# 基于 Java EE 的企业应用系统设计
# Spring MVC

王晓东
wangxiaodong@ouc.edu.cn

中国海洋大学

December 22, 2017


# References

1. Spring MVC: A Tutorial (Second Edition) (ISBN 9781771970310)

# 大纲

Java Web 应用的开发演化

MVC 模式示例

Spring MVC

数据绑定和表单标签库

# 接下来…

Java Web 应用的开发演化

MVC 模式示例

Spring MVC

数据绑定和表单标签库

# JSP 方式

JSP 在 HTML 代码里写 Java 代码完成业务逻辑。

```
<%
    String name = request.getParameter("name");
    String password = request.getParameter("password");

    UserHandler userHandler = new UserHandler();
    if(userHandler.authenticate(name, password)) {
%>
<p>Congratulations, login successfully. </p>
<%
     } else {
%>
<p>Sorry, login failed.</p>
<%
     }
%>
```

# JSP 方式

## ❖ 仅有的一点优势

1. 无需额外的配置文件，无需框架的帮助，即可完成逻辑。
2. 简单易上手。

## ❖ 劣势

1. Java 代码由于混杂在一个 HTML 环境中而显得混乱不堪，可读性非常差。一个 JSP 文件有时候会变成几十 K，甚至上百 K，经常难以定位逻辑代码的所在。
2. 编写代码时非常困惑，不知道代码到底应该写在哪里，也不知道别人是不是已经曾经实现过类似的功能，到哪里去引用。
3. 突然之间，某个需求发生了变化。于是，每个人蒙头开始全程替换，还要小心翼翼的，生怕把别人的逻辑改了。
4. 逻辑处理程序需要自己来维护生命周期，对于类似数据库事务、日志等众多模块无法统一支持。

# JSP 方式

## ❖ 仅有的一点优势

1. 无需额外的配置文件，无需框架的帮助，即可完成逻辑。
2. 简单易上手。

## ❖ 劣势

1. Java 代码由于混杂在一个 HTML 环境中而显得混乱不堪，可读性非常差。一个 JSP 文件有时候会变成几十 K，甚至上百 K，经常难以定位逻辑代码的所在。
2. 编写代码时非常困惑，不知道代码到底应该写在哪里，也不知道别人是不是已经曾经实现过类似的功能，到哪里去引用。
3. 突然之间，某个需求发生了变化。于是，每个人蒙头开始全程替换，还要小心翼翼的，生怕把别人的逻辑改了。
4. 逻辑处理程序需要自己来维护生命周期，对于类似数据库事务、日志等众多模块无法统一支持。

# JSP 方式

## ❖ 仅有的一点优势

1. 无需额外的配置文件，无需框架的帮助，即可完成逻辑。
2. 简单易上手。

## ❖ 劣势

1. Java 代码由于混杂在一个 HTML 环境中而显得混乱不堪，可读性非常差。一个 JSP 文件有时候会变成几十 K，甚至上百 K，经常难以定位逻辑代码的所在。
2. 编写代码时非常困惑，不知道代码到底应该写在哪里，也不知道别人是不是已经曾经实现过类似的功能，到哪里去引用。
3. 突然之间，某个需求发生了变化。于是，每个人蒙头开始全程替换，还要小心翼翼的，生怕把别人的逻辑改了。
4. 逻辑处理程序需要自己来维护生命周期，对于类似数据库事务、日志等众多模块无法统一支持。

# JSP 方式

## ❖ 仅有的一点优势

1. 无需额外的配置文件，无需框架的帮助，即可完成逻辑。
2. 简单易上手。

## ❖ 劣势

1. Java 代码由于混杂在一个 HTML 环境中而显得混乱不堪，可读性非常差。一个 JSP 文件有时候会变成几十 K，甚至上百 K，经常难以定位逻辑代码的所在。
2. 编写代码时非常困惑，不知道代码到底应该写在哪里，也不知道别人是不是已经曾经实现过类似的功能，到哪里去引用。
3. 突然之间，某个需求发生了变化。于是，每个人蒙头开始全程替换，还要小心翼翼的，生怕把别人的逻辑改了。
4. 逻辑处理程序需要自己来维护生命周期，对于类似数据库事务、日志等众多模块无法统一支持。

