1 Spring MVC 起步

1.1 Spring MVC 请求

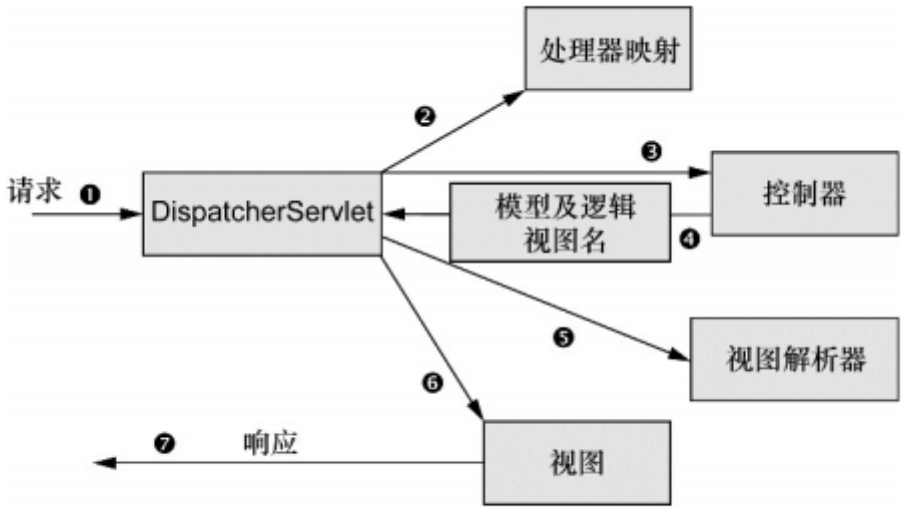

图5.1 一路上请求会将信息带到很多站点，并生产期望的结果

Spring MVC所有请求都会通过前端控制器Servlet

- **DispatcherServlet 前端控制器** 任务是：将请求发送给Spring MVC控制器。控制器是一个用于处理请求的Spring组件。控制器可以有多个，DispatcherServlet会查询一个或多个处理器映射来确定请求的下一站。
- **处理器映射会更具请求所携带的URL信息来决策** 一旦选择了合适的控制器，DispatcherServlet会将请求发送给选定的控制器
- **控制器** 请求到达控制器并等待控制器处理信息。实际上控制器本省只处理很少甚至不处理工作，而是将业务逻辑委托给一个或多个服务进行处理
- **模型** 控制器处理后，会产生一些信息返回给用户并再浏览器上显示。这些信息被称为模型。不过原始信息是不够的--这些信息需要以友好的方式进行格式化，一般是HTML。所以信息需要发送给一个**视图**，通常是jsp

  控制器最后的一件事就是将模型数据打包，并且标示出用于渲染输出的视图名。然后将请求连同模型和视图名发送回DispatcherServlet

  这样，控制器就不会与特定的视图相耦合，传递个DispatcherServlet的试图并不直接标识某个特定的JSP。他甚至并不能确定试图就是JSP，而仅仅传递一个逻辑名称，用来查找产生结果的真正试图。DispatcherServlet将会使用视图解析器来讲逻辑视图名匹配为一个特定的视图实现。
- **视图** DispatcherServlet已经知道又哪个视图渲染结果，最后是视图的实现（可能是JSP），在这里交付模型数据。视图将是一个你模型数据渲染输出，这个对象会通过响应对象传递给客户端

1.2 搭建Spring MVC

```
@Configuration
@EnableWebMvc
public class webConfig{

}
```

- **没有配置视图解析器。**Spring默认使用BeanNameView-Resolver，会查找Id与视图名称匹配的bean，并且查找的bean要实现View接口
- **没有启动组件扫描。**Spring只能找到显式生命在配置类中的控制器
- **DispatcherServlet会映射为应用的默认Servlet** 它会处理所有的请求，包括对静态资源的请求，如图片和样式表

```
@Configuration
@EnableWebMvc
@ComponentScan("spittr.web")
public class WebConfig extends WebMvcConfigurerAdapter {

    @Bean
    public ViewResolver viewResolver() {
        InternalResourceViewResolver resolver = new
InternalResourceViewResolver();
        resolver.setPrefix("/WEB-INF/views/");
        resolver.setSuffix(".jsp");
        return resolver;
    }

    @Override
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer
configurer) {
        configurer.enable();
    }

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        // TODO Auto-generated method stub
        super.addResourceHandlers(registry);
    }

}
```

