<https://segmentfault.com/a/1190000004343063?_ea=575820>

## 主要内容

* 将web请求映射到Spring控制器
* 绑定form参数
* 验证表单提交的参数

写在前面：关于Java Web，首先推荐一篇文章——[写给java web一年左右工作经验的人](http://my.oschina.net/aaron74/blog/282304)，这篇文章的作者用精练的话语勾勒除了各种Java框架的缘由和最基本的原理。我们在学习Spring的过程中也要切记，不仅要知道怎么做？还要深究背后的思考和权衡。

对于很多Java程序员来说，他们的主要工作就是开发Web应用，如果你也在做这样的工作，那么你一定会了解到构建这类系统所面临的挑战，例如状态管理、工作流和参数验证等。HTTP协议的无状态性使得这些任务极具挑战性。

Spring的web框架用于解决上述提到的问题，基于Model-View-Controller（MVC）模型，Spring MVC可以帮助开发人员构建灵活易扩展的Web  
应用。

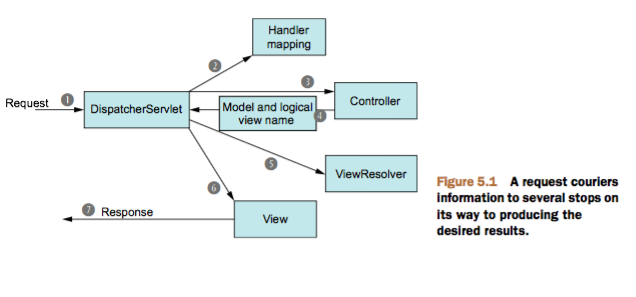
这一章将涉及Spring MVC框架的主要知识，由于基于注解开发是目前Spring社区的潮流，因此我们将侧重介绍如何使用注解创建控制器，进而处理各类web请求和表单提交。在深入介绍各个专题之前，首先从一个比较高的层面观察和理解下Spring MVC的工作原理。

## 5.1 Spring MVC入门

### 5.1.1 request的处理过程

用户每次点击浏览器界面的一个按钮，都发出一个web请求（request）。一个web请求的工作就像一个快递员，负责将信息从一个地方运送到另一个地方。

从web请求离开浏览器（**1**）到返回响应，中间经历了几个节点，在每个节点都进行一些操作用于交换信息。下图展示了Spring MVC应用中web请求会遇到的几个节点。



请求旅行的第一站是Spring的DispatcherServlet，和大多数Javaweb应用相同，Spring MVC通过一个单独的前端控制器过滤分发请求。当Web应用委托一个servlet将请求分发给应用的其他组件时，这个servlert称为前端控制器（front controller）。在Spring MVC中，DispatcherServlet就是前端控制器。

DispatcherServlet的任务是将请求发送给某个Spring控制器。控制器（controller）是Spring应用中处理请求的组件。一般在一个应用中会有多个控制器，DispatcherServlet来决定把请求发给哪个控制器处理。DispatcherServlet会维护一个或者多个处理器映射（**2**），用于指出request的下一站——根据请求携带的URL做决定。

一旦选好了控制器，DispatcherServlet会把请求发送给指定的控制器（**3**），控制器中的处理方法负责从请求中取得用户提交的信息，然后委托给对应的业务逻辑组件（service objects）处理。

控制器的处理结果包含一些需要传回给用户或者显示在浏览器中的信息。这些信息存放在模型（model）中，但是直接把原始信息返回给用户非常低效——最好格式化成用户友好的格式，例如HTML或者JSON格式。为了生成HTML格式的文件，需要把这些信息传给指定的视图（view），一般而言是JSP。

控制器的最后一个任务就是将数据打包在模型中，然后指定一个视图的逻辑名称（由该视图名称解析HTML格式的输出），然后将请求和模型、视图名称一起发送回DispatcherServlet（**4**）。

注意，控制器并不负责指定具体的视图，返回给DispatcherServlet的视图名称也不会指定具体的JSP页面（或者其他类型的页面）；控制器返回的仅仅是视图的逻辑名称，DispatcherServlet用这个名称查找对应的视图解析器（**5**），负责将逻辑名称转换成对应的页面实现，可能是JSP也可能不是。

现在DispatcherServlet就已经知道将由哪个视图渲染结果，至此一个请求的处理就基本完成了。最后一步就是视图的实现（**6**），最经典的是JSP。视图会使用模型数据填充到视图实现中，然后将结果放在HTTP响应对象中（**7**）。

### 5.1.2 设置Spring MVC

如上一小节的图展示的，看起来需要填写很多配置信息。幸运地是，Spring的最新版本提供了很多容易配置的选项，降低了Spring MVC的学习门槛。这里我们先简单配置一个Spring MVC应用，作为这一章将会不断完善的例子。

#### CONFIGURING DISPATCHERSERVLET

DispatcherServlet是Spring MVC的核心，每当应用接受一个HTTP请求，由DispatcherServlet负责将请求分发给应用的其他组件。

在旧版本中，DispatcherServlet之类的servlet一般在web.xml文件中配置，该文件一般会打包进最后的war包种；但是Spring 3引入了注解，我们在这一章将展示如何基于注解配置Spring MVC。

既然不适用web.xml文件，你需要在servlet容器中使用Java配置DispatcherServlet，具体的代码列举如下：

**package** org.test.spittr.config;

**import** org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

**public** **class** SpittrWebAppInitializer

**extends** AbstractAnnotationConfigDispatcherServletInitializer {

@Override

**protected** **Class**<?>[] getRootConfigClasses() { //根容器

**return** **new** **Class**<?>[] { RootConfig.**class** };

}

@Override

**protected** **Class**<?>[] getServletConfigClasses() { //Spring mvc容器

**return** **new** **Class**<?>[] { WebConfig.**class** };

}

@Override

**protected** String[] getServletMappings() { //DispatcherServlet映射,从"/"开始

**return** **new** String[] { "/" };

}

}

spitter这个单词是我们应用的名称，SpittrWebAppInitializer类是整个应用的总配置类。

AbstractAnnotationConfigDispatcherServletInitializer这个类负责配置DispatcherServlet、初始化Spring MVC容器和Spring容器。getRootConfigClasses()方法用于获取Spring应用容器的配置文件，这里我们给定预先定义的RootConfig.class；getServletConfigClasses负责获取Spring MVC应用容器，这里传入预先定义好的WebConfig.class；getServletMappings()方法负责指定需要由DispatcherServlet映射的路径，这里给定的是"/"，意思是由DispatcherServlet处理所有向该应用发起的请求。

