

## 07、单元测试

### 1、JUnit5 的变化

**Spring Boot 2.2.0 版本开始引入 JUnit 5 作为单元测试默认库**

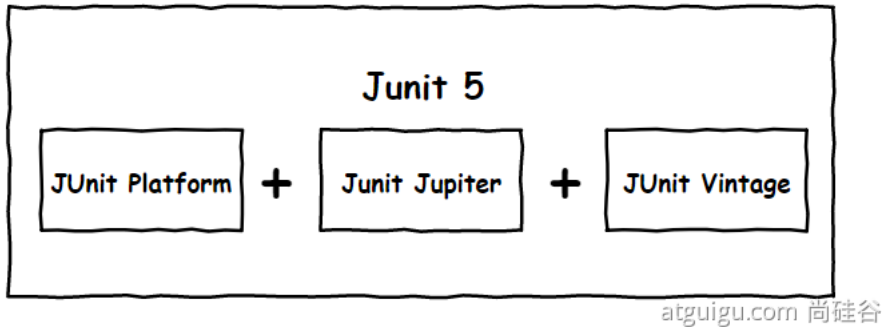
作为最新版本的JUnit框架，JUnit5与之前版本的JUnit框架有很大的不同。由三个不同子项目的几个不同模块组成。

**JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage**

**JUnit Platform:** JUnit Platform是在JVM上启动测试框架的基础，不仅支持JUnit自制的测试引擎，其他测试引擎也都可以接入。

**JUnit Jupiter:** JUnit Jupiter提供了JUnit5的新的编程模型，是JUnit5新特性的核心。内部 包含了一个**测试引擎**，用于在JUnit Platform上运行。

**JUnit Vintage:** 由于JUnit已经发展多年，为了照顾老的项目，JUnit Vintage提供了兼容JUnit4.x,JUnit3.x的测试引擎。
















注意:

**SpringBoot 2.4 以上版本移除了默认对 Vintage 的依赖。如果需要兼容junit4需要自行引入（不能使用junit4的功能 @Test）**

**JUnit 5' s Vintage Engine Removed from `spring-boot-starter-test`,如果需要继续兼容junit4需要自行引入vintage**

```
1 <dependency>
2   <groupId>org.junit.vintage</groupId>
3   <artifactId>junit-vintage-engine</artifactId>
4   <scope>test</scope>
5   <exclusions>
6     <exclusion>
7       <groupId>org.hamcrest</groupId>
8       <artifactId>hamcrest-core</artifactId>
9     </exclusion>
10  </exclusions>
11 </dependency>
```

- ▼  org.junit.jupiter:junit-jupiter:5.7.0 (test)
  - ▼  org.junit.jupiter:junit-jupiter-api:5.7.0 (test)
    -  org.apiguardian:apiguardian-api:1.1.0 (test)
    -  org.opentest4j:opentest4j:1.2.0 (test)
    - >  org.junit.platform:junit-platform-commons:1.7.0 (test)
    - >  org.junit.jupiter:junit-jupiter-params:5.7.0 (test)
    - ▼  org.junit.jupiter:junit-jupiter-engine:5.7.0 (test)
      -  org.apiguardian:apiguardian-api:1.1.0 (test omitted for duplicate)
      - ▼  org.junit.platform:junit-platform-engine:1.7.0 (test)
        -  org.apiguardian:apiguardian-api:1.1.0 (test omitted for duplicate)
        -  org.opentest4j:opentest4j:1.2.0 (test omitted for duplicate)
        -  org.junit.platform:junit-platform-commons:1.7.0 (test omitted for duplicate)
        -  org.junit.jupiter:junit-jupiter-api:5.7.0 (test omitted for duplicate)

```

1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-test</artifactId>
4   <scope>test</scope>
5 </dependency>

```

现在版本：

```

1 @SpringBootTest
2 class Boot05WebAdminApplicationTests {
3
4
5     @Test
6     void contextLoads() {
7
8     }
9 }
10
11

```

以前：

@SpringBootTest + @RunWith(SpringTest.class)

SpringBoot整合JUnit以后。

- 编写测试方法：@Test标注（注意需要使用junit5版本的注解）
- JUnit类具有Spring的功能，@Autowired、比如 @Transactional 标注测试方法，测试完成后自动回滚

## 2、JUnit5常用注解

JUnit5的注解与JUnit4的注解有所变化

<https://junit.org/junit5/docs/current/user-guide/#writing-tests-annotations> <<https://junit.org/junit5/docs/current/user-guide/#writing-tests-annotations>>

- @Test :表示方法是测试方法。但是与JUnit4的@Test不同，他的职责非常单一不能声明任何属性，拓展的测试将会由Jupiter提供额外测试
- @ParameterizedTest :表示方法是参数化测试，下方会有详细介绍

