Spring 的帝国

之

SpringMVC

叩丁狼教育：任小龙

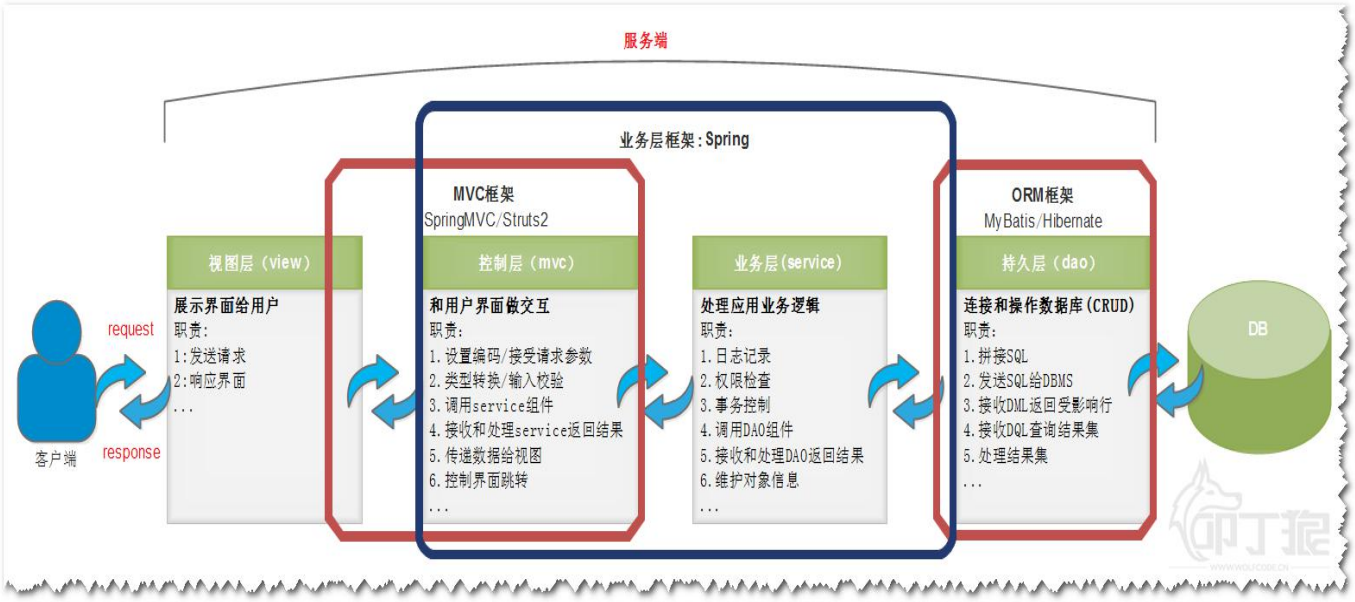
春水初生，春林初盛，春风十里，不如你！——冯唐

1. 动手写 MVC 框架

1.1. MVC 思想

1.1.1. 三层架构

Web 开发的最佳实践就是根据功能职责的不同，划分为控制层、业务层、持久层。



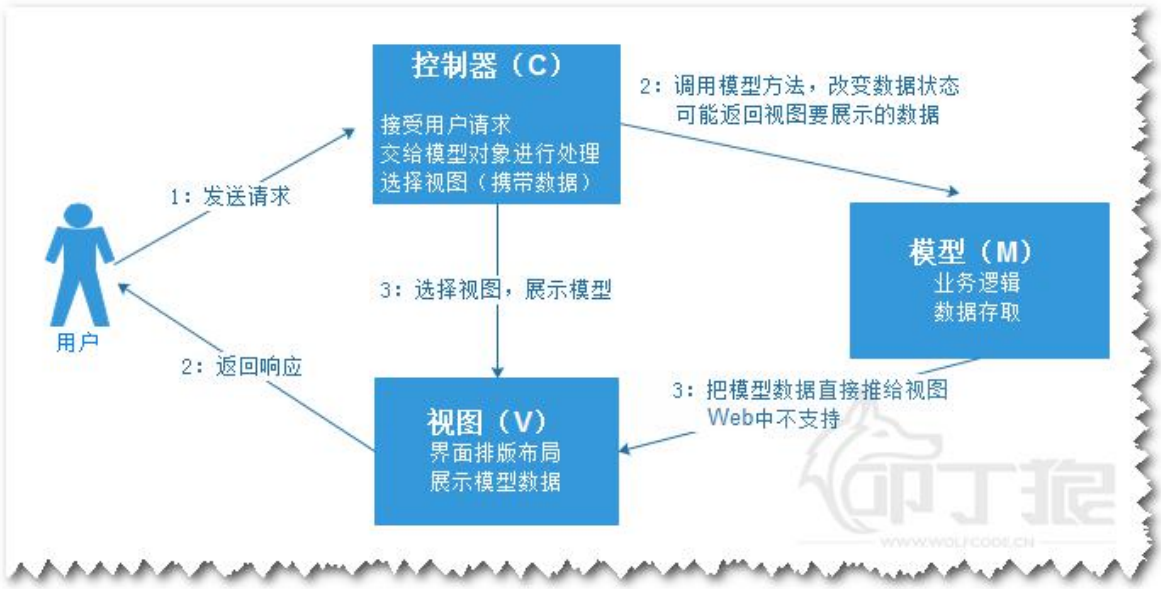
1.1.2. MVC 原理

MVC 模型：是一种架构型的模式，本身不引入新功能，只是帮助我们将开发的结构组织的更加合理，使展示与模型分离、流程控制逻辑、业务逻辑调用与展示逻辑分离------>责任分离。

Model（模型）：数据模型，包含要展示的数据和业务功能。

View（视图）：用户界面，在界面上显示模型数据。

Controller（控制器）：起调度作用，接收用户请求、调用业务处理请求、共享模型数据并跳转界面。



在 CS 架构标准的 MVC 中模型能主动推数据给视图进行更新（观察者设计模式，在模型上注册视图，当模型更新时自动更新视图）。

但在 Web 开发中模型是无法主动推给视图(无法主动更新用户界面）,因为在 Web 开发是请求-响应模型。

Front Controller 模式要求在 WEB 应用系统的前端（Front）设置一个入口控制器（Controller），所有的 request 请求都被发往该控制器统一处理，处理所有请求共同的操作。

学习 MVC 框架: 都得先配置前端控制器。

1.1.3. MVC 框架功能

MVC 框架的功能作用（WEB 开发常见功能）：

MVC 令程序开发有章可循，但是表现层的困惑来了，因为需要处理的功能太多：

设置请求编码、接受请求参数、输入校验、参数类型转换、把参数封装成对象、

文件上传、处理响应、国际化处理、自定义标签等。

1.2. 前端控制器

1.2.1. 什么是前端控制器

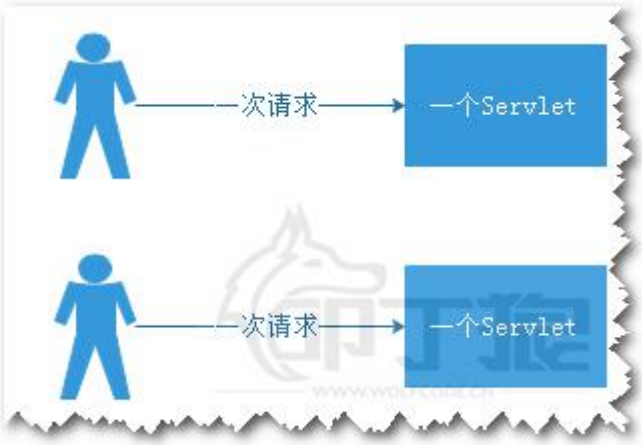
Front Controller 模式要求在 WEB 应用系统的前端（Front）设置一个入口控制器（Controller），

是用来提供一个集中的请求处理机制，所有的请求都被发往该控制器统一处理，然后把请求分发给各自相

应的处理程序。

一般用来做一个共同的处理，如权限检查，授权，日志记录等。因为前端控制的集中处理请求的能力，因此提高了可重用性和可拓展性。

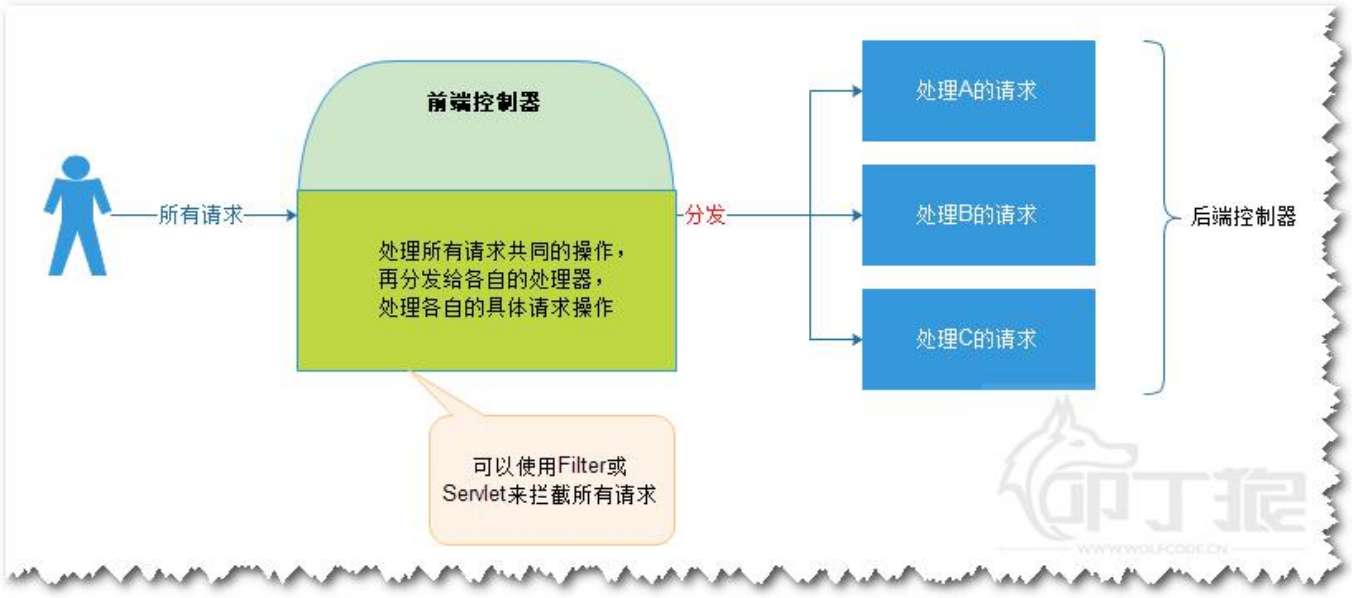
没有前端控制器前：



存在问题：如果每次请求需要做相同处理操作，此时每隔 Servlet 都要重复处理。

1.2.2.前端控制器原理

有了前端控制器：



一般的，我们把处理请求的对象称之为处理器。

Apache 习惯称之为 Action，如 EmployeeAction。

Spring 习惯称之为 Controller，如 EmployeeController。

从这里能看出使用 MVC 框架必须在 web.xml 中配置前端控制器，一般的要么是要 Filter，要么是 Servlet。

Struts2 基于 Filter、SpringMVC 基于 Servlet。

1.3. EasyMVC

1.3.1.注解

用来标注在处理器类上。

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Controller {
}
```

用来标注在处理器方法上。

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface RequestMapping {
  String value();
}
```

1.3.2.模型对象

封装每一个处理请求的方法：

```
public class ControllerBean {
  private Class<?> controllerClass;
  private Method controllerMethod;

  public ControllerBean(Class<?> controllerClass, Method controllerMethod) {
    this.controllerClass = controllerClass;
    this.controllerMethod = controllerMethod;
  }

  public Class<?> getControllerClass() {
    return controllerClass;
  }

  public Method getControllerMethod() {
    return controllerMethod;
  }
}
```

用来封装结果的模型数据和结果视图。

```
public class ModelAndView {
```

```
  private String viewName;
  private Map<String, Object> model = new HashMap<>();

  public void setViewName(String viewName) {
    this.viewName = viewName;
  }

  public ModelAndView addObject(String attributeName, Object attributeValue) {
    this.model.put(attributeName, attributeValue);
    return this;
  }

  public String getViewName() {
    return viewName;
  }

  public Map<String, Object> getModel() {
    return model;
  }
}
```