# 需求的变化

如果有一种方式能够将页面上的那些 Java 代码抽取出来，让页面上尽量少出现 Java 代码该有多好！

于是许多人开始使用 servlet 来处理那些业务逻辑。

# Servlet 方式

```
public class LoginServlet extends HttpServlet {

  @Override
  protected void doPost(HttpServletRequest req,
  HttpServletResponse resp) throws ServletException, IOException {
    String message = null;
    RequestDispatcher dispatcher = req.
      getRequestDispatcher("/result.jsp");
    String name = req.getParameter("name");
    String password = req.getParameter("password");

    UserHandler userHandler = new UserHandler();
    if(userHandler.authenticate(name, password)) {
      message = "恭喜你，登录成功";
    } else {
      message = "对不起，登录失败";
    }

    req.setAttribute("message", message);
    dispatcher.forward(req, resp);
  }
}
```

# Servlet 方式

同时，我们需要在 web.xml 中为这个 servlet 配置 url 的请求映射关系。

```
<servlet>
  <servlet-name>Login</servlet-name>
  <servlet-class>com.demo2do.servlet.LoginServlet</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>Login</servlet-name>
  <url-pattern>/Login</url-pattern>
</servlet-mapping>
```

# 框架方式

时代进一步发展，人们发现简单的 JSP 和 Servlet 已经很难满足人们懒惰的要求了。于是，人们开始试图总结一些公用的 Java 类，来解决 Web 开发过程中碰到的问题。如 Struts、Spring MVC，它们非常先进地实现了MVC 模式。

# 那么我们需要什么？

在回顾写代码的历史之后，回头来看我们到底需要什么？
无论是使用 JSP，还是使用 Struts，或是 Spring MVC，我们至少都需要一些必须的元素：

1. **数据**在这个例子中，就是 name 和 password。他们共同构成了程序数据的核心载体。事实上，我们往往会有一个 User 类来封装 name 和 password，这样会使得我们的程序更加 OO。无论怎么说，数据会穿插在这个程序的各处，成为程序运行的核心。
2. 页面展示
3. 处理具体业务的场所

# 那么我们需要什么？

在回顾写代码的历史之后，回头来看我们到底需要什么？
无论是使用 JSP，还是使用 Struts，或是 Spring MVC，我们至少都需要一些必须的元素：

1. 数据
2. **页面展示**在这个例子中，就是 login.jsp。没有这个页面，一切的请求、验证和错误展示也无从谈起。在页面上，我们需要利用 HTML，把我们需要展现的数据都呈现出来。同时我们也需要完成一定的页面逻辑，例如，错误展示，分支判断等。
3. 处理具体业务的场所

# 那么我们需要什么？

在回顾写代码的历史之后，回头来看我们到底需要什么？
无论是使用 JSP，还是使用 Struts，或是 Spring MVC，我们至少都需要一些必须的元素：

1. 数据
2. 页面展示
3. **处理具体业务的场所**不同阶段，处理具体业务的场所就不太一样。原来用 JSP 和 Servlet，后来用 Struts 的 Action、Spring 的 Controller。