#### A TALE OF TWO APPLICATION CONTEXT

当DispatcherServlet启动时，会创建一个Spring MVC应用容器并开始加载配置文件中定义好的beans。通过getServletConfigClasses()方法，可以获取由DispatcherServlet加载的定义在WebConfig.class中的beans。

在Spring Web应用中，还有另一个Spring应用容器，这个容器由ContextLoaderListener创建。

我们希望DispatcherServlet仅加载web组件之类的beans，例如controllers（控制器）、view resolvers（视图解析器）和处理器映射（handler mappings）；而希望ContextLoaderListener加载应用中的其他类型的beans——例如业务逻辑组件、数据库操作组件等等。

实际上，AbstractAnnotationConfigDispatcherServletInitializer创建了DispatcherServlet和ContextLoaderListener：getServletConfigClasses()返回的配置类定义了Spring MVC应用容器中的beans；getRootConfigClasses()返回的配置类定义了Spring应用根容器中的beans。**【书中没有说的】：**Spring MVC容器是根容器的子容器，子容器可以看到根容器中定义的beans，反之不行。

注意：通过AbstractAnnotationConfigDispatcherServletInitializer配置DispatcherServlet仅仅是传统的web.xml文件方式的另一个可选项。尽管你也可以使用AbstractAnnotationConfigDispatcherServletInitializer的一个子类引入web.xml文件来配置，但这没有必要。

这种方式配置DispatcherServlet需要支持Servlert 3.0的容器，例如Apache Tomcat 7或者更高版本的。

#### ENABLING SPRING MVC

正如可以通过多种方式配置DispatcherServlet一样，也可以通过多种方式启动Spring MVC特性。原来我们一般在xml文件中使用<mvc:annotation-driven>元素启动注解驱动的Spring MVC特性。

这里我们仍然使用JavaConfig配置，最简单的Spring MVC配置类代码如下：

**package** **org**.test.spittr.config;

**import** **org**.springframework.context.annotation.Configuration;

**import** **org**.springframework.web.servlet.config.annotation.EnableWebMvc;

@**Configuration**

@EnableWebMvc

public class WebConfig {

}

@Configuration表示这是Java配置类；@EnableWebMvc注解用于启动Spring MVC特性。

仅仅这些代码就可以启动Spring MVC了，虽然它换缺了一些必要的组件：

* 没有配置视图解析器。这种情况下，Spring会使用BeanNameViewResolver，这个视图解析器通过查找ID与逻辑视图名称匹配且实现了View接口的beans。
* 没有启动Component-scanning。
* DispatcherServlet作为默认的servlet，将负责处理所有的请求，包括对静态资源的请求，例如图片和CSS文件等。

因此，我们还需要在配置文件中增加一些配置，使得这个应用可以完成最简单的功能，代码如下：

**package** org.test.spittr.config;

**import** org.springframework.context.annotation.**Bean**;

**import** org.springframework.context.annotation.**ComponentScan**;

**import** org.springframework.context.annotation.**Configuration**;

**import** org.springframework.web.servlet.**ViewResolver**;

**import** org.springframework.web.servlet.config.annotation.**DefaultServletHandlerConfigurer**;

**import** org.springframework.web.servlet.config.annotation.**EnableWebMvc**;

**import** org.springframework.web.servlet.config.annotation.**WebMvcConfigurerAdapter**;

**import** org.springframework.web.servlet.view.**InternalResourceViewResolver**;

**@Configuration**

**@EnableWebMvc**

**@ComponentScan**("org.test.spittr.web")

public **class** **WebConfig** **extends** **WebMvcConfigurerAdapter**{

**@Bean**

public **ViewResolver** viewResolver() { //配置JSP视图解析器

**InternalResourceViewResolver** resolver = **new** **InternalResourceViewResolver**();

resolver.setPrefix("/WEB-INF/views/");

resolver.setSuffix(".jsp");

//可以在JSP页面中通过${}访问beans

resolver.setExposeContextBeansAsAttributes(true);

**return** resolver;

}

**@Override**

public void configureDefaultServletHandling(**DefaultServletHandlerConfigurer** configurer) {

configurer.enable(); //配置静态文件处理

}

}

首先，通过@ComponentScan("org.test.spittr.web")注解指定bean的自动发现机制作用的范围，待会会看到，被@Controller等注解修饰的web的bean将被发现并加载到spring mvc应用容器。这样就不需要在配置类中显式定义任何控制器bean了。

然后，你通过@Bean注解添加一个ViewResolverbean，具体来说是InternalResourceViewResolver。后面我们会专门探讨视图解析器，这里的三个函数的含义依次是：setPrefix()方法用于设置视图路径的前缀；setSuffix()用于设置视图路径的后缀，即如果给定一个逻辑视图名称——"home"，则会被解析成"/WEB-INF/views/home.jsp"； setExposeContextBeansAsAttributes(true)使得可以在JSP页面中通过${ }访问容器中的bean。

最后，WebConfig继承了WebMvcConfigurerAdapter类，然后覆盖了其提供的configureDefaultServletHandling()方法，通过调用configer.enable()，DispatcherServlet将会把针对静态资源的请求转交给servlert容器的default servlet处理。

RootConfig的配置就非常简单了，唯一需要注意的是，它在设置扫描机制的时候，将之前WebConfig设置过的那个包排除了；也就是说，这两个扫描机制作用的范围正交。RootConfig的代码如下：

