yxysuanfa

【Java实战㉖】深入Java单元测试：JUnit 5实战指南 - 详解

目录

一、单元测试概述

1.1 单元测试概念

单元测试是软件开发过程中至关重要的一环，它专注于对软件中最小的可测试单元进行验证。在 Java 中，这些最小单元通常指的是方法或函数。通过精心设计针对这些单元的测试用例，我们能够深入检查每个单元的功能是否符合预期。

比如，在一个简单的数学计算类中，有一个用于两数相加的方法add(int a, int b)。单元测试就会针对这个方法编写多个测试用例，包括正常的相加情况，如add(2, 3)是否返回5；也会考虑边界情况，如add(0, 0)的结果，以及特殊情况，如传入最大整数相加是否会溢出等。通过这样细致的测试，确保add方法在各种情况下都能正确工作，为整个软件系统的稳定性奠定基础。

1.2 单元测试优势

- **提前发现 bug**：在开发过程中，越早发现问题，修复成本就越低。单元测试由开发人员在代码编写阶段执行，能够及时捕捉到代码中的逻辑错误、边界条件处理不当等问题，避免这些问题在后续集成测试、系统测试甚至生产环境中才被发现，从而节省大量的时间和精力。
- **便于重构**：随着项目的演进，代码需要不断重构以提高可维护性和扩展性。有了完善的单元测试，开发人员在重构代码时可以更加放心，因为只要单元测试全部通过，就可以基本保证重构后的代码功能没有改变。例如，当对一个复杂算法进行优化时，运行单元测试可以快速验证优化后的代码是否依然正确。
- **提升代码质量**：编写单元测试的过程促使开发人员更加深入地思考代码的功能和逻辑，从而编写出更健壮、更清晰的代码。同时，单元测试也可以作为一种文档，记录代码的预期行为和使用方式，方便后续维护和理解。

1.3 JUnit 5 框架组成

- **JUnit Jupiter**：它是 JUnit 5 的核心模块，提供了全新的编程模型和扩展模型。在编程模型方面，它引入了一系列新的注解，如@Test、@BeforeEach、@AfterEach等，让编写测试代码更加简洁和灵

昵称：yxysuanfa
园龄：8年6个月
粉丝：25
关注：0
+加关注

< 2025年10月 >

日	一	二	三	四	五	六
28	29	30	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	1
2	3	4	5	6	7	8

搜索

找找看

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签

最新随笔

1.OpenCV基础操作与图像处理 - 指南
2.asmlaw80a.dll asmlaw130i.dll asmlaw120a.dll asmkern80a.dll asmkern130i.dll ASMKERN120A.dll asm - 实践
3.应用虚幻引擎时间轴制作一个弹跳小球
4.Zookeeper 技术详细介绍 - 指南
5.完整教程：Logit论文阅读
6.JavaScript学习笔记(十四)：ES6 Set函数详解 - 指南
7.Nx项目中利用Vitest对原生JS组件进行单元测试
8.MySQL 在金融高效的系统中的应用：强一致性与高可用架构实战
9.【Linux&&vs code】Xshell远程配备到VS Code环境配置指南
10.详细介绍：verilog中的FIR滤波器和自控中一阶低通滤波器的区别和共性

我的标签

人工智能(64)
开发语言(39)
Java(33)
运维(31)
前端(28)
数据库(27)
python(23)
linux(22)
算法(21)
c++(20)
更多

活。在扩展模型上，允许开发者创建自定义的测试扩展，以满足特定的测试需求。

- **JUnit Vintage**：主要用于兼容旧版本的 JUnit 测试，即 JUnit 3 和 JUnit 4 的测试用例。在项目从旧版本的 JUnit 迁移到 JUnit 5 时，这个模块可以确保旧的测试用例依然能够正常运行，为项目的平稳过渡提供保障。
- **JUnit Platform**：作为整个 JUnit 5 测试框架的基础，它为在 JVM 上启动测试框架提供了必要的支持，定义了 TestEngine API，用于开发在平台上运行的测试引擎，使得 JUnit 5 能够与各种 IDE 和构建工具集成，方便开发者使用。

1.4 JUnit 5 环境搭建

以 Maven 项目为例，在pom.xml文件中添加 JUnit 5 依赖：

```
<dependency>
  <groupId>org.junit.jupiter<
    /groupId>
    <artifactId>junit-jupiter-api<
      /artifactId>
      <version>5.10.0<
        /version>
        <scope>test<
          /scope>
          <
          /dependency>
          <dependency>
            <groupId>org.junit.jupiter<
              /groupId>
              <artifactId>junit-jupiter-engine<
                /artifactId>
                <version>5.10.0<
                  /version>
                  <scope>test<
                    /scope>
                    <
                    /dependency>
```

