## 转载自：http://jinnianshilongnian.iteye.com/blog/2004660

## 1.1 以前的测试方式

package com.sishuok.mvc.controller;

//省略import

public class UserControllerTest {

private UserController userController;

@Before

public void setUp() {

userController = new UserController();

//安装userCtroller依赖 比如userService

}

@Test

public void testView() {

MockHttpServletRequest req = new MockHttpServletRequest();

ModelAndView mv = userController.view(1L, req);

ModelAndViewAssert.assertViewName(mv, "user/view");

ModelAndViewAssert.assertModelAttributeAvailable(mv, "user");

}

}

****准备控制器****：我们通过new方式创建一个，然后手工查找依赖注入进去（比如从spring容器获取/new的）；

****Mock Request****：此处使用Spring提供的Mock API模拟一个HttpServletRequest，其他的Servlet API也提供了相应的Mock类，具体请查看Javadoc；

****访问控制器方法****：通过直接调用控制器方法进行访问，此处无法验证Spring MVC框架的类型转换、数据验证等是否正常；

****ModelAndViewAssert****：通过这个Assert API验证我们的返回值是否正常；

对于单元测试步骤请参考：[加速Java应用开发速度3——单元/集成测试+CI](http://jinnianshilongnian.iteye.com/blog/1893135)

这种方式的缺点已经说过了，如不能走Spring MVC完整流程（不能走Servlet的过滤器链、SpringMVC的类型转换、数据验证、数据绑定、拦截器等等），如果做基本的测试没问题，这种方式就是纯粹的单元测试，我们想要的功能其实是一种集成测试，不过后续部分不区分。

## 1.2 安装测试环境

spring mvc测试框架提供了两种方式，独立安装和集成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）。

### 独立测试方式

public class UserControllerStandaloneSetupTest {

private MockMvc mockMvc;

@Before

public void setUp() {

UserController userController = new UserController();

mockMvc = MockMvcBuilders.standaloneSetup(userController).build();

}

}

1、首先自己创建相应的控制器，注入相应的依赖

2、通过MockMvcBuilders.standaloneSetup模拟一个Mvc测试环境，通过build得到一个MockMvc

3、MockMvc：是我们以后测试时经常使用的API，后边介绍

### 集成Web环境方式

//XML风格

@RunWith(SpringJUnit4ClassRunner.class)

@WebAppConfiguration(value = "src/main/webapp")

@ContextHierarchy({

@ContextConfiguration(name = "parent", locations = "classpath:spring-config.xml"),

@ContextConfiguration(name = "child", locations = "classpath:spring-mvc.xml")

})

//注解风格

//@RunWith(SpringJUnit4ClassRunner.class)

//@WebAppConfiguration(value = "src/main/webapp")

//@ContextHierarchy({

// @ContextConfiguration(name = "parent", classes = AppConfig.class),

// @ContextConfiguration(name = "child", classes = MvcConfig.class)

//})

public class UserControllerWebAppContextSetupTest {

@Autowired

private WebApplicationContext wac;

private MockMvc mockMvc;

@Before

public void setUp() {

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

}

}

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、@ContextHierarchy：指定容器层次，即spring-config.xml是父容器，而spring-mvc.xml是子容器，请参考《[第三章 DispatcherServlet详解 ——跟开涛学SpringMVC](http://jinnianshilongnian.iteye.com/blog/1602617)》

3、通过@Autowired WebApplicationContext wac：注入web环境的ApplicationContext容器；

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

## 1.3、HelloWorld

@Test

public void testView() throws Exception {

MvcResult result = mockMvc.perform(MockMvcRequestBuilders.get("/user/1"))

.andExpect(MockMvcResultMatchers.view().name("user/view"))

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

.andDo(MockMvcResultHandlers.print())

.andReturn();

Assert.assertNotNull(result.getModelAndView().getModel().get("user"));