**package** org.test.spittr.config;

**import** org.springframework.context.**annotation**.ComponentScan;

**import** org.springframework.context.**annotation**.Configuration;

**import** org.springframework.context.**annotation**.FilterType;

**import** org.springframework.web.servlet.config.**annotation**.EnableWebMvc;

**@Configuration**

**@ComponentScan(basePackages = {"org.test.spittr"},**

**excludeFilters = {@ComponentScan.Filter(type = FilterType.ANNOTATION, value = EnableWebMvc.class)**})

**public** **class** **RootConfig** {

}

### 5.1.3 Spittr应用简介

这一章要用的例子应用，从Twitter获取了一些灵感，因此最开始叫Spitter；然后又借鉴了最近比较流行的网站Flickr，因此我们也把e去掉，最终形成Spittr这个名字。这也有利于区分领域名称（类似于twitter，这里用spring实现，因此叫spitter）和应用名称。

Spittr应用有两个关键的领域概念：spitters（应用的用户）和spittles（用户发布的状态更新）。在这一章中，将专注于构建该应用的web层，创建控制器和显示spittles，以及处理用户注册的表单。

基础已经打好了，你已经配置好了DispatcherServlet，启动了Spring MVC特性等，接下来看看如何编写Spring MVC控制器。

## 5.2 编写简单的控制器

在Spring MVC应用中，控制器类就是含有被@RequestMapping注解修饰的方法的类，其中该注解用于指出这些方法要处理的请求类型。

我们从最简单的请求"/"开始，用于渲染该应用的主页，HomeController的代码列举如下：

**package** **org**.test.spittr.web;

**import** **org**.springframework.stereotype.Controller;

**import** **org**.springframework.web.bind.annotation.RequestMapping;

**import** **org**.springframework.web.bind.annotation.RequestMethod;

@**Controller**

public class HomeController {

@**RequestMapping**(**value** = "/", method = RequestMethod.GET)

public String home() {

**return** "**home**";

}

}

@Controller是一个模式化的注解，它的作用跟@Component一样；Component-scanning机制会自动发现该控制器，并在Spring容器中创建对应的bean。

HomeController中的home()方法用于处理[*http://localhost*](http://localhost/):8080/这个URL对应的"/"请求，且仅处理GET方法，方法的内容是返回一个逻辑名称为"home"的视图。DispatcherServlet将会让视图解析器通过这个逻辑名称解析出真正的视图。

根据之前配置的InternalResourceViewResolver，最后解析成/WEB-INF/views/home.jsp，home.jsp的内容列举如下：

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<%@ page contentType="text/html;charset=UTF-8" language="java" session="false" %>

<html>

<head>

<title>Spittr</title></head>

<body>

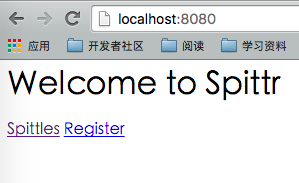
<h1>Welcome to Spittr</h1>

<a href="<c:url value="/spittles" /> ">Spittles</a>

<a href="<c:url value="/spitter/register"/> ">Register</a>

</body>

</html>

启动应用，然后访问[*http://localhost*](http://localhost/):8080/，Spittr应用的主页如下图所示：  


### 5.2.1 控制器测试

控制器的测试通过Mockito框架进行，首先在pom文件中引入需要的依赖库：

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

</dependency>

<!-- test support -->

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-all</artifactId>

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

</dependency><dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

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

</dependency>

然后，对应的单元测试用例HomeControllerTest的代码如下所示：

**package** org.test.spittr.web;

**import** org.junit.Before;**import** org.junit.Test;

**import** org.springframework.test.web.servlet.MockMvc;

**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.setup.MockMvcBuilders.\*;

**public** **class** **HomeControllerTest** {

MockMvc mockMvc;

**@Before**

**public** **void** **setupMock**() {

HomeController controller = **new** HomeController();

mockMvc = standaloneSetup(controller).build();

}

**@Test**

**public** **void** **testHomePage**() **throws** Exception {

mockMvc.perform(get("/"))

.andExpect(view().name("home"));

}

}

首先stanaloneSetup()方法通过HomeController的实例模拟出一个web服务，然后使用perform执行对应的GET请求，并检查返回的视图的名称。MockMvcBuilders类有两个静态接口，代表两种模拟web服务的方式：独立测试和集成测试。上面这段代码是独立测试，我们也尝试了集成测试的方式，最终代码如下：

package org.test.spittr.web;

**import** org.junit.Before;**import** org.junit.Test;

**import** org.junit.runner.RunWith;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.test.context.ContextConfiguration;

**import** org.springframework.test.context.ContextHierarchy;

**import** org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

**import** org.springframework.test.context.web.WebAppConfiguration;

**import** org.springframework.test.web.servlet.MockMvc;

**import** org.springframework.web.context.WebApplicationContext;

**import** org.test.spittr.config.RootConfig;

**import** org.test.spittr.config.WebConfig;

**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.setup.MockMvcBuilders.\*;

@**RunWith**(**SpringJUnit4ClassRunner**.**class**)

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

@**ContextHierarchy**({

@**ContextConfiguration**(name = "parent", classes = **RootConfig**.**class**),

@**ContextConfiguration**(name = "child", classes = **WebConfig**.**class**)})

**public** **class** **HomeControllerTest** {

@**Autowired**

**private** **WebApplicationContext** context;

**MockMvc** mockMvc;

@**Before**

**public** void setupMock() {

//HomeController controller = new HomeController();

//mockMvc = standaloneSetup(controller).build();

mockMvc = webAppContextSetup(context).build();

}

@**Test**

**public** void testHomePage() **throws** **Exception** {

mockMvc.perform(**get**("/"))

.andExpect(view().name("home"));

}

}

### 5.2.2 定义类级别的请求处理

上面一节对之前的HomeController进行了简单的测试，现在可以对它进行进一步的完善：将@RequestMapping从修饰函数改成修饰类，代码如下：

**package** **org**.test.spittr.web;

**import** **org**.springframework.stereotype.Controller;

**import** **org**.springframework.web.bind.annotation.RequestMapping;

**import** **org**.springframework.web.bind.annotation.RequestMethod;

@**Controller**

@RequestMapping(value = "/")

public class HomeController {

@**RequestMapping**(**method** = RequestMethod.GET)

public String home() {

**return** "**home**";

}

}

在新的HomeController中，"/"被移动到类级别的@RequestMapping中，而定义HTTP方法的@RequestMapping仍然用于修饰home()方法。RequestMapping注解可以接受字符串数组，即可以同时映射多个路径，因此我们还可以按照下面这种方式修改：

