JUnit5 框架

概述

- 网址：https://www.junit.org
- Github: https://github.com/junit-team/junit5
- 创始人：Kent Beck + Eric Gemma（VS Code）
- 最新版本：5.6.1/4.13
- IDE支持：Eclipse，IDEA，VS Code
- 构建工具支持：Gradle，Maven，Ant

 XUnit:

  - C/C++：googletest
  - 数据库集成测试：dbUnit
- JUnit 5：**需要 JDK 8+**
  - JUnit Platform
    - 启动测试框架
    - 定义 `TestEngine` 接口
    - 提供控制台启动器（`Console Launcher`）
  - JUnit Jupiter：编写基于 JUnit 5 的测试和扩展
  - JUnit Vintage：为 JUnit 4 和 JUnit 3 提供 TestEngine，使老的测试程序能够运行在 JUnit Platform 上
- 使用 JUnit 5

```
<!-- Maven -->
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-api</artifactId>
  <version>5.6.1</version>
```

```
    <scope>test</scope>
  </dependency>

  <!-- Spring Boot -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    <exclusions>
      <exclusion>
        <groupId>org.junit.vintage</groupId>
        <artifactId>junit-vintage-engine</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
```

编写测试

- `AAA` 模式
  - Arrange：准备测试环境。根据测试用例的描述进行准备
  - Act：调用被测对象。把测试用例中的测试数据作为参数传递到被测试对象的方法，Act结束后返回实际结果
  - Assert：断言。把实际结果与测试用例当中的预期结果进行对比

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import example.util.Calculator;
import org.junit.jupiter.api.Test;

class MyFirstJUnitJupiterTests { // 测试类
  private final Calculator calculator = new Calculator();

  @Test
  void addition() { // 测试方法
    assertEquals(2, calculator.add(1, 1)); // 断言
  }
}
```

- 测试类
  - 顶级类，静态成员类
  - **必须包含至少一个测试方法**
  - **必须不能是抽象类**
  - **只能有一个构造函数**
  - 单元测试类一般命名为 `*Test[s]`
  - 集成测试类一般命名为`*IntegrationTest[s]`
- 测试方法
  - 被`@Test`，`@RepeatedTest`，`@ParameterizedTest`，`@TestFactory`，`@TestTemplate`等注解直接修饰的的**实例**（不能是静态）方法
- 生命周期方法：被`@BeforeAll`，`@AfterAll`，`@BeforeEach`，`@AfterEach`等注解直接修饰的任意方法
- **测试方法和生命周期方法可以包含在测试类中，也可以从父类或接口中继承**
- **测试方法和生命周期方法必须不能是抽象的，也必须不能返回任何值**
- **测试类，测试方法，生命周期方法不要求是`public`，但不能是`private`**

案例：基本用法

- 编写被测组件

```
public final class Money {
    private final String currency;
    private final double amount;

    public Money(String currency, double amount) {
        if(amount < 0) {
            throw new NegativeMoneyAmountException();
        }
        this.amount = amount;
        this.currency = currency;
    }

    public static Money dollar(double amount) {
        return new Money("USD", amount);
    }

    public static Money renminbi(double amount) {
        return new Money("RMB", amount);
    }

    public Money add(Money money) {
        if(money.currency.equals(this.currency)) {
            return new Money(this.currency, amount);
        }
        throw new BadCurrencyException(money.currency);
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Money money = (Money) o;
        return Double.compare(money.amount, amount) == 0 &&
                Objects.equals(currency, money.currency);
    }

    @Override
    public int hashCode() {
        return Objects.hash(currency, amount);
    }
}
```

- 编写测试

```
@DisplayName("Test My Money Class")
// @Disabled("先不运行本次测试")
class MoneyTest {
    @Test
    // @Tag("fast")
    // @Order(1)
    // @EnabledOnOs(WINDOWS) // 只在 Windows 上运行
    // @EnabledOnJre(JAVA_8) // 至少需要 JRE 8 版本
```

```
    @DisplayName("test equality with different currency and same amount")
    void test_equality_with_different_currency() {
        Money dollar5 = Money.dollar(5);
        Money renminbi5 = Money.renminbi(5);
        Assertions.assertNotEquals(dollar5, renminbi5);
    }

    @Test
    // @Order(2)
    @DisplayName("test equality with same currency and amount")
    void test_equality_with_same_currency_and_amount() {
        Money dollar5 = Money.dollar(5);
        Money anotherDollar5 = Money.dollar(5);
        Assertions.assertEquals(dollar5, anotherDollar5);
    }

    @Test
    // @Disabled("先不测试不同货币造成的异常")
    void test_bad_currency_exception() {
        Money dollar5 = Money.dollar(5);
        Money renminbi5 = Money.renminbi(5);
        Assertions.assertThrows(BadCurrencyException.class, () ->
dollar5.add(renminbi5));
        Assertions.assertDoesNotThrow(() -> dollar5.add(dollar5));
    }