1.3.3.工具类

操作字节码对象工具类：

```
public class ClassUtil {
  // 获取指定包(包括子包)中指定注解的所有类并返回
  public static List<Class<?>> getClassListByAnnotation(String packageName,
      Class<? extends Annotation> annotationClass) {
    List<Class<?>> classList = new ArrayList<Class<?>>();
    try {
      Enumeration<URL> urls = Thread.currentThread().getContextClassLoader()
          .getResources(packageName.replaceAll("\\.", "/"));
      while (urls.hasMoreElements()) {
        URL url = urls.nextElement();
        if (url != null) {
          String protocol = url.getProtocol();
          if (protocol.equals("file")) {
            String packagePath = url.getPath();
            addClassByAnnotation(classList, packagePath, packageName, annotationClass);
          } else if (protocol.equals("jar")) {
            JarURLConnection jarURLConnection = (JarURLConnection) url.openConnection();
            JarFile jarFile = jarURLConnection.getJarFile();
            Enumeration<JarEntry> jarEntries = jarFile.entries();
            while (jarEntries.hasMoreElements()) {
```

```
                JarEntry jarEntry = jarEntries.nextElement();
                String jarEntryName = jarEntry.getName();
                if (jarEntryName.endsWith(".class")) {
                  String          className          =          jarEntryName.substring(0,
jarEntryName.lastIndexOf("."))
                      .replaceAll("/", ".");
                  Class<?> cls = Class.forName(className);
                  if (cls.isAnnotationPresent(annotationClass)) {
                    classList.add(cls);
                  }
                }
              }
            }
          }
        } catch (Exception e) {
          e.printStackTrace();
        }
        return classList;
      }

      private static void addClassByAnnotation(List<Class<?>> classList, String packagePath,
String packageName,
          Class<? extends Annotation> annotationClass) {
        try {
          File[] files = getClassFiles(packagePath);
          if (files != null) {
            for (File file : files) {
              String fileName = file.getName();
              if (file.isFile()) {
                String className = getClassName(packageName, fileName);
                Class<?> cls = Class.forName(className);
                if (cls.isAnnotationPresent(annotationClass)) {
                  classList.add(cls);
                }
              } else {
                String subPackagePath = getSubPackagePath(packagePath, fileName);
                String subPackageName = getSubPackageName(packageName, fileName);
                addClassByAnnotation(classList,       subPackagePath,       subPackageName,
annotationClass);
              }
            }
          }
        } catch (Exception e) {
```

```
            e.printStackTrace();
          }
        }

        private static File[] getClassFiles(String packagePath) {
          return new File(packagePath).listFiles(new FileFilter() {
            public boolean accept(File file) {
              return     (file.isFile()     &&     file.getName().endsWith(".class"))     ||
file.isDirectory();
            }
          });
        }

        private static String getClassName(String packageName, String fileName) {
          String className = fileName.substring(0, fileName.lastIndexOf("."));
          if (hasLength(packageName)) {
            className = packageName + "." + className;
          }
          return className;
        }

        private static String getSubPackagePath(String packagePath, String filePath) {
          String subPackagePath = filePath;
          if (hasLength(packagePath)) {
            subPackagePath = packagePath + "/" + subPackagePath;
          }
          return subPackagePath;
        }

        private static String getSubPackageName(String packageName, String filePath) {
          String subPackageName = filePath;
          if (hasLength(packageName)) {
            subPackageName = packageName + "." + subPackageName;
          }
          return subPackageName;
        }

        public static boolean hasLength(String str) {
          return str != null && !"".equals(str.trim());
        }

        public static boolean isEmpty(String str) {
          return !hasLength(str);
        }
```

```
}
```

EasyMVC 工具类（可以最后抽取）：

```
public class WebUtil {
  public static Properties prop = new Properties();
  static {
    try {
       Properties prop = new Properties();
       prop.load(Thread.currentThread().getContextClassLoader()
              .getResourceAsStream("mvc.properties"));
    } catch (IOException e) {
       e.printStackTrace();
    }
  }

  //每一个 url 对应一个 Controller 方法
  public static Map<String, ControllerBean> urlMap = new HashMap<>();

  //初始化应用
  public static void initWebApp() {
    List<Class<?>> classList = ClassUtil.getClassListByAnnotation(
              prop.getProperty("mvc.scanPackags"),Controller.class);
    for (Class<?> clazz : classList) {
       Method[] controllerMethods = clazz.getDeclaredMethods();
       if (controllerMethods != null && controllerMethods.length > 0) {
         for (Method controllerMethod : controllerMethods) {
           RequestMapping rm = controllerMethod.getAnnotation(RequestMapping.class);
           if (rm != null) {
             String url = rm.value();
             urlMap.put(url, new ControllerBean(clazz, controllerMethod));
           }
         }
       }
    }
  }

  //处理结果
  public static void handleResult(HttpServletRequest request,
      HttpServletResponse response, ModelAndView mv) throws Exception {
    String path = prop.getProperty("mvc.view_perfix") +
                  mv.getViewName() + prop.getProperty("mvc.view_suffix");
    Map<String, Object> attrs = mv.getModel();
    for (Map.Entry<String, Object> entry : attrs.entrySet()) {
      request.setAttribute(entry.getKey(), entry.getValue());
    }
```

```
    request.getRequestDispatcher(path).forward(request, response);
  }

  public static ControllerBean getControllerBeanByUrl(String url) {
    return urlMap.get(url);
  }
}
```

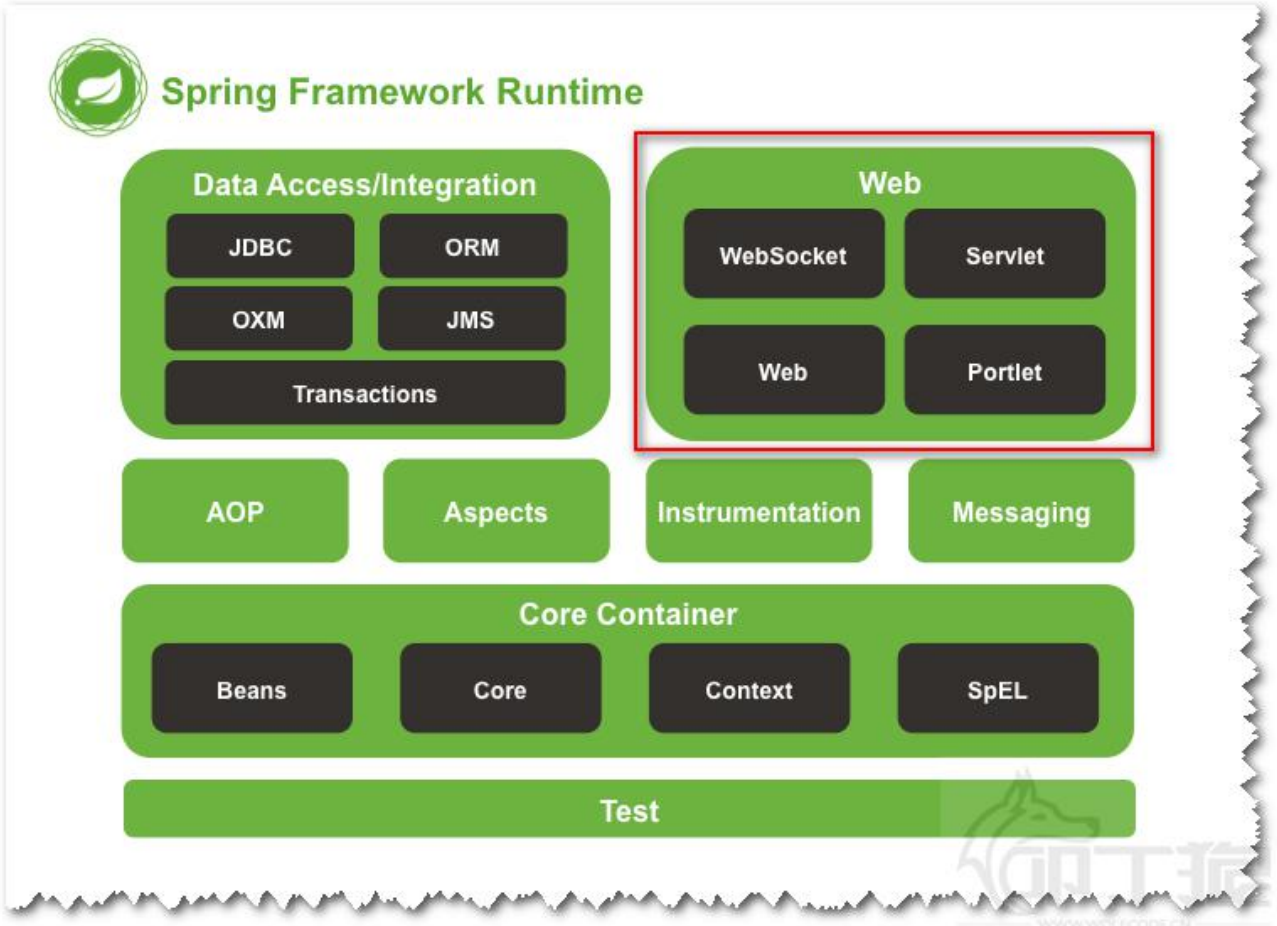
1.3.4.前端控制器

```
public class DispatcherServlet extends HttpServlet {
  private static final long serialVersionUID = 1L;

  public void init() throws ServletException {
    WebUtil.initWebApp();
  }

  protected void service(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    // 获取当前请求相关数据
    String requestURL = request.getRequestURI();
    if (requestURL.equals("/favicon.ico")) {
      return;
    }

    if (requestURL != null && requestURL.contains(".")) {
      requestURL = requestURL.substring(0, requestURL.lastIndexOf("."));
    }
    //==========================================================================
    ControllerBean controllerBean = WebUtil.getControllerBeanByUrl(requestURL);
    if (controllerBean == null) {
      return;
    }
    Class<?> controllerClass = controllerBean.getControllerClass();
    Method controllerMethod = controllerBean.getControllerMethod();
    Object ret = null;
    try {
      Object controllerObject = controllerClass.newInstance();
      ret = controllerMethod.invoke(controllerObject);
      if (ret != null && ret.getClass() == ModelAndView.class) {
        WebUtil.handleResult(request, response, (ModelAndView) ret);
      }
    } catch (Exception e) {
      e.printStackTrace();
```

```
        }

    }
}
```

2. 走进 SpringMVC

2.1. Spring Web 框架



2.1.1.SpringMVC

什么是 Spring MVC？

MVC 框架，它解决 WEB 开发中常见的问题(参数接收、文件上传、表单验证、国际化、等等)，而且使用简单，与 Spring 无缝集成。

Spring3.0 后全面超越 Struts2，成为最优秀的 MVC 框架 （更安全，性能更好，更简单）。

支持 RESTful 风格的 URL 请求 ，非常容易与其他视图技术集成，如 Velocity、FreeMarker。

采用了松散耦合可插拔组件结构，比其他 MVC 框架更具扩展性和灵活性。

2.1.2. Spring WebFlux

Spring WebFlux 是 Spring 另一个 Web 框架，基于异步非阻塞的。

SpringMVC 是同步阻塞的 IO 模型，资源浪费相对来说比较严重，当我们在处理一个比较耗时的任务时，比如上传文件，服务器的线程一直在等待接收文件，而 Spring WebFlux 可以做到异步非阻塞。

看文档

2.1.3. SpringMVC 和 Struts2

SpringMVC 和 Struts2 对比：

①、Spring MVC 的前端控制器是 Servlet， 而 Struts2 是 Filter。

②、Spring MVC 会稍微比 Struts2 快些， Spring MVC 是基于方法设计 ,处理器是单例 ,而 Sturts2 是基于类，每次发一次请求都会实例一个 新的 Action 对象，Action 是多例的。

③、Spring MVC 更加简洁，开发效率 Spring MVC 比 Struts2 高，如支持 JSR303 校验，且处理 AJAX 请求更方便。

④、Struts2 的 OGNL 表达式使页面的开发效率相比 Spring MVC 更高些，但是 Spring MVC 也不差。

2.2. 第一个程序

准备环境：

搭建 Web 项目、拷贝依赖的 jar

开发步骤：

①：配置前端控制器：DispatcherServlet

②：配置处理器映射器：BeanNameUrlHandlerMapping

③：配置处理器适配器：SimpleControllerHandlerAdapter

④：配置视图解析器：InternalResourceViewResolver

⑤：开发和配置处理器：HelloController

2.2.1.环境准备

依赖的 jar:

依赖的 jar：

1、添加 Spring 的核心包

com.springsource.org.apache.commons.logging-1.1.1.jar

spring-core-5.x.x.RELEASE.jar

spring-beans-5.x.x.RELEASE.jar

spring-context-5.x.x.RELEASE.jar

spring-expression-5.x.x.RELEASE.jar

2、添加 SpringMVC 的核心包

spring-web-5.x.x.RELEASE.jar spring 对 web 项目的支持

spring-webmvc-5.x.x.RELEASE.jar spring mvc 核心包

在 web.xml 中，配置前端控制器（初始化 Spring 容器）：

```
<servlet>
    <servlet-name>springMVC</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:mvc.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>springMVC</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

2.2.2.Java 代码

先在控制台打印，再跳转到 JSP 页面。

```
public class HelloController implements Controller {
  public ModelAndView handleRequest(HttpServletRequest request,
          HttpServletResponse response) throws Exception {
    System.out.println("hello.....");
    ModelAndView mv = new ModelAndView();
    mv.addObject("helloMsg", "你好 SpringMVC");
    mv.setViewName("/WEB-INF/views/hello/welcome.jsp");
    return mv;
  }
}
```

此时处理器类习惯以 Controller 结尾，并要求必须实现于 Controller 接口。

2.2.3.XML 配置

mvc.xml 文件：

```
<!--1:处理器映射器 -->
<!-- 处理器映射器将处理器(handler)的 name 作为 URL 查找 -->
<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping" />
<!--2:处理器适配器 -->
<!--
    ①:所有的适配器都需要实现于 HandlerAdapter 接口
    ②:要求所有的处理器需要实现于 Controller 接口
 -->
<bean class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter" />
<!--3:视图解析器 -->
<!-- 缺省使用 JSTL -->
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver" />

<!--4:处理器 -->
<bean id="/hello" class="cn.wolfcode.hello.HelloController" />
```

2.2.4.第一个程序小结

web.xml 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>classpath:mvc.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>springMVC</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

- load-on-startup 元素：