}

1、mockMvc.perform执行一个请求；

2、MockMvcRequestBuilders.get("/user/1")构造一个请求

3、ResultActions.andExpect添加执行完成后的断言

4、ResultActions.andDo添加一个结果处理器，表示要对结果做点什么事情，比如此处使用MockMvcResultHandlers.print()输出整个响应结果信息。

5、ResultActions.andReturn表示执行完成后返回相应的结果。

### 整个测试过程非常有规律：

1、准备测试环境

2、通过MockMvc执行请求

3.1、添加验证断言

3.2、添加结果处理器

3.3、得到MvcResult进行自定义断言/进行下一步的异步请求

4、卸载测试环境

## 1.4、了解测试API

Spring mvc测试框架提供了测试MVC需要的API，主要包括Servlet/JSP Mock、MockMvcBuilder、MockMvc、RequestBuilder、ResultMatcher、ResultHandler、MvcResult等。另外提供了几个静态工厂方法便于测试：MockMvcBuilders、MockMvcRequestBuilders、MockMvcResultMatchers、MockMvcResultHandlers。在使用时请使用静态方法导入方便测试，如：

import static org.springframework.test.web.servlet.setup.MockMvcBuilders.\*;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.\*;

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.\*;

import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.\*;

### Servlet/JSP API Mock

提供了对Servlet 3 相应API的Mock，如：

MockServletContext

MockHttpServletRequest

MockHttpServletResponse

……

具体请查看spring-test模块的org.springframework.mock.web包。

### MockMvcBuilder/MockMvcBuilders

MockMvcBuilder是用来构造MockMvc的构造器，其主要有两个实现：StandaloneMockMvcBuilder和DefaultMockMvcBuilder，分别对应之前的两种测试方式。对于我们来说直接使用静态工厂MockMvcBuilders创建即可：

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；

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；

其中DefaultMockMvcBuilder还提供了如下API：

addFilters(Filter... filters)/addFilter(Filter filter, String... urlPatterns)：添加javax.servlet.Filter过滤器

defaultRequest(RequestBuilder requestBuilder)：默认的RequestBuilder，每次执行时会合并到自定义的RequestBuilder中，即提供公共请求数据的；

alwaysExpect(ResultMatcher resultMatcher)：定义全局的结果验证器，即每次执行请求时都进行验证的规则；

alwaysDo(ResultHandler resultHandler)：定义全局结果处理器，即每次请求时都进行结果处理；

dispatchOptions：DispatcherServlet是否分发OPTIONS请求方法到控制器；

StandaloneMockMvcBuilder继承了DefaultMockMvcBuilder，又提供了如下API：

setMessageConverters(HttpMessageConverter<?>...messageConverters)：设置HTTP消息转换器；

setValidator(Validator validator)：设置验证器；

setConversionService(FormattingConversionService conversionService)：设置转换服务；

addInterceptors(HandlerInterceptor... interceptors)/addMappedInterceptors(String[] pathPatterns, HandlerInterceptor... interceptors)：添加spring mvc拦截器；

setContentNegotiationManager(ContentNegotiationManager contentNegotiationManager)：设置内容协商管理器；

setAsyncRequestTimeout(long timeout)：设置异步超时时间；

setCustomArgumentResolvers(HandlerMethodArgumentResolver... argumentResolvers)：设置自定义控制器方法参数解析器；

setCustomReturnValueHandlers(HandlerMethodReturnValueHandler... handlers)：设置自定义控制器方法返回值处理器；

setHandlerExceptionResolvers(List<HandlerExceptionResolver> exceptionResolvers)/setHandlerExceptionResolvers(HandlerExceptionResolver... exceptionResolvers)：设置异常解析器；

setViewResolvers(ViewResolver...resolvers)：设置视图解析器；

setSingleView(View view)：设置单个视图，即视图解析时总是解析到这一个（仅适用于只有一个视图的情况）；

setLocaleResolver(LocaleResolver localeResolver)：设置Local解析器；

setFlashMapManager(FlashMapManager flashMapManager)：设置FlashMapManager，如存储重定向数据；

setUseSuffixPatternMatch(boolean useSuffixPatternMatch)：设置是否是后缀模式匹配，如“/user”是否匹配"/user.\*"，默认真即匹配；

setUseTrailingSlashPatternMatch(boolean useTrailingSlashPatternMatch)：设置是否自动后缀路径模式匹配，如“/user”是否匹配“/user/”，默认真即匹配；

addPlaceHolderValue(String name, String value) ：添加request mapping中的占位符替代；

因为StandaloneMockMvcBuilder不会加载Spring MVC配置文件，因此就不会注册我们需要的一些组件，因此就提供了如上API用于注册我们需要的相应组件。

