异常名）

timeout 检测测试方法的执行时间

Mockito.mock()

Mockito 框架提供的方法，Mock 对象在单元测试中用于将被测试代码与其依赖项隔离开来，从而使测试可以隔离执行，而不依赖于真正的外部组件。

Mockito.mock()方法用于创建给定类或接口的模拟对象。它将类或接口作为参数并返回该类或接口的实例，其所有方法都以默认行为存在，例如返回 null、0 或空集合。

然后，开发人员可以使用 Mockito 的 API 配置模拟对象的行为，以定义测试用例期间方法调用的特定期望和响应。

Mockito.mockStatic()

mock 静态类和静态方法。

```
import org.junit.jupiter.api.Test;
import org.mockito.MockedStatic;
import org.mockito.Mockito;

import static org.assertj.core.api.Assertions.assertThat;


public class MockedStaticDemo {
    static class Buddy {
        static String name() {
            return "John";
        }
    }

    @Test
    void lookMomICanMockStaticMethods() {
        assertThat(Buddy.name()).isEqualTo("John");

        try (MockedStatic<Buddy> theMock = Mockito.mockStatic(Buddy.class)) {
            theMock.when(Buddy::name).thenReturn("Rafael");
            assertThat(Buddy.name()).isEqualTo("Rafael");
        }

        assertThat(Buddy.name()).isEqualTo("John");
    }
```

```
}
```

ReflectionTestUtils.setField()