- **@ComponentScan** 会扫描spitter.web包查找组件。我们所编写的控制器及那个会带有@Controller注解，这会使其称为组件扫描时的候选bean
- **ViewResolver bean** 它会查找JSp文件，在查找的时候，他会在视图名称上加一个特定的前缀和后缀(例如名为home的视图将会解析为/WEB-INF/view/home.jsp)
- **WebConfig 扩展 WebMvcConfigurerAdapter** 重写了configureDefaultServletHandling，通过调用DefaultServletHandlerConfigurer的enable()方法，要求DispatcherServlet将对静态资源的请求抓发到Servlet容器中默认的Servlet上，而不是使用DispatcherServlet自生

```
@Configuration
@ComponentScan(basePackages = {"spittr"}, excludeFilters = {@Filter(type =
FilterType.CUSTOM, value = WebPackage.class)})
public class RootConfig {
}
```

RootConfig 使用了@ComponentScan，这样，就可以用非Web组件来充实完善RootConfig

2 编写控制器

Spring MVC中，控制器只是方法上添加了@RequestMapping注解的类，这个注解声明了他所要处理的请求

```
@Controller//声明一个控制器
public class HomeController {

    @RequestMapping(value = "/", method = GET)
    public String home(Model model) {
        return "home";//视图名为home
    }
}
```

HomeController带有@Controller，组件扫描会自动找到HomeController，将其声明问Spring应用上下文中共的一个bean

@RequestMapping，它的value属性制定了这个方法要处理的请求路径，method属性细化了它所处理的HTTP方法。

home()方法返回了"home",这个String将会被Spring MVC解读为要渲染的视图名称。DispatcherServlet会要求视图解析器将这个逻辑名称解析为实际的视图。

由于配置了InternalResourceViewResolver，视图名'home'将会解析为"/WEB-INF/views/home.jsp"路径的JSP

2.1 测试控制器

```
public class HomeControllerTest {

    @Test
    public void testHomePage() throws Exception {
        HomeController controller = new HomeController();
        MockMvc mockMvc = standaloneSetup(controller).build();
        mockMvc.perform(get("/")).andExpect(view().name("home"));
    }
}
```

使用MockMvc，发起了对"/"的GET请求，并断言视图的名称为home

2.2 定义类级别的请求处理

```
@Controller
@RequestMapping("/")
public class HomeController {
    @RequestMapping(method = GET)
    public String home() {
        return "home";
    }
}
```

路径转移到类级别的@RequestMapping，而HTTP方法依然映射再方法级别上。当哦内阁制其再类级别上添加@RequestMapping时，这个注解会应用到控制器的所有处理器方法上。处理器方法上的@RequestMapping会对类级别上的@RequestMapping声明进行补充。

2.3 传递模型数据到视图中

```
@Controller
@RequestMapping("/spittles")
public class SpittleController {
```

```
    private static final String MAX_LONG_AS_STRING = "9223372036854775807";

    private SpittleRepository spittleRepository;

    @Autowired
    public SpittleController(SpittleRepository spittleRepository) {
        this.spittleRepository = spittleRepository;
    }

    @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);
    }

    @RequestMapping(value = "/{spittleId}", method = RequestMethod.GET)
    public String spittle(
            @PathVariable("spittleId") long spittleId,
            Model model) {
        model.addAttribute(spittleRepository.findOne(spittleId));
        return "spittle";
    }

    @RequestMapping(method = RequestMethod.POST)
    public String saveSpittle(SpittleForm form, Model model) throws Exception {
        spittleRepository.save(new Spittle(null, form.getMessage(), new Date(),
                form.getLongitude(), form.getLatitude()));
        return "redirect:/spittles";
    }

}
```

```
@Test
public void houldShowRecentSpittles() throws Exception {
    List<Spittle> expectedSpittles = createSpittleList(20);
    SpittleRepository mockRepository = mock(SpittleRepository.class);//Mock
Repository
    when(mockRepository.findSpittles(Long.MAX_VALUE, 20))
            .thenReturn(expectedSpittles);

    SpittleController controller = new SpittleController(mockRepository);
    MockMvc mockMvc = standaloneSetup(controller)// Mock Spring MVC
            .setSingleView(new InternalResourceView("/WEB-
INF/views/spittles.jsp"))
            .build();

    mockMvc.perform(get("/spittles"))
            .andExpect(view().name("spittles"))
            .andExpect(model().attributeExists("spittleList"))
            .andExpect(model().attribute("spittleList",
                    hasItems(expectedSpittles.toArray())));
}
```

