Junit 语言整合

JUnit中的各种断言(Assertion)操作:

1. assertTrue/assertFalse

- 用途：断言一个布尔条件是真或假
- 语法：

```
assertTrue(condition)
assertTrue(condition, message)  // 带错误信息
assertFalse(condition)
assertFalse(condition, message)  // 带错误信息
```

- 示例：

```
assertTrue(5 > 3);  // 通过
assertTrue(10 > 20, "10应该大于20"); // 失败并显示消息
assertFalse(3 > 5);  // 通过
```

2. assertSame/assertNotSame

- 用途：断言两个对象引用是否为同一个对象(比较引用而不是值)
- 语法：

```
assertSame(expected, actual)
assertSame(expected, actual, message)
assertNotSame(expected, actual)
assertNotSame(expected, actual, message)
```

- 示例：

```
String str1 = new String("hello");
String str2 = new String("hello");
String str3 = str1;
```

```
assertNotSame(str1, str2);  // 通过，虽然值相同但是不是同一个对象
assertSame(str1, str3);     // 通过，因为str3引用了str1的对象
```

3. assertEquals/assertNotEquals

- 用途：断言两个对象的值是否相等(比较值而不是引用)
- 语法：

```
assertEquals(expected, actual)
assertEquals(expected, actual, message)
assertNotEquals(expected, actual)
assertNotEquals(expected, actual, message)
```

- 示例：

```
String str1 = new String("hello");
String str2 = new String("hello");
assertEquals(str1, str2);  // 通过，因为值相同
assertEquals(5, 2 + 3);    // 通过
assertNotEquals(5, 6);     // 通过
```

4. assertArrayEquals

- 用途：断言两个数组是否相等(长度相同且对应位置的元素相等)
- 语法：

```
assertArrayEquals(expected, actual)
assertArrayEquals(expected, actual, message)
```

- 示例：

```
int[] arr1 = {1, 2, 3};
int[] arr2 = {1, 2, 3};
int[] arr3 = {1, 2, 4};
```

```
assertArrayEquals(arr1, arr2);  // 通过
assertArrayEquals(arr1, arr3);  // 失败
```

5. assertThrows

- 用途：断言某段代码会抛出特定类型的异常
- 语法：

```
assertThrows(expectedExceptionClass, executable)
```

- 示例：

```
Calculator calculator = new Calculator();
assertThrows(ArithmeticException.class,
    () -> calculator.divide(10, 0));  // 通过，因为除以0会抛出ArithmeticException
```

特别注意：

1. 所有断言方法都可以添加可选的message参数，在测试失败时显示
2. assertSame和assertEquals的区别在于前者比较对象引用，后者比较对象值
3. 断言方法的选择应该基于测试的具体需求，使用最能表达测试意图的方法

这些断言方法共同构成了JUnit测试框架的基础，可以帮助我们验证代码的正确性。选择合适的断言方法有助于使测试代码更清晰、更有表达力。

各个@分别代表什么：

两类，第一类是代表生命周期：

```
@BeforeAll
// - 在所有测试方法执行之前运行一次
// - 必须是static方法
// - 用于设置重量级资源(如数据库连接)

@BeforeEach
```

```
// - 在每个测试方法执行之前运行
// - 不能是static方法
// - 用于设置/重置测试对象

@AfterEach
// - 在每个测试方法执行之后运行
// - 不能是static方法
// - 用于清理测试方法的环境

@AfterAll
// - 在所有测试方法执行完成后运行一次
// - 必须是static方法
// - 用于清理重量级资源
```

示例：

```
class MyTest {
    @BeforeAll
    static void initAll() {
        System.out.println("初始化所有测试");
    }

    @BeforeEach
    void init() {
        System.out.println("初始化单个测试");
    }

    @Test
    void test1() {
        // 测试代码
    }

    @AfterEach
    void cleanup() {
        System.out.println("清理单个测试");
    }

    @AfterAll
    static void cleanupAll() {
        System.out.println("清理所有测试");
    }
}
```

All 和 each的差别在于All方法只执行一次，而each会在每个对应的方法前都执行一次（大概可以理解为对应each我们就是重置回初始状态，对应all就是设定初始状态？）

第二类是代表注解

```
@DisplayName("测试名称")
// - 用于设置测试的显示名称，使测试报告更易读
// 示例：
@DisplayName("测试加法功能")
void testAddition() {
    assertEquals(4, 2 + 2);
}

@Timeout(value = 1)
// - 用于设置测试方法的超时时间
// - 默认单位是秒，可配置
// 示例：
@Timeout(value = 100, unit = TimeUnit.MILLISECONDS)
void testWithTimeout() {
    // 测试代码需要在100毫秒内完成
}

@RepeatedTest(次数)
// - 用于重复执行测试多次
// - 不需要额外的@Test注解
// 示例：
@RepeatedTest(3)
void repeatedTest() {
    assertTrue(true);
}  // 这个测试会执行3次
```

注意事项：

1. @BeforeAll 和 @AfterAll 必须是静态方法，因为它们在测试类实例化之前/后执行
2. @BeforeEach 和 @AfterEach 用于实例方法，因为它们需要操作测试实例的状态
3. @RepeatedTest 使用时不需要再加 @Test 注解，否则会收到警告