```
<load-on-startup>1</load-on-startup>
```

load-on-startup 元素是可选的，如果存在，则随着 Web 服务器的启动而构建并调用 init 方法做初始化操作，如不存在，则在第一次请求该 Servlet 才加载。

- url-pattern 元素（后面详细讲）：

配置如 *.do、*.htm 是最传统方式，不会导致静态文件（jpg, js, css）被拦截，但不支持 RESTful 风格。

配置成 /，可以支持流行的 RESTfull 风格，但会导致静态文件（jpg、 js、 css）被拦截后不能正常显示。

配置成 /*，是错误的方式，可以请求到 Controller 中，但调转到 jsp 时被拦截，不能渲染 jsp 视图。

- init-param 元素：

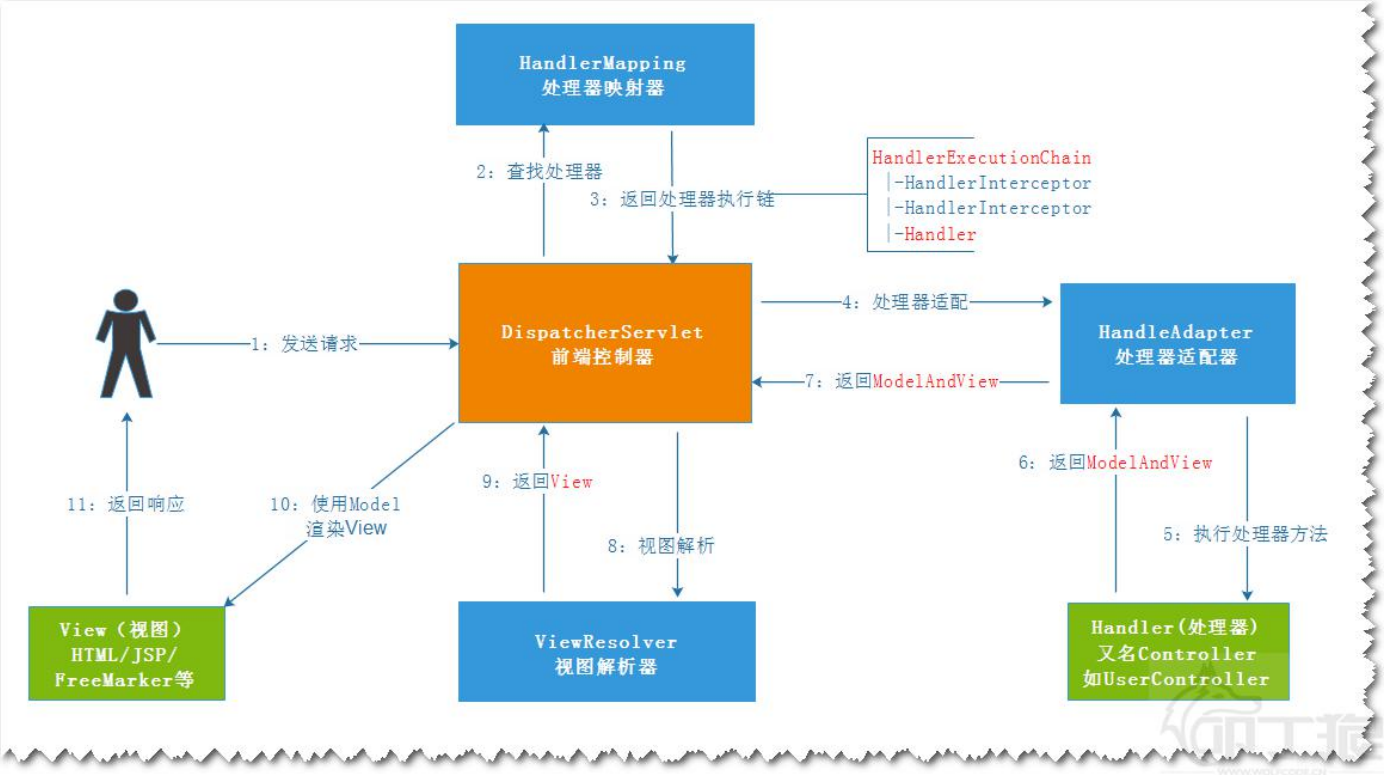
```
<init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:mvc.xml</param-value>
</init-param>
```

DispatcherServlet 默认会去 WEB-INF 目录下去按照 *servletName*-servlet.xml 方式去加载 SpringMVC 配置文件。如 *servlet-name* 元素文本内容是 hello，则去 WEB-INF 目录下寻找 hello-servlet.xml 文件。一般的，我们从 classpath 路径，显示去加载 SpringMVC 的配置文件。

- 多个 Controller 的请求 URL 必须保证唯一性。

2.3. SpringMVC 执行流程

2.3.1.流程分析



SpringMVC 流程：

01、用户发送出请求到前端控制器 DispatcherServlet。

02、DispatcherServlet 收到请求调用 HandlerMapping（处理器映射器）。

03、HandlerMapping 找到具体的处理器(通过 XML 或注解配置)，生成处理器对象及处理器拦截器(如果有)，再一起返回给 DispatcherServlet。

04、DispatcherServlet 调用 HandlerAdapter（处理器适配器）。

05、HandlerAdapter 经过适配调用具体的处理器的某个方法（Handler/Controller）。

06、Controller 执行完成返回 ModelAndView 对象。

07、HandlerAdapter 将 Controller 返回的 ModelAndView 再返回给 DispatcherServlet。

08、DispatcherServlet 将 ModelAndView 传给 ViewReslover（视图解析器）。

09、ViewReslover 解析后返回具体 View（视图）。

10、DispatcherServlet 根据 View 进行渲染视图（即将模型数据填充至视图中）。

11、DispatcherServlet 响应用户。

2.3.2.涉及组件分析

1、前端控制器 DispatcherServlet（不需开发），由框架提供，在 web.xml 中配置。

作用：接收请求，响应结果，相当于转发器，中央处理器。

2、处理器映射器 HandlerMapping(不需开发),由框架提供。

作用：根据请求的 url 查找 Handler(处理器)，可以通过 XML 和注解方式来映射。

3、处理器适配器 HandlerAdapter(不需开发)，由框架提供。

作用：按照特定规则（HandlerAdapter 要求的规则）去执行 Handler。

4、处理器 Handler(也称之为 Controller，需要开发)

注意：编写 Handler 时按照 HandlerAdapter 的要求去做，这样适配器才可以去正确执行 Handler。

作用：接受用户请求信息，调用业务方法处理请求，也称之为后端控制器。

5、视图解析器 ViewResolver(不需开发)，由框架提供

作用：进行视图解析，把逻辑视图名解析成真正的物理视图。

SpringMVC 框架支持多种 View 视图技术，包括：JstlView、FreemarkerView、pdfView 等。

6、视图 View(需要开发)

作用：把数据展现给用户的页面

View 是一个接口，实现类支持不同的 View 技术（jsp、freemarker、pdf 等）

使用 Debug 断点调试分析流程。

2.4. 使用注解开发

使用基于注解的 Controller 的好处：

1、一个 Controller 类可以有多个处理方法。

2、请求映射不需要在 XML 中配置，开发效率更高。

DispatcherServlet.properties 文件（部分）：

```
org.springframework.web.servlet.HandlerMapping=org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping,\
   org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping

org.springframework.web.servlet.HandlerAdapter=org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter,\
   org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,\
   org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter

org.springframework.web.servlet.ViewResolver=org.springframework.web.servlet.view.InternalResourceViewResolver
```

在 spring3.1 之前使用：

注解映射器：org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping

注解适配器：org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter

在 spring3.1 之后使用：

注解映射器：org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping

注解适配器：org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter

2.4.1. Java 代码

```
@Controller
public class HelloController {

  @RequestMapping("/sayHello")
  public ModelAndView sayHello() {
    System.out.println("hello.....");
    ModelAndView mv = new ModelAndView();
    mv.addObject("helloMsg", "你好 SpringMVC");
    mv.setViewName("/WEB-INF/views/hello/welcome.jsp");
    return mv;
  }
}
```

2.4.2.Spring 版型注解回顾

Java EE 应用三层架构：

控制层(mvc)	@Controller
业务逻辑层(Service)	@Service
数据持久层(DAO)	@Resposotory
其他组件使用通用注解	@Component

2.4.3.SpringMVC 注解

MVC 注解解析器：

<mvc:annotation-driver>

会自动注册 RequestMappingHandlerMapping、RequestMappingHandlerAdapter、ExceptionHandlerExceptionResolver 三个 bean。

除此之外，还支持：

支持使用 ConversionService 实例对表单参数进行类型转换.

支持使用 @NumberFormat、@DateTimeFormat 注解完成数据格式化操作.

支持使用 @Valid 注解对 JavaBean 实例进行 JSR303 验证.

支持使用 @RequestBody 和@ResponseBody 注解读写 JSON.

RequestMapping 注解

RequestMapping 是一个用来处理请求地址映射的注解，可用于类或方法上。用于类上，表示类中的所有响应请求的方法都是以该地址作为父路径。RequestMapping 注解有六个属性，分成三类进行说明。

1、 value， method；

value： 指定请求的 URL；

method： 指定请求的 method 类型， GET、POST、PUT、DELETE 等；

窄化请求映射

2、 consumes，produces（MediaType 类）；

consumes： 指定处理请求的提交内容类型（Content-Type），例如 application/json, text/html;

produces: 指定返回的内容类型，如 application/json; charset=UTF-8 仅当 request 请求头中的(Accept)类型中包含该指定类型才返回；

3、 params，headers；

params： 指定 request 中必须包含某些参数值是，才让该方法处理。

headers： 指定 request 中必须包含某些指定的 header 值，才能让该方法处理请求。

2.4.4.XML 配置

```
<!-- 组件扫描 -->
<context:component-scan base-package="cn.wolfcode.hello"/>
<!--MVC 注解解析器-->
<mvc:annotation-driven/>
```

2.5. 静态资源访问

为了效果明显，可以在 web 根目录新建一个 hello.html，在里面包含一张图片。

当对前端控制器设置为拦截资源的路径（url-pattern）为/时，此时出现不能访问静态资源的问题。

导致原因：

在 Tomcat 中处理静态资源的 servlet（default）所映射路径为就是/。

在启动项目的时候，在 Tomcat 中的 web.xml 是先加载，项目的 web.xml 是后加载，如果配置了相同的映射路径，项目中的 web.xml 会覆盖 Tomcat 中 web.xml 相同的配置。

也就是说，SpringMVC 中的 DispatcherServlet 的映射路径覆盖了 Tomcat 默认对静态资源的处理的路径。此时 SpringMVC 会把静态资源当做是 Controller，寻找并访问，当然结果肯定是找不到。

2.5.1.方式一-排除法

在 web.xml 中追加需要放行的静态资源，这种方式比较 SB，不用：

```
<servlet>
    <servlet-name>default</servlet-name>
</servlet>
<servlet-mapping>
    <servlet-name>default</servlet-name>
    <url-pattern>*.html</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>default</servlet-name>
    <url-pattern>*.png</url-pattern>
</servlet-mapping>
```

2.5.2.方式二-default-servlet-handler

<mvc:default-servlet-handler/>：开发中运用比较多

将在 SpringMVC 上下文中定义一个 DefaultServletHttpRequestHandler ,它会对进入 DispatcherServlet 的请求进行筛查，如果发现是没有经过映射的请求，就将该请求交由 Tomcat 默认的 Servlet 处理，如果不是静态资源的请求，才由 DispatcherServlet 继续处理。

2.5.3.方式三-资源映射

<mvc:resources location="/" mapping="/**"/> ：开发中运用比较多

<mvc:default-servlet-handler/>将静态资源的处理经由 SpringMVC 框架交回 Web 服务器，<mvc:resources/>而更进一步，由 SpringMVC 框架自己处理静态资源，并添加一些有用的附加功能。允许静态资源放在在任何地方，如 WEB-INF 目录下，类路径下等，甚至可以将 JS 等静态文件打包到 jar 包中。通过 location 属性指定静态资源的位置，由于 location 属性是 Resource 类型，所以可以使用前缀如 classpath 等。

```
<mvc:resources location="/WEB-INF/" mapping="/**"/>
<mvc:resources location="/WEB-INF/,classpath:/static/" mapping="/**"/>
```

2.5.4.小结

url-pattern 拦截方式：

配置如 *.do、*.htm 是最传统方式，不会导致静态文件（jpg、js、css）被拦截，但不支持 RESTful 风格。

配置成 /，可以支持流行的 RESTfull 风格，但会导致静态文件（jpg、 js、 css）被拦截后不能正常显示。

配置成 /*，是错误的方式，可以请求到 Controller 中，但调转到 jsp 时被拦截，不能渲染 jsp 视图。

3. 请求和响应

Controller 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。

下面是可以在请求处理方法中出现的参数类型：

- javax.servlet.ServletRequest 或 javax.servlet.http.HttpServletRequest。
- javax.servlet.ServletResponse 或 javax.servlet.http.HttpServletResponse。
- javax.servlet.http.HttpSession。
- org.springframework.web.context.request.WebRequest 或 org.springframework.web.context.request.NativeWebRequest。
- java.util.Locale。
- java.io.InputStream 或 java.io.Reader。
- java.io.OutputStream 或 java.io.Writer。
- java.security.Principal。
- HttpEntity<?>paramters
- java.util.Map/org.springframework.ui.Model /。
- org.springframework.ui.ModelMap。
- org.springframework.web.servlet.mvc.support.RedirectAttributes。
- org.springframework.validation.Errors /。
- org.springframework.validation.BindingResult。
- 命令或表单对象。
- org.springframework.web.bind.support.SessionStatus。
- org.springframework.web.util.UriComponentsBuilder。
- 带@PathVariable、@MatrixVariable、@RequestParam、@RequestHeader、@RequestBody或@RequestPart 注释的对象。

请求处理方法可以返回如下类型的对象：

- ModelAndView。
- Model。
- 包含模型的属性的 Map。
- View。
- 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。

3.1. 处理器方法响应处理

响应传值方式：

1）：Controller 方法的返回类型

2）：Controller 共享数据到视图页面

3.1.1.返回 void

情况一、Controller 方法的返回值类型为 void: 此时就是把 Controller 当做 Servlet 来用(SB)。

在 Controller 方法形参上可以定义 request 和 response，使用 request 或 response 指定响应结果：

①：使用 request 请求转发页面：

```
request.getRequestDispatcher("页面路径").forward(request, response);
```

②：通过 response 重定向页面：

```
response.sendRedirect("url")
```

③：通过 response 指定响应结果，例如响应 json 数据如下：

```
response.setContentType("text/json;charset=utf-8");
response.getWriter().print(JSON 字符串);
```

传统代码：

```
@RequestMapping("/test1")
  public void test1(HttpServletRequest request, HttpServletResponse response)
                throws ServletException, IOException {
    request.setAttribute("hello", "叩丁狼教育 1");
    request.setAttribute("msg", "SpringMVC");
    request.getRequestDispatcher("/WEB-INF/views/response/result.jsp")
                .forward(request, response);
  }
```

下载案例：

