切片测试和集成测试

2021年10月3日 20:07

切片测试和集成测试使用相同的技术，都需要依赖Spring框架

切片测试

我们在这个例子中把Controller层和Service层切开，单独测试Controller层
因为只有一个GoodsController，所以我们做了一个GoodsControllerTest的类
在类前面，有两个标签

```
@SpringBootTest(classes = DemoApplication.class)  //标识本类是一个SpringBootTest
@AutoConfigureMockMvc   //配置模拟的MVC，这样可以不启动服务器测试
public class GoodsControllerTest {
    @Autowired
```

第一个是@SpringBootTest，用来表示当前的类是一个SpringBoot测试类
我们要测试Controller的话，它需要依赖于Spring的环境，因此我们在后面的括号里写了和生产代码用同样的环境

```
package cn.edu.xmu.restfuldemo;

import ...

/**
 * @author Ming Qiu
 */
@SpringBootApplication(exclude = {DataSourceAutoConfiguration.class})
public class DemoApplication {


    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }

}
```

这里的 exclude={DataSourceAutoConfiguration.class} 意思是：
当你项目需要自定义数据源的时候，或者不想系统自动注入数据源的时候，就把它排除掉

第二个注解是@AutoConfigureMockMvc，是为了提供一个模拟的环境
因为真正的Controller需要一个Servlet容器，还需要网络等去发送接收Http请求，会很麻烦
所以我们在测试Controller时往往用一个模拟环境，去模拟Http的Request/Response过程
同时，这个Auto是为了给Mvc提供一些自动配置，其实我们也可以在标签后面给他配置一些值

```
public class GoodsControllerTest {
    @Autowired
    private MockMvc mvc;

    @MockBean
    private GoodsService goodsService;
```

之后我们可以看到两个对象
一个是MockMvc对象
一个是加了@MockBean的Service对象，因为我们要把Controller和Service切割开，所以我们用这个注解去模拟一个Service层对象
在之后的方法里我们可以去实现GoodsService中的一些方法

小知识：getInstance
这里我们看到创建对象用的是GoodsFactory下的getInstance

```
public class GoodsFactory {

    private static GoodsFactory instance =  null;

    public static GoodsFactory getInstance(){
        if (instance == null){
            synchronized (GoodsFactory.class){
                if (instance == null){
                    instance = new GoodsFactory();
                }
            }
        }
        return instance;
    }
```

这个一般用在单例模式中，保证这个类在实例化之后有且只有一个实例

```
@Test   //标识此方法为测试方法
public void getGoodByIdTest() throws Exception {
    given(goodsService.findById(eq( value: 1))).willReturn(GoodsFactory.getInstance().createGoods( id: 1));
    String responseString = this.mvc.perform(get( urlTemplate: "/goods/1"))
            .andExpect(status().isOk())
            .andExpect(content().contentType("application/json;charset=UTF-8"))
            .andReturn().getResponse().getContentAsString();

    String expectedResponse = "{\"errno\":0,\"data\":{\"id\":1,\"goodsSn\":\"111111\",\"name\":\"红米4X\",\"catego

    JSONAssert.assertEquals(expectedResponse, responseString,  strict: true);
}
```

这是第一个方法，前面要加上@Test，表示这是一个测试方法
第一句是一个given，用来定义需要用到的Service层方法（在这里看做一个未实现的Service方法）
通过given，定义了如果findId的参数传进来是1，那么willReturn将返回一个id为1的Goods对象
（这里的eq是Mockito的ArgumentMatcher，用来做参数匹配）

有了这样的前提，我们就可以用MockMvc进行发送请求
首先是perform发出请求/goods/1
andExpect是期望的返回值，status.isOk是200
同时期望的contentType是json和utf-8
然后要拿到真正的返回值，所以使用.andReturn().getResponse().getContentAsString()

之后我们写一个期望的字符串expectedResponse，因为我们知道拿到的是一个Json字符串，所以我们写的也是一个json字符串

之后我们使用JSONAssert这个包，可以用来比较两个Json字符串
assertEquals三个参数，第一个是期望得到的，第二个是用来比较的返回值，第三个用来控制是否严格比较
严格比较：期望值和返回值必须完全一致
不严格比较：返回值可以比期望值多一些，只需要期望值包含在返回值中即可

其余类似的方法大同小异

```java
@Test
public void createGoodTest() throws Exception {

    GoodsVo g = GoodsFactory.getInstance().createGoodsVo();
    Goods goods = g.createGoods();
    goods.setId(1);
    goods.setGoodsSn("11111");
    given(goodsService.createGoods(any())).willReturn(goods);

    String goodJson = JacksonUtil.toJson(g);

    String responseString = this.mvc.perform(post( urlTemplate: "/goods").contentType("a
            .andExpect(status().isCreated())
            .andExpect(content().contentType("application/json;charset=UTF-8"))
            .andReturn().getResponse().getContentAsString();

    String expectedResponse = "{\"errno\":0,\"data\":{\"id\":1,\"goodsSn\":\"11111\",

    JSONAssert.assertEquals(expectedResponse, responseString,  strict: true);
}
```

这里我们要注意这个any

这个方法是createGoodTest，这里的参数需要用到一个Vo对象，但这个传进去的vo对象和我们创建的不一定是同一个，所以我们用这个any将两个进行统一，让后面的return返回我们建出来的vo对象

集成测试

集成测试在GoodsControllerIntegrationTest中

代码和切片测试基本一样，不同的是我们没有去模拟GoodsService对象，也没有去规定Service的返回值