**Spring-Boot学习笔记**

# Spring-Boot的API单元测试

## 1 MockMvc详解

### 1.1 为何使用MockMvc

对模块进行集成测试时，希望能够通过输入URL对Controller进行测试，如果通过启动服务器，建立http client进行测试，这样会使得测试变得很麻烦，比如，启动速度慢，测试验证不方便，依赖网络环境等，所以为了可以对Controller进行测试，引入了MockMVC。

MockMvc实现了对Http请求的模拟，能够直接使用网络的形式，转换到Controller的调用，这样可以使得测试速度快、不依赖网络环境，而且提供了一套验证的工具，这样可以使得请求的验证统一而且很方便。

### 1.2 常用注解

@RunWith(SpringJUnit4ClassRunner.class)  
@SpringApplicationConfiguration(ApplicationConfig.class)  
以上两个注解在spring1.4无法使用，不能导包时可以使用1.5及以上的如下注解：

**在测试类的类上标注如下两个注解：**

*@RunWith*(SpringRunner.class):

@RunWith就是一个运行器，

@RunWith(SpringRunner.class),让测试运行于Spring测试环境

*@SpringBootTest：*

*@AutoConfigureMockMvc:*

注入一个MockMvc实例，（该注解可以省略）

### 1.3 测试环境

spring mvc测试框架提供了两种方式，独立安装测试和集成Web环境测试（此种方式并不会集成真正的web环境，而是通过相应的Mock API进行模拟测试，无须启动服务器）。

MockMvcBuilder是用来构造MockMvc的构造器，其主要有两个实现：

StandaloneMockMvcBuilder和webAppContextSetup。

1. MockMvcBuilders.webAppContextSetup(WebApplicationContext context)：

指定WebApplicationContext，将会从该上下文获取相应的控制器并得到相应的MockMvc；

② MockMvcBuilders.standaloneSetup(Object... controllers)：

通过参数指定一组控制器，这样就不需要从上下文获取了，比如this.mockMvc= MockMvcBuilders.standaloneSetup(this.controller).build();

### 1.4 测试过程

1、准备测试环境。 2、通过MockMvc执行请求

3、添加验证断言。 4、添加结果处理器。

5、得到MvcResult进行自定义断言/进行下一步的异步请求**（**ResultMatcher用来匹配执行完请求后的结果验证，是放在断言里的，其就一个match(MvcResult result)断言方法，如果匹配失败将抛出相应的异常**）**

6、卸载测试环境。

### 1.5 独立安装测试

#### 1.5.1 测试逻辑

1、通过MockMvcBuilders.standaloneSetup模拟一个Mvc测试环境，通过build得到一个MockMvc构造器。

2、MockMvc调用perform，执行一个RequestBuilder请求，调用controller的业务处理逻辑。

3、perform返回ResultActions，返回操作结果，通过ResultActions，提供了统一的验证方式。

4、使用StatusResultMatchers对请求结果进行验证。

5、使用ContentResultMatchers对请求返回的内容进行验证。

#### 1.5.2 RequestBuilder/MockMvcRequestBuilders

RequestBuilder是用来构建请求的，从名字可以看出，RequestBuilder用来构建请求的，其提供了一个方法buildRequest(ServletContext servletContext)用于构建MockHttpServletRequest；其主要有两个子类

1.MockHttpServletRequestBuilder；

2.MockMultipartHttpServletRequestBuilder（如文件上传使用），即用来Mock客户端请求需要的所有数据。

##### 1.5.2.1 MockMvcRequestBuilders主要API

MockHttpServletRequestBuilder get(String urlTemplate, Object...urlVariables)：

根据uri模板和uri变量值得到一个GET请求方式的MockHttpServletRequestBuilder；如get(/user/{id}, 1L)；

MockHttpServletRequestBuilder post(String urlTemplate, Object... urlVariables)：

同get类似，但是是POST方法；

MockHttpServletRequestBuilder put(String urlTemplate, Object... urlVariables)：

同get类似，但是是PUT方法；

MockHttpServletRequestBuilder delete(String urlTemplate, Object... urlVariables) ：

同get类似，但是是DELETE方法；

MockHttpServletRequestBuilder options(String urlTemplate, Object... urlVariables)：

同get类似，但是是OPTIONS方法；

MockHttpServletReqestBuilder request(HttpMethod httpMethod, String urlTemplate, Object... urlVariables)：

提供自己的Http请求方法及uri模板和uri变量，如上API都是委托给这个API；

MockMultipartHttpServletRequestBuilder fileUpload(String urlTemplate, Object... urlVariables)：

提供文件上传方式的请求，得到MockMultipartHttpServletRequestBuilder；

RequestBuilder asyncDispatch(final MvcResult mvcResult)：

创建一个从启动异步处理的请求的MvcResult进行异步分派的RequestBuilder；

### 1.5.3 ResultActions

调用MockMvc.perform(RequestBuilder requestBuilder)后将得到ResultActions，对ResultActions有以下三种处理:

1. ResultActions.andExpect：添加执行完成后的断言。添加ResultMatcher验证规则，验证控制器执行完成后结果是否正确;
2. ResultActions.andDo：添加一个结果处理器，比如此处使用.andDo(MockMvcResultHandlers.print())输出整个响应结果信息，可以在调试的时候使用；
3. ResultActions.andReturn：表示执行完成后返回相应的结果。

### 1.5.4 例子

例一：

String example= "{"id":1, "name":"kqzu"}";

mockMvc.perform(post("/user")  // 路径

.contentType(MediaType.APPLICATION\_JSON)   //用contentType表示具体请求中的媒体类型信息，MediaType.APPLICATION\_JSON\_UTF8表示互联网媒体类型的json数据格式（见备注）

.content(example)

.accept(MediaType.APPLICATION\_JSON)) //accept指定客户端能够接收的内容类型

.andExpect(content().contentType("application/json;charset=UTF-8")) //验证响应contentType == application/json;charset=UTF-8

.andExpect(jsonPath("$.id").value(1)) //验证id是否为1，jsonPath的使用

.andExpect(jsonPath("$.name).value("kqzhu");  // 验证name是否等于Zhukeqian

String errorExample = "{"id":1, "name":"kqzhu"}";

MvcResult result = mockMvc.perform(post("/user")

.contentType(MediaType.APPLICATION\_JSON)

.content(errorExample)

.accept(MediaType.APPLICATION\_JSON)) //执行请求

.andExpect(status().isBadRequest())

//400错误请求，status().isOk()正确；status().isNotFound() 验证控制器不存在

.andReturn();  //返回MvcResult

例二：

public void add() throws Exception {

Map map = new HashMap();

map.put("username","tough");

map.put("password","123456");

map.put("phone","17623677587");

String jsonObject = JSONObject.toJSONString(map);

System.out.println(jsonObject);

MockHttpServletRequestBuilder requestBuilder = MockMvcRequestBuilders.post("/v1/sys/user/add")

.contentType(MediaType.APPLICATION\_JSON)

.content(jsonObject);

ResultActions result = mvc.perform(requestBuilder);

MvcResult mvcResult = result.andExpect(MockMvcResultMatchers.status().isOk())

.andDo(MockMvcResultHandlers.print())

.andReturn();// 返回执行请求的结果

System.out.println(mvcResult.getResponse().getContentAsString());

}

### 1.6 快速入门

这个方法中包含了大多数的常见情况：

@RunWith(SpringRunner.class)

@SpringBootTest

@AutoConfigureMockMvc

public class CourseControllerTest {

@Autowired

private MockMvc mockMvc;

//ObjectMapper是一个可以重复使用的对象

@Autowired

private ObjectMapper mapper;

@Test

public void courseListTest() throws Exception{

mockMvc.perform(MockMvcRequestBuilders.get("/course"))

.andExpect(MockMvcResultMatchers.status().isOk());

}

@Test

public void createTest() throws Exception{

String json = "{\"name\":\"Mock测试\",\"classHour\":\"4课时\",\"id\":\"rn:practice:Course:5af27fa5d34f435e581e5bbf\"}";

//将json格式字符串转换成Course对象里的属性值

Course course = mapper.readValue(json,Course.class);

//perform,执行一个RequestBuilders请求，会自动执行SpringMVC的流程并映射到相应的控制器执行处理

mockMvc.perform(

//构造一个post请求

MockMvcRequestBuilders.post("/course")

.contentType(MediaType.APPLICATION\_JSON\_UTF8)

//使用writeValueAsString()方法来获取对象的JSON字符串表示

.content(mapper.writeValueAsString(course)))

//andExpect，添加ResultMathcers验证规则，验证控制器执行完成后结果是否正确，【这是一个断言】

.andExpect(MockMvcResultMatchers.status().is(200))

.andExpect(MockMvcResultMatchers.content().contentType(MediaType.APPLICATION\_JSON\_UTF8))

//假定返回的结果中，"name" 值为 "Mock测试2",如果不是的话，会抛出异常java.lang.AssertionError，并给出期望值和实际值

.andExpect(MockMvcResultMatchers.jsonPath("$.name").value("Mock测试2"))

//添加ResultHandler结果处理器，比如调试时 打印结果(print方法)到控制台

.andDo(print())

//返回相应的MvcResult

.andReturn();

}

### 1.7 一些常用的测试

#### 1.7.1 测试普通控制器

mockMvc.perform(get("/user/{id}", 1)) //执行请求

.andExpect(model().attributeExists("user")) //验证存储模型数据

.andExpect(view().name("user/view")) //验证viewName

.andExpect(forwardedUrl("/WEB-INF/jsp/user/view.jsp"))//验证视图渲染时forward到的jsp（ResultMatcher forwardedUrl(final String expectedUrl)：验证处理完请求后转发的url（绝对匹配）；）

.andExpect(status().isOk())//验证状态码

.andDo(print()); //输出MvcResult到控制台

#### 1.7.2 得到MvcResult自定义验证

MvcResult result = mockMvc.perform(get("/user/{id}", 1))//执行请求

.andReturn(); //返回MvcResult

Assert.assertNotNull(result.getModelAndView().getModel().get("user")); //自定义断言

#### 1.7.3 验证请求参数绑定到模型数据及Flash属性

mockMvc.perform(post("/user").param("name", "zhang")) //执行传递参数的POST请求(也可以post("/user?name=zhang"))