### MockMvc

使用之前的MockMvcBuilder.build()得到构建好的MockMvc；这个是mvc测试的核心API，对于该API的使用方式如下：

MvcResult result = mockMvc.perform(MockMvcRequestBuilders.get("/user/1"))

.andExpect(MockMvcResultMatchers.view().name("user/view"))

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

.andDo(MockMvcResultHandlers.print())

.andReturn();

perform：执行一个RequestBuilder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；

andExpect：添加ResultMatcher验证规则，验证控制器执行完成后结果是否正确；

andDo：添加ResultHandler结果处理器，比如调试时打印结果到控制台；

andReturn：最后返回相应的MvcResult；然后进行自定义验证/进行下一步的异步处理；

另外还提供了以下API：

setDefaultRequest：设置默认的RequestBuilder，用于在每次perform执行相应的RequestBuilder时自动把该默认的RequestBuilder合并到perform的RequestBuilder中；

setGlobalResultMatchers：设置全局的预期结果验证规则，如我们通过MockMvc测试多个控制器时，假设它们都想验证某个规则时，就可以使用这个；

setGlobalResultHandlers：设置全局的ResultHandler结果处理器；

### RequestBuilder/MockMvcRequestBuilders

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

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；

MockHttpServletRequestBuilder 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；

接下来再看看MockHttpServletRequestBuilder和MockMultipartHttpServletRequestBuilder API：

MockHttpServletRequestBuilder API：

MockHttpServletRequestBuilder header(String name, Object... values)/MockHttpServletRequestBuilder headers(HttpHeaders httpHeaders)：添加头信息；

MockHttpServletRequestBuilder contentType(MediaType mediaType)：指定请求的contentType头信息；

MockHttpServletRequestBuilder accept(MediaType... mediaTypes)/MockHttpServletRequestBuilder accept(String... mediaTypes)：指定请求的Accept头信息；

MockHttpServletRequestBuilder content(byte[] content)/MockHttpServletRequestBuilder content(String content)：指定请求Body体内容；

MockHttpServletRequestBuilder cookie(Cookie... cookies)：指定请求的Cookie；

MockHttpServletRequestBuilder locale(Locale locale)：指定请求的Locale；

MockHttpServletRequestBuilder characterEncoding(String encoding)：指定请求字符编码；

MockHttpServletRequestBuilder requestAttr(String name, Object value) ：设置请求属性数据；

MockHttpServletRequestBuilder sessionAttr(String name, Object value)/MockHttpServletRequestBuilder sessionAttrs(Map<String, Object> sessionAttributes)：设置请求session属性数据；

MockHttpServletRequestBuilder flashAttr(String name, Object value)/MockHttpServletRequestBuilder flashAttrs(Map<String, Object> flashAttributes)：指定请求的flash信息，比如重定向后的属性信息；

MockHttpServletRequestBuilder session(MockHttpSession session) ：指定请求的Session；

MockHttpServletRequestBuilder principal(Principal principal) ：指定请求的Principal；

MockHttpServletRequestBuilder contextPath(String contextPath) ：指定请求的上下文路径，必须以“/”开头，且不能以“/”结尾；

MockHttpServletRequestBuilder pathInfo(String pathInfo) ：请求的路径信息，必须以“/”开头；

MockHttpServletRequestBuilder secure(boolean secure)：请求是否使用安全通道；

MockHttpServletRequestBuilder with(RequestPostProcessor postProcessor)：请求的后处理器，用于自定义一些请求处理的扩展点；