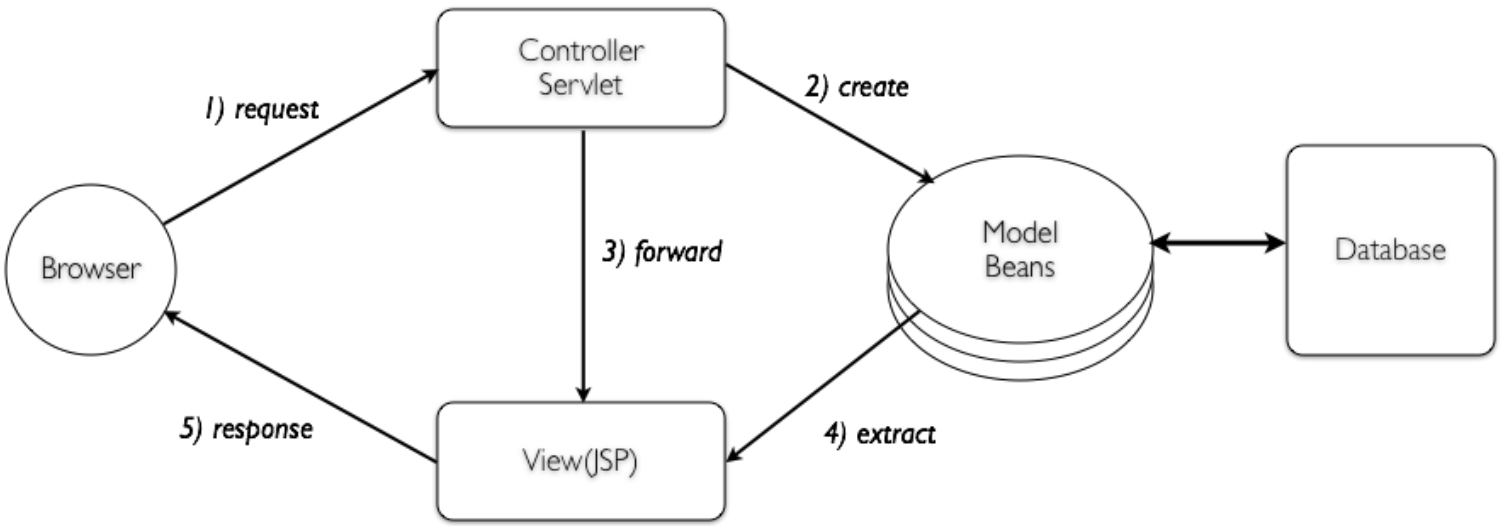
# MVC

上面的这些必须出现的元素，在不同的时代被赋予了不同的表现形式，有的受到时代的束缚，其表现形式非常落后，有的已经不再使用。但是拨开这些外在的表现形式，我们就可以发现，这就是我们已经熟悉的 MVC。

- 数据 ⇨ Model
- 页面展示 ⇨ View
- 处理具体业务的场所 ⇨ Control

框架不重要。只要能够深刻理解 MVC 的概念，框架只是几个 jar 包而已。

# MVC




**MVC Design Model**

# MVC 的特点

1. 多个视图可以对应一个模型，可以减少代码的复制，在模型发生改变时，易于维护。
2. 模型返回的数据与显示逻辑分离。模型数据可以应用任何显示技术，例如，使用 JSP、Velocity 模板或者直接产生 Excel。
3. 应用被分为三层，降低各层耦合，提高了可扩展性。
4. 控制层把不同模型和视图组合在一起，完成不同的请求，控制层包含了用户请求权限的概念。
5. MVC 符合软件工程化管理的思想，不同层各司其职，有利于通过工程化和工具化产生管理程序代码。

# MVC

数据是动的，数据在 View 和 Control 层一旦运动起来，就会产生许多的问题：

- 数据从 View 层传递到 Control 层，如何使得一个个扁平的字符串，转化成一个个生龙活虎的 Java 对象。
- 数据从 View 层传递到 Control 层，如何方便的进行数据格式和内容的校验？
- 数据从 Control 层传递到 View 层，一个个 Java 对象，又如何在页面上以各种各样的形式展现出来。
- 如果试图将数据请求从 View 层发送到 Control 层，你如何才能知道你要调用的究竟是哪个类，哪个方法？一个 Http 的请求，又如何与 Control 层的 Java 代码建立起关系来？

# MVC

数据是动的，数据在 View 和 Control 层一旦运动起来，就会产生许多的问题：

- 数据从 View 层传递到 Control 层，如何使得一个个扁平的字符串，转化成一个个生龙活虎的 Java 对象。
- 数据从 View 层传递到 Control 层，如何方便的进行数据格式和内容的校验？
- 数据从 Control 层传递到 View 层，一个个 Java 对象，又如何在页面上以各种各样的形式展现出来。
- 如果试图将数据请求从 View 层发送到 Control 层，你如何才能知道你要调用的究竟是哪个类，哪个方法？一个 Http 的请求，又如何与 Control 层的 Java 代码建立起关系来？

# MVC

数据是动的，数据在 View 和 Control 层一旦运动起来，就会产生许多的问题：

- 数据从 View 层传递到 Control 层，如何使得一个个扁平的字符串，转化成一个个生龙活虎的 Java 对象。
- 数据从 View 层传递到 Control 层，如何方便的进行数据格式和内容的校验？
- 数据从 Control 层传递到 View 层，一个个 Java 对象，又如何在页面上以各种各样的形式展现出来。
- 如果试图将数据请求从 View 层发送到 Control 层，你如何才能知道你要调用的究竟是哪个类，哪个方法？一个 Http 的请求，又如何与 Control 层的 Java 代码建立起关系来？