@Controller

@RequestMapping({"/", "/homepage"})

public class HomeController {

}

}

### 5.2.3 给视图传入模型数据

对于DispatcherServlet传来的请求，控制器通常不会实现具体的业务逻辑，而是调用业务层的接口，并且将业务层服务返回的数据放在模型对象中返回给DispatcherServlet。

在Spittr应用中，需要一个页面显示最近的spittles列表。首先需要定义数据库存取接口，这里不需要提供具体实现，只需要用Mokito框架填充模拟测试数据即可。SpittleRepository接口的代码列举如下：

**package** org.test.spittr.data;

**import** java.util.List;

**public** **interface** **SpittleRepository** {

List<Spittle> **findSpittles**(**long** max, **int** count);

}

SpittleRepository接口中的findSpittles()方法有两个参数：max表示要返回的Spittle对象的最大ID；count表示指定需要返回的Spittle对象数量。为了返回20个最近发表的Spittle对象，则使用List<Spittle> recent = spittleRepository.findSpittle(Long.MAX\_VALUE, 20)这行代码即可。该接口要处理的实体对象是Spittle，因此还需要定义对应的实体类，代码如下：

**package** org.test.spittr.**data**;

**import** org.apache.commons.lang.builder.EqualsBuilder;

**import** org.apache.commons.lang.builder.HashCodeBuilder;

**import** java.util.Date;

**public** **class** **Spittle** {

**private** **final** Long id;

**private** **final** String message;

**private** **final** Date time;

**private** Double latitude;

**private** Double longitude;

**public** Spittle(String message, Date time) {

**this**(message, time, null, null);

}

**public** Spittle(String message,Date time, Double latitude, Double longitude) {

**this**.id = null;

**this**.time = time;

**this**.latitude = latitude;

**this**.longitude = longitude;

**this**.message = message;

}

**public** Long getId() {

**return** id;

}

**public** String getMessage() {

**return** message;

}

**public** Date getTime() {

**return** time;

}

**public** Double getLongitude() {

**return** longitude;

}

**public** Double getLatitude() {

**return** latitude;

}

**@Override**

**public** boolean equals(Object obj) {

**return** EqualsBuilder.reflectionEquals(**this**, obj,

new String[]{"message","latitude", "longitude"});

}

**@Override**

**public** int hashCode() {

**return** HashCodeBuilder.reflectionHashCode(**this**,

new String[]{"message", "latitude", "longitude"});

}

}

Spittle对象还是POJO，并没什么复杂的。唯一需要注意的就是，利用Apache Commons Lang库的接口，用于简化equals和hashCode方法的实现。参考[Apache Commons EqualsBuilder and HashCodeBuilder](http://www.codeproject.com/Articles/143431/Apache-Commons-EqualsBuilder-and-HashCodeBuilder)

首先为新的控制器接口写一个测试用例，利用Mockito框架模拟repository对象，并模拟出request请求，代码如下：

package org.test.spittr.web;

**import** org.junit.Test;

**import** org.springframework.test.web.servlet.MockMvc;

**import** org.springframework.web.servlet.view.InternalResourceView;

**import** org.test.spittr.data.Spittle;**import** org.test.spittr.data.SpittleRepository;**import** java.util.ArrayList;

**import** java.util.Date;**import** java.util.List;

**import** **static** org.hamcrest.Matchers.hasItems;

**import** **static** org.mockito.Mockito.\*;

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

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

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

public **class** **SpittleControllerTest** {

**@Test**

public **void** shouldShowRecentSpittles() throws Exception {

//step1 准备测试数据

List<Spittle> expectedSpittles = createSpittleList(20);

SpittleRepository mockRepository = mock(SpittleRepository.**class**);

when(mockRepository.findSpittles(Long.MAX\_VALUE, 20))

.thenReturn(expectedSpittles);

SpittleController controller = **new** SpittleController(mockRepository);

MockMvc mockMvc = standaloneSetup(controller)

.setSingleView(**new** InternalResourceView("/WEB-INF/views/spittles.jsp"))

.build();

//step2 and step3

mockMvc.perform(**get**("/spittles"))

.andExpect(view().name("spittles"))

.andExpect(model().attributeExists("spittleList"))

.andExpect(model().attribute("spittleList",

hasItems(expectedSpittles.toArray())));

}

private List<Spittle> createSpittleList(int count) {

List<Spittle> spittles = **new** ArrayList<Spittle>();

**for** (int i = 0; i < count; i++) {

spittles.add(**new** Spittle("Spittle " + i, **new** Date()));

}

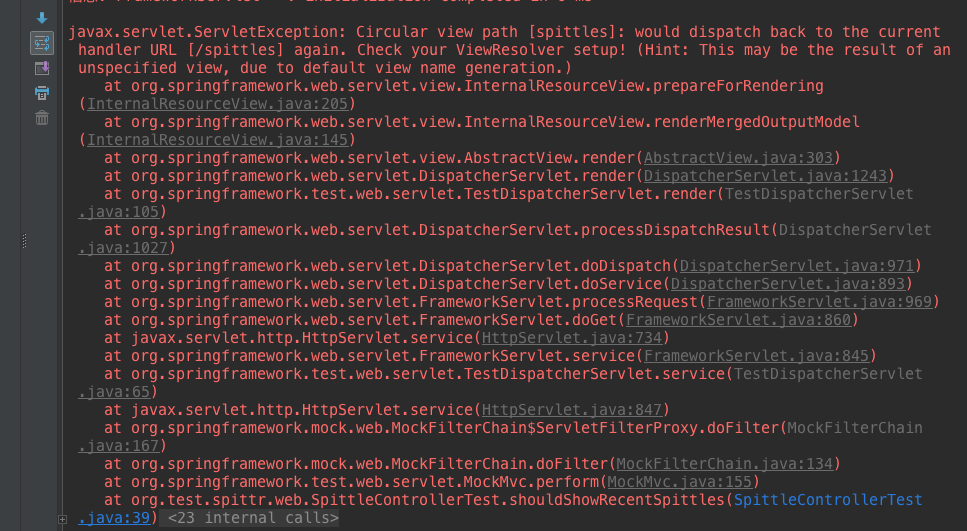
**return** spittles;

