853769620

我们正好基于之前快速写好的一个所谓“用户管理”的模块，完整的模块，我们来做所有代码组件的一个单元测试

（1）JUnit

（2）Mockito做测试替身，隔离各个代码组件

（3）Hamcrest去做复杂的断言

（4）spring test等框架去在spring boot、spring等框架的环境下，去进行相关的测试

1、JUnit的介绍

JUnit的使用非常容易上手，测试类放在src/main/test下，然后类名一般用Test结尾

通常来说，一个测试类对应一个系统类，类中的每个方法，用于测试一个方法，方法一般以test开头，每个方法用@Test标注

如果有测试开始前要执行的操作，用@Before标注一个方法来执行；如果有测试结束之后要执行的操作，用@After标注一个方法来执行，一般会将多个测试方法中的初始化和资源清理代码放在这些方法中

每次运行一个方法，都会创建一个类的实例，将各个测试方法互相之间隔离开来，避免互相影响，导致测试失败

在每个测试方法中，最后都会用Assert类来进行断言，判断测试是否通过，就是看一下测试得到的结果，是不是我们期望的，如果不是会报错，表示测试不通过

JUnit还有一个Suite的概念，一次性运行多个测试用例，算是一个测试套件，比如下面这样子【用的少】

@RunWith(Suite.class)

@SuiteClasses({TestSuite1.class, TestSuite2.class})

public class TestSuitMain {

}

我们一般都是，写好一些代码组件，然后就针对这些代码组件，去立即写对应的单元测试，单元测试是一个类一个类的执行

到了最后，你都要集成测试，mvn test，将所有的单元测试全部跑一遍

2、spring-boot-starter-test

spring boot对单元测试提供了很好的支持，只要依赖spring-boot-starter-test即可，这个依赖会自动导入单元测试需要的所有依赖，包括了JUnit、AssertJ、Hamcrest以及其他的一些包，而且这个spring-boot-starter-test一般是设置为test scope即可

具体来说，spring-boot-starter-test导入之后，会包含下面这些东西：

（1）JUnit：最经典的单元测试框架

（2）Spring Test、Spring Boot Test：是spring和spring boot环境下，对测试的一个支持

（3）AssertJ：是用来进行断言的

（4）Hamcrest：是用来进行复杂断言，复杂的表达式

（5）Mockito：测试替身的模拟

（6）JSONassert：都是对json数据进行操作的

（7）JsonPath

3、测试spring boot应用程序

3.1 spring boot单元测试脚手架

// @RunWith的意思，是不要使用默认方式进行单元测试，而是使用指定的类来提供单元测试

// 所有的spring测试都是找SpringRunner.class

// 注解含义：在执行单元测试的时候，不是直接去执行里面的单元测试的方法，因为那些方法执行之前，是需要做一些准备工作的，它是需要先初始化一个spring容器，所以得找到这个SpringRunner类，来先准备好spring容器，再执行各个测试方法

@RunWith(SpringRunner.class)

// 这个是spring boot提供的，会一直找到一个Application类，只要包含了@SpringBootApplication的就算，然后会先启动这个类，来给单元测试提供环境

@SpringBootTest

public class EmployeeServiceTest {

   // 这里就可以从启动的spring上下文中，将EmployeeService注入到这里来，供我们进行测试

   @Autowired

   private EmployeeService employeeService;

   @Test

   public void testFindById() {

  }

}

3.2 测试service组件

要考虑的地方有3点：

（1）测试之前自动构造好数据，测试结束之后自动回滚数据构造

（2）将service依赖的service进行模拟打桩进来

（3）可能需要在数据库中构造好数据

spring boot会默认在单元测试结束之后，进行数据库事务回滚；spring boot会集成mockito框架来模拟依赖service进行打桩；spring boot有一个@Sql注解，可以在测试开始前执行SQL语句初始化数据

import static org.mockito.BDDMockito.given;

import static org.mockito.Mockito.\*;

@RunWith(SpringRunner.class)

@SpringBootTest

@Transactional

@Rollback(true)

public class EmployeeServiceTest {

   @Autowired

   private EmployeeService employeeService;

   // 这里是基于Mockito框架模拟出来了一个EmployeeMapper

   // 可能我们到这里为止，只想要测试EmployeeService，而EmployeeService依赖了EmployeeMapper，那么我们就需要自己模拟一个EmployeeMapper出来

   // 这里定义的@MockBean，会模拟生成一个EmployeeMapper，放入spring容器中去，然后被EmployeeService给引用到

   @MockBean

   private EmployeeMapper employeeMapper;

   @Test

   public void testService() {

       int employeeId = 1;

       Employee employee = new Employee();

       employee.setId(1);

       employee.setName("张三");

       employee.setAge(20);

       // 这行代码是对employeeMapper的findById方法进行模拟

       // 无论给这个方法传入什么参数

       // 都会返回一个Employee对象，这个是我们预先定义好的employee对象

       given(this.employeeMapper.findById(anyLong()))

              .willReturn(employee);

       Employee resultEmployee = employeeService.findById(employeeId);

       assertEquals(employee, resultEmployee);

  }

}

3.3 测试Controller

@RunWith(SpringRunner.class)

// 这个代表是Mvc测试，可以传入要测试的Controller类

@WebMvcTest(EmployeeController.class)

public class EmployeeControllerTest {

   // 这个是专门用来测试mvc的类，可以模拟发起http请求

   @Autowired

   private MockMvc mvc;

   // 这里是模拟以一个service组件

   @MockBean

   private EmployeeService employeeService;

   @Test

   public void testMvc() throws Exception {

       int employeeId = 1;