如果需要兼容 JUnit 4 的测试，还需添加：

JAVA 复制 全屏

```
<dependency>
  <groupId>org.junit.vintage<
    /groupId>
    <artifactId>junit-vintage-engine<
      /artifactId>
      <version>5.10.0<
        /version>
        <scope>test<
          /scope>
          <
          /dependency>
```

在 IDE 中的配置步骤如下：

- **IntelliJ IDEA**：新建项目时，若选择 Maven 项目，在创建过程中可直接在pom.xml添加上述依赖，IDEA 会自动下载相关库。若项目已创建，打开pom.xml添加依赖后，点击右上角的Maven图标，选择Reload All Maven Projects即可。创建测试类时，在src/test/java目录下新建类，无需额外配置即可使用 JUnit 5 进行测试。
- **Eclipse**：新建 Java 项目后，右键点击项目，选择Properties，在弹出的窗口中选择Java Build Path，切换到Libraries标签，点击Add Library，选择JUnit，然后选择JUnit 5，点击Finish。之后在src/test/java目录下创建测试类即可使用 JUnit 5。

二、JUnit 5 核心功能实战

2.1 测试类与测试方法

在 JUnit 5 中，使用@Test注解来标记一个方法为测试方法。例如：

```
import org.junit.jupiter.api.Test;
public class CalculatorTest
```

```
{
@Test
public void testAdd() {
// 测试逻辑
}
}
```

@DisplayName注解则可以为测试类或测试方法自定义显示名称，使测试报告更加易读。例如：

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
@DisplayName("计算器测试类")
public class CalculatorTest
{
@Test
@DisplayName("加法测试方法")
public void testAdd() {
// 测试逻辑
}
}
```

2.2 测试生命周期

- **@BeforeEach**：标注在方法上，在每一个测试方法（使用@Test、@RepeatedTest、@ParameterizedTest或@TestFactory注解的方法）执行之前都会执行该方法。常用于初始化测试所需的资源，如创建对象实例、建立数据库连接等。例如：

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
public class UserServiceTest
{
private UserService userService;
@BeforeEach
public void setUp() {
userService = new UserService();
}
@Test
public void testAddUser() {
// 测试添加用户的逻辑
}
}
```

- **@AfterEach**：标注在方法上，在每一个测试方法执行之后都会执行该方法。主要用于清理测试过程中产生的资源，如关闭数据库连接、删除临时文件等。例如：

```
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
public class UserServiceTest
{
private UserService userService;
@BeforeEach
public void setUp() {
userService = new UserService();
}
@Test
public void testAddUser() {
// 测试添加用户的逻辑
}
@AfterEach
public void tearDown() {
// 清理资源的逻辑
}
}
```

- **@BeforeAll**：标注在静态方法上，在当前测试类中所有的测试方法执行之前执行一次。适用于初始化一些在整个测试类中都需要共享的资源，如加载配置文件、初始化数据库连接池等。例如：

```
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
public class DatabaseTest
{
private static DatabaseConnection connection;
@BeforeAll
public static void setUp() {
connection = DatabaseConnection.getConnection();
}
@Test
public void testQuery() {
// 测试数据库查询的逻辑
}
}
```

- **@AfterAll**：标注在静态方法上，在当前测试类中所有的测试方法执行完毕之后执行一次。用于释放那些在@BeforeAll中初始化的共享资源，如关闭数据库连接池、释放文件锁等。例如：

```
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
public class DatabaseTest
{
private static DatabaseConnection connection;
@BeforeAll
public static void setUp() {
connection = DatabaseConnection.getConnection();
}
@Test
public void testQuery() {
// 测试数据库查询的逻辑
}
@AfterAll
public static void tearDown() {
connection.close();
}
}
```

2.3 断言方法

断言方法是 JUnit 5 中用于验证测试结果是否符合预期的关键工具。常用的断言方法有：

- **assertEquals**：用于验证两个值是否相等。例如：

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;
public class MathUtilsTest
{
@Test
public void testAdd() {
int result = MathUtils.add(2, 3);
assertEquals(5, result);
}
}
```

- **assertTrue**：用于验证某个条件是否为真。例如：

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertTrue;
public class StringUtilsTest
{
@Test
```

```
public void testIsEmpty() {
assertTrue(StringUtils.isEmpty(""));
}
}
```

