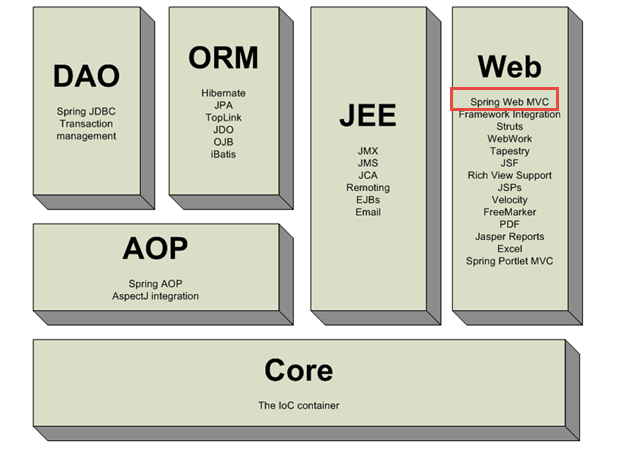
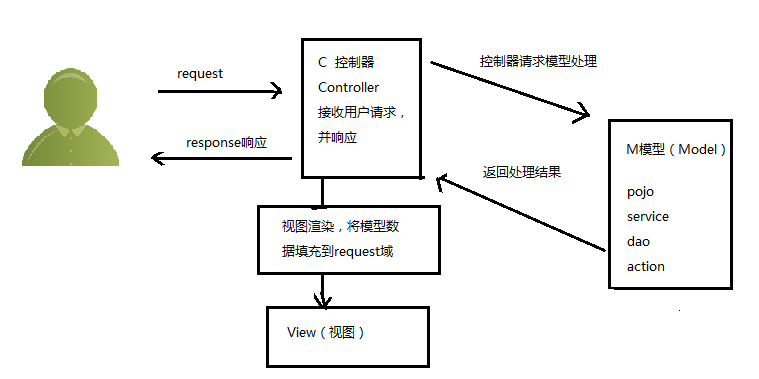
# 1、SpringMVC基本用法

SpringMVC是Spring框架的一个模块，SpringMVC和Spring无需通过中间整合层进行整合。SpringMVC是一个基于MVC的web框架，即Spring Web MVC。Spring Web MVC和Struts2都属于表现层的框架，它是Spring框架的一部分，我们可以从Spring的整体结构中看得出来：



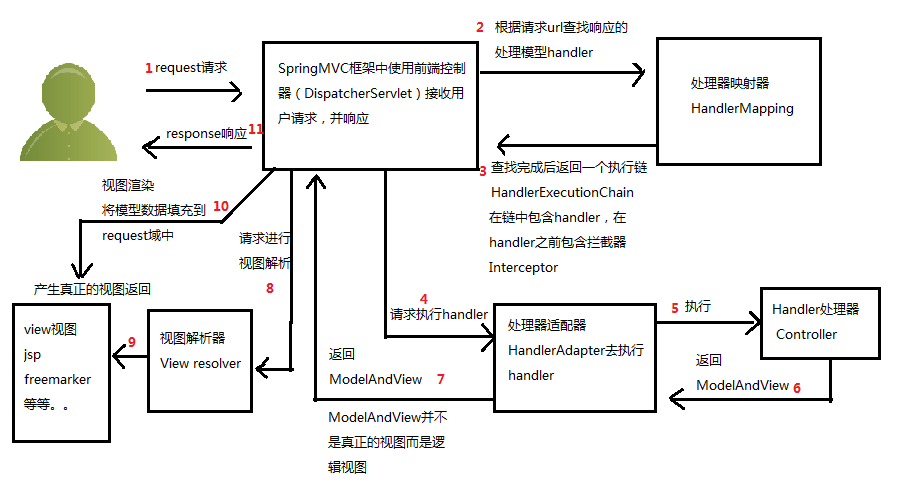
## 1.1 Web MVC原理

首先来一下MVC，MVC是一种开发模式，在大量的软件开发过程中所积累出来的经验，然后根据经验抽象出来的一种开发模式。下面来看一下MVC设计模式B/S架构的系统下的应用：