}

}

单元测试的基本组成是：准备测试数据、调用待测试接口、校验接口的执行结果。对于shouldShowRecentSpittles()这个用例我们也可以这么分割：首先规定在调用SpittleRepository接口的findSpittles()方法时将返回20个Spittle对象。

这里选择独立测试，跟HomeControllerTest不同的地方在于，这里构建MockMvc对象时还调用了setSingleView()函数，这是为了防止mock框架从控制器解析view名字。在很多情况下并没有这个必要，但是对于SpittleController控制器来说，视图名称和路径名称相同，如果使用默认的视图解析器，则MockMvc会混淆这两者而失败，报出如下图所示的错误：



在这里其实可以随意设置InternalResourceView的路径，但是为了和WebConfig中的配置相同。

通过get方法构造GET请求，访问"/spittles"，并确保返回的视图名称是"spittles"，返回的model数据中包含spittleList属性，且对应的值为我们之前创建的测试数据。

最后，为了使用hasItems，需要在pom文件中引入hamcrest库，代码如下

<dependency>

<groupId>org.hamcrest</groupId>

<artifactId>hamcrest-library</artifactId>

<version>1.3</version>

</dependency>

现在跑单元测试的话，必然会失败，因为我们还没有提供SpittleController的对应方法，代码如下：

**package** org.test.spittr.web;

**import** org.springframework.beans.factory.**annotation**.Autowired;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.ui.Model;

**import** org.springframework.web.bind.**annotation**.RequestMapping;

**import** org.springframework.web.bind.**annotation**.RequestMethod;

**import** org.test.spittr.**data**.SpittleRepository;

**@Controller**

**@RequestMapping("/spittles")**

**public** **class** **SpittleController** {

**private** SpittleRepository spittleRepository;

**@Autowired**

SpittleController(SpittleRepository spittleRepository) {

**this**.spittleRepository = spittleRepository;

}

**@RequestMapping(method = RequestMethod.GET)**

**public** String spittles(Model model) {

model.addAttribute(

spittleRepository.findSpittles(Long.MAX\_VALUE, 20));

**return** "spittles";

}

}

Model对象本质上是一个Map，spittles方法负责填充数据，然后跟视图的逻辑名称一起回传给DispatcherServlet。在调用addAttribute方法的时候，如果不指定key字段，则key字段会从value的类型推导出，在这个例子中默认的key字段是spittleList。

如果你希望显式指定key字段，则可以按照如下方式指定：

@RequestMapping(**method** = **RequestMethod**.**GET**)

**public** **String** **spittles**(Model model) {

model.addAttribute("spittleList",

spittleRepository.findSpittles(Long.MAX\_VALUE, 20));

return "spittles";

}

另外，如果你希望尽量少使用Spring规定的数据类型，则可以使用Map代替Model。

还有另一种spittles方法的实现，如下所示：

@RequestMapping(**method** = **RequestMethod**.**GET**)

**public** **List**<**Spittle**> **spittles**() {

return spittleRepository.findSpittles(Long.MAX\_VALUE, 20));

}

这个版本和之前的不同，并没有返回一个逻辑名称以及显式设置Model对象，这个方法直接返回Spittle列表。在这种情况下，Spring会将返回值直接放入Model对象，并从值类型推导出对应的关键字key；然后从路径推导出视图逻辑名称，在这里是spittles。

无论你选择那种实现，最终都需要一个页面——spittles.jsp。JSP页面使用JSTL库的<c:forEach>标签获取model对象中的数据，如下所示：

<c:forEach items="${spittleList}" var="spittle" >

<li id="spittle\_<c:out value="spittle.id"/>" >

<div class="spittleMessage">

<c:out value="${spittle.message}" />

</div>

<div>

<span class="spittleTime"><c:out value="${spittle.time}" /></span>

<span class="spittleLocation">

(<c:out value="${spittle.latitude}" />,

<c:out value="${spittle.longitude}" />)</span>

</div>

</li>

</c:forEach>

尽管SpittleController还是很简单，但是它比HomeController复杂了一点，不过，这两个控制器都没有实现的一个功能是处理表单输入。接下来将扩展SpittleController，使其能够处理表单上输入。

## 5.3 访问request输入

Spring MVC提供了三种方式，可以让客户端给控制器的handler传入参数，包括：

* 查询参数（Query parameters）
* 表单参数（Form parameters）
* 路径参数（Path parameters）

### 5.3.1 获取查询参数

Spittr应用需要一个页面显示spittles列表，目前的SpittleController仅能返回最近的所有spittles，还不能提供根据spittles的生成历史进行查询。如果你想提供这个功能，首先要提供用户一个传入参数的方法，从而可以决定返回历史spittles的那一个子集。

spittles列表是按照ID的生成先后倒序排序的：下一页spittles的第一条spittle的ID应正好在当前页的最后一条spittle的ID后面。因此，为了显示下一页spttles，应该能够传入仅仅小于当前页最后一条spittleID的参数；并且提供设置每页返回几个spittles的参数count。

* before参数，代表某个Spittle的ID，包含该ID的spittles集合中所有的spittles都在当前页的spittles之前发布；
* count参数，代表希望返回结果中包含多少条spittles。

我们将改造5.2.3小节实现的spittles()方法，来处理上述两个参数。首先编写测试用例：

@**Test**

public void shouldShowRecentSpittles\_NORMAL() throws **Exception** {

**List**<**Spittle**> expectedSpittles = createSpittleList(50);

**SpittleRepository** mockRepository = mock(**SpittleRepository**.**class**);

when(mockRepository.findSpittles(238900, 50))

.thenReturn(expectedSpittles);