.andExpect(handler().handlerType(UserController.class)) //验证执行的控制器类型

.andExpect(handler().methodName("create")) //验证执行的控制器方法名

.andExpect(model().hasNoErrors()) //验证页面没有错误

.andExpect(flash().attributeExists("success")) //验证存在flash属性

.andExpect(view().name("redirect:/user")); //验证视图

#### 1.7.4 文件上传

byte[] bytes = new byte[] {1, 2};

mockMvc.perform(fileUpload("/user/{id}/icon", 1L).file("icon", bytes)) //执行文件上传

.andExpect(model().attribute("icon", bytes)) //验证属性相等性

.andExpect(view().name("success")); //验证视图

#### 1.7.5 JSON请求/响应验证

String requestBody = "{\"id\":1, \"name\":\"zhang\"}";

mockMvc.perform(post("/user")

.contentType(MediaType.APPLICATION\_JSON).content(requestBody)

.accept(MediaType.APPLICATION\_JSON)) //执行请求

.andExpect(content().contentType(MediaType.APPLICATION\_JSON)) //验证响应contentType

.andExpect(jsonPath("$.id").value(1)); //使用Json path验证JSON 请参考http://goessner.net/articles/JsonPath/

String errorBody = "{id:1, name:zhang}";

MvcResult result = mockMvc.perform(post("/user")

.contentType(MediaType.APPLICATION\_JSON).content(errorBody)

.accept(MediaType.APPLICATION\_JSON)) //执行请求

.andExpect(status().isBadRequest()) //400错误请求

.andReturn();

Assert.assertTrue(HttpMessageNotReadableException.class.isAssignableFrom(result.getResolvedException().getClass()));//错误的请求内容体

#### 1.7.6 异步测试

//Callable

MvcResult result = mockMvc.perform(get("/user/async1?id=1&name=zhang")) //执行请求

.andExpect(request().asyncStarted())

.andExpect(request().asyncResult(CoreMatchers.instanceOf(User.class))) //默认会等10秒超时

.andReturn();

mockMvc.perform(asyncDispatch(result))

.andExpect(status().isOk())

.andExpect(content().contentType(MediaType.APPLICATION\_JSON))

.andExpect(jsonPath("$.id").value(1));

#### 1.7.7 全局配置

mockMvc = webAppContextSetup(wac)

.defaultRequest(get("/user/1").requestAttr("default", true)) //默认请求 如果其是Mergeable类型的，会自动合并的哦mockMvc.perform中的RequestBuilder

.alwaysDo(print()) //默认每次执行请求后都做的动作

.alwaysExpect(request().attribute("default", true)) //默认每次执行后进行验证的断言

.build();

mockMvc.perform(get("/user/1"))

.andExpect(model().attributeExists("user"));

### 1.8 集成web环境测试

MockMvcBuilders.webAppContextSetup(WebApplicationContext context)：指定WebApplicationContext，将会从该上下文获取相应的控制器并得到相应的MockMvc；

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration("classpath\*:spring/\*.xml")

@WebAppConfiguration

public class IncotermsRestServiceTest {

@Autowired

private WebApplicationContext wac;

private MockMvc mockMvc;

@Before

public void setup() {

this.mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).build(); //构造MockMvc

}

...

}

注意：

(1)@WebAppConfiguration：测试环境使用，用来表示测试环境使用的ApplicationContext将是WebApplicationContext类型的；value指定web应用的根；

(2)通过@Autowired WebApplicationContext wac：注入web环境的ApplicationContext容器；

(3)然后通过MockMvcBuilders.webAppContextSetup(wac).build()创建一个MockMvc进行测试；

例子：

@RunWith(JUnit4ClassRunner.class)

@WebAppConfiguration

@ContextConfiguration(locations = { "classpath\*:spring/\*.xml" })

public class TestApiTwo extends AbstractJUnit4SpringContextTests {

@Autowired

public WebApplicationContext wac;

public MockMvc mockMvc;

public MockHttpSession session;

@Before

public void before() throws Exception {

mockMvc = MockMvcBuilders.webAppContextSetup(wac).build();

}

@Test

public void testGetSequence() {

try {

MvcResult mvcResult =

mockMvc.perform(MockMvcRequestBuilders.post("/api/getSequence"))

.andExpect(MockMvcResultMatchers.status().is(200))

.andDo(MockMvcResultHandlers.print())

.andReturn();

int status = mvcResult.getResponse().getStatus();

System.out.println("请求状态码：" + status);

String result = mvcResult.getResponse().getContentAsString();

System.out.println("接口返回结果：" + result);

JSONObject resultObj = JSON.parseObject(result);

// 判断接口返回json中success字段是否为true

Assert.assertTrue(resultObj.getBooleanValue("success"));

} catch (Exception e) {

e.printStackTrace();

}

}

}