## 1.2 SpringMVC框架原理

从上图中看到MVC的基本框架后，那么下面学习SpringMVC框架就有了一个大体的思路。因为SpringMVC就是MVC模型的一种框架。所以学习SpringMVC框架主要就是来看SpringMVC中的C（控制层）、M（模型层）、V（视图层）的各种技术实现即可。如下图所示：



根据上图所示，使用SpringMVC的步骤如下：

1. 发起请求到前端控制器DispatcherServlet 主要作用：处理请求，不执行具体操作(调用相应的model层处理)
2. 前端控制DispatcherServlet请求HandlerMapping查找Handler，（XML配置或注解）
3. 处理器映射器HandlerMapping向前端控制器返回handler执行链
4. 前端控制器调用处理器适配器去执行handler
5. 处理器适配器执行handler
6. handler执行完成后返回给处理器适配器ModelAndView（SpringMVC框架的一个底层对象，包括model、view）
7. 处理器适配器向前端控制器返回ModelAndView
8. 前端控制器请求视图解析器去进行视图解析，根据逻辑视图名解析成真正的视图（jsp）
9. 视图解析器向前端控制器返回view

10、前端控制器进行视图渲染，将模型数据（在ModelAndView）填充到request域中。

11、前端控制器向用户响应结果。

以上步骤涉及到的组件：

1. 前端控制器DispatcherServlet，作用：接收请求，响应结果，相当于转发**（不需要程序人员开发）**
2. 处理器映射器HandlerMapping，作用：根据请求url查找handler**（不需要程序人员开发）**
3. 处理器适配器HandlerAdapter，作用：按照特定的规则去执行handler，所谓的特定规则，即HandlerAdapter指定的规则。**（不需要程序人员开发）**
4. Handler处理器。**（需要程序人员开发）**
5. 视图解析器Viewresolver，作用：进行视图解析，根据逻辑视图名解析成真正的视图view**（不需要程序人员开发）**
6. 视图View，作用：View是一个接口，实现类支持不同的View类型（jsp、freemarker等）**（需要程序人员开发，View不需要但是页面需要开发）**

## 1.3 SpringMVC入门程序

### 第一步：配置SpringMVC的前端控制器

SpringMVC的前端控制器是个Servlet，所以在web应用的web.xml文件中来配置该前端控制器，如下图所示：

<servlet>

<servlet-name>springmvc</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<!--

使用contextConfigLocation参数用于配置SpringMVC需要加载的配置文件(处理器映射器、处理器适配器、视图解析器等）。如果没有配置，SpringMVC默认加载的是/WEB-INF/下面的servlet名称-servlet.xml文件，即springmvc-servlet.xml文件。

-->

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>**classpath:config/springmvc.xml**</param-value>

</init-param>

</servlet>

说明：

classpath就是代表 /WEB-INF /classes/ 这个路径

servlet配置完成后，需要对该servlet类来配置相应的servlet-mapping如下图所示：

<servlet-mapping>

<servlet-name>springmvc</servlet-name>

<url-pattern>\*.action</url-pattern>

</servlet-mapping>

url-pattern的配置有两种方式：

第一种：\*.action；访问以.action结尾的任何请求都由DispatcherServlet进行解析。

第二种：/，所有访问的地址都由DispatcherServlet进行解析，对于静态文件的解析需要配置不让DispatcherServlet进行解析，使用此种方式可以实现RESTful风格的url。

### 第二步：配置处理器适配器

在classpath下的springmvc.xml文件中配置处理器映射器。SpringMVC框架是Spring的一个内部模块，所以springmvc.xml文件肯定需要同spring的配置文件兼容，即保持相同的头信息。如下所示：

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:mvc="http://www.springframework.org/schema/mvc"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc.xsd