```
@RequestMapping("/download")
public void down(OutputStream out) throws Exception {
    Path source = Paths.get("c:/abc.rar");
    Files.copy(source, out);
}
```

但是在开发中我们经常会使用到 Servlet API 对象，此时可以直接以参数的形式传递进去，或者以 DI 方式注入。

3.1.2.返回 ModelAndView

情况二、返回 ModelAndView 类型和共享数据。

Controller 方法中定义 ModelAndView 对象并返回，对象中设置 model 数据并指定 view。

```
    @RequestMapping("/test2")
    public ModelAndView test2() {
        ModelAndView mv = new ModelAndView();
        mv.addObject("hello", "叩丁狼教育 2");
        mv.addObject("msg", "SpringMVC");
        mv.addObject("will");
        mv.addObject("lucy");
        mv.setViewName("/WEB-INF/views/response/result.jsp");
        return mv;
      }
```

ModelAndView 把数据添加到 Model 中的两类方法：

addObject(String key,Object value):设置共享数据的 key 和 value

addObject(Object value):设置共享数据的 value，key 为该 value 类型首字母小写。

每次一个方法都需要配置很长的物理视图地址，我们可以统一把视图的前缀和后缀配置起来，只需要告诉框架逻辑视图名称就可以了。

此时默认的物理视图地址为：视图前缀 + 逻辑视图名称 + 视图后缀

在 XMl 中配置视图解析器：

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

添加视图解析器后：

```
    @RequestMapping("/test2")
      public ModelAndView test2() {
        ModelAndView mv = new ModelAndView();
        mv.addObject("hello", "叩丁狼教育 2");
        mv.addObject("msg", "SpringMVC");
        mv.addObject("will");
        mv.addObject("lucy");
        mv.setViewName("response/result");
        return mv;
      }
```

3.1.3.返回 String

情况三、返回 String 类型和共享数据(使用广泛)，此时和 Model 参数组合使用。

1、返回值为 String，此时物理视图路径为：前缀+逻辑视图名+后缀

2、redirect 前缀方式，表示重定向，

相当于 response.sendRedirect()，重定向后浏览器地址栏变为重定向后的地址，不共享之前请求的数据。

3、forward 前缀方式，表示请求转发，

相当于使用 forward(request,response)方式，转发后浏览器地址栏不变，共享之前请求中的数据。

3.2. 请求跳转

3.2.1.请求转发

forward 前缀方式，表示请求转发，转发后浏览器地址栏不变，共享之前请求中的数据，相当于：

request.getRequestDispatcher(path).forward(request,response)

代码如下：

```
@RequestMapping("/test3_2")
public String test3_2(Model model) {
   return "forward:/hello.html";
}
```

3.2.2.URL 重定向

redirect 前缀方式，表示重定向，重定向后浏览器地址栏变为重定向后的地址，不共享之前请求的数据，相当于：

response.sendRedirect(path)

代码如下：

```
@RequestMapping("/test3_1")
public String test3_1(Model model) {
   return "redirect:/hello.html";
}
```

思考：hello 和 /hello 区别？

3.2.3.URL 重定向共享数据

使用请求转发可以轻松携带数据到目标页面，但是容易造成表单重复提交问题。使用重定向的好处是可以

避免表单重复提交，但是不便的是无法轻松地传值给目标页面。

Spring 从 3.1 开始通过 Flash 属性可以提供了重定向传值方式，此时参数需要在处理方法上添加一个新的参数类型 RedirectAttributes，并在处理方法中保存需要共享的数据：

```
redirectAttributes.addFlashAttribute(String name,Obejct value);
```

代码如下：

```
@RequestMapping("/a")
public String a(RedirectAttributes ra) {
    ra.addAttribute("msg1", "普通方式");
    ra.addFlashAttribute("msg2", "更好的方式");
    System.out.println("aaa...");
    return "redirect:/b";
}

@RequestMapping("/b")
public ModelAndView b(String msg1, @ModelAttribute("msg2")String msg2) {
    System.out.println("bbbb...");
    System.out.println(msg1);
    System.out.println(msg2);
    return null;
}
```

3.3. 处理器方法参数处理

一个完整的 POST 请信息：

3.3.1.Servlet API 参数

情况一、为了操作 Servlet API 对象，此时可以直接以参数形式传递，也可以直接使用 DI 注入。

```
@Autowired
 private HttpServletRequest request;
 @Autowired
 private HttpServletResponse response;
 @Autowired
 private HttpSession session;
 @Autowired
 private ServletContext servletContexct;
 @RequestMapping("/test1")
 public void test1(HttpServletRequest request) {
   System.out.println(request.getParameter("username"));
   System.out.println(this.request);
   System.out.println(this.response);
   System.out.println(this.session);
   System.out.println(this.servletContexct);
 }
```

3.3.2.简单类型参数

情况二、简单类型参数和 RequestParam 注解。

1、如果请求参数和 Controller 方法的形参同名，可以直接接收。

2、如果请求参数和 Controller 方法的形参不同名，使用@RequestParam 注解贴在形参上，设置对应的请求参数名称。

Controller 方法的形参和请求参数同名:

```
@RequestMapping("/test2")
 public ModelAndView test2(String username, int age) {
    User u = new User();
    u.setUsername(username);
    u.setAge(age);
    System.out.println(u);
    return null;
 }
```

Controller 方法的形参和请求参数不同名:

```
 @RequestMapping("/test2_2")
 public ModelAndView test2_2(@RequestParam("username") String name, int age) {
   User u = new User();
   u.setUsername(name);
   u.setAge(age);
```

```
        System.out.println(u);
        return null;
    }
```

@RequestParam 使用于参数上，用于将请求参数映射到指定参数变量上。

3.3.3. POST 请求时中文乱码处理

在 POST 请求中使用请求参数是中文，出现乱码怎么解决？

以前设置请求编码都是我们自己写：

```
request.setCharacterEncoding("UTF-8");
```

而在 SringMVC 只需在 web.xml 中配置一个编码过滤器：

```
    <filter>
      <filter-name>characterEncoding</filter-name>
      <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
      <init-param>
         <param-name>encoding</param-name>
         <param-value>UTF-8</param-value>
      </init-param>
    </filter>
    <filter-mapping>
      <filter-name>characterEncoding</filter-name>
      <url-pattern>/*</url-pattern>
    </filter-mapping>
```

3.3.4. RESTful 风格传参

情况三、支持 RESTfull

RESTful 是一种软件架构风格，严格上说是一种编码风格，其充分利用 HTTP 协议本身语义从而提供了一组设计原则和约束条件。主要用于客户端和服务器交互类的软件，该风格设计的软件可以更简洁，更有层次，更易于实现缓存等机制。

在后台使用 RequestMapping 标签后，可以类似{参数名}占位符的方式传参，此时需要通过 PathVariable 注解可以将 URL 中占位符参数绑定到控制器处理方法的形参中。

此时客户端请求地址如：http://localhost:8080/delete/3

```
    @RequestMapping("/delete/{id}")
    public ModelAndView value6(@PathVariable("id") Long id) {
        System.out.println("delete"+id);
        return null;
    }
```

3.3.5.数组和 List 类型参数

情况四、一个参数多个参数值。

接收数组类型参数：

```
@RequestMapping("/test4")
 public ModelAndView test4(Long[] ids) {
    System.out.println(Arrays.asList(ids));
    return null;
 }
```

浏览器请求 URL：/request/test4?ids=10&ids=20&ids=30

使用 List 类型封装参数，必须绑定在对象上，不能直接作为 Controller 方法参数。

```
@Getter@Setter
public class FormBean{
    private List<Long> ids= new ArrayList<Long>();
}
```

接收 List 类型参数：

```
@RequestMapping("/test5")
 public ModelAndView test5(FormBean fb) {
    System.out.println(fb);
    return null;
 }
```

浏览器请求 URL：/request/test5?ids[0]=10&ids[1]=20&ids[2]=30

总结，此时操作多参数值时使用数组会更简单，以后操作 List 集合一般都是出于对象中，不会直接操作。

3.3.6.JavaBean 类型参数

情况五、把请求信息直接封装到传参对象。

```
@Getter@Setter
public class User {
    private Long id;
    private String username;
    private int age;
}
```

能够自动把请求参数封装到 Controller 方法的形参对象中，此时请求参数必须和对象的属性同名、。

```
@RequestMapping("/test3")
public ModelAndView test3(User u) {
    System.out.println(u);
    return null;
}
```

默认会把复合类型参数共享到视图中去，共享的 key 名为参数类型首字母小写，如果要修改共享数据的 key

名，使用 ModelAttribute 注解。

3.4. ModelAttribute

ModelAttribute 注解作用：

1、给共享的数据设置 key 名，可以绑定 Controller 的方法的形参上或返回值上。

2、可以标注一个非请求的处理方法，被标注的方法在每次调用该控制器类中的请求处理方法时先被调用。

被 ModelAttribute 注解标注的方法可以返回一个对象，此时直接把返回值放入 Model 中。

也可以返回 void，此时还需要搭配 Model 对象来设置共享数据。

- 默认会把复合类型参数共享到视图中去，共享的 key 名为类型首字母小写，如果要修改共享数据的 key 名，使用 ModelAttribute 注解。

```
@RequestMapping("/test3")
public String test3( User u) {
    System.out.println(u);
    return "request/request";
 }
```

- 返回对象类型和共享数据（把当前请求的 url 作为逻辑视图名称），一般用于返回 JSON 数据。

```
@RequestMapping("/test4")
public User test4() {
    User u = new User();
    u.setId(123L);
    u.setUsername("will");
    u.setAge(17);
    return u;
}
```

此时把当前请求的 URL 作为逻辑视图名，即跳转到 test4 页面。此时共享数据的 key 为返回对象类型首字母小写，即 user。

可以通过 ModelAttribute 注解来设置 Model 共享的 key 名。

```
  @RequestMapping("/test4")
  @ModelAttribute("myUser")
  public User test4() {
  }
```

此时放在 model 中的共享数据的 key 为 myUser。

ModelAttribute 标注一个非请求的处理方法：

```
@ModelAttribute
public void doWork(Model model) {
    System.out.println("执行.................");
```

```
        model.addAttribute("..","..");
    }
```

在开发中，一般的我们怎样使用 ModelAttribute？

```
@RequestMapping("bidrequest_publishaudit_list")
public String publishAuditList(
                @ModelAttribute("qo") BidRequestQueryObject qo, Model model) {
   qo.setBidRequestState(BidConst.BIDREQUEST_STATE_PUBLISH_PENDING);
   model.addAttribute("pageInfo", bidRequestService.query(qo));
   return "bidrequest/publish_audit";
}
```

3.5. 其他请求信息

- 获取请求头信息

@RequestHeader：完成请求头（header）数据到处理器功能处理方法的方法参数上的绑定；

- 获取 Cookie 信息

@CookieValue：完成 Cookie 数据到处理器功能处理方法的方法参数上的绑定；

```
    @RequestMapping("test11")
    public ModelAndView test1(//
          @RequestHeader("User-Agent") String userAgent,//
          @CookieValue("JSESSIONID") String cookieName) {
       System.out.println("test1.................");
       System.out.println(userAgent);
       System.out.println(cookieName);
       return null;
     }
```

- 操作 HttpSession
- 默认情况下模型数据是保存到 request 作用域的，如果需要保存到 session 作用域，此时得使用 SessionAttributes 注解。SessionAttributes 注解贴在处理器类上，用于声明 HttpSession 级别存储的属性，通常列出模型属性（如 @ModelAttribute）对应的名称，则这些属性会自动保存到 session 中。

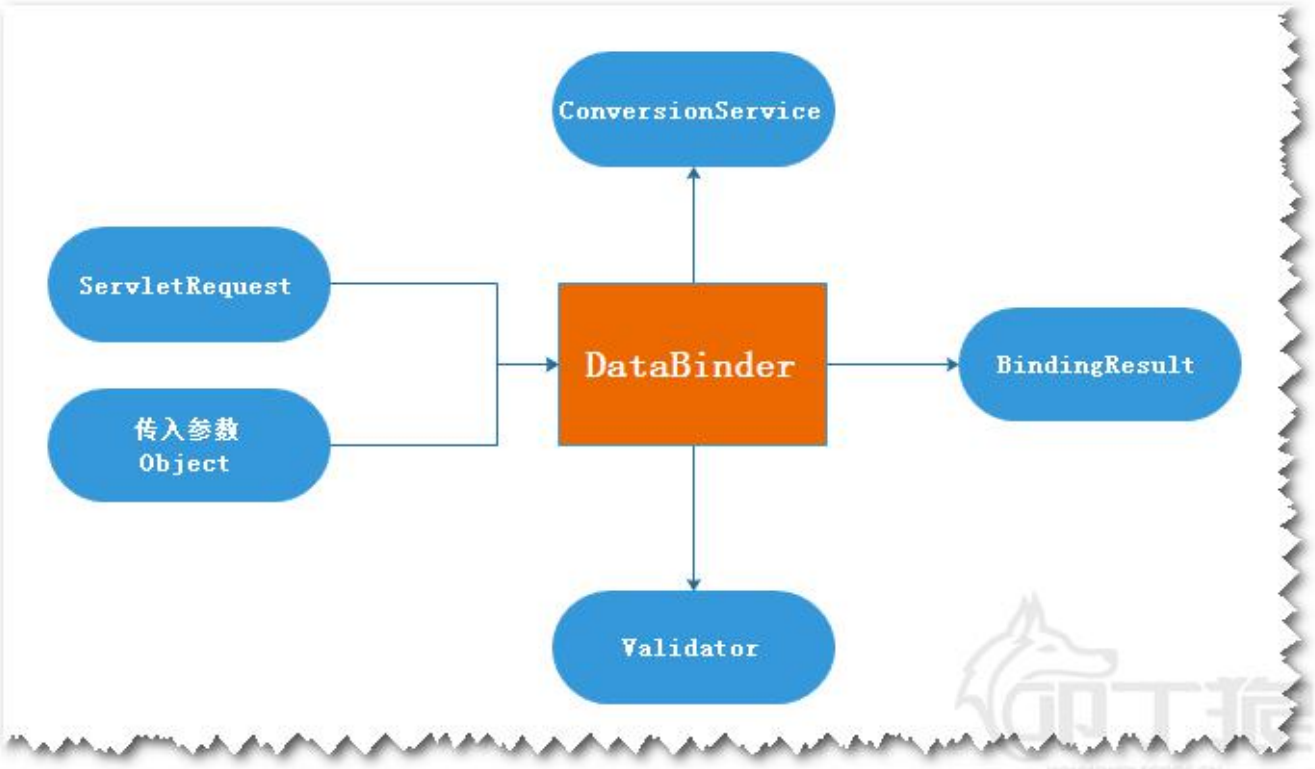
```
    @Controller
    @SessionAttributes("errorMsg")
    public class HelloController {
      @RequestMapping("/test2")
      public String test2(Model model) {
        System.out.println("test2.................");
```