# MVC

数据是动的，数据在 View 和 Control 层一旦运动起来，就会产生许多的问题：

- 数据从 View 层传递到 Control 层，如何使得一个个扁平的字符串，转化成一个个生龙活虎的 Java 对象。
- 数据从 View 层传递到 Control 层，如何方便的进行数据格式和内容的校验？
- 数据从 Control 层传递到 View 层，一个个 Java 对象，又如何在页面上以各种各样的形式展现出来。
- 如果试图将数据请求从 View 层发送到 Control 层，你如何才能知道你要调用的究竟是哪个类，哪个方法？一个 Http 的请求，又如何与 Control 层的 Java 代码建立起关系来？

# 框架

框架是为了解决一个又一个在 Web 开发中所遇到的问题而诞生的。不同的框架，都是为了解决不同的问题，但是对于程序员而言，他们仅仅是 jar 包而已。框架的优缺点的评论，也完全取决于其对问题解决程度和解决方式的优雅性的评论。所以，千万不要为了学习框架而学习框架，而是要为了解决问题而学习框架!

# 接下来…

Java Web 应用的开发演化

MVC 模式示例

Spring MVC

数据绑定和表单标签库

# 应用功能说明

应用功能设定为输入一个产品信息并展示，流程为：

- 用户填写产品表单并提交；
- 应用保存产品并展示一个完成页面，显示已保存的产品信息。

# 程序设计

## ❖ 应用支持以下两个 Action

- 发送输入表单到浏览器上，展示“添加产品”表单，其对应的 URI 应包含字符串 product_input；
- 保存产品并返回完成页面，对应的 URI 包含字符串 product_save。

## ❖ 应用所包含的组件构成

- 一个 Product 类，作为 product 的领域对象；
- 一个 ProductForm 类，封装了 HTML 表单的输入项；
- 一个 ControllerServlet 类，本示例应用的控制器；
- 一个 SaveProductAction 类；
- 两个 JSP 视图页面（ProductForm.jsp 和 ProductDetail.jsp）；
- 一个 CSS 文件，定义了两个 JSP 页面的显示风格。

# 接下来…

Java Web 应用的开发演化

MVC 模式示例

Spring MVC

数据绑定和表单标签库

# 采用 Spring MVC 框架开发 Web 应用的优势

## ❖ Spring MVC 具备的能加速开发的功能列表

1. Spring MVC 是 Spring 框架的一部分，可以利用 Spring 提供的其他能力。
2. Spring MVC 中提供了 Dispatcher Servlet 而无需额外开发。
3. Spring MVC 中使用基于 XML 的配置文件，可以编辑配置而无需重新编译应用程序。
4. Spring MVC 实例化控制器，并根据用户输入来构造 bean。
5. Spring MVC 可以自动绑定用户输入并正确地转换数据类型。
6. Spring MVC 内置了常见的校验器，可以校验用户输入，若校验不通过则重定向回输入表单。
7. Spring MVC 支持国际化和本地化，支持根据用户区域显示多国语言。
8. Spring MVC 支持多种视图技术，包括 JSP 技术、Velocity 和 FreeMarker 等。

# Spring MVC 的 DispatcherServlet

Spring MVC 自带了一个开箱即用的 DispatcherServlet，要使用这个 servlet 需要在部署描述符 web.xml 中配置。

org.springfamework.web.servlet.DispatcherServlet

```
<servlet>
  <servlet-name>springmvc</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>springmvc</servlet-name>
  <!-- Map all requests to the DispatcherServlet -->
  <url-pattern>/</url-pattern>
</servlet-mapping>
```

servlet 元素内的 load-on-startup 元素可选。如果它存在，则在应用程序启动时装载 servlet 并调用它的 init 方法；若不存在，则在该 servlet 的第一个请求到来时加载。

# Spring MVC 的 DispatcherServlet

## ❖ DispatcherServlet 的默认配置文件

DispatcherServlet 会使用 Spring MVC 诸多默认组件。初始化时会寻找在应用程序的 WEB-INF 目录下的配置文件，该配置文件的命名规则为: **servletName-servlet.xml**。
其中，servletName 是在部署描述符中的 DispatcherServlet 的名称。

## ❖ 指定任意位置的 DispatcherServlet 配置文件

```
<servlet>
  <servlet-name>springmvc</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/confiq/simple-config.xml</param-value>
  </init-param〉
  <load-on-startup>l</load-on-startup>
</servlet>
```