">

<!--

配置处理器适配器

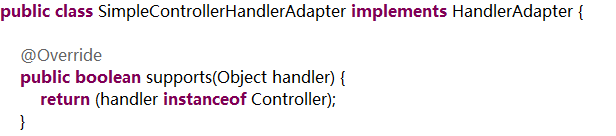
SpringMVC框架通过bean是否实现 HandleAdapter接口来判断该bean是否是一个适配器

-->

<bean class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter" />

</beans>

配置完成适配器，适配器的作用就是来执行相应的Handler处理器，那么怎么编写代码才算是一个Handler处理器，通过SimpleControllerHandlerAdapter的原码查看，此类中包含如下一个方法supports，如下所示：



通过此方法可以得知，只有实现了Controller接口的类，才能够被Handler适配器当成是Handler处理器来进行调用。所以进入下一步，编写Handler。

### 第三步：编写Handler

package com.langsin.controller;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.ModelAndView;

import org.springframework.web.servlet.mvc.Controller;

public class Test implements Controller {

@Override

public ModelAndView handleRequest(HttpServletRequest arg0, HttpServletResponse arg1) throws Exception {

// TODO Auto-generated method stub

ModelAndView view = new ModelAndView();

view.setViewName("/WEB-INF/jsp/Test.jsp");

view.addObject("msg", "我的第一个servletMVC程序。");

return view;

}

}

### 第四步：配置处理器映射器

在springmvc.xml文件中配置处理器映射器，如下所示：

<!--

配置处理器映射器

SpringMVC提供了一个BeanNameUrlHandlerMapping类，此类一个映射器，该类的作用是将handlerBean的name属性当做url进行匹配。

-->

<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>

<bean name="/test.action" class="com.langsin.controller.Test"/>

上图中的test.action就是应用中访问的url路径。

测试：

地址栏输入：

http://localhost:8080/网站/test.action

### 第五步：配置视图解析器

在springmvc.xml文件中配置是解析器，如下所示：

<!-- 配置视图解析器，解析jsp，默认使用jstl标签库 -->

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<!--前缀-->

<property name="prefix" value="/WEB-INF/jsp/" />

<!--后缀-->

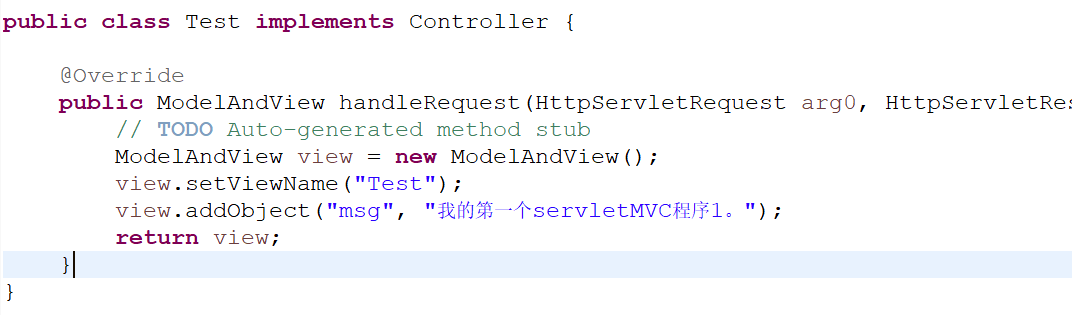
<property name="suffix" value=".jsp" />

</bean>

prefix属性：执行访问路径中固定的前缀路径。

suffix属性：执行访问路径中固定的后缀路径。

使用此种方式：第三步中的代码可修改如下：



## 1.4 非注解处理器映射器、适配器

**SimpleUrlHandlerMapping处理器映射器：**

通过上面的入门程序中，对于处理器映射器BeanNameUrlHandlerMapping就是一个非注解的处理器映射器，只不过这个映射器稍微有点缺陷。缺陷就是不能对bean的url映射进行集中管理。