**SpittleController** controller = **new** **SpittleController**(mockRepository);

**MockMvc** **mockMvc** = **standaloneSetup**(controller)

.**setSingleView**(new **InternalResourceView**("/**WEB**-**INF**/views/spittles.jsp"))

.**build**();

**mockMvc**.**perform**(get("/spittles?max=238900&count=50"))

.**andExpect**(view().**name**("spittles"))

.**andExpect**(model().**attributeExists**("spittleList"))

.**andExpect**(model().**attribute**("spittleList",

hasItems(expectedSpittles.toArray())));

}

这个测试用例的关键在于：为请求"/spittles"传入两个参数，max和count。这个测试用例可以测试提供参数的情况，两个测试用例都应该提供，这样可以覆盖到所有测试条件。改造后的spittles方法列举如下：

@RequestMapping(method = RequestMethod.GET)

public List<Spittle> spittles(

@RequestParam("max") long max,

@RequestParam("count") int count) {

**return** **spittleRepository**.findSpittles(max, count);

}

如果SpittleController的handle方法需要默认处理同时处理两种情况：提供了max和count参数，或者没有提供的情况，代码如下：

@RequestMapping(method = RequestMethod.GET)

public List<Spittle> spittles(

@RequestParam(value = "max", defaultValue = MAX\_LONG\_AS\_STRING) long max,

@RequestParam(value = "count", defaultValue = "20") int count) {

**return** **spittleRepository**.findSpittles(max, count);

}

其中MAX\_LONG\_AS\_STRING是Long的最大值的字符串形式，定义为：private static final String MAX\_LONG\_AS\_STRING = Long.MAX\_VALUE + "";，默认值都给定字符串形式，不过一旦需要绑定到参数上时，就会自动转为合适的格式。

### 5.3.2 通过路径参数获取输入

假设Spittr应用应该支持通过指定ID显示对应的Spittle，可以使用@RequestParam给控制器的处理方法传入参数ID，如下所示：

**@RequestMapping(value = "/show", method = RequestMethod.GET)**

**public** String showSpittle(

**@RequestParam("spittle\_id")** long spittleId,

Model model) {

model.addAttribute(spittleRepository.findOne(spittleId));

**return** "spittle";

}

这个方法将处理类似/spittles/show?spittle\_id=12345的请求，尽管这可以工作，但是从基于资源管理的角度并不理想。理想情况下，某个指定的资源应该可以通过路径指定，而不是通过查询参数指定，因此GET请求最好是这种形式：/spittles/12345。

首先编写一个测试用例，代码如下：

@**Test**

public void testSpittle() throws **Exception** {

**Spittle** expectedSpittle = **new** **Spittle**("**Hello**", new **Date**());

**SpittleRepository** **mockRepository** = **mock**(**SpittleRepository**.class);

**when**(mockRepository.findOne(12345)).**thenReturn**(expectedSpittle);

**SpittleController** **controller** = **new** **SpittleController**(mockRepository);

**MockMvc** **mockMvc** = **standaloneSetup**(controller).**build**();

**mockMvc**.**perform**(get("/spittles/12345"))

.**andExpect**(view().**name**("spittle"))

.**andExpect**(model().**attributeExists**("spittle"))

.**andExpect**(model().**attribute**("spittle", expectedSpittle));

}

该测试用例首先模拟一个repository、控制器和MockMvc对象，跟之前的几个测试用例相同。不同之处在于这里构造的GET请求——/spittles/12345，并希望返回的视图逻辑名称是spittle，返回的模型对象中包含关键字spittle，且与该key对应的值为我们创建的测试数据。

为了实现路径参数，Spring MVC在@RequestMapping注解中提供占位符机制，并在参数列表中通过@PathVariable("spittleId")获取路径参数，完整的处理方法列举如下：

**@RequestMapping(value = "/{spittleId}", method = RequestMethod.GET)**

**public** String showSpittle(

**@PathVariable("spittleId")** long spittleId,

Model model) {

model.addAttribute(spittleRepository.findOne(spittleId));

**return** "spittle";

}

@PathVariable注解的参数应该和@RequestMapping注解中的占位符名称完全相同；如果函数参数也和占位符名称相同，则可以省略@PathVariable注解的参数，代码如下所示：

**@RequestMapping(value = "/{spittleId}", method = RequestMethod.GET)**

**public** String showSpittle(

**@PathVariable** long spittleId,

Model model) {

model.addAttribute(spittleRepository.findOne(spittleId));

**return** "spittle";

}

这么写确实可以使得代码更加简单，不过需要注意：如果要修改函数参数名称，则要同时修改路径参数的占位符名称。

## 5.4 处理表单

Web应用通常不仅仅是给用户显示数据，也接受用户的表单输入，最典型的例子就是账号注册页面——需要用户填入相关信息，应用程序按照这些信息为用户创建一个账户。

关于表单的处理有两个方面需要考虑：显示表单内容和处理用户提交的表单数据。在Spittr应用中，需要提供一个表单供新用户注册使用；需要一个SpitterController控制器显示注册信息。

**package** **org**.test.spittr.web;

**import** **org**.springframework.stereotype.Controller;

**import** **org**.springframework.web.bind.annotation.RequestMapping;

**import** **org**.springframework.web.bind.annotation.RequestMethod;

@**Controller**

@RequestMapping("/spitter")

public class SpitterController {

@**RequestMapping**(**value** = "/register", method = RequestMethod.GET)

public String showRegistrationForm() {

**return** "**registerForm**";

}

}

修饰showRegistrationForm()方法的@RequestMapping(value = "/register", method = RequestMethod.GET)注解，和类级别的注解一起，表明该方法需要处理类似"/spitter/register"的GET请求。这个方法非常简单，没有输入，且仅仅返回一个逻辑名称——"registerForm"。

即使showRegistrationForm()方法非常简单，也应该写个单元测试，代码如下所示：

**@Test**

**public** **void** **shouldShowRegistrationForm**() **throws** Exception {

SpitterController controller = **new** SpitterController();

MockMvc mockMvc = standaloneSetup(controller).build();

mockMvc.perform(get("/spitter/register"))