MockMultipartHttpServletRequestBuilder继承自MockHttpServletRequestBuilder，又提供了如下API：

MockMultipartHttpServletRequestBuilder file(String name, byte[] content)/MockMultipartHttpServletRequestBuilder file(MockMultipartFile file)：指定要上传的文件；

### ResultActions

调用MockMvc.perform(RequestBuilder requestBuilder)后将得到ResultActions，通过ResultActions完成如下三件事：

ResultActions andExpect(ResultMatcher matcher) ：添加验证断言来判断执行请求后的结果是否是预期的；

ResultActions andDo(ResultHandler handler) ：添加结果处理器，用于对验证成功后执行的动作，如输出下请求/结果信息用于调试；

MvcResult andReturn() ：返回验证成功后的MvcResult；用于自定义验证/下一步的异步处理；

### ResultMatcher/MockMvcResultMatchers

ResultMatcher用来匹配执行完请求后的结果验证，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；spring mvc测试框架提供了很多\*\*\*ResultMatchers来满足测试需求。注意这些\*\*\*ResultMatchers并不是ResultMatcher的子类，而是返回ResultMatcher实例的。Spring mvc测试框架为了测试方便提供了MockMvcResultMatchers静态工厂方法方便操作；具体的API如下：

HandlerResultMatchers handler()：请求的Handler验证器，比如验证处理器类型/方法名；此处的Handler其实就是处理请求的控制器；

RequestResultMatchers request()：得到RequestResultMatchers验证器；

ModelResultMatchers model()：得到模型验证器；

ViewResultMatchers view()：得到视图验证器；

FlashAttributeResultMatchers flash()：得到Flash属性验证；

StatusResultMatchers status()：得到响应状态验证器；

HeaderResultMatchers header()：得到响应Header验证器；

CookieResultMatchers cookie()：得到响应Cookie验证器；

ContentResultMatchers content()：得到响应内容验证器；

JsonPathResultMatchers jsonPath(String expression, Object ... args)/ResultMatcher jsonPath(String expression, Matcher<T> matcher)：得到Json表达式验证器；

XpathResultMatchers xpath(String expression, Object... args)/XpathResultMatchers xpath(String expression, Map<String, String> namespaces, Object... args)：得到Xpath表达式验证器；

ResultMatcher forwardedUrl(final String expectedUrl)：验证处理完请求后转发的url（绝对匹配）；

ResultMatcher forwardedUrlPattern(final String urlPattern)：验证处理完请求后转发的url（Ant风格模式匹配，@since spring4）；

ResultMatcher redirectedUrl(final String expectedUrl)：验证处理完请求后重定向的url（绝对匹配）；

ResultMatcher redirectedUrlPattern(final String expectedUrl)：验证处理完请求后重定向的url（Ant风格模式匹配，@since spring4）；

得到相应的\*\*\*ResultMatchers后，接着再调用其相应的API得到ResultMatcher，如ModelResultMatchers.attributeExists(final String... names)判断Model属性是否存在。具体请查看相应的API。再次就不一一列举了。

****ResultHandler/MockMvcResultHandlers****

ResultHandler用于对处理的结果进行相应处理的，比如输出整个请求/响应等信息方便调试，Spring mvc测试框架提供了MockMvcResultHandlers静态工厂方法，该工厂提供了ResultHandler print()返回一个输出MvcResult详细信息到控制台的ResultHandler实现。

****MvcResult****

即执行完控制器后得到的整个结果，并不仅仅是返回值，其包含了测试时需要的所有信息，如：

MockHttpServletRequest getRequest()：得到执行的请求；

MockHttpServletResponse getResponse()：得到执行后的响应；

Object getHandler()：得到执行的处理器，一般就是控制器；

HandlerInterceptor[] getInterceptors()：得到对处理器进行拦截的拦截器；

ModelAndView getModelAndView()：得到执行后的ModelAndView；

Exception getResolvedException()：得到HandlerExceptionResolver解析后的异常；

FlashMap getFlashMap()：得到FlashMap；

Object getAsyncResult()/Object getAsyncResult(long timeout)：得到异步执行的结果；

## 1.5 测试示例

### 测试普通控制器

//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

.andExpect(status().isOk())//验证状态码

.andDo(print()); //输出MvcResult到控制台

### 测试普通控制器，但是URL错误，即404

//找不到控制器，404测试

MvcResult result = mockMvc.perform(get("/user2/{id}", 1)) //执行请求

.andDo(print())