# Controller

## ❖ 实现 Controller 的方法

- 实现 org.springframework.web.servlet.mvc.Controller 接口开发控制器，这个接口包含 handleRequest 方法：

```
ModelAndView handleRequest(HttpServletRequest request,
  HttpServletResponse response);
```

其实现类可以访问对应请求的 HttpServletRequest 和 HttpServletResponse 对象，还必须返回一个包含视图路径或视图模型的 ModelAndView 对象。Controller 接口的实现类只能处理一个单一动作。

- 基于注解的控制器可以同时支持多个请求处理动作处理，并且无须实现任何接口。

# Spring MVC 示例

☛ References: springmvc-intro-01

# View Resolver

Spring MVC 中的视图解析器负责解析视图，可以通过在配置文件中定义一个 ViewResolver 来配置视图解析器。

```
  <bean id="viewResolver" class="org.springframework.web.servlet.
␣␣view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/jsp/" />
    <property name="suffix" value=".jsp" />
  </bean>
```

视图解析器配置有前缀和后缀两个属性，View 路径将缩短。例如，仅需提供“myPage”，而不必再设置视图路径为/WEB-INF/jsp/myPage.jsp，视图解析器将会自动增加前缀和后缀。

# 基于注解的控制器

## ❖ 使用基于注解的控制器的几个优点

1. Controller 和 RequestMapping 注释类型是 Spring MVC API 最重要的两个注释类型。
2. 一个控制器类可以处理多个动作（而一个实现了 Controller 接口的控制器只能处理一个动作）。
3. 基于注解的控制器的请求映射不需要存储在配置文件中。使用 RequestMapping 注释类型，可以对一个方法进行请求处理。

# Controller 注解类型

org.springframework.stereotype.Controller 注解类型用于指示 Spring 类的实例是一个控制器。

❖ @Controller 示例

```
package com.example.controller;
import org.springframework.stereotype.Controller;
...

@Controller
public class CustomerController {
  // request-handling methods here
}
```

# Controller 注解类型

Spring 使用扫描机制找到应用程序中所有基于注解的控制器类。为了确保 Spring 能找到控制器，需要在 Spring MVC 的配置文件中完成以下配置：

**声明 springcontext**

```
<beans
xmlns:context="http://www.springfrarnework.org/schema/context">
```

**应用 <component-scan> 元素指定控制器类的基本包**

```
<context:component-scan base-package="basePackage" />
```

例如，若所有的控制器类都在 com.example.controller 及其子包下，则需要写一个如下所示的 <component-scan> 元素：

```
<context:component-scan base-package="com.example.controller" />
```

# RequestMapping 注解类型

org.springframework.web.bind.annotation.RequestMapping 在控制类的内部为每一个动作开发相应的处理方法，要让 Spring 知道用哪一种方法来处理它的动作，需要使用该注解类型映射 URI 与方法。

采用 @RequestMapping 注解的方法将成为请求处理方法，并由调度程序在接收到对应 URL 请求时调用。

## ❖ @RequestMapping 示例

```
package com.example.controller;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class CustomerController {
  @RequestMapping(value = "/customer-input")
  public String inputCustomer() {
    // do something here
    return "CustomerForm";
  }
}
```

# RequestMapping 注解类型

RequestMapping 除了具有 value 属性外，还有其他属性：

method 用来指示该方法仅处理哪些 HTTP 方法。

```
@RequestMappinq(value = "/order-process",
method={RequestMethod.POST, RequestMethod.PUT})
```

# RequestMapping 注解类型

RequestMapping 注解类型也可以用来注解一个控制器类。

```
@Controller
@RequestMapping("/customer")
public class CustomerController {
  @RequestMapping (value ="/delete", method = RequestMethod.POST)
  public String deleteCustomer() {
    // do something here
    return ...;
  }
}
```

在这种情况下，所有的方法都将映射为相对于类级别的请求。由于控制器类的映射使用“/customer”，而 deleteCustomer 方法映射为“/delete”，则如下 URL 会映射到该方法上：

```
1    http://domain/context/customer/delete
```

# 请求处理方法编写

每个请求处理方法可以有多个不同类型的参数，以及一个多种类型的返回结果。

例如，如果在请求处理方法中需要访问 HttpSession 对象，则可以添加的 HttpSession 作为参数，Spring 会将对象正确地传递给方法。

```
@RequestMapping("/uri")
public String myMethod(HttpSession session) {
  ...
  session.addAttribute(key, value);
  ...
}
```