SpringMVC还提供了另外一个非注解的处理器映射器SimpleUrlHandlerMapping，这个映射器可以实现对bean

的url的集中管理，首先配置一个handler Bean通过bean ID与property元素中的prop子元素关联，key属性配置的是访问的业务的url，prop元素内容为bean ID，表示当访问到/test1.action业务时，SpringMVC通过映射器查询到test handler，然后将它交由Handler适配器去执行。如下所示：

<!-- SimpleUrlHandlerMapping处理器映射器，便于集中管理处理器映射 -->

<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">

<property name="mappings">

<props>

<prop key="/test.action">test</prop>

<prop key="/test2.action">test2</prop>

</props>

</property>

</bean>

<bean id="test" class="com.langsin.controller.Test"/>

<bean id="test2" class="com.langsin.controller.Test2"/>

**HttpRequestHandlerAdapter适配器：**

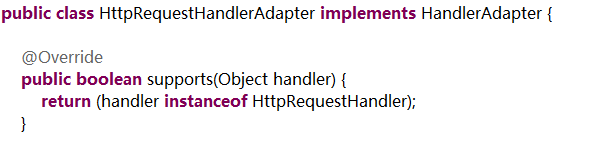
在上面的入门程序中，org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter就是一个非注解的适配器。此适配器要求所有的Handler处理器必须实现Controller接口。使用方式参考入门程序。

org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter是SpringMVC提供的另外一种非注解的处理器适配器，配置如下图所示：

<!-- 配置另一个处理器适配器 -->

<bean class="org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter" />

查看该类的原码如下：



此适配器要求Handler处理器必须实现HttpRequestHandler接口，测试类如下所示：

public class Test3 implements HttpRequestHandler {

@Override

public void handleRequest(HttpServletRequest arg0, HttpServletResponse arg1) throws ServletException, IOException {

// TODO Auto-generated method stub

arg0.setAttribute("msg", "我的第一个springMVC 程序3 ");

arg0.getRequestDispatcher("/WEB-INF/jsp/Test.jsp").forward(arg0, arg1);

}

}

通过上面测试类的编写，大家会发现这种编程非常类似JSP、Servlet中Servlet的代码编写。大家可能会认为这种方式不如上面实现Controller接口的handler更加直观。但是这种方式却比较灵活，因为实现Controller接口的handler返回值就是一个ModelAndView对象，如果需要服务器端响应后，返回json格式的数据，返回image对象数据，它就不能实现。而使用第二种方式，可以通过修改response对象，设置响应格式，所以在开发时可根据需要实现不同类型的handler。

## 1.5 注解处理器映射器、适配器

### 1.5.1 配置注解的映射器、适配器

<!-- 配置注解的处理器映射器 -->

<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping" />

<!-- 配置注解的处理器适配器 -->

<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter" />

在3.1之后的注解的处理器映射器、适配器添加了很多的优化以及参数设置，所以在开发时，推荐使用3.1之后的版本。3.1版本中，在SpringMVC中提供了一个mvc:annotation-driven元素，用于简化映射器、适配器在springmvc.xml文件中的配置。不仅可以替代上面两行的配置，而且mvc:annotation-driven的驱动类还默认加载了多个参数绑定的方法，比如json转换解析器就默认加载。所以开发时推荐使用该方式。如下所示：

<!-- 配置注解的处理器映射器 -->

<mvc:annotation-driven />

### 1.5.2 开发注解的处理器Handler

package com.langsin.controller;

import org.springframework.stereotype.Controller;

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

import org.springframework.web.servlet.ModelAndView;

@Controller

public class Test {

@RequestMapping("test.action")

public ModelAndView test(){

// TODO Auto-generated method stub

ModelAndView view = new ModelAndView();

view.setViewName("Test");

view.addObject("msg", "我的第一个servletMVC程序1。");

return view;