.andExpect(view().name("registerForm"));

}

为了接受用户的输入，需要提供一个JSP页面——registerForm.jsp，该页面的代码如下所示：

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<%@ page contentType="text/html;charset=UTF-8" language="java" %>

<html>

<head>

<title>Spittr</title>

</head>

<body>

<h1>Register</h1>

<form method="POST">

First Name: <input type="text" name="firstName" /><br/>

Last Name: <input type="text" name="lastName"/><br/>

Username: <input type="text" name="username"/><br/>

Password: <input type="password" name="password" /><br/>

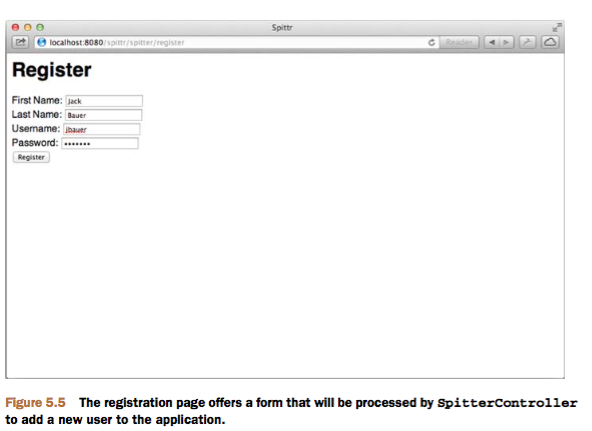
<input type="submit" value="Register" />

</form>

</body>

</html>

上述JSP页面在浏览器中渲染图如下所示：



因为<form>标签并没有设置action参数，因此，当用户单击submit按钮的时候，将向后台发出/spitter/register的POST请求。这就需要我们为SpitterController编写对应的处理方法。

### 5.4.1 编写表单控制器

在处理来自注册表单的POST请求时，控制器需要接收表单数据，然后构造Spitter对象，并保存在数据库中。为了避免重复提交，应该重定向到另一个页面——用户信息页。

按照惯例，首先编写测试用例，如下所示：

**@Test**

**public** **void** **shouldProcessRegistration**() **throws** Exception {

SpitterRepository mockRepository = mock(SpitterRepository.class);

Spitter unsaved = **new** Spitter("Jack", "Bauer", "jbauer", "24hours");

Spitter saved = **new** Spitter(24L, "Jack", "Bauer", "jbauer", "24hours");

when(mockRepository.save(unsaved)).thenReturn(saved);

SpitterController controller = **new** SpitterController(mockRepository);

MockMvc mockMvc = standaloneSetup(controller).build();

mockMvc.perform(post("/spitter/register")

.param("firstName", "Jack")

.param("lastName", "Bauer")

.param("username", "jbauer")

.param("password", "24hours"))

.andExpect(redirectedUrl("/spitter/jbauer"));

//Verified save(unsaved) is called atleast once

verify(mockRepository, atLeastOnce()).save(unsaved);

}

显然，这个测试比之前验证显示注册页面的测试更加丰富。首先设置好SpitterRepository对象、控制器和MockMvc对象，然后构建一个POST请求——/spitter/register，且该请求会携带四个参数，用于模拟submit的提交动作。

在处理POST请求的最后一般需要利用重定向到一个新的页面，以防浏览器刷新引来的重复提交。在这个例子中我们重定向到/spitter/jbaure，即新添加的用户的个人信息页面。

最后，该测试用例还需要验证模拟对象mockRepository确实用于保存表单提交的数据了，即save()方法之上调用了一次。

在SpitterController中添加处理表单的方法，代码如下：

**@RequestMapping(value = "/register", method = RequestMethod.POST)**

**public** String processRegistration(Spitter spitter) {

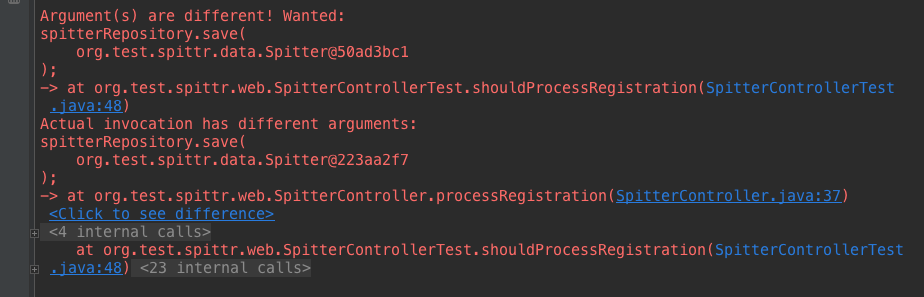
spitterRepository.save(spitter);

**return** "redirect:/spitter/" + spitter.getUsername();

}

shouldShowRegistrationForm()这个方法还在，新加的处理方法processRegistration()以Spitter对象为参数，Spring利用POST请求所携带的参数初始化Spitter对象。

现在执行之前的测试用例，发现一个错误如下所示：



我分析了这个错误，原因是测试用例的写法有问题：verify(mockRepository, atLeastOnce()).save(unsaved);这行代码表示，希望调用至少保存unsave这个对象一次，而实际上在控制器中执行save的时候，参数对象的ID是另一个——根据参数新创建的。回顾我们写这行代码的初衷：确保save方法至少被调用一次，而保存哪个对象则无所谓，因此，这行语句改成verify(mockRepository, atLeastOnce());后，再次执行测试用例就可以通过了。

注意：无论使用哪个框架，请尽量不要使用verify，也就是传说中的Mock模式，那是把代码拉入泥潭的开始。参见[你应该更新的Java知识之常用程序库](http://www.im47.cn/post/2013/2013-05-21-ni-ying-gai-geng-xin-de-javazhi-shi-zhi-chang-yong-cheng-xu-ku#toc_0)

当InternalResourceViewResolver看到这个函数返回的重定向URL是以view标志开头，就知道需要把该URL当做重定向URL处理，而不是按照视图逻辑名称处理。在这个例子中，页面将被重定向至用户的个人信息页面。因此，我们还需要给SpitterController添加一个处理方法，用于显示个人信息，showSpitterProfile()方法代码如下：