测试先创建SpittleRepository的Mock实现，这个实现从findSpittles()方法返回20个Spittle对象，然后将这个Repository注入到一个新的SpittleController，然后创建MokMvc并使用这个控制器。

MokMvc调用setSingleView()方法。这样mock框架就甭能解析控制器中的视图名了。(很多场景没必要)

该测试对"/spittles"发起请求，然后断言视图的名称为spittles并且模型中包含名为spittleList的属性，在spittleList中包含预期的内容。

```
@RequestMapping(method = RequestMethod.GET)
public String spittles(Model model) {
    //将spittle添加到模型中，
    model.addAttribute(spittleRepository.findSpittles(Long.MAX_VALUE, 20));
    //model.addAttribute("spittleList",
spittleRepository.findSpittles(Long.MAX_VALUE, 20));
    return "spittles";//返回视图名
}
```

Model实际上是一个Map，他会传递给视图，这样数据就能渲染到客户端。当调用addAttribute方法时，如果不指定key，key会根据值的对象类型推断确定。由于本例是List，会推断为spittleList

还可以编写为

```
@RequestMapping(method = RequestMethod.GET)
public List<Spittle> spittles() {
    return spittleRepository.findSpittles(Long.MAX_VALUE, 20);//返回视图名
}
```

- 没有返回视图名称也没有显示的设置模型。这个方法返回的是Spittle列表当处理器方法像这样返回对象或者集合。这个值会放入到模型中。模型的key会根据类型推断得出
- 逻辑视图名称会格局请求路径推断得出。因为该方法处理针对"/spittles"的GET请求，因此视图名称是spittles(去掉开头的斜线)

  模型中会存储一个Spittle列表，key为spittleList，然后这个列表会发送到名为spittles的视图中。按照配置的InternalResourceViewResolver的方式，视图的JSP将会是 "/WEB-INF/views/spittles.jsp"

3 接受请求的输入

Spring MVC允许以多种方式将客户端的数据传送到控制器的处理器方法中：

- **查询参数 (Query Parameter)**
- **表单参数 (Form Parameter)**
- **路径变量 (Paht Variable)**

3.1 处理查询参数

```
@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);
}
```

```
@Test
```

```
public void shouldShowPagedSpittles() 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())));
}
```

3.2 通过路径参数接受输入

```
@RequestMapping(value = "/show", method = RequestMethod.GET)
public String spittle(
    @RequestParam("spittleId") long spittleId,Model model) {

    model.addAttribute(spittleRepository.findOne(spittleId));
    return "spittle";
}
```

该方法会处理"spittles/show?spittleId=12345"这样的请求。但是正常情况下要是别的资源应该通过URL路径进行标示，俄日不是通过查询参数。"spittles/12345"发起请求更优，可以识别要查询的资源

@RequestMapping需要包含变量部分，Spring MVC允许在@RequestMapping路径中添加占位符。占位符的名称要用大括号{}括起来。路径的其他部分要与所处的请求完全匹配，但是占位符部分可以是任意的值。

```
@RequestMapping(value = "/{spittleId}", method = RequestMethod.GET)
public String spittle(
        @PathVariable("spittleId") long spittleId,
        Model model) {
    model.addAttribute(spittleRepository.findOne(spittleId));
    return "spittle";
}
```

```
@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));
}
```

spittleId参数上添加了，@PathVariable("spittleId")，表明在请求路径中，不管占位符部分的值是什么都会传递到处理器方法的spittleId参数中。**@PathVariable如果没有value属性，会是假设占位符的名称与方法的参数名相同。如果需要重命名参数，必须要同时修改占位符的名称，使其互相匹配**