# 编写请求处理方法

## ❖ 可以在请求处理方法中出现的参数类型

1. javax.servlet.ServletRequest 或 HttpServletRequest
2. javax.servlet.ServletResponse 或 HttpServletResponse
3. javax.servlet.http.HttpSession
4. org.springframework.web.context.request.WebRequest 或 NativeWebRequest
5. java.util.Locale
6. java.io.InputStream 或 Reader
7. java.io.OutputStream 或 Writer
8. java.security.Principal
9. HttpEntity<?>
10. java.util.Map / org.springframework.ui.Model
11. org.springframework.ui.ModelMap
12. org.springframework.web.servlet.mvc.support.RedirectAttributes
13. org.springframework.validation.Errors
14. org.springframework.validation.BindingResult
15. 命令或表单对象
16. org.springframework.web.bind.support.SessionStatus
17. org.springframework.web.util.UriComponentsBuilder
18. 带 @PathVariable @MatrixVariable 注释的对象
19. @RequestParam @RequestHeader @RequestBody @RequestPart

# 编写请求处理方法

## ❖ 请求处理方法可以返回的对象类型

ModelAndView

Model

Map 包含模型的属性

View

String 代表逻辑视图名的 String

void

提供对 Servlet 的访问，以响应 HTTP 头部和内容

HttpEntity 或 ResponseEntity 对象

Callable

DeferredResult

其他任意类型 Spring 将其视作输出给 View 的对象模型

# 应用基于注解的控制器

## ❖ 配置文件 springmvc-servlet.xml 片段

```
<context:component-scan base-package="controller"/>
<mvc:annotation-driven/>
<mvc:resources mapping="/css/**" location="/css/"/>
<mvc:resources mapping="/*.html" location="/"/>
```

- <component-scan/> 元素指示 Spring MVC 扫描目标包中的类；
- <annotation-driven/> 元素[1] 注册用于支持基于注解的控制器的请求处理方法的 bean 对象；
- <resources/> 元素指示 Spring MVC 哪些静态资源不通过 DispatcherServlet，而是需要单独处理。

[1]注：如果没有 <annotation-driven/>，<resources/> 元素会阻止任意控制器被调用。若不需要使用 resources，则不需要 <annotation-driven/> 元素。

# 应用基于注解的控制器

❖ 控制器类 ProductController 包含两个请求处理方法

```
@Controller
public class ProductController {
  private static final Log logger = LogFactory.
    getLog(ProductController.class);

  @RequestMapping(value= "/input-product")
  public String inputProduct() {
    logger.info("inputProduct␣called");
    return "ProductForm";
  }
}
// 接下页
```

# 应用基于注解的控制器

```
@RequestMapping(value = "/save-product")
public String saveProduct(ProductForm productForm, Model model) {
  logger.info("saveProduct␣called");
  // no need to create and instantiate a ProductForm
  // create Product
  Product product = new Product();
  product.setName(productForm.getName());
  product.setDescription(productForm.getDescription());
  try {
    product.setPrice(new BigDecimal(productForm.getPrice()));
  } catch (NumberFormatException e) {
  }
  // add product
  model.addAttribute("product", product); // tag01
  return "ProductDetails";
}
```

注：saveProduct 方法的 org.springframework.ui.Model 类型参数。无论是否会使用，Spring MVC 都会在每一个请求处理方法被调用时创建一个 Model 实例，用于维护需要显示在视图中的属性。如 tag01，Product 实例可以像被添加到 HttpServletRequest 中那样访问。

# 应用 @Autowired 和 @Service 进行依赖注入

org.springfamework.beans.factory.annotation.Autowired

- 将依赖注入到 Spring MVC 控制器的最简单方法是通过注解 @Autowired 到字段或方法。

org.springframework.stereotype.Service

- 为了能被作为依赖注入，类必须要声明为 @Service 注解类型。
- @Service 注解类型指示类是一个服务。
- 配置文件中需要添加一个 <component-scan/> 元素来扫描依赖包：

```
<context:component-scan base-package="dependencyPackage"/>
```