```
        model.addAttribute("errorMsg", "可不可以显示");
        return "redirect:/abc.jsp";
      }
    }
```

abc.jsp 测试

```
    ${requestScope.errorMsg}
    ${sessionScope.errorMsg}
```

3.6. 数据绑定流程



SpringMVC 通过反射机制对目标处理方法的签名进行分析，将请求信息绑定到处理方法的形参中，数据绑定的核心组件是 DataBinder 类。

数据绑定流程：

1、框架把 ServletRequest 对象和请求参数传递给 DataBinder；

2、DataBinder 首先调用 Spring Web 环境中的 ConversionService 组件，进行数据类型转换和格式化等操作，将 ServletRequest 中的信息填充到形参对象中；

3、DataBinder 然后调用 Validator 组件对已经绑定了请求消息数据的形参对象进行数据合法性校验；

4、DataBinder 最后输出数据绑定结果对象 BindingResult。

BindingResult 包含了已完成数据绑定的形参对象和校验错误信息对象。

最终 SpringMVC 框架会把 BindingResult 中的数据，分别赋给相应的处理方法。

3.7. 多对象封装传参

现在有两个类 Cat 和 Dog，分别都有相同的属性 name 和 age。

```
@Data
public class Cat {
    private String name;
    private int age;
}
@Data
public class Dog {
    private String name;
    private int age;
}
```

现在期望在提交表单时，同时保存 Cat 和 Dog 的信息。

```
 <form action="/request/save" method="POST">
       狗狗姓名:<input type="text" name="dog.name"/><br/>
       狗狗年龄:<input type="text" name="dog.age"/><br/>
       猫猫姓名:<input type="text" name="cat.name"/><br/>
       猫猫年龄:<input type="text" name="cat.age"/><br/>
       <input type="submit" value="提交"/>
   </form>
```

Controller 方法：

```
@RequestMapping("/save")
public ModelAndView save(Cat cat, Dog dog) {
    System.out.println(cat);
    System.out.println(dog);
    return null;
 }
```

当需要把表单数据封装到多个不同对象中去的时候， 如果各个对象中都有相同的属性如 name，此时请求参数 name 就不清楚到底该把值封装到哪一个对象中去。更何况缺省情况下 SpringMVC 也不支持 Struts2 中类似于对象名.属性名传参方式。

此时需要我们来对象数据设置绑定规则。

InitBinder 注解：自定义数据绑定注册支持，用于将请求参数转换到命令对象属性的对应类型；

由@InitBinder 注解标注的方法，可以对 WebDataBinder 对象进行初始化。而 WebDataBinder 是 DataBinder 的子类，用于完成由请求参数到 JavaBean 的属性绑定。

@InitBinder 标注的方法不能有返回值，它必须声明为 void 。

@InitBinder 标注的方法的参数通常是 WebDataBinder。

代码如下：

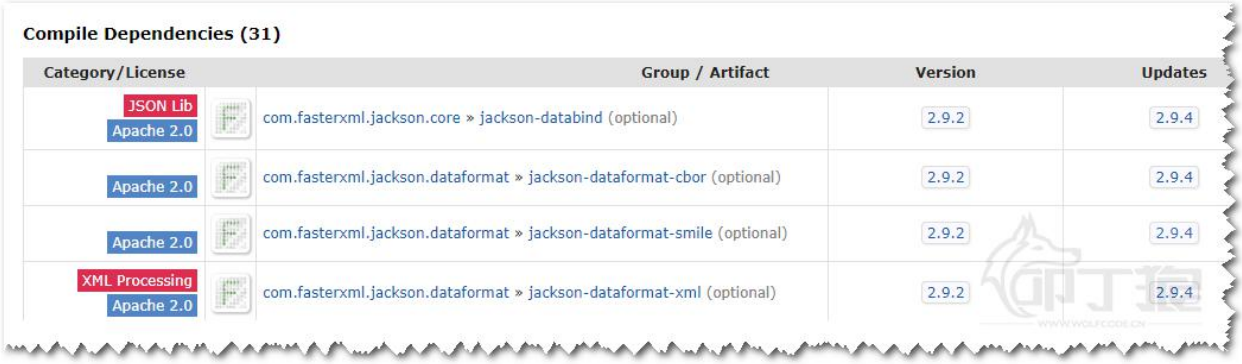
```
//绑定到 Cat 类型
@InitBinder("cat")
public void initBinderCatType(WebDataBinder binder) {
    binder.setFieldDefaultPrefix("cat.");
}
//绑定到 Dog 类型
@InitBinder("dog")
public void initBinderDogType(WebDataBinder binder) {
    binder.setFieldDefaultPrefix("dog.");
}
```

3.8. JSON 数据处理

3.8.1.Jackson

jackson 是一个 Java 开源的 JSON 工具库，性能超高，可以轻松的将 Java 对象转换成 json 对象和 xml 文档，同样也可以将 json、xml 转换成 Java 对象。

下载当前 Spring 版本依赖的 jackson 版本。



Compile Dependencies (31)

Category/License	Group / Artifact	Version	Updates
JSON Lib Apache 2.0	com.fasterxml.jackson.core » jackson-databind (optional)	2.9.2	2.9.4
Apache 2.0	com.fasterxml.jackson.dataformat » jackson-dataformat-cbor (optional)	2.9.2	2.9.4
Apache 2.0	com.fasterxml.jackson.dataformat » jackson-dataformat-smile (optional)	2.9.2	2.9.4
XML Processing Apache 2.0	com.fasterxml.jackson.dataformat » jackson-dataformat-xml (optional)	2.9.2	2.9.4

依赖库为：

```
jackson-core-2.9.4.jar
jackson-databind-2.9.4.jar
jackson-annotations-2.9.4.jar
```

3.9. JSON 处理

SpringMVC 提供了处理 JSON 格式请求和响应的 HttpMessageConverter——MappingJackson2HttpMessageConverter，利用 Jackson 开源库处理 JSON 格式的请求和响应信息。

SpringMVC 默认支持 jackson 处理 JSON 数据。

处理 JSON 的注解：

RequestBody 注解，处理请求，用于读取 HTTP 请求的内容(JSON 格式字符串)，转换为 Java 对象。

ResponseBody 注解，处理响应，用于将 Controller 的方法返回的对象，转换为 JSON 字符串。

RestController 是一个复合组件，主要是用来指出 RESTful 的。

@RestController 注解 = @Controller 注解 + @ResponseBody 注解

RequestBody 注解：

如果 HTTP 请求类型和数据格式是 JSON(contentType=application/json;charset=utf-8)，此时必须使用@RequestBody 贴在 Controller 方法参数上。

如果 HTTP 请求的数据格式是 key-value 就不需要使用@RequestBody（更多的时候）。

ResponseBody 注解（使用较多）：

小结：

application/x-www-form-urlencoded，传统的 key-value 格式数据，处理很方便，当然 RequestBody 也能处理。

application/json,application/xml 格式的数据，必须使用 RequestBody 来处理。

把单个对象转换为 JSON：

```
@RequestMapping("/test1")
@ResponseBody
public User test1() {
  User u = new User();
  u.setId(123L);
  u.setUsername("will");
  u.setAge(17);
  return u;
}
```

把多个对象转换为 JSON：

```
@RequestMapping("/test2")
@ResponseBody
public List<User> test2() {
  User u = new User();
  u.setId(123L);
  u.setUsername("will");
```

```
    u.setAge(17);
    return Arrays.asList(u, u, u);
  }
```

把字符串转换为 JOSN：

```
  @RequestMapping(value="/test3",produces="application/json; charset=UTF-8")
  @ResponseBody
  public String test3() {
    return "你好,JSON";
  }
```

把 Map 转换为 JSON：

```
  @RequestMapping("/test4")
  @ResponseBody
  public Map<String,Object> test4() {
    Map<String,Object> map = new HashMap<>();
    map.put("id", "123");
    map.put("name", "Will");
    map.put("age", 17);
    return map;
  }
```

3.10. 日期类型处理

3.10.1. 前台往后台传参（String->Date）

- 方式一、在对象字段或 Controller 形参上添加上@DateTimeFormat(pattern="...")注解

```
@RequestMapping("/test1")
public ModelAndView test1(@DateTimeFormat(pattern = "yyyy-MM-dd") Date d) {
    System.out.println("d = [" + d + "]");
    return null;
}
```

更多的时候，Date 参数都是作为对象的字段存在，此时需要在该字段上贴。

```
@Getter@Setter
public class User{
   @DateTimeFormat(pattern = "yyyy-MM-dd")
   private Date hiredate;
}
```

- 方式二、在 Controller 类中加入以下代码.

```
@InitBinder
public void initBinderDateType(WebDataBinder binder) {
    SimpleDateFormat sdf = new SimpleDateFormat();
```

```
        sdf.applyPattern("yyyy-MM-dd");
        binder.registerCustomEditor(java.util.Date.class, new CustomDateEditor(sdf, true));
    }
```

但此时只能在该 Controller 类中使用，其他 Controller 中的 Date 类型还是不能转换。

- 方式三、使用@ControllerAdvice 注解

```
@ControllerAdvice
public class DateFormatControllerAdvice {
    @InitBinder
    public void initBinderDateType(WebDataBinder binder){
        SimpleDateFormat sdf = new SimpleDateFormat();
        sdf.applyPattern("yyyy-MM-dd");
        binder.registerCustomEditor(java.util.Date.class,
                                    new CustomDateEditor(sdf,true));
    }
}
```

此时需要保证 DateFormatControllerAdvice 类所在包能被组件扫描到。

3.10.2. JSP 显示 Date 类型（Date->String）

在 JSP 中显示 Date 类型的数据，有时候会呈现出英美式风格，此时需要使用 JSTL 中的日期格式化标签——fmt 标签。

依赖 JSTL 标签库:

```
taglibs-standard-spec-1.2.5.jar
taglibs-standard-impl-1.2.5.jar
```

JSP 代码如下：

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>
<%
    request.setAttribute("myDate",new java.util.Date());
%>
<fmt:formatDate value="${myDate}" pattern="yyyy-MM-dd"/>
```

3.10.3. 后台往前台响应 JSON 格式 Date 类型处理

如果以 JSON 格式数据响应数据到前台，若返回数据中存在 Date 类型的字段，SpringVC 自动把 Date 类型数据的解析成毫秒值，此时解决方案有：

方式一、在需要解析的字段上添加注解。

```
@JsonFormat(pattern = "yyyy-MM-dd-HH:mm:ss", timezone = "GMT+8")
private Date hiredate;
```

方式二、在 mvc.xml 中配置全局的解析器.

```
<mvc:annotation-driven>
        <mvc:message-converters>
            <bean class="org.springframework.http.converter
                    .json.MappingJackson2HttpMessageConverter">
                <property name="objectMapper">
                    <bean class="com.fasterxml.jackson.databind.ObjectMapper">
                        <property name="dateFormat">
                            <bean class="java.text.SimpleDateFormat">
                                <constructor-arg type="java.lang.String"
                                      value="yyyy-MM-dd HH:mm:ss" />
                            </bean>
                        </property>
                    </bean>
                </property>
            </bean>
        </mvc:message-converters>
</mvc:annotation-driven>
```

4. SpringMVC 案例

拷贝之前 Spring 和 JDBC 整合开发的案例，保证 CRUD 测试通过，把项目修改成 Web 应用。

4.1. 基于 XML 配置

4.1.1. mvc.xml 配置

```
<import resource="classpath:applicationContext.xml"/>
<!-- 组件扫描 -->
<context:component-scan base-package="cn.wolfcode" />
<!-- 处理静态资源 -->
<mvc:default-servlet-handler />

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

<!--MVC 注解解析器 -->
<mvc:annotation-driven />
```

4.1.2. Controller 代码

```
@Controller
@RequestMapping("/employee")
public class EmployeeController {
  @Autowired
  private IEmployeeService employeeService;

  @RequestMapping("/list")
  public String list(Model model) {
    model.addAttribute("employees", employeeService.listAll());
    return "employee/list";
  }

  @RequestMapping("/input")
  public String input(Model model, Long id) {
    if (id != null) {
      model.addAttribute("employee", employeeService.get(id));
```

```
    }
    return "employee/input";
  }

  @RequestMapping("/saveOrUpdate")
  public String saveOrUpdate(Model model, Employee e) {
    if (e.getId() == null) {
      employeeService.save(e);
    } else {
      employeeService.update(e);
    }
    return "redirect:/employee/list";
  }

  @RequestMapping("/delete")
  public String delete(Long id) {
    if (id != null) {
      employeeService.delete(id);
    }
    return "redirect:/employee/list";
  }
}
```