- **@RepeatedTest** :表示方法可重复执行，下方会有详细介绍
- **@DisplayName** :为测试类或者测试方法设置展示名称
- **@BeforeEach** :表示在每个单元测试之前执行
- **@AfterEach** :表示在每个单元测试之后执行
- **@BeforeAll** :表示在所有单元测试之前执行
- **@AfterAll** :表示在所有单元测试之后执行
- **@Tag** :表示单元测试类别，类似于JUnit4中的@Categories
- **@Disabled** :表示测试类或测试方法不执行，类似于JUnit4中的@Ignore
- **@Timeout** :表示测试方法运行如果超过了指定时间将会返回错误
- **@ExtendWith** :为测试类或测试方法提供扩展类引用

Java | Copy

```
1  import org.junit.jupiter.api.Test; //注意这里使用的是jupiter的Test注解！！
2
3
4
5  public class TestDemo {
6
7
8      @Test
9      @DisplayName("第一次测试")
10     public void firstTest() {
11         System.out.println("hello world");
12     }
13 }
```

### 3、断言 (assertions)

断言 (assertions) 是测试方法中的核心部分，用来对测试需要满足的条件进行验证。这些断言方法都是 `org.junit.jupiter.api.Assertions` 的静态方法。JUnit 5 内置的断言可以分成如下几个类别：

**检查业务逻辑返回的数据是否合理。**

**所有的测试运行结束以后，会有一个详细的测试报告；**

#### 1、简单断言

用来对单个值进行简单的验证。如：

方法	说明
assertEquals	判断两个对象或两个原始类型是否相等
assertNotEquals	判断两个对象或两个原始类型是否不相等
assertSame	判断两个对象引用是否指向同一个对象
assertNotSame	判断两个对象引用是否指向不同的对象
assertTrue	判断给定的布尔值是否为 true
assertFalse	判断给定的布尔值是否为 false
assertNull	判断给定的对象引用是否为 null
assertNotNull	判断给定的对象引用是否不为 null

```
1  @Test
2
3  @DisplayName("simple assertion")
4  public void simple() {
5      assertEquals(3, 1 + 2, "simple math");
6      assertNotEquals(3, 1 + 1);
7
8
9      assertNotSame(new Object(), new Object());
10     Object obj = new Object();
11     assertEquals(obj, obj);
12
13     assertFalse(1 > 2);
14     assertTrue(1 < 2);
15
16     assertNull(null);
17     assertNotNull(new Object());
18 }
```

## 2、数组断言

通过 `assertArrayEquals` 方法来判断两个对象或原始类型的数组是否相等

```
1  @Test
2
3  @DisplayName("array assertion")
4  public void array() {
5      assertArrayEquals(new int[]{1, 2}, new int[] {1, 2});
6  }
```

## 3、组合断言

`assertAll` 方法接受多个 `org.junit.jupiter.api.Executable` 函数式接口的实例作为要验证的断言，可以通过 `lambda` 表达式很容易的提供这些断言

```
1  @Test
2
3  @DisplayName("assert all")
4  public void all() {
5      assertAll("Math",
6          () -> assertEquals(2, 1 + 1),
7          () -> assertTrue(1 > 0)
8      );
9  }
```

## 4、异常断言

在JUnit4时期，想要测试方法的异常情况时，需要用`@Rule`注解的`ExpectedException`变量还是比较麻烦的。而JUnit5提供了一种新的断言方式 `Assertions.assertThrows()` ,配合函数式编程就可以进行使用。

Java | Copy

```
1  @Test
2  @DisplayName("异常测试")
3  public void exceptionTest() {
4      ArithmeticException exception = Assertions.assertThrows(
5          //抛出断言异常
6          ArithmeticException.class, () -> System.out.println(1 % 0));
7
8  }
```

## 5、超时断言

JUnit5还提供了**Assertions.assertTimeout()** 为测试方法设置了超时时间

Java | Copy

```
1  @Test
2  @DisplayName("超时测试")
3  public void timeoutTest() {
4      //如果测试方法时间超过1s将会异常
5      Assertions.assertTimeout(Duration.ofMillis(1000), () -> Thread.sleep(500));
6  }
```

## 6、快速失败

通过 fail 方法直接使得测试失败

Java | Copy

```
1  @Test
2  @DisplayName("fail")
3  public void shouldFail() {
4      fail("This should fail");
5  }
```

