# 单元测试：单元测试与 Mock：使用 Mockito 进行 Mock 对象创建

## 1 单元测试基础

### 1.1 单元测试的概念与重要性

单元测试是软件开发过程中的一个重要组成部分，它主要关注于测试软件中的最小可测试单元，通常是单个函数或方法。单元测试的目的是验证这些单元在独立于其他部分的情况下是否能够正确运行。这有助于确保代码的健壮性，减少集成时的错误，提高软件质量，同时也便于代码的维护和重构。

#### 1.1.1 重要性

1. **错误检测**：单元测试可以在开发早期阶段检测到错误，避免将错误遗留到后期，减少修复成本。
2. **代码质量**：通过单元测试，开发者可以确保代码的每个部分都按预期工作，提高代码的可靠性和质量。
3. **重构支持**：单元测试为代码重构提供了安全网，确保在修改代码后，原有的功能仍然正常。
4. **文档作用**：良好的单元测试可以作为代码的活文档，帮助新开发者理解代码的意图和功能。
5. **持续集成**：单元测试是持续集成流程中的关键环节，确保每次代码提交后，软件的基本功能仍然可用。

### 1.2 单元测试的生命周期

单元测试的生命周期通常包括以下几个阶段：

1. **编写测试**：首先，根据代码的功能和需求编写测试用例。
2. **运行测试**：使用测试框架运行测试，如 JUnit。
3. **评估结果**：检查测试结果，确定测试是否通过。
4. **修复错误**：如果测试失败，定位并修复代码中的错误。
5. **重构代码**：在确保测试通过后，可以安全地重构代码。
6. **重复**：上述过程在软件开发的整个生命周期中不断重复，以确保代码质量。

### 1.3 JUnit 简介与基本用法

JUnit 是一个流行的 Java 单元测试框架，它简化了单元测试的编写和执行过程。JUnit 支持自动化测试，可以与持续集成工具如 Jenkins 配合使用，提高测

试效率。

### 1.3.1 基本用法

下面是一个使用 JUnit 进行单元测试的简单示例：

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class CalculatorTest {

    private Calculator calculator = new Calculator();

    /**
     * 测试加法功能
     */
    @Test
    public void testAdd() {
        int result = calculator.add(5, 3);
        assertEquals(8, result);
    }

    /**
     * 测试减法功能
     */
    @Test
    public void testSubtract() {
        int result = calculator.subtract(5, 3);
        assertEquals(2, result);
    }
}
```

在这个例子中，我们创建了一个 CalculatorTest 类，用于测试 Calculator 类的加法和减法功能。每个测试方法都使用@Test 注解标记，表示这是一个测试方法。assertEquals 方法用于验证方法的返回值是否与预期相符。

### 1.3.2 JUnit 注解

JUnit 提供了多种注解来控制测试的执行：

- @Test：标记一个方法为测试方法。
- @Before：在每个测试方法执行前运行的方法，用于设置测试环境。
- @After：在每个测试方法执行后运行的方法，用于清理测试环境。
- @BeforeClass：在所有测试方法执行前运行一次的方法，通常用于初始化类级别的资源。
- @AfterClass：在所有测试方法执行后运行一次的方法，用于释放类级别的资源。