4.1.3.JSP 代码

list.jsp：

```
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<h3>员工列表</h3>
<a href="/employee/input">新增</a>
<table border="1" cellspacing="0" cellpadding="0" width="500">
  <tr>
    <th>ID</th>
    <th>用户名</th>
    <th>密码</th>
    <th>入职日期</th>
    <th>操作</th>
  </tr>
  <c:forEach items="${employees}" var="e">
    <tr>
      <td>${e.id}</td>
      <td>${e.username}</td>
      <td>${e.password}</td>
```

```
        <td>${e.hiredate}</td>
        <td>
          <a href="/employee/delete?id=${e.id}">删除</a>
          <a href="/employee/input?id=${e.id}">编辑</a>
        </td>
      </tr>
    </c:forEach>
  </table>
```

input.jsp：

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<h3>编辑员工</h3>
<form action="/employee/saveOrUpdate" method="post">
  <input type="hidden" name="id" value="${employee.id}"/>
  账号<input type="text" name="username"  value="${employee.username}"/><br/>
  密码<input type="text" name="password"  value="${employee.password}"/><br/>
  入职<input type="text" name="hiredate"  value="${employee.hiredate}"/><br/>
  <input type="submit" value="保存"/>
</form>
```

4.1.4.登录操作

login.jsp：

```
<span style="color:red">
    ${errorMsg}
</span>
<%
        session.invalidate();
%>
<form action="/login" method="post">
    账号<input type="text" name="username"/><br/>
    密码<input type="text" name="password"/><br/>
    <input type="submit" value="登录"/>
</form>
```

LoginController：

```
@RequestMapping("/login")
  public String list(String username, String password, HttpSession session) {
    Employee current = employeeService.login(username, password);
    if (current == null) {
      session.setAttribute("errorMsg", "账号或密码不正确!");
      return "redirect:/login.jsp";
    }
```

```
        session.setAttribute("user_in_session", current);
        return "redirect:employee/list";
      }
```

EmployeeServiceImpl 类：

```
    public Employee login(String username, String password) {
        return dao.checkLogin(username, password);
    }
```

DAO 方法：

```
    public Employee checkLogin(String username, String password) {
        List<Employee> list = jdbcTemplate.query(
            "SELECT id,username,password,age,hiredate FROM employee
                        WHERE username = ? AND password=?",
            new Object[] { username, password }, (rs, rowNum) -> {
              Employee e = new Employee();
              e.setId(rs.getLong("id"));
              e.setUsername(rs.getString("username"));
              e.setPassword(rs.getString("password"));
              e.setAge(rs.getInt("age"));
              e.setHiredate(rs.getDate("hiredate"));
              return e;
            });
        return list.size() == 1 ? list.get(0) : null;
      }
```

4.1.5. RequestContextHolder

SpringMVC 提供了一个非常方便的工具类，能够在任意地方很方便的获取 request 和 session 对象。

RequestContextHolder 顾名思义，持有上下文的 Request 容器。

在新的 Spring 版本中不需要配置监听器，但如果和第三方框架集成，有时候需要在 web.xml 中配置监听器：

```
<listener>
    <listener-class>org.springframework.web.context.request.RequestContextListener</listener-class>
</listener>
```

封装 UserContext 类：

```
    public class UserContext {
        public static final String USER_IN_SESSION = "user_in_session";

        private static HttpSession getSession() {
            return
                 ((ServletRequestAttributes)
                 (RequestContextHolder.getRequestAttributes())).getRequest().getSession();
        }
```

```
    public static void setCurrentUser(Employee current) {
        if(current!=null) {
        getSession().setAttribute(USER_IN_SESSION, current);
        }else {
            getSession().invalidate();
        }
    }

    public static Employee getCurrentUser() {
        return (Employee) getSession().getAttribute(USER_IN_SESSION);
    }
}
```

EmployeeServiceImpl 类：

```
public void login(String username, String password) {
    Employee current = dao.checkLogin(username, password);
    if (current == null) {
        throw new RuntimeException("账号或密码错误");
    }
    UserContext.setCurrentUser(current);
}
```

LoginController：

```
@Controller
public class LoginController {
    @Autowired
    private IEmployeeService employeeService;

    @RequestMapping("/login")
    public String list(String username, String password, HttpSession session) {
        try {
            employeeService.login(username, password);
        } catch (Exception e) {
            session.setAttribute("errorMsg", e.getMessage());
            return "redirect:/login.jsp";
        }
        return "redirect:employee/list";
    }
}
```

4.2. 基于 Annotation 配置

4.2.1. XML 配置

```
<context:property-placeholder
      location="classpath:db.properties" system-properties-mode="NEVER" />

  <bean id="dataSource" init-method="init" destroy-method="close"
            class="com.alibaba.druid.pool.DruidDataSource"  >
    <property name="driverClassName" value="${jdbc.driverClassName}" />
    <property name="url" value="${jdbc.url}" />
    <property name="username" value="${jdbc.username}" />
    <property name="password" value="${jdbc.password}" />
    <property name="initialSize" value="${jdbc.initialSize}" />
  </bean>

  <bean id="txManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
  </bean>

  <tx:annotation-driven transaction-manager="txManager" />
```

4.2.2. DAO 代码

```
@Repository
@Transactional
public class EmployeeDAOImpl implements IEmployeeDAO {
  private JdbcTemplate jdbcTemplate;

  @Autowired
  public void setDataSource(DataSource ds) {
    this.jdbcTemplate = new JdbcTemplate(ds);
  }
  //TODO
}
```

4.2.3. Service 代码

```
@Service
@Transactional
public class EmployeeServiceImpl implements IEmployeeService {
```

```
  @Autowired
  private IEmployeeDAO dao;
  //TODO
}
```

4.3. 基于 JavaConfig 配置

4.3.1. Web 启动器

```
public class MyWebInitializer extends
                AbstractAnnotationConfigDispatcherServletInitializer {
  //根容器
  protected Class<?>[] getRootConfigClasses() {
    return new Class<?>[] { RootConfig.class };
  }

  //SpringMVC 容器
  protected Class<?>[] getServletConfigClasses() {
    return new Class<?>[] { WebConfig.class };
  }
  //映射路径
  protected String[] getServletMappings() {
    return new String[] { "/" };
  }

  protected void registerContextLoaderListener(ServletContext servletContext) {
    super.registerContextLoaderListener(servletContext);
    //编码过滤器
    Dynamic addFilter = servletContext
          .addFilter("characterEncodingFilter", CharacterEncodingFilter.class);
    addFilter.setInitParameter("encoding", "UTF-8");
    addFilter.addMappingForUrlPatterns(EnumSet.of(DispatcherType.REQUEST), true, "/*");
  }
}
```

4.3.2. RootConfig

```
@Configuration
@ComponentScan(basePackages = { "cn.wolfcode" }, excludeFilters = {
    @ComponentScan.Filter(type = FilterType.ANNOTATION, value = EnableWebMvc.class) })
@PropertySource("classpath:db.properties")
@EnableTransactionManagement
```

```
public class RootConfig {
  @Value("${jdbc.driverClassName}")
  private String driverClassName;
  @Value("${jdbc.url}")
  private String url;
  @Value("${jdbc.username}")
  private String username;
  @Value("${jdbc.password}")
  private String password;
  @Value("${jdbc.initialSize}")
  private int initialSize;

  @Bean
  public DataSource dataSource() {
    DruidDataSource ds = new DruidDataSource();
    ds.setDriverClassName(driverClassName);
    ds.setUrl(url);
    ds.setUsername(username);
    ds.setPassword(password);
    ds.setInitialSize(initialSize);
    return ds;
  }

  @Bean
  public DataSourceTransactionManager dataSourceTransactionManager(DataSource ds) {
    return new DataSourceTransactionManager(ds);
  }
}
```

4.3.3. WebConfig

```
@Configuration
@EnableWebMvc
@ComponentScan
public class WebConfig implements WebMvcConfigurer {
  @Bean
  public CheckLoginInterceptor checkLoginInterceptor() {
    return new CheckLoginInterceptor();
  }
  //添加拦截器
  public void addInterceptors(InterceptorRegistry registry) {
     registry.addInterceptor(checkLoginInterceptor())
          .addPathPatterns("/**").excludePathPatterns("/login");
  }
```

```
    //配置 jsp 视图
    @Bean
    public ViewResolver viewResolver() {
      InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
      viewResolver.setPrefix("/WEB-INF/views/");
      viewResolver.setSuffix(".jsp");
      return viewResolver;
    }
    //配置静态资源处理
    public void configureDefaultServletHandling(
           DefaultServletHandlerConfigurer configurer) {
      configurer.enable();
    }
  }
```

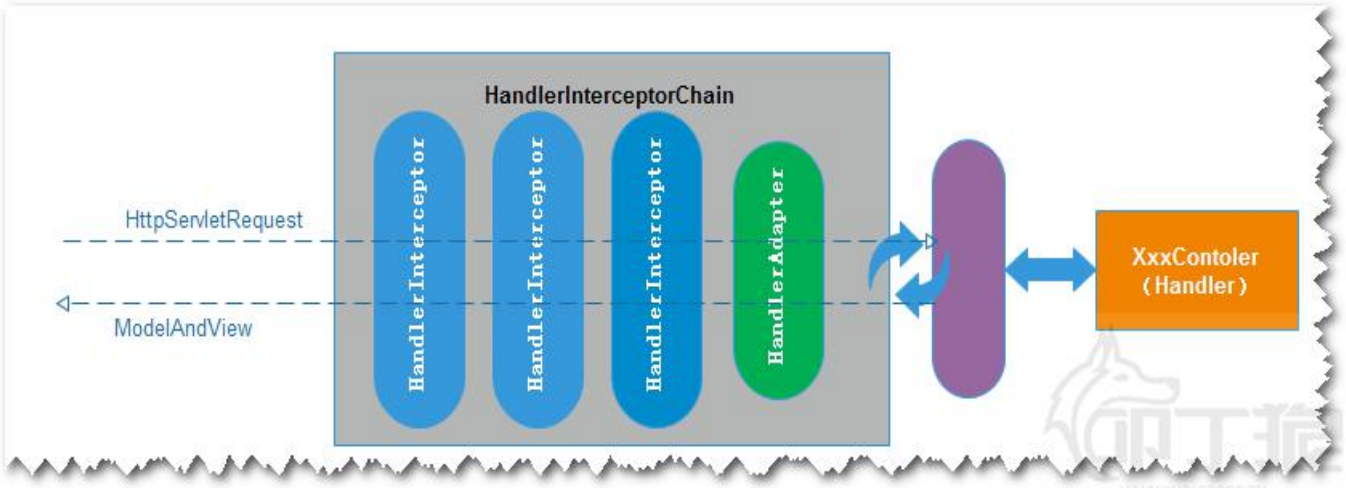
拦截器最后加上。

5. 拦截器

当收到请求时，DispatcherServlet 将请求交给处理器映射（HandlerMapping），让它找出对应该请求的 HandlerExeutionChain 对象。HandlerExeutionChain 对象是一个执行链，包含一个处理该请求的处理器（Handler），同时包括多个对该请求实施拦截的拦截器（HandlerInterceptor）。

当 HandlerMapping 返回 HandlerExecutionChain 后，DispatcherServlet 将请求交给定于在 HandlerExecutionChain 中的拦截器和处理器一并处理。

HandlerExecutionChain 结构：



位于处理器链末端的是一个 Handler，DispatcherServlet 通过 HandlerAdapter 适配器对 Handler 进行封装，并按统一的适配器接口对 Handler 处理方法进行调用。

5.1. 拦截原理

拦截器类必须实现接口：org.springframework.web.servlet.HandlerInterceptor。

```
public interface HandlerInterceptor {
   default boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)
         throws Exception {
      return true;
   }
   default void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,
         @Nullable ModelAndView modelAndView) throws Exception {
   }
   default void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler,
         @Nullable Exception ex) throws Exception {
   }
}
```

拦截器到底做了什么，我们可以通过查看拦截器的接口方法来进行了解。

1、preHandle 方法

在请求达到 Handler 之前，先执行这个前置处理方法。当该方法返回 false 时，请求直接返回，不会传递到链中的下一个拦截器，更不会传递到处理链末端的 Handler 中。

只有返回 true 时，请求才向链中的下一个处理节点传递。

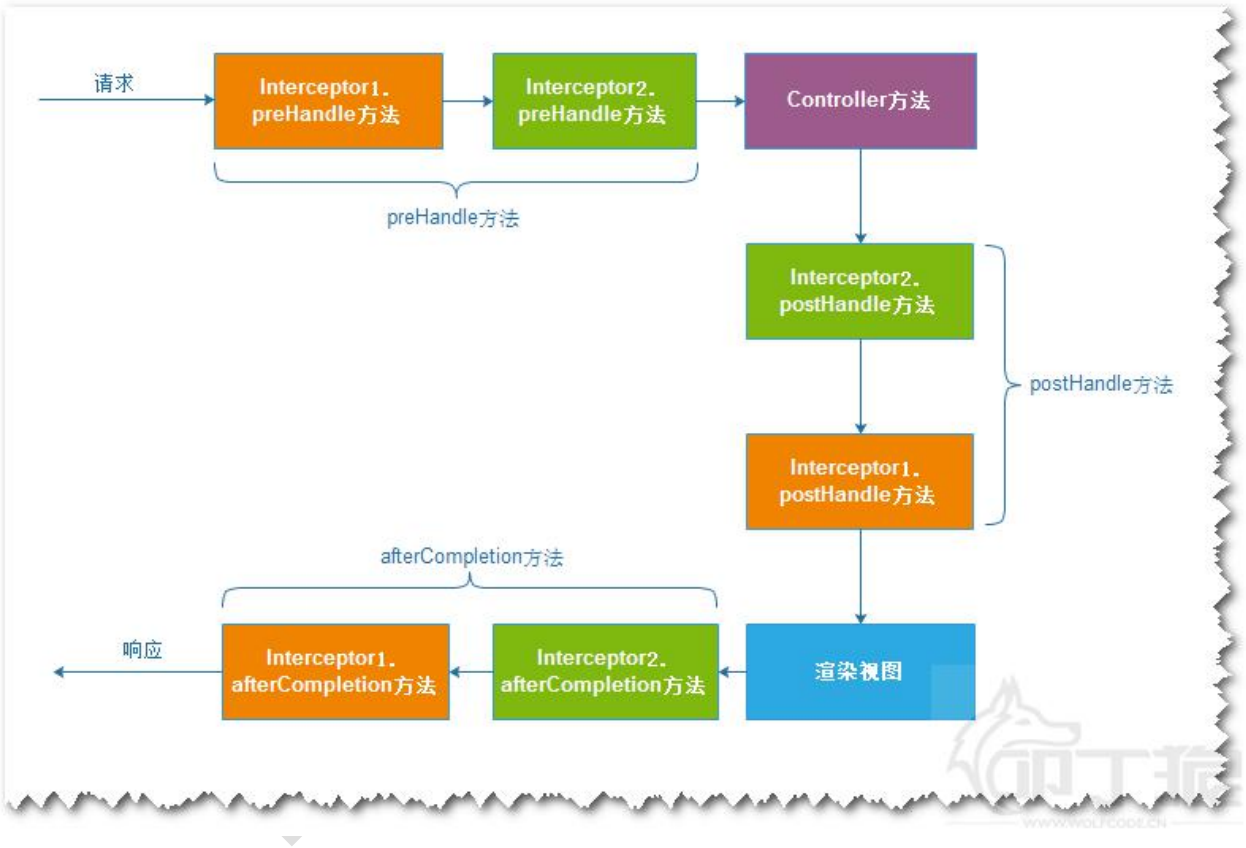
2、postHandle 方法：

控制器方法执行后，视图渲染之前执行(可以加入统一的响应信息)。

3、afterCompletion 方法：

视图渲染之后执行(处理 Controller 异常信息，记录操作日志，清理资源等)

拦截器工作流程图：



5.2. 拦截器开发

开发 SpringMVC 拦截器步骤：

1、定义拦截器类,实现接口 org.springframework.web.servlet.HandlerInterceptor

2、在 mvc.xml 中配置拦截器

CheckLoginInterceptor 类：