- **assertNotNull**：用于验证某个对象是否不为空。例如：

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertNotNull;
public class UserServiceTest
{
@Test
public void testGetUserById() {
User user = userService.getUserById(1);
assertNotNull(user);
}
}
```

2.4 异常测试

在测试过程中，常常需要验证某些代码在特定情况下是否会抛出预期的异常。JUnit 5 提供了两种主要的方式来进行异常测试：

- **@Test (expected = 异常类.class)**：在 JUnit 4 中就已存在，在 JUnit 5 中仍然可用。通过在@Test注解中使用expected属性指定预期抛出的异常类型。例如：

```
import org.junit.jupiter.api.Test;
public class MathUtilsTest
{
@Test(expected = ArithmeticException.class)
public void testDivideByZero() {
MathUtils.divide(10, 0);
}
}
```

- **assertThrows**：JUnit 5 中新增的方式，通过assertThrows方法来验证代码块是否抛出预期的异常，并可以进一步对异常的属性进行验证。例如：

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertThrows;
public class MathUtilsTest
{
@Test
public void testDivideByZero() {
ArithmeticException exception = assertThrows(
ArithmeticException.class,
() ->
MathUtils.divide(10, 0)
);
assertEquals("/ by zero", exception.getMessage());
}
}
```

这种方式不仅能验证异常是否抛出，还能获取异常对象，从而对异常的详细信息（如错误消息、堆栈跟踪等）进行断言验证。

三、单元测试进阶实战

3.1 参数化测试

在 JUnit 5 中，参数化测试是一项非常实用的功能，它允许我们使用不同的参数多次运行同一个测试方法，从而覆盖更多的测试场景。通过@ParameterizedTest注解结合@ValueSource、@MethodSource等注解来实现。

比如，我们有一个用于判断数字是否为偶数的方法isEven(int num)，可以编写如下参数化测试：

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;
import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.Assertions.assertTrue;
public class MathUtilsTest
{
@ParameterizedTest
@ValueSource(ints = {
2, 4, 6, 8
})
public void testIsEven(int num) {
assertTrue(MathUtils.isEven(num));
}
@ParameterizedTest
@ValueSource(ints = {
1, 3, 5, 7
})
public void testIsNotEven(int num) {
assertFalse(MathUtils.isEven(num));
}
}
```

在上述代码中，@ParameterizedTest注解表明这是一个参数化测试方法。@ValueSource(ints = {2, 4, 6, 8})提供了一组测试数据，testIsEven方法会针对这组数据中的每一个值执行一次，验证isEven方法在这些偶数输入下的正确性。同理，testIsNotEven方法针对奇数数据进行测试。

如果需要更复杂的参数设置，可以使用@MethodSource注解。例如：

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.MethodSource;
import java.util.stream.Stream;
public class MathUtilsTest
{
static Stream<
Integer> evenNumbersProvider() {
return Stream.of(2, 4, 6, 8);
}
@ParameterizedTest
@MethodSource("evenNumbersProvider")
public void testIsEven(int num) {
assertTrue(MathUtils.isEven(num));
}
}
```

这里通过evenNumbersProvider方法返回一个包含偶数的Stream，@MethodSource("evenNumbersProvider")指定该方法作为参数数据源，testIsEven方法会根据这个数据源中的数据依次执行测试。

3.2 测试套件

在实际项目中，通常会有多个测试类。使用测试套件可以将多个相关的测试类组织在一起，方便批量执行。在JUnit 5 中，使用@Suite注解来创建测试套件。

假设我们有两个测试类CalculatorTest和MathUtilsTest，可以创建一个测试套件类AllTestsSuite：

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.runner.RunWith;
import org.junit.platform.suite.api.SelectClasses;
import org.junit.platform.suite.api.Suite;
@RunWith(JUnitPlatform.class)
@Suite
@SelectClasses({
CalculatorTest.class,
MathUtilsTest.class
})
public class AllTestsSuite
{
```

```
// 测试套件类不需要任何方法，仅用于组织测试类
}
```

在上述代码中，@RunWith(JUnitPlatform.class)指定使用 JUnit Platform 运行测试，@Suite注解表明这是一个测试套件，@SelectClasses注解指定要包含在测试套件中的测试类。运行AllTestsSuite时，JUnit 会依次执行CalculatorTest和MathUtilsTest中的所有测试方法，大大提高了测试效率，尤其适用于集成测试或回归测试场景。

3.3 Mockito 框架

在单元测试中，被测对象往往依赖其他对象。这些依赖对象可能是数据库连接、网络服务调用等，直接使用真实的依赖对象进行测试会带来很多问题，比如测试环境依赖、测试速度慢等。Mockito 框架就是为了解决这些问题而诞生的，它可以模拟依赖对象的行为，实现隔离测试。

以一个用户服务类UserService依赖用户仓库类UserRepository为例：

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.when;
@ExtendWith(MockitoExtension.class)
public class UserServiceTest
{
@Mock
private UserRepository userRepository;
@Test
public void testGetUserById() {
// 模拟userRepository.findById(1L)的返回值
User mockUser = new User(1L, "张三", "zhangsan@example.com");
when(userRepository.findById(1L)).thenReturn(mockUser);
UserService userService = new UserService(userRepository);
User user = userService.getUserById(1L);
assertEquals(mockUser, user);
}
}
```