}

}

加载该Handler可以使用Spring的批量加载扫描的方式，如下所示：

<!-- 自动扫描 -->

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

小结：

处理器映射器：

非注解的处理器映射器（了解）

注解的处理器映射器（掌握）

对标记@Controller注解的类，Spring容器自动加载该Bean并管理，对于该Bean中的使用@RequestMapping注解进行修饰的方法进行映射。[在@RequestMapping里边定义映射的url路径。使用注解的映射器不需要在xml中配置url和Handler](mailto:在@RequestMapping里边定义映射的url路径。使用注解的映射器不需要在xml中配置url和Handler)之间的映射关系。

处理器适配器：

非注解的处理器适配（了解）

注解的处理器适配器（掌握）

注解的适配器与注解的映射器是配对使用，也就是说注解的适配器不能使用非注解映射进行映射。

## 1.6 Spring+SpringMVC+Mybatis整合开发

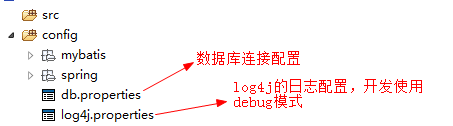
通过SpringMVC入门程序的学习，以及对非注解、注解处理器映射器、处理器适配器的了解，大体知道了SpringMVC的基本的开发流程，下面就以用户查询功能模块为示例，来学习Spring框架、SpringMVC框架、MyBatis框架的整合开发。

### 1、引入JAR包

首先将应用所需要的JAR包引入到项目中，包括log4j、mysql、dbcp、commons-io、spring、mybatis、mybatis-spring-1.3.0等Jar文件。

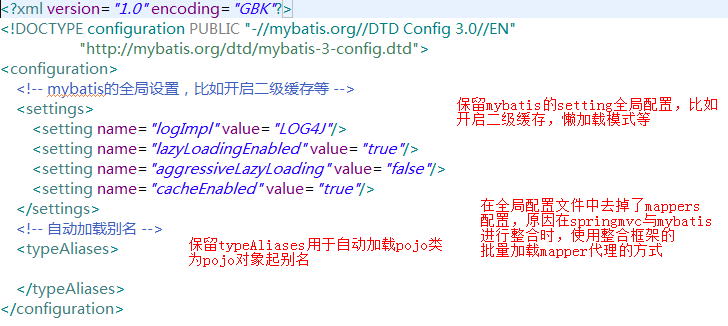
### 2、配置资源文件

配置相关的资源文件，比如log4j、数据库链接配置等，如下图所示：



### 3、mybatis的sqlMapConfig.xml

编写mybatis框架的配置文件sqlMapConfig.xml，在使用Spring框架进行整合开发时会加入不同框架的配置文件，如下图所示：



代码：

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>

<!-- 加载属性文件 -->

<properties resource="config/db.properties"></properties>

<!-- MyBatis的全局参数设置，基本用来进行MyBatis的优化处理 -->

<settings>

<setting name="logImpl" value="LOG4J"/>

<setting name="lazyLoadingEnabled" value="true"/>

<setting name="aggressiveLazyLoading" value="false"/>

<setting name="cacheEnabled" value="true"/>

<setting name="lazyLoadTriggerMethods" value=""/>

</settings>

<!-- 别名 -->

<typeAliases>

<package name="com.langsin.dto"/>

</typeAliases>

</configuration>

### 4、Spring与MyBatis整合

将Spring与mybatis进行整合，配置文件为spring-dao.xml，如下图所示：

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd

">

<!-- c3p0连接池 -->

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource" destroy-method="close">