```
public class CheckLoginInterceptor extends HandlerInterceptorAdapter {
    public boolean preHandle(HttpServletRequest request,
                             HttpServletResponse response, Object handler)
            throws Exception {
        if (UserContext.getCurrentUser() == null) {
            response.sendRedirect("/login.jsp");
            return false;
        }
        return true;
    }
}
```

mvc.xml 配置：

```
<mvc:interceptors>
   <mvc:interceptor>
     <mvc:mapping path="/**" />
     <mvc:exclude-mapping path="/login" />
     <bean class="cn.wolfcode.CheckLoginInterceptor" />
   </mvc:interceptor>
</mvc:interceptors>
```

拦截路径说明：

/* ：只能拦截一级路径

/**：可以拦截一级或多级路径

5.3. JavaConfig 添加拦截器

```
@Configuration
@EnableWebMvc
@ComponentScan
public class WebConfig implements WebMvcConfigurer {
  @Bean
  public CheckLoginInterceptor checkLoginInterceptor() {
    return new CheckLoginInterceptor();
  }
  //添加拦截器
  public void addInterceptors(InterceptorRegistry registry) {
     registry.addInterceptor(checkLoginInterceptor())
          .addPathPatterns("/**").excludePathPatterns("/login");
  }
  //配置 jsp 视图
  @Bean
  public ViewResolver viewResolver() {
```

```
        InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
        viewResolver.setPrefix("/WEB-INF/views/");
        viewResolver.setSuffix(".jsp");
        return viewResolver;
    }
    //配置静态资源处理
    public void configureDefaultServletHandling(
            DefaultServletHandlerConfigurer configurer) {
        configurer.enable();
    }
}
```

拦截器最后加上。

6. 异常处理

在 Java EE 项目的开发中，不管是对底层的数据库操作过程，还是业务层的处理过程，还是控制层的处理过程，都不可避免会遇到各种可预知的、不可预知的异常需要处理。

每个过程都单独处理异常，系统的代码耦合度高，工作量大且不好统一，维护的工作量也很大。

那么，能不能将所有类型的异常处理从各处理过程解耦出来，这样既保证了相关处理过程的功能较单一，也实现了异常信息的统一处理和维护？答案是肯定的。下面将介绍使用 Spring MVC 统一处理异常的解决和实现过程。

Spring MVC 处理异常有 3 种方式：

（1）使用 Spring MVC 提供的简单异常处理器 SimpleMappingExceptionResolver；

（2）实现 Spring 的异常处理接口 HandlerExceptionResolver 自定义自己的异常处理器；

（3）使用@ExceptionHandler 注解实现异常处理；

使用方式一和方式三配置统一处理全局异常。

6.1. SimpleMappingExceptionResolver

```
<bean class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
    <!-- 定义默认的异常处理页面，当该异常类型的注册时使用 -->
    <property name="defaultErrorView" value="commons/error" />
    <!-- 定义异常处理页面用来获取异常信息的变量名，默认名为 exception -->
    <property name="exceptionAttribute" value="ex" />
    <!-- 定义需要特殊处理的异常，用类名或完全路径名作为 key，异常也页名作为值 -->
    <property name="exceptionMappings">
      <value>
        cn.wolfcode.p2p.web.exception.SecurityException=commons/nopermission
        <!-- 这里还可以继续扩展不同异常类型的异常处理 -->
      </value>
    </property>
 </bean>
```

6.2. ExceptionHandler

通过使用@ControllerAdvice 注解定义统一的异常处理类，而不是在每个 Controller 中逐个定义，ExceptionHandler 注解用来定义方法针对的异常类型。

```
@ControllerAdvice
```

```
public class HandlerExceptionControllerAdvice {
  @ExceptionHandler(Exception.class)
  public String error(Exception ex, Model model) {
    model.addAttribute("errorMsg", ex.getMessage());
    return "error";
  }
}
```

7. 数据校验

输入校验是 Web 开发任务之一，在 SpringMVC 中有两种方式可以实现，分别是使用 Spring 自带的验证框架和使用 JSR 303 实现，也称之为 spring-validator 和 JSR303-validator。

在开发中更建议使用 JSR303-validator。

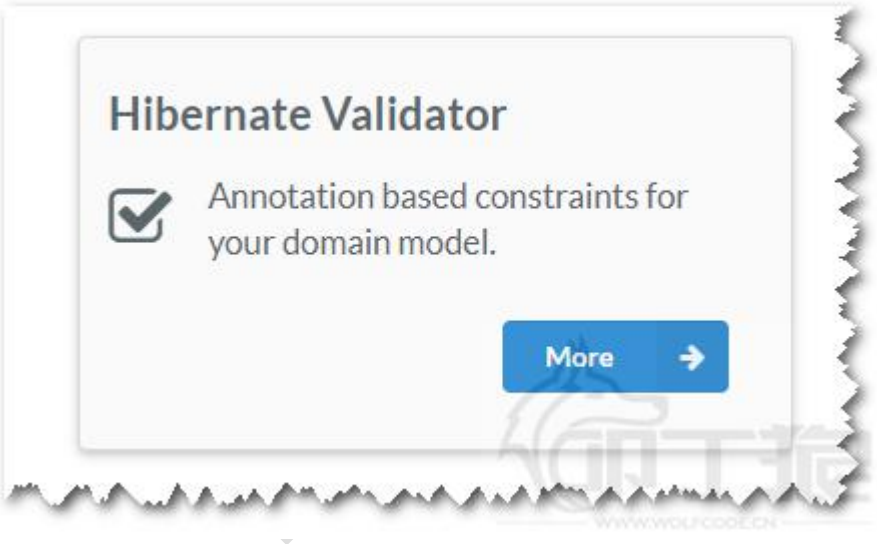
7.1. JSR303 约束

Spring 在进行数据绑定时，可同时调用校验框架完成数据校验工作。

在 Spring MVC 中，可直接通过注解驱动的方式进行数据校验 Spring 的 LocalValidatorFactroyBean 既实现了 Spring 的 Validator 接口，也实现了 JSR 303 的 Validator 接口。只要在 Spring 容器中定义了一个 LocalValidatorFactoryBean ，即可将其注入到需要数据校验的 Bean 中。

Spring 本身并没有提供 JSR303 的实现，所以必须将 JSR303 的实现者的 jar 包放到类路径下。

JSR303 仅仅是校验的标准，我们这里使用的是 Hibernate Validator 框架实现方案：



依赖的 jar：

```
hibernate-validator-6.0.7.Final.jar
validation-api-2.0.1.Final.jar
jboss-logging-3.3.0.Final.jar
classmate-1.3.1.jar
```

7.2. JSR303 约束规则

JSR 303 约束

属性	描述	范例
@AssertFalse	应用于 boolean 属性，该属性值必须为 False	@AssertFalse boolean hasChildren;
@AssertTrue	应用于 boolean 属性，该属性值必须为 True	@AssertTrue boolean isEmpty;
@DecimalMax	该属性值必须为小于或等于指定值的小数	@DecimalMax("1.1") BigDecimal price;
@DecimalMin	该属性值必须为大于或等于指定值的小数	@DecimalMin("0.04") BigDecimal price;
@Digits	该属性值必须在指定范围内。integer 属性定义该数值的最大整数部分，fraction 属性定义该数值的最大小数部分	@Digits(integer=5, fraction=2) BigDecimal price;
@Future	该属性值必须是未来的一个日期	@Future Date shippingDate;
@Max	该属性值必须是一个小于或等于指定值的整数	@Max(150) int age;
@Min	该属性值必须是一个大于或等于指定值的整数	@Max(150) int age;
@NotNull	该属性值不能为 Null	@NotNull String firstName;
@Null	该属性值必须为 Null	@Null String testString;
@Past	该属性值必须是过去的一个日期	@Past Date birthDate;
@Pattern	该属性值必须与指定的常规表达式相匹配	@Pattern(regext="\\d{3 }") String areaCode;
@Size	该属性值必须在指定范围内	Size(min=2, max=140) String description;

<mvc:annotation-driven/>会默认装配好一个 LocalValidatorFactoryBean，通过在处理方法的形参上标注@Validated 注解，即可让 Spring MVC 在完成数据绑定后执行数据校验的工作。

在已经标注了 JSR303 注解的表单/命令对象前标注一个 Validated 注解，Spring MVC 框架将请求参数绑定到该参数对象后，就会调用校验框架，根据注解声明的校验规则来实施校验。

Spring MVC 通过对处理方法签名的规约来保存校验结果的，前一个表单/命令对象的校验结果保存到随后的参数中，这个保存校验结果的参数必须是 BindingResult 或 Errors 类型，这两个类都位于 org.springframework.validation 包中。

需校验的对象和其绑定结果对象或错误对象必须成对出现的，它们之间不允许声明其他的参数。

7.3. 校验操作

需要校验的模型对象：

```
@Data
public class Employee {
  private Long id;
  @NotNull(message = "用户名不能为空")
  @Size(max = 10, min = 6, message = "用户名在 6 到 10 位之间")
  private String username;
  @NotNull(message = "用户名不能为空")
  @Size(max = 10, min = 6, message = "用户名在 6 到 10 位之间")
  private String password;
  @NotNull(message = "年龄不能为空")
  @Min(value = 18, message = "最小年龄为 18")
  @Max(value = 60, message = "最大年龄为 18")
  private Integer age;
  @DateTimeFormat(pattern = "yyyy-MM-dd")
  private Date hiredate;
}
```

Controller 方法：

```
 @RequestMapping("/saveOrUpdate")
  public String saveOrUpdate(Model model,
              @Validated Employee e, BindingResult bindingResult) {
    List<ObjectError> errors = bindingResult.getAllErrors();
    if (errors.size() > 0) {
       model.addAttribute("errors", errors);
       return "employee/input";
    }
    if (e.getId() == null) {
       employeeService.save(e);
    } else {
       employeeService.update(e);
    }
    return "redirect:/employee/list";
 }
```

input.jsp 代码：

```
<c:forEach items="${errors}" var="err">
  <span style="color: red">${err.defaultMessage}</span><br/>
</c:forEach>

<form action="/employee/saveOrUpdate" method="post">
```

```
  <input type="hidden" name="id" value="${employee.id}"/>
  账号：<input type="text" name="username"  value="${employee.username}"/><br/>
  密码：<input type="text" name="password"  value="${employee.password}"/><br/>
  入职：<input type="text" name="hiredate"  value="${employee.hiredate}"/><br/>
  <input type="submit" value="保存"/>
</form>
```

7.4. 国际化操作

需要校验的模型对象：

```
@Data
public class Employee {
  private Long id;
  @NotNull(message = "{employee.username.notNull}")
  @Size(max = 10, min = 6, message = "{employee.username.size}")
  private String username;
  @NotNull(message = "{employee.password.notNull}")
  @Size(max = 10, min = 6, message = "{employee.password.size}")
  private String password;
  @NotNull(message = "{employee.age.notNull}")
  @Min(value = 18, message = "{employee.age.min}")
  @Max(value = 60, message = "{employee.age.max}")
  private Integer age;
  @DateTimeFormat(pattern = "yyyy-MM-dd")
  private Date hiredate;
}
```

ValidationMessages.properties:

```
employee.username.notNull=\u7528\u6237\u540D\u4E0D\u80FD\u4E3A\u7A7A
employee.username.size=\u7528\u6237\u540D\u57286\u523010\u4F4D\u4E4B\u95F4

employee.password.notNull=\u5BC6\u7801\u4E0D\u80FD\u4E3A\u7A7A
employee.password.size=\u5BC6\u7801\u57286\u523010\u4F4D\u4E4B\u95F4

employee.age.notNull=\u5E74\u9F84\u4E0D\u80FD\u4E3A\u7A7A
employee.age.min=\u5E74\u9F84\u6700\u5C0F\u4E3A18
employee.age.max=\u5E74\u9F84\u6700\u5927\u4E3A60
```

8. 表单标签

通过 SpringMVC 的表单标签可以实现将模型数据中的属性和 HTML 表单元素相绑定，以实现表单数据更便捷编辑和表单值的回显功能。

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

SpringMVC 所有表单元素：

表单标签库中的标签

标签	描述
form	渲染表单元素
input	渲染 ____ 元素
password	渲染 元素
hidden	渲染<input type="hidden"/>元素
textarea	渲染 textarea 元素
checkbox	渲染一个☐元素
checkboxes	渲染多个☐元素
radiobutton	渲染一个☐元素
radiobuttons	渲染多个☐元素
select	渲染一个选择元素
option	渲染一个可选元素
options	渲染一个可选元素列表
errors	在 span 元素中渲染字段错误

8.1. form 元素

表单标签的属性

属性	描述
acceptCharset	定义服务器接受的字符编码列表
commandName	暴露表单对象之模型属性的名称，默认为 command
cssClass	定义要应用到被渲染 form 元素的 CSS 类
cssStyle	定义要应用到被渲染 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	暴露表单支持对象的模型属性名称，默认为 command

modelAttribute 已经取代 commandName 属性。

8.2. input 元素

input 标签的属性

属性	描述
cssClass	定义要应用到被渲染 input 元素的 CSS 类
cssStyle	定义要应用到被渲染 input 元素的 CSS 样式
cssErrorClass	定义要应用到被渲染 input 元素的 CSS 类，如果 bound 属性中包含错误，则覆盖 cssClass 属性值
htmlEscape	接受 true 或者 false，表示是否应该对被渲染的值进行 HTML 转义
path	要绑定的属性路径

8.3. password 元素

password 标签的属性

属性	描述
cssClass	定义要应用到被渲染 input 元素的 CSS 类
cssStyle	定义要应用到被渲染 input 元素的 CSS 样式
cssErrorClass	定义要应用到被渲染 input 元素的 CSS 类，如果 bound 属性中包含错误，则覆盖 cssClass 属性值
htmlEscape	接受 true 或者 false，表示是否应该对被渲染的值进行 HTML 转义
path	要绑定的属性路径
showPassword	表示应该显示或遮盖密码，默认值为 false

8.4. hidden 元素

hidden 标签的属性

属性	描述
htmlEscape	接受 true 或者 false，表示是否应该对被渲染的值进行 HTML 转义
path	要绑定的属性路径

8.5. textarea 元素

textarea 标签的属性

属性	描述
cssClass	定义要应用到被渲染 input 元素的 CSS 类
cssStyle	定义要应用到被渲染 input 元素的 CSS 样式
cssErrorClass	定义要应用到被渲染 input 元素的 CSS 类，如果 bound 属性中包含错误，则覆盖 cssClass 属性值
htmlEscape	接受 true 或者 false，表示是否应该对被渲染的值进行 HTML 转义
path	要绑定的属性路径

8.6. 复选框

8.6.1. checkbox 元素

checkbox 标签的属性

属性	描述
cssClass	定义要应用到被渲染 input 元素的 CSS 类
cssStyle	定义要应用到被渲染 input 元素的 CSS 样式
cssErrorClass	定义要应用到被渲染 input 元素的 CSS 类，如果 bound 属性中包含错误，则覆盖 cssClass 属性值
htmlEscape	接受 true 或者 false，表示是否应该对被渲染的（多个）值进行 HTML 转义
label	要作为标签用于被渲染复选框的值
path	要绑定的属性路径

8.6.2. checkboxes 元素

checkboxes 标签的属性

属性	描述
cssClass	定义要应用到被渲染 input 元素的 CSS 类
cssStyle	定义要应用到被渲染 input 元素的 CSS 样式
cssErrorClass	定义要应用到被渲染 input 元素的 CSS 类，如果 bound 属性中包含错误，则覆盖 cssClass 属性值
delimiter	定义两个 input 元素之间的分隔符，默认没有分隔符
element	给每个被渲染的 input 元素都定义一个 HTML 元素，默认为“span”
htmlEscape	接受 true 或者 false，表示是否应该对被渲染的（多个）值进行 HTML 转义
items	用于生成 input 元素的对象的 Collection、Map 或者 Array
itemLabel	item 属性中定义的 Collection、Map 或者 Array 中的对象属性，为每个 input 元素提供标签
itemValue	item 属性中定义的 Collection、Map 或者 Array 中的对象属性，为每个 input 元素提供值
path	要绑定的属性路径

8.7. 单选按钮

8.7.1. radiobutton 元素

radiobutton 标签的属性

属性	描述
cssClass	定义要应用到被渲染 input 元素的 CSS 类
cssStyle	定义要应用到被渲染 input 元素的 CSS 样式
cssErrorClass	定义要应用到被渲染 input 元素的 CSS 类，如果 bound 属性中包含错误，则覆盖 cssClass 属性值
htmlEscape	接受 true 或者 false，表示是否应该对被渲染的（多个）值进行 HTML 转义
label	要作为标签用于被渲染复选框的值
path	要绑定的属性路径

8.7.2. radiobuttons 元素

radiobuttons 标签的属性

属性	描述
cssClass	定义要应用到被渲染 input 元素的 CSS 类
cssStyle	定义要应用到被渲染 input 元素的 CSS 样式
cssErrorClass	定义要应用到被渲染 input 元素的 CSS 类，如果 bound 属性中包含错误，则覆盖 cssClass 属性值
delimiter	定义两个 input 元素之间的分隔符，默认没有分隔符
element	给每一个被渲染的 input 元素都定义一个 HTML 元素，默认为“span”
htmlEscape	接受 true 或者 false，表示是否应该对被渲染的（多个）值进行 HTML 转义
items	用于生成 input 元素的对象的 Collection、Map 或者 Array
itemLabel	item 属性中定义的 Collection、Map 或者 Array 中的对象属性，为每个 input 元素提供标签
itemValue	item 属性中定义的 Collection、Map 或者 Array 中的对象属性，为每个 input 元素提供值
path	要绑定的属性路径

8.8. select 元素

select 标签的属性

属性	描述
cssClass	定义要应用到被渲染 input 元素的 CSS 类
cssStyle	定义要应用到被渲染 input 元素的 CSS 样式
cssErrorClass	定义要应用到被渲染 input 元素的 CSS 类，如果 bound 属性中包含错误，则覆盖 cssClass 属性值
htmlEscape	接受 true 或者 false，表示是否应该对被渲染的（多个）值进行 HTML 转义
items	用于生成 input 元素的对象的 Collection、Map 或者 Array
itemLabel	item 属性中定义的 Collection、Map 或者 Array 中的对象属性，为每个 input 元素提供标签
itemValue	item 属性中定义的 Collection、Map 或者 Array 中的对象属性，为每个 input 元素提供值
path	要绑定的属性路径

8.8.1. option 和 options 元素

option 标签的属性

属性	描述
cssClass	定义要应用到被渲染 input 元素的 CSS 类
cssStyle	定义要应用到被渲染 input 元素的 CSS 样式
cssErrorClass	定义要应用到被渲染 input 元素的 CSS 类，如果 bound 属性中包含错误，则覆盖 cssClass 属性值
htmlEscape	接受 true 或者 false，表示是否应该对被渲染的（多个）值进行 HTML 转义

options 标签的属性

属性	描述
cssClass	定义要应用到被渲染 input 元素的 CSS 类
cssStyle	定义要应用到被渲染 input 元素的 CSS 样式
cssErrorClass	定义要应用到被渲染 input 元素的 CSS 类，如果 bound 属性中包含错误，则覆盖 cssClass 属性值
htmlEscape	接受 true 或者 false，表示是否应该对被渲染的（多个）值进行 HTML 转义
items	用于生成 input 元素的对象的 Collection、Map 或者 Array
itemLabel	item 属性中定义的 Collection、Map 或者 Array 中的对象属性，为每个 input 元素提供标签
itemValue	item 属性中定义的 Collection、Map 或者 Array 中的对象属性，为每个 input 元素提供值

8.9. errors 元素

errors 标签的属性

属性	描述
cssClass	定义要应用到被渲染 input 元素的 CSS 类
cssStyle	定义要应用到被渲染 input 元素的 CSS 样式
属性	描述
Delimiter	分隔多个错误消息的分隔符
element	定义一个包含错误消息的 HTML 元素
htmlEscape	接受 true 或者 false，表示是否应该对被渲染的（多个）值进行 HTML 转义
path	要绑定的错误对象路径

8.10. 案例

8.10.1. input.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
  pageEncoding="UTF-8"%>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<style type="text/css">
  .text_error {
    color: red;
  }
</style>
<h3>编辑员工</h3>

<form:form  modelAttribute="employee" action="/employee/saveOrUpdate" method="post">
  <form:hidden path="id"/>
 账号:<form:input path="username"/>
            <form:errors path="username" cssClass="text_error"/><br/>
  密码:<form:password path="password" showPassword="true"/>
            <form:errors path="password"  cssClass="text_error"/><br/>
  年龄:<form:input path="age"/>
            <form:errors path="age" cssClass="text_error"/><br/>
  入职:<form:input path="hiredate"/><br/>
   <input type="submit" value="保存" />
</form:form>
```

8.10.2. Controller 方法

```
@RequestMapping("/input")
public String input(Model model, Long id) {
    if (id != null) {
      model.addAttribute("employee", employeeService.get(id));
    }else {
      model.addAttribute("employee", new Employee());
    }
    return "employee/input";
}
```

9. 文件上传和下载

9.1. 文件上传

前台表单代码：

```
<form action="/upload" method="POST" enctype="multipart/form-data">
    姓名:<input type="text" name="username"/><br/>
    邮箱:<input type="text" name="age"/><br/>
    文件:<input type="file" name="pic"><br>
        <input type="submit" value="提交"/>
</form>
```

使用 MultipartFile 作为形参，能将前台传入的文件自动注入到该参数中。

```
@Controller
public class FileUploadController {
  @Autowired
  private ServletContext servletContext;

  @RequestMapping("/upload")
  public ModelAndView save(User u, MultipartFile pic) throws Exception {
    System.out.println("u = " + u);
    String saveDir = servletContext.getRealPath("/upload");
    String uuid = UUID.randomUUID().toString();
    String ext = FilenameUtils.getExtension(pic.getOriginalFilename());
    String fileName = uuid + "." + ext;
    System.out.println(Paths.get(saveDir, fileName).toString());
    Files.copy(pic.getInputStream(), Paths.get(saveDir, fileName));
    return null;
  }
}
```

如果使用动态 Web 项目，此时上传文件保存路径不在 Web 根路径下，这是动态 Web 项目部署问题，在实际生产环境中没问题。

9.1.1. Apache 上传组件

需要导入 fileupload 依赖包 io 的包

```
com.springsource.org.apache.commons.fileupload-1.2.0.jar
com.springsource.org.apache.commons.io-1.4.0.jar
```

配置文件上传解析器：

```
<bean id="multipartResolver"
        class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <!-- 设置上传文件的最大尺寸为 1MB -->
    <property name="maxUploadSize">
      <value>1048576</value>
    </property>
</bean>
```

配置文件上传解析器：bean 的名字是固定的

9.1.2.Servlet3 上传

使用 Servlet3 的文件上传方式，不再依赖 Apache 组件，围绕 MultipartConfig 注解和 Part 接口完成。

MultipartConfig 注解：

maxFileSize：上传文件的最大容量，缺省值为-1 表示没有限制，大于指定值会被拒绝。

maxRequestSize：表示一次请求允许最大容量，缺省值为-1，表示没有限制。

location：表示 Part 调用 write 方法时，要将已上传的文件临时保存到磁盘中的位置。

fileSizeThreshold：上传文件超过该容量界限，会被写入到磁盘。

文件上传解析器：

```
<bean id="multipartResolver"
 class="org.springframework.web.multipart.support.StandardServletMultipartResolver"/>
```

在 web.xml 中做上传配置：

```
<servlet>
    <servlet-name>springDispatcherServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>classpath:mvc.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
    <multipart-config>
      <location>c:/temp</location>
      <max-file-size>1048576</max-file-size>
      <max-request-size>10485760</max-request-size>
      <file-size-threshold>10240</file-size-threshold>
    </multipart-config>
  </servlet>
  <servlet-mapping>
    <servlet-name>springDispatcherServlet</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
```

9.2. 文件下载

9.2.1.传统方式

```
    @RequestMapping("/download")
    public void download(String fileName, HttpServletRequest request, HttpServletResponse
response) throws Exception {
        //GET 请求中文乱码处理
        fileName = new String(fileName.getBytes("ISO-8859-1"), "UTF-8");
        response.setContentType("application/x-msdownload");
        String userAgent = request.getHeader("User-Agent");
        if (userAgent.contains("IE")) {
           response.setHeader("Content-Disposition",
              "attachment;filename=" + URLEncoder.encode(fileName, "utf-8"));
        } else {
           response.setHeader("Content-Disposition",
               "attachment;filename=" + new String(fileName.getBytes("UTF-8"), "ISO8859-1"));
        }
        String dir = request.getServletContext().getRealPath("/WEB-INF/down");
        Files.copy(Paths.get(dir, fileName), response.getOutputStream());
     }
```

9.2.2.SpringMVC 方式

```
    @RequestMapping("/download")
    public        ResponseEntity<byte[]>        download(@RequestHeader("User-Agent")String
userAgent,String fileName, HttpServletRequest request, HttpServletResponse response)
                throws Exception {
            //fileName = new String(fileName.getBytes("ISO-8859-1"), "UTF-8");
            String dir = request.getServletContext().getRealPath("/WEB-INF/down");
            HttpHeaders headers = new HttpHeaders();
            //设置下载显示文件名称
            if (userAgent.contains("IE")) {
                headers.setContentDispositionFormData("attachment",
URLEncoder.encode(fileName, "utf-8"));
            }else {
                headers.setContentDispositionFormData("attachment",                    new
String(fileName.getBytes("UTF-8"), "ISO8859-1"));
            }
            //保存文件
```

```
            headers.setContentType(MediaType.APPLICATION_OCTET_STREAM);

            return    new    ResponseEntity<>(FileUtils.readFileToByteArray(new    File(dir,
fileName)), headers,
                    HttpStatus.CREATED);
    }
```