**@RequestMapping(value = "/{username}", method = RequestMethod.GET)**

**public** String showSpitterProfile(

**@PathVariable** String username, Model model) {

Spitter spitter = spitterRepository.findByUsername(username);

model.addAttribute(spitter);

**return** "profile";

}

showSpitterProfile()方法根据username从SpitterRepository中查询Spitter对象，然后将该对象存放在model对象中，并返回视图的逻辑名称profile。

profile.jsp的页面代码如下所示：

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<%@ page contentType="text/html;charset=UTF-8" language="java" %>

<html>

<head>

<title>Your Profile</title>

</head>

<body>

<h1>Your Profile</h1>

<c:out value="${spitter.username}"/><br/>

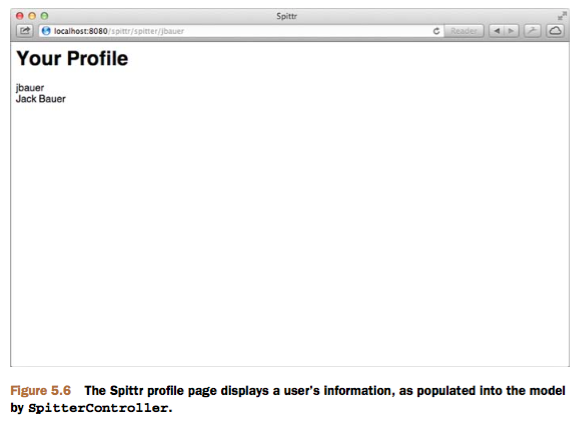
<c:out value="${spitter.firstName}"/><br/>

<c:out value="${spitter.lastName}" /><br/>

</body>

</html>

上述代码的渲染图如下图所示：



### 5.4.2 表单验证

如果用户忘记输入username或者password就点了提交，则可能创建一个这两个字段为空字符串的Spitter对象。往小了说，这是丑陋的开发习惯，往大了说这是会应发安全问题，因为用户可以通过提交一个空的表单来登录系统。

综上所述，需要对用户的输入进行有效性验证，一种验证方法是为processRegistration()方法添加校验输入参数的代码，因为这个函数本身非常简单，参数也不多，因此在开头加入一些If判断语句还可以接受。

除了使用这种方法，换可以利用Spring提供的Java验证支持（a.k.a JSR-303）。从Spring 3.0开始，Spring支持在Spring MVC项目中使用Java Validation API。

首先需要在pom文件中添加依赖：

<dependency>

<groupId>javax.validation</groupId>

<artifactId>validation-api</artifactId>

</dependency>

然后就可以使用各类具体的注解，进行参数验证了，以Spitter类的实现为例：

**package** org.test.spittr.**data**;

**import** javax.validation.constraints.NotNull;

**import** javax.validation.constraints.Size;

**public** **class** **Spitter** {

**private** Long id;

**@NotNull**

**@Size(min = 5, max = 16)**

**private** String username;

**@NotNull**

**@Size(min = 5, max = 25)**

**private** String password;

**@NotNull**

**@Size(min = 2, max = 30)**

**private** String firstName;

**@NotNull**

**@Size(min = 2, max = 30)**

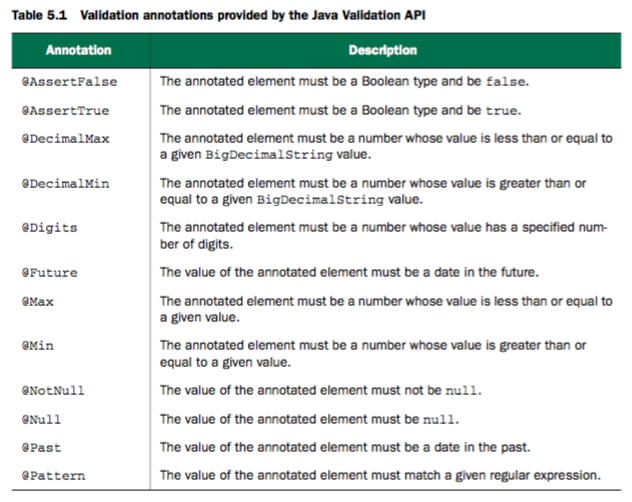
**private** String lastName;

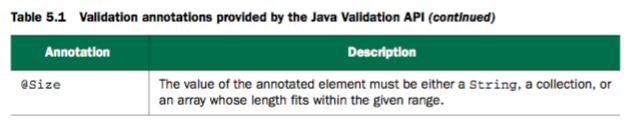
....

}

@NotNull注解表示被它修饰的字段不能为空；@Size字段用于限制指定字段的长度范围。在Spittr应用的含义是：用户必须填写表单中的所有字段，并且满足一定的长度限制，才可以注册成功。

除了上述两个注解，Java Validation API提供了很多不同功能的注解，都定义在javax.validation.constraints包种，下表列举出这些注解：





在Spittr类的定义中规定验证条件后，需要在控制器的处理方法中应用验证条件。

**@RequestMapping(value = "/register", method = RequestMethod.POST)**

**public** String processRegistration(

**@Valid** Spitter spitter,

Errors errors) {

**if** (errors.hasErrors()) {

**return** "registerForm";

}

spitterRepository.save(spitter);

**return** "redirect:/spitter/" + spitter.getUsername();

}

如果用户输入的参数有误，则返回registerForm这个逻辑名称，浏览器将返回到表单填写页面，以便用户重新输入。当然，为了更好的用户体验，还需要提示用户具体哪个字段写错了，应该怎么改；最好是在用户填写之前就做出提示，这就需要前端工程师做很多工作了。

## 5.5 总结

这一章比较适合Spring MVC的入门学习资料。涵盖了Spring MVC处理web请求的处理过程、如何写简单的控制器和控制器方法来处理Http请求、如何使用mockito框架测试控制器方法。

基于Spring MVC的应用有三种方式读取数据：查询参数、路径参数和表单输入。本章用两节介绍了这些内容，并给出了类似错误处理和参数验证等关键知识点。