# 应用 @Autowired 和 @Service 进行依赖注入

## ❖ 应用 @Autowired 和 @Service 进行依赖注入示例

```
@Controller
public class ProductController {
  ...
  @Autowired
  private ProductService productService; // tag01
  ...
  @RequestMapping(value = "/save-product", method = RequestMethod.POST)
  public String saveProduct(ProductForm productForm, RedirectAttributes redirectAttributes) {
    Product product = new Product();
    product.setName(productForm.getName());
    product.setDescription(productForm.getDescription());
    try {
      product.setPrice(new BigDecimal(productForm.getPrice()));
    } catch (NumberFormatException e) {
    }
    // add product
    Product savedProduct = productService.add(product); // tag02
    redirectAttributes.addFlashAttribute("message", "The␣product␣was␣successfully␣added.");
    return "redirect:/view-product/" + savedProduct.getId();
  }
```

# 重定向和 Flash 属性

在前序示例 ProductController 类中的 saveProduct 方法以如下所示的行结束：

```
return "redirect:/view-product/" + savedProduct.getId();
```

此处使用重定向至安全页面（URL 改变），而不是转发（URL 不变）来防止当用户重新加载页面时 saveProduct 被二次调用。

## ❖ 重定向页面传值

- 由于重定向经过客户端，无法轻松地传值给目标页面；而采用转发则可以简单地将属性添加到 Model 供目标视图访问。
- Spring（3.1 以及更高版本）通过 Flash 属性提供了一种供重定向传值的方法。
- 要使用 Flash 属性，必须在 Spring MVC 配置文件中有一个 <annotation-driven/> 元素；必须在方法上添加一个类型 org.springframework.web.servlet.mvc.support.RedirectAttributes 参数。

# 请求参数和路径变量

请求参数和路径变量都可以用于发送值给服务器，二者都是 URL 的一部分。

## ❖ 操作请求参数

请求参数采用 key=value 形式，并用“&”分隔。

```
1  http://localhost:8080/myapp/retrieve-product?productId=3
```

传统的 Servlet 编程使用 HttpServletRequest 的 getParameter 方法来获取一个请求参数值。

# 请求参数和路径变量

## ❖ Spring MVC 获取请求参数值

通过使用 org.springframework.web.bind.annotation.RequestParam 注解类型来注解方法参数。
如下方法包含了一个获取请求参数 productld 值的参数：

```
public void sendProduct(@RequestParam int productld) {
 ...
}
```

@RequestParam 注解的参数类型不一定是字符串。

# 请求参数和路径变量

## ❖ 操作路径变量

```
@RequestMapping(value = "/view-product/{id}")
public String viewProduct(@PathVariable Long id, Model model) {
  Product product = productService.get(id);
  model.addAttribute("product", product);
  return "ProductView";
}
```

- 需要在 RequestMapping 注解的值属性中添加一个变量，该变量必须放在花括号之间。例如，RequestMapping 注解定义了一个名为 id 的路径变量。
- 在方法参数列表中添加一个同名变量，并加上 @PathVariable 注解。
- 可以在请求映射中使用多个路径变量。

```
@RequestMapping(value = "/view-product/{userId}/{orderId}")
```

# @ModelAttribute

- Spring MVC 在每次调用请求处理方法时，都会创建 Model 类型的一个实例，若使用该实例则可以在方法中添加一个 Model 类型的参数。
- 还可以使用在方法中添加 `org.springframework.web.bind.annotation.ModelAttribute` 注解类型来访问 Model 实例。
- 带 @ModelAttribute 注解的方法会将其输入的或创建的参数对象添加到 Model 对象中（若方法中没有显式添加）。

# @ModelAttribute

## ❖ @ModelAttribute 注解方法的参数

Spring MVC 将在每次调用 submitOrder 方法时创建一个 Order 实例。

```
@RequestMapping(method = RequestMethod.POST)
public String submitOrder(@ModelAttribute("newOrder") Order order, Model model) {
  ...
}
```

输入或创建的 Order 实例将用 newOrder 键值添加到 Model 对象中。如果未定义键值名，则将使用该对象类型的名称。

# @ModelAttribute

## ❖ @ModelAttribute 注解方法

- @ModelAttribute 的第二个用途是标注一个非请求的处理方法。
- 被 @ModelAttribute 注释的方法会在每次调用该控制器类的请求处理方法时被调用。这意味着，如果一个控制器类有两个请求处理方法，以及有一个 @ModelAttribute 注解的方法，Spring MVC 会在调用请求处理方法之前调用带 @ModelAttribute 注解的方法。

# @ModelAttribute

带 @ModelAttribute 注解的方法可以返回一个对象或一个 void 类型：

- 方法返回一个对象，则返回对象会自动添加到 Model 中。

```
@ModelAttribute
public Product addProduct(@RequestParam String productId) {
  return productService.get(productId);
}
```