## 4、前置条件 (assumptions)

JUnit 5 中的前置条件（**assumptions【假设】**）类似于断言，不同之处在于**不满足的断言会使得测试方法失败**，而不满足的**前置条件只会使得测试方法的执行终止**。前置条件可以看成是测试方法执行的前提，当该前提不满足时，就没有继续执行的必要。

```
1  @DisplayName("前置条件")
2
3  public class AssumptionsTest {
4      private final String environment = "DEV";
5
6
7      @Test
8      @DisplayName("simple")
9      public void simpleAssume() {
10         assertTrue(Objects.equals(this.environment, "DEV"));
11         assertFalse(() -> Objects.equals(this.environment, "PROD"));
12     }
13
14
15
16
17     @Test
18     @DisplayName("assume then do")
19     public void assumeThenDo() {
20         assumingThat(
21             Objects.equals(this.environment, "DEV"),
22             () -> System.out.println("In DEV")
23         );
24     }
25 }
```

`assertTrue` 和 `assertFalse` 确保给定的条件为 `true` 或 `false`，不满足条件会使得测试执行终止。`assumingThat` 的参数是表示条件的布尔值和对应的 `Executable` 接口的实现对象。只有条件满足时，`Executable` 对象才会被执行；当条件不满足时，测试执行并不会终止。

## 5、嵌套测试

JUnit 5 可以通过 Java 中的内部类和 `@Nested` 注解实现嵌套测试，从而可以更好的把相关的测试方法组织在一起。在内部类中可以使用 `@BeforeEach` 和 `@AfterEach` 注解，而且嵌套的层次没有限制。

```
1  @DisplayName("A stack")
2
3  class TestingAStackDemo {
4
5
6      Stack<Object> stack;
7
8
9      @Test
10     @DisplayName("is instantiated with new Stack()")
11     void isInstantiatedWithNew() {
12         new Stack<>();
13     }
14
15
16
17     @Nested
18     @DisplayName("when new")
19     class WhenNew {
20
21
22
23         @BeforeEach
24         void createNewStack() {
25             stack = new Stack<>();
26         }
27
28
29
30         @Test
31         @DisplayName("is empty")
32         void isEmpty() {
33             assertTrue(stack.isEmpty());
34         }
35
36
37
38         @Test
39         @DisplayName("throws EmptyStackException when popped")
40         void throwsExceptionWhenPopped() {
41             assertThrows(EmptyStackException.class, stack::pop);
42         }
43
44
45
46         @Test
47         @DisplayName("throws EmptyStackException when peeked")
48         void throwsExceptionWhenPeeked() {
49             assertThrows(EmptyStackException.class, stack::peek);
50         }
51     }
52 }
```

## 6、参数化测试

参数化测试是JUnit5很重要的一个新特性，它使得用不同的参数多次运行测试成为了可能，也为我们的单元测试带来许多便利。

利用@ValueSource等注解，指定入参，我们将可以使用不同的参数进行多次单元测试，而不需要每新增一个参数就新增一个单元测试，省去了很多冗余代码。

**@ValueSource**: 为参数化测试指定入参来源，支持八大基础类以及String类型,Class类型

**@NullSource**: 表示为参数化测试提供一个null的入参

**@EnumSource**: 表示为参数化测试提供一个枚举入参

**@CsvFileSource**: 表示读取指定CSV文件内容作为参数化测试入参

**@MethodSource**: 表示读取指定方法的返回值作为参数化测试入参(注意方法返回需要是一个流)

当然如果参数化测试仅仅只能做到指定普通的入参还达不到让我觉得惊艳的地步。让我真正感到他的强大之处的地方在于他可以支持外部的各类入参。如:CSV,YML,JSON 文件甚至方法的返回值也可以作为入参。只需要去实现**ArgumentsProvider**接口，任何外部文件都可以作为它的入参。

Java | Copy

```
1  @ParameterizedTest
2
3  @ValueSource(strings = {"one", "two", "three"})
4  @DisplayName("参数化测试1")
5
6  public void parameterizedTest1(String string) {
7      System.out.println(string);
8      Assertions.assertTrue(StringUtils.isNotBlank(string));
9  }
10
11
12
13
14  @ParameterizedTest
15  @MethodSource("method")    //指定方法名
16  @DisplayName("方法来源参数")
17
18  public void testWithExplicitLocalMethodSource(String name) {
19      System.out.println(name);
20      Assertions.assertNotNull(name);
21  }
22
23
24  static Stream<String> method() {
25      return Stream.of("apple", "banana");
26  }
```

## 7、迁移指南

在进行迁移的时候需要注意如下的变化：

- 注解在 org.junit.jupiter.api 包中，断言在 org.junit.jupiter.api.Assertions 类中，前置条件在 org.junit.jupiter.api.Assumptions 类中。
- 把@Before和@After 替换成@BeforeEach和@AfterEach。
- 把@BeforeClass和@AfterClass 替换成@BeforeAll和@AfterAll。
- 把@Ignore 替换成@Disabled。
- 把@Category 替换成@Tag。
- 把@RunWith、@Rule和@ClassRule 替换成@ExtendWith。