.andExpect(status().isNotFound()) //验证控制器不存在

.andReturn();

Assert.assertNull(result.getModelAndView()); //自定义断言

### 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

### 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

### 验证请求参数验证失败出错

mockMvc.perform(post("/user").param("name", "admin")) //执行请求

.andExpect(model().hasErrors()) //验证模型有错误

.andExpect(model().attributeDoesNotExist("name")) //验证存在错误的属性

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

### 文件上传

//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

### JSON请求/响应验证

测试时需要安装jackson Json和JsonPath依赖：

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>${jackson2.version}</version>

</dependency>

<dependency>

<groupId>com.jayway.jsonpath</groupId>

<artifactId>json-path</artifactId>

<version>${jsonpath.version}</version>

<scope>test</scope>

</dependency>

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

### XML请求/响应验证

测试时需要安装spring oxm和xstream依赖：

<dependency>

<groupId>com.thoughtworks.xstream</groupId>

<artifactId>xstream</artifactId>

<version>${xsream.version}</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-oxm</artifactId>

<version>${spring.version}</version>

<scope>test</scope>

</dependency>

//XML请求/响应

String requestBody = "<user><id>1</id><name>zhang</name></user>";

mockMvc.perform(post("/user")

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

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

.andDo(print())

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

.andExpect(xpath("/user/id/text()").string("1")); //使用XPath表达式验证XML 请参考http://www.w3school.com.cn/xpath/

String errorBody = "<user><id>1</id><name>zhang</name>";

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

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

.accept(MediaType.APPLICATION\_XML)) //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

### 异常处理

//异常处理

MvcResult result = mockMvc.perform(get("/user/exception")) //执行请求

.andExpect(status().isInternalServerError()) //验证服务器内部错误

.andReturn();

Assert.assertTrue(IllegalArgumentException.class.isAssignableFrom(result.getResolvedException().getClass()));

//静态资源

mockMvc.perform(get("/static/app.js")) //执行请求

.andExpect(status().isOk()) //验证状态码200

.andExpect(content().string(CoreMatchers.containsString("var")));//验证渲染后的视图内容包含var

mockMvc.perform(get("/static/app1.js")) //执行请求

.andExpect(status().isNotFound()); //验证状态码404

### 静态资源

//静态资源

mockMvc.perform(get("/static/app.js")) //执行请求

.andExpect(status().isOk()) //验证状态码200

.andExpect(content().string(CoreMatchers.containsString("var")));//验证渲染后的视图内容包含var

mockMvc.perform(get("/static/app1.js")) //执行请求

.andExpect(status().isNotFound()); //验证状态码404

### ****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));

//DeferredResult

result = mockMvc.perform(get("/user/async2?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));

此处请在第一次请求时加上 andExpect(request().asyncResult(CoreMatchers.instanceOf(User.class)))这样会等待结果返回/超时，无须自己设置线程等待了；此处注意request().asyncResult一定是在第一次请求发出；然后第二次通过asyncDispatch进行异步请求。

### 添加自定义过滤器

mockMvc = webAppContextSetup(wac).addFilter(new MyFilter(), "/\*").build();

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

.andExpect(request().attribute("filter", true));

### 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"));

****只要记住测试步骤，按照步骤操作，整个测试过程是非常容易理解的：****

1、准备测试环境

2、通过MockMvc执行请求

3.1、添加验证断言

3.2、添加结果处理器

3.3、得到MvcResult进行自定义断言/进行下一步的异步请求

4、卸载测试环境