  @ParameterizedTest
  @MethodSource("moneyProvider")
  void palindromes(Money m1, Money m2, boolean equality) {
    assertEquals(equality, m1.equals(m2));
  }

  static Stream<Arguments> moneyProvider() {
     return Stream.of(
            Arguments.of(Money.dollar(5), Money.dollar(5), true),
            Arguments.of(Money.dollar(5), Money.dollar(6), false),
            Arguments.of(Money.renminbi(5), Money.dollar(5), false),
            Arguments.of(Money.renminbi(5), Money.dollar(6), false));
  }
}
```

- 生命周期方法
  - `@BeforeEach`：每个测试之前执行的操作。一般为重置被测试对象的状态
  - `@BeforeAll`：第一个测试之前执行的操作。静态方法。一般为准备测试环境，如数据库连接等
  - `@AfterEach`：每个测试之后执行的操作。如，删除测试过程中生成的临时文件等
  - `@AfterAll`：最后一个测试之后执行的操作。静态方法。一般为销毁测试环境，如释放数据库连接

参数测试（Parameterized Tests）

- 使用多份不同的测试数据执行同一个测试
- `@ParameterizedTest`：声明参数测试
- `@ValueSource`：测试数据的来源

```
@ParameterizedTest
@ValueSource(strings = { "racecar", "radar", "able was I ere I saw elba" })
void palindromes(String candidate) {
  assertTrue(StringUtils.isPalindrome(candidate));
}
```

- 参数测试依赖：`org.junit.jupiter:junit-jupiter-params:5.6.1`
- 参数化测试方法的参数必须满足：
  - [可选] 第一个参数为参数索引。索引源由 `ArgumentsProvider` 提供
  - [可选] 第二个参数为聚合器（aggregator）。聚合器从数据源将数据聚合为对象
  - [可选] 第三个参数为 `ParameterResolver` 接口的实现

参数源（Sources of Arguments）

- `@ValueSource`：指定值数组。数组的类型可以为基本类型和 `String`，`Class` 类型

```
@ParameterizedTest
@ValueSource(ints = { 1, 2, 3 })
void testWithValueSource(int argument) {
  assertTrue(argument > 0 && argument < 4);
}
```

- `@EnumSource`：使用枚举指定参数源

```
@ParameterizedTest
@EnumSource(mode = EXCLUDE, names = { "ERAS", "FOREVER" })
void testWithEnumSourceExclude(ChronoUnit unit) {
assertFalse(EnumSet.of(ChronoUnit.ERAS, ChronoUnit.FOREVER).contains(unit));
}
```

- `@MethodSource`：指定测试类或外部类的工厂方法来生成参数
  - 测试类中的工厂方法必须是 `static`，除非被 `@TestInstance(Lifecycle.PER_CLASS)` 所注解
  - 外部类中的工厂方法必须是 `static` 的
  - 每个工厂方法必须生成一个参数的流（stream of arguments）。流中的每个参数将被作为 `参数测试方法` 的参数进行使用

```
@ParameterizedTest
@MethodSource("stringProvider") void
testWithExplicitLocalMethodSource(String argument) {
  assertNotNull(argument);
}
static Stream<String> stringProvider() {
  return Stream.of("apple", "banana");
}
```

  - 如果 `参数测试方法` 声明了多个参数，则 `工厂方法` 需要返回 `Stream<Arguments>`

```java
@ParameterizedTest
@MethodSource("stringIntAndListProvider")
void testWithMultiArgMethodSource(String str, int num, List<String> list) {
  assertEquals(5, str.length());
  assertTrue(num >=1 && num <=2);
  assertEquals(2, list.size());
}
static Stream<Arguments> stringIntAndListProvider() {
  return Stream.of(
    arguments("apple", 1, Arrays.asList("a", "b")),
    arguments("lemon", 2, Arrays.asList("x", "y")));
}
```

- `@CsvSource`：从 `CSV` 文件中获取参数
- `@ArgumentsSource`： 通过实现`ArgumentsProvider`提供自定义，可重用的工厂方法

```java
@ParameterizedTest
@ArgumentsSource(MyArgumentsProvider.class)
void testWithArgumentsSource(String argument) {
  assertNotNull(argument);
}

public class MyArgumentsProvider implements ArgumentsProvider {
  @Override
  public Stream<? extends Arguments> provideArguments(ExtensionContext
context) {
    return Stream.of("apple", "banana").map(Arguments::of);
  }
}
```

参数转换

- 如果参数源提供的数据类型与`参数化测试方法`的参数类型不一致时，需要转换

参数聚合

测试模板

- `@TestTemplate`：提供了测试用例的模板

JUnit最佳实践

- 使用测试，不要使用System.out或调试
- 主代码与测试代码交替编写
  - 小步推进，编织安全网
  - 增强开发人员的信心
- 尽可能覆盖所有功能
- 每个公开方法都要被测到（Setter/Getter?）
- 经常性重构
- 为缺陷添加测试
- 测试类名为XxxTest，其中Xxx为被测的类名
- 测试方法名字要准确，要能够描述用例

JUnit思考题

- 如何测试返回类型为void的函数？ //Mockito
- 如何测试私有方法？或需要测试私有方法吗？
- 如何测试线程？
- 如何测试图形界面？
- 测试代码需要维护吗？
- JUnit框架可以用来做集成测试吗？
- 需要几个测试方法？
- 写这么多测试代码值得吗？