在上述代码中，首先通过@Mock注解创建了UserRepository的模拟对象userRepository。在testGetUserById方法中，使用when(userRepository.findById(1L)).thenReturn(mockUser)模拟了userRepository.findById(1L)方法的行为，使其返回一个预设的模拟用户对象mockUser。然后创建UserService对象并调用getUserById方法，最后断言返回的用户对象与模拟用户对象一致。这样就实现了对UserService的隔离测试，不受UserRepository实际实现的影响，提高了测试的独立性和可重复性。

3.4 单元测试实战案例

下面以一个UserService层的方法测试为例，综合展示 JUnit 5 的各种功能和技术的使用。假设UserService有一个注册用户的方法registerUser(User user)，该方法依赖UserRepository来保存用户信息，并且需要对用户输入进行一些验证。

首先，创建UserService和UserRepository接口及实现类：

```
// UserRepository接口
public interface UserRepository {
User save(User user);
}
// UserRepository实现类
public class UserRepositoryImpl
implements UserRepository {
@Override
public User save(User user) {
// 实际保存用户到数据库的逻辑，这里简化为直接返回用户对象
return user;
}
}
// UserService接口
public interface UserService {
boolean registerUser(User user);
}
```

```
// UserService实现类
public class UserServiceImpl
implements UserService {
private final UserRepository userRepository;
public UserServiceImpl(UserRepository userRepository) {
this.userRepository = userRepository;
}
@Override
public boolean registerUser(User user) {
if (user == null || user.getUsername() == null || user.getPassword() == null) {
return false;
}
// 简单的用户名长度验证
if (user.getUsername().length() <
3) {
return false;
}
User savedUser = userRepository.save(user);
return savedUser != null;
}
}
```

然后，编写UserService的单元测试类：

```
import org.junit.jupiter.api.*;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.when;
@DisplayName("用户服务测试")
@ExtendWith(MockitoExtension.class)
public class UserServiceTest
{
@Mock
private UserRepository userRepository;
private UserService userService;
@BeforeEach
public void setUp() {
userService = new UserServiceImpl(userRepository);
}
@Test
@DisplayName("正常注册用户测试")
public void testRegisterUserSuccess() {
User user = new User("testUser", "testPassword", "test@example.com");
when(userRepository.save(user)).thenReturn(user);
boolean result = userService.registerUser(user);
assertTrue(result);
}
@Test
@DisplayName("注册用户为空测试")
public void testRegisterUserWithNullUser() {
boolean result = userService.registerUser(null);
assertFalse(result);
}
@Test
@DisplayName("注册用户用户名过短测试")
public void testRegisterUserWithShortUsername() {
User user = new User("ab", "testPassword", "test@example.com");
boolean result = userService.registerUser(user);
assertFalse(result);
}
@Test
@DisplayName("注册用户保存失败测试")
```

```
public void testRegisterUserSaveFailure() {
User user = new User("testUser", "testPassword", "test@example.com");
when(userRepository.save(user)).thenReturn(null);
boolean result = userService.registerUser(user);
assertFalse(result);
}
}
```

在这个测试类中：

- 使用@DisplayName为测试类添加了有意义的显示名称。
- 通过@Mock创建了UserRepository的模拟对象，并在@BeforeEach方法中创建UserService对象，将模拟的UserRepository注入其中。
- 编写了四个测试方法，分别测试正常注册用户、注册用户为空、注册用户用户名过短以及注册用户保存失败的情况。每个测试方法都使用了断言来验证方法的返回结果是否符合预期，同时利用 Mockito 框架模拟了UserRepository的save方法的不同行为，以覆盖各种可能的测试场景，确保UserService的registerUser方法在各种情况下都能正确工作。

yxysuanfa
粉丝 - 25 关注 - 0
0 0
+加关注

« 上一篇： 实用指南：DevOps历程--Drone安装使用详细教程
» 下一篇： 实用指南：星穹无损合约：以信任为基石，开启DeFi新纪元

posted @ 2025-09-16 10:36 yxysuanfa 阅读(102) 评论(0) 收藏 举报