4 处理表单

```
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;

import spittr.Spitter;
import spittr.data.SpitterRepository;

@Controller
@RequestMapping("/spitter")
public class SpitterController {

    private SpitterRepository spitterRepository;

    @Autowired
    public SpitterController(SpitterRepository spitterRepository) {
        this.spitterRepository = spitterRepository;
    }

    @RequestMapping(value = "/register", method = GET)
    public String showRegistrationForm() {
        return "registerForm";
    }
}
```

showRegistrationForm()没有任何输入只是返回名为registerForm的视图，意味着会使用”/WEB-INF/views/registerForm.jsp“来渲染注册表单。

```
@Test
public void shouldShowRegistration() throws Exception {
    SpitterRepository mockRepository = mock(SpitterRepository.class);//Mock
Repository
    SpitterController controller = new SpitterController(mockRepository);
    MockMvc mockMvc = standaloneSetup(controller).build();
    mockMvc.perform(get("/spitter/register"))
            .andExpect(view().name("registerForm"));
}
```

```
@Controller
@RequestMapping("/spitter")
public class SpitterController {

    private SpitterRepository spitterRepository;

    @Autowired
    public SpitterController(SpitterRepository spitterRepository) {
        this.spitterRepository = spitterRepository;
```

```
    }

    @RequestMapping(value = "/register", method = GET)
    public String showRegistrationForm() {
        return "registerForm";
    }

    @RequestMapping(value = "/register", method = POST)
    public String processRegistration(
            @Valid Spitter spitter,
            Errors errors) {
        if (errors.hasErrors()) {
            return "registerForm";
        }

        spitterRepository.save(spitter);
        return "redirect:/spitter/" + spitter.getUsername();
    }
```

processRegistration()方法，接受一个Spitter对象作为参数。这个对象的属性将会使用请求终同名属性参数进行填充。灾后放回一个String类型，用来指定视图。这里不仅返回了视图的名称供视图解析器朝朝目标是图，而且返回的值还带有重定向的格式。

当InternalResourViewResolver解析到视图个是中共的”redirect:“前缀时。他就知道要将其解析为重定向的规则，而不是视图的名称。如果username为test，视图将会重定向到”spitter/test“.

- **forward: 前缀**，当InternalResourViewResolverforward: 前缀。请求将会前往forward指定的URL路径。

同时需要添加方法

```
@RequestMapping(value = "/{username}", method = GET)
public String showSpitterProfile(@PathVariable String username, Model model) {
    Spitter spitter = spitterRepository.findByUsername(username);
    model.addAttribute(spitter);
    return "profile";
}
```

Java校验API定义了多个注解，这些注解可以放到属性上，从而限制谢谢属性的值

注解	描述
@AssertFalse	所注解的元素必须是Boolean类型，并且值为false
@AssertTrue	所注解的元素必须是Boolean类型，并且值为true
@DecimalMax	所注解的元素必须是数字，并且小于等于给定的BigDecimalString值
@DecimalMin	所注解的元素必须是数字，并且大于等于给定的BigDecimalString值
@Digits	所注解的元素必须是数字，并且必须有指定的位数
@Future	所注解的元素必须是将来的日期
@Max/@Min	所注解的元素必须是数字，并且小于等于/大于等于给定的值
@NotNull/@Null	所注解的元素不能/必须为null
@Past	所注解的元素必须是过去的日期
@Pattern	所注解的元素必须匹配给定的正则表达式
@Size	所注解的元素值必须是string、集合或数组，并且它的长度要符合给定的范围

```
@Data
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;

    public Spitter() {
    }

    public Spitter(String username, String password, String firstName, String
lastName) {
        this(null, username, password, firstName, lastName);
    }

    public Spitter(Long id, String username, String password, String firstName,
String lastName) {
        this.id = id;
        this.username = username;
```

```
        this.password = password;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Override
    public boolean equals(Object that) {
        return EqualsBuilder.reflectionEquals(this, that, "firstName",
"lastName", "username", "password");
    }

    @Override
    public int hashCode() {
        return HashCodeBuilder.reflectionHashCode(this, "firstName", "lastName",
"username", "password");
    }

}
```

```
@RequestMapping(value = "/register", method = POST)
public String processRegistration(
        @Valid Spitter spitter,
        Errors errors) {
    if (errors.hasErrors()) {//校验出错重新返回表单
        return "registerForm";
    }

    spitterRepository.save(spitter);
    return "redirect:/spitter/" + spitter.getUsername();
}
```

Spitter参数添加了@Valid注解，这会告知Spring，需要且报这个对象满足校验限制。属性限制不能组织表单提交。方法依然会被调用。这时候就需要处理检验的错误。如果校验出现错误，这些错误可以通过Errors对象进行访问，现在这个对象以作为方法的参数(**Errors 参数要紧跟在带有@Valid注解的参数后面，@Valid注解所标注的就是要检验的参数**)