- 方法返回 void，则还必须添加一个 Model 类型的参数，并自行将实例添加到 Model 中。

```
@ModelAttribute
public void populateModel(@RequestParam String id, Model model) {
  model.addAttribute(new Account(id));
}
```

# 接下来…

Java Web 应用的开发演化

MVC 模式示例

Spring MVC

数据绑定和表单标签库

# 数据绑定

- 数据绑定是将用户输入绑定到领域模型的一种特性。
  有了数据绑定，类型总是为 String 的 HTTP 请求参数，可用于填充不同类型的对象属性。数据绑定使得 Form Bean(如 ProductForm) 变成多余。
- 为了高效地使用数据绑定，还需要 Spring 的表单标签库支持。

# 数据绑定示例

## 未采用数据绑定

```
@RequestMapping(value="save-product")
public String saveProduct(ProductForm productForm, Model model) (
  logger.info("saveProduct called");
  Product product = new Product();
  product.setName(productForm.getName()) ;
  product.setDescription(productForm.getDescription());
  try {
    product.setPrice(Float.parseFloat(productForm.getPrice()));
  } catch (NurnberFormatException e) {
  }
```

需要解析 ProductForm 的 price 属性，将 HTTP String 类型的请求参数转换为 float 以填充 Product 的 price 属性。

## 采用数据绑定

```
@RequestMapping(value="save-product")
public String saveProduct(Product product, Model model)
```

# 表单标签库

- 表单标签库中包含了可以用在 JSP 页面中渲染 HTML 元素的标签。
- 为了使用这些标签，必须在 JSP 页面的开头处声明 taglib 指令。

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

# 表单标签库

## ❖ 表单标签库中的标签

| 标签 | 描述 |
| --- | --- |
| form | 渲染表单元素 |
| input | 渲染 <input type=”text” /> 元素 |
| password | 渲染 <input type=”password” /> 元素 |
| textarea | 渲染 textarea 元素 |
| select | 渲染一个选择元素 |
| checkbox | 渲染一个 <input type=”checkbox” /> 元素 |
| ... | ... |
| errors | 在 span 元素中渲染字段错误 |

# 表单标签库

## ❖ form 标签

```
<form:form commandName="book" action="save-book" method="post">
</form:form>
```

### form 标签的属性

| 属性 | 描述 |
| --- | --- |
| **commandName** | 暴露表单对象模型属性的名称，默认为 command |
| cssClass | 定义渲染 form 元素的 CSS 类 |
| cssStyle | 定义渲染 form 元素的 CSS 样式 |
| htmlEscape | 接受 true 或者 false，表示被渲染的值是否应该进行 HTML 转义 |
| modelAttribute | 暴露 form backing object 的模型属性名称，默认为 command |
| acceptCharset | 定义服务器接受的字符编码列表 |

# 表单标签库

## ❖ form 标签示例

BookAddForm.jsp

```
<form:form commandName="book" action="save-book" method="post">
</form:form>
```

BookController.java

```
@RequestMapping(value = "/input-book")
public String inputBook(Model model) {
  ...
  model.addAttribute("book", new Book());

  return "BookAddForm";
}
```

代码中创建了一个 Book 对象，并添加到 Model。如果没有 Model 属性，BookAddForm.jsp 页面就会抛出异常，因为表单标签无法找到在其 commandName 属性中指定的 form backing Object。

# 表单标签库

## ❖ input 标签

input 标签最重要的属性是**path**，它将这个输入字段绑定到 form backing object 的一个属性。

例如，若 <form> 标签的 commandName 属性值为 book，且 input 标签的 path 属性值为 isbn，则 input 标签将被绑定到 book 对象的 isbn 属性。

```
<form:input id="isbn" path="isbn" cssErrorClass ="errorBox" / >
```

这个 input 标签被绑定到 form backing object 的 isbn 属性。

# 表单标签库

## ❖ textarea 标签

```
<form:textarea path="note" tabindex="4" rows="5" cols="80"/>
```

以上 textarea 标签被绑定到 form backing object 的 note 属性。

☛ 其他标签请自行学习

# 数据绑定示例

☞ springmvc-databinding-01

https://github.com/pauldeck/springmvc-2ed/tree/master/chapter-05

# THE END

wangxiaodong@ouc.edu.cn