       // 这里是模拟service组件的行为

       Employee employee = new Employee();

       employee.setId(1);

       employee.setName("张三");

       employee.setAge(20);

       given(this.employeeService.findById(employeeId)).willReturn(employee);

       // 这里是模拟发起一个http请求

       mvc.perform(get("/employee/{id}", employeeId))

              .andExpect(model().attribute("id", 1))

              .andExpect(model().attribute("name", "张三"))

              .andExpect(model().attribute("age", 20));

  }

}

3.4 mvc请求模拟

模拟GET请求：mvc.perform(get("/employee/{id}", 1))

模拟POST请求：mvc.perform(post("/employee/{id}", 1))

模拟文件上传：mvc.perform(multipart("/upload").file("file", "文件内容".getBytes("UTF-8")))

模拟表单请求：mvc.perform(post("/employee").param("name", "张三").param("age", 20))

模拟session：mvc.perform(get("/employee").sessionAttr(name, value))

模拟cookiei：mvc.perform(get("/employee").cookie(new Cookie(name ,value)))

模拟HTTP body内容，比如json：mvc.perform(get("/employee").content(json))

模拟设置HTTP header：

mvc.perform(get("/employee/{id}"), employeeId)

        .contentType("application/x-www-form-urlencoded")

        .accept("application/json")

        .header(header, value)

3.5 比较mvc请求返回结果

mvc.perform(get("/employee"))

        .andExpect(status().isOk()) // 响应是否为200状态码

        .andExpect(content().contentType(MediaType.APPLICATION\_JSON)) // 返回的content type是不是application/json

        .andExpect(jsonPath("$.name").value("张三")) // 返回内容本身检查

mvc.perform(post("/employee"))

        .andExpect(model().attribute("name", "张三"))

        .andExpect(model().attributeExists("name"))

4、Mockito使用

假设我们有一个接口，需要来模拟这个接口的实现类对象

public interface EmployeeService {

   public Employee findById(Long id);

   public boolean add(Employee employee);

}

import static org.mockito.Mockito.\*;

@RunWith(MockitoJUnitRunner.class)

public class EmployeeServiceTest {

   @Test

   public void test() {

       // 模拟出来一个实现了EmployeeService接口的对象

       EmployeeService employeeService = mock(EmployeeService.class);

       // 这个对象的findById()方法无论传入什么参数，都是返回预定义的一个对象

       when(employeeService.findById(anyLong())).thenReturn(employee);

       // 尝试调用

       Employee resultEmployee = employeeServicie.findById(1);

       // 比较结果

       assertEquals(employee, resultEmployee);

  }

}

一般是不建议在单元测试中使用any类方法的，而是用eq(1)之类的，传入指定的值，而且用了eq()的意义就是，如果传递的不是这个值，就报错

还可以检查一个接口被调用的次数

when(employeeService.findById(eq(1))).thenReturn(employee);

Employee resultEmployee = employeeService.findById(1);

assertEquals(employee, resultEmployee);

// 这里就是检查employeeService的findById(1)调用是否为2次

verify(employeeService ,times(2)).findById(eq(1));

此外，还可以检查多个接口被调用的顺序

EmployeeService employeeService = mock(EmployeeService.class);

when(employeeService.findById(eq(1))).thenReturn(employee);

when(employeeService.add(eq(employee))).thenResult(true);

boolean addResult = employeeService.add(employee);

Employee resultEmployee = employeeService.findById(1);

// 这里就是在验证，是否按照下面的顺序完成的调用

InOrder inOrder = inOrder(employeeService);

inOrder.verify(employeeService).add(employee);

inOrder.verify(employeeService).findById(1);

还可以模拟抛出异常

EmployeeService employeeService = mock(EmployeeService.class);

when(employeeService.findById(lt(0))).thenThrow(new IllegalArgumentException("employeeId不能小于0"))

Employee reseultEmployee = employeeService.findById(-1);

5、DAO层单元测试

5.1 @Sql

可以使用@Sql先构造好需要的数据

@RunWith(SpringRunner.class)

@SpringBootTest

@Transactional

@Rollback(true)

public class EmployeeMapperTest {

   @Autowired

   private EmployeeMapper employeeMapper;

   @Test

   // 这个@Sql注解，就会在我们执行测试之前，先执行sql语句，初始化数据

   @Sql({"employee.sql"})

   public void testAddEmployee() {

       Employee employee = new Employee();

       employee.setName("李四");

       employee.setAge(30);

       Long employeeId = employeeService.add(employee);

       employee.setId(employeeId);

       assertTrue(employeeId > 0);

       // 接着需要从数据库中查询数据来比较

       Employee resultEmployee = employeeMapper.findById(1);

       assertEquals(employee, resultEmployee);

  }

}

6、assertThat和Hamcrest

* 单元测试结构
  + @Before
  + @Test
  + @After
* 断言
  + assertEquals
  + assertTrue / assertFalse
  + assertNull / assertNotNull
  + assertSame / assertNotSame
  + assertArrayEquals
  + assertThat
* 测试异常
  + @Test(expected = NullPointException.class)
* 主动失败
  + fail
* JUnit + Hamcrest
  + assertThat(str.indexOf("hello"), is(not(-1)))
  + assertThat(str.contains("hello"), equals(true))
  + assertThat(str, containsString("hello"))
  + is、not
  + equalTo / sameInstance、nullValue / notNullValue、instanceOf
  + hasProperty
  + hasEntry、hasKey、hasValue、hasItem / hasItems、hasItemInArray、in
  + greaterThan、greaterThanOrEqualTo、lessThan、lessThanOrEqualTo
  + containsString、endsWith、startsWith
* JUnit + Mockito
  + when().thenReturn()