<property name="driverClass" value="com.mysql.jdbc.Driver"/>

<property name="jdbcUrl" value="jdbc:mysql://127.0.0.1:3306/ssmtest"/>

<property name="user" value="root"/>

<property name="password" value="123456"/>

<property name="maxPoolSize" value="200"/>

<property name="minPoolSize" value="2" />

<property name="initialPoolSize" value="2" />

<property name="maxIdleTime" value="200"/>

</bean>

<!-- sqlSessionFactory -->

<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">

<!-- 数据库连接池 -->

<property name="dataSource" ref="dataSource" />

<!-- 加载mybatis的全局配置文件 -->

<property name="configLocation" value="classpath:config/sqlMapConfig.xml" />

</bean>

<!-- mapper 扫描器 -->

<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">

<!-- 扫描包路径，如果需要扫描多个包，中间使用半角逗号隔开 -->

<property name="basePackage" value="com.langsin.dao"></property>

<property name="sqlSessionFactoryBeanName" value="sqlSessionFactory"/>

</bean>

</beans>

**注意：**

**需要导入mybatis-spring-1.3.0.jar 包。否则org.mybatis.spring.SqlSessionFactoryBean会找不到。**

### 5、配置SpringMVC控制层

SpringMVC的配置文件 spring-mvc.xml，主要配置视图解析器，处理器映射器、处理器适配器，在配置文件中使用mvc:annotation-dirven驱动来加载映射器与适配器。如下图所示：

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:mvc="http://www.springframework.org/schema/mvc"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc.xsd

">

<!-- 配置注解的处理器映射器 -->

<mvc:annotation-driven />

<!-- 自动扫描 -->

<context:component-scan base-package="com.langsin" />

<!-- 配置视图解析器，解析jsp，默认使用jstl标签库 -->

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<!--前缀-->

<property name="prefix" value="/WEB-INF/jsp/" />

<!--后缀-->

<property name="suffix" value=".jsp" />

</bean>

</beans>

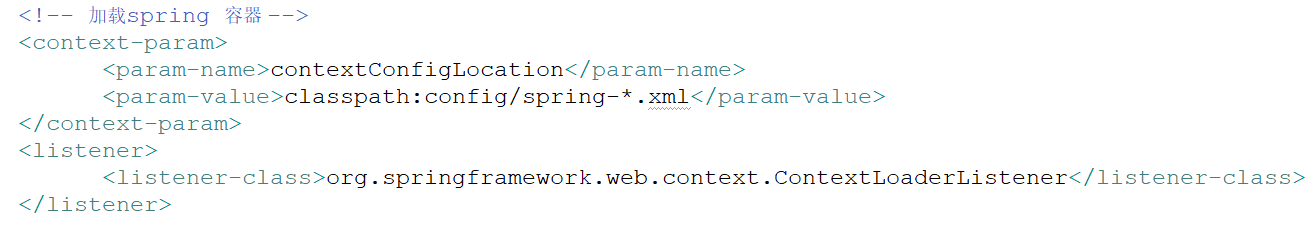
### 6、配置Spring容器

以上所有配置文件配置完成后，在web.xml文件中通过配置SpringMVC的前端控制器，则可以完成将springmvc.xml配置文件加载到应用的容器中去，如下所示：



此操作仅仅是完成了SpringMVC模块的加载，但是Spring容器还没有加载到Web应用中来，所以下一步需要将Spring容器加载到Web应用的容器中，通过Spring框架提供的org.springframework.web.context.

ContextLoaderListener监听器来完成，此监听器在启动时会自动加载web.xml配置文件中的context-param元素中指定contextConfigLocation的value值，如下所示：



完整配置：

<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd" id="WebApp\_ID" version="3.0">

<display-name>MybatisStudy1</display-name>

<servlet>

<servlet-name>springmvc</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<!--

使用contextConfigLocation参数用于配置SpringMVC需要加载的配置文件(处理器映射器、处理器适配器、视图解析器等）。如果没有配置，SpringMVC默认加载的是/WEB-INF/下面的servlet名称-servlet.xml文件，即springmvc-servlet.xml文件。

-->

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:config/spring-mvc.xml</param-value>

</init-param>

</servlet>

<servlet-mapping>

<servlet-name>springmvc</servlet-name>

<url-pattern>\*.action</url-pattern>

</servlet-mapping>

<!-- 加载spring 容器 -->

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:config/spring-\*.xml</param-value>

</context-param>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<welcome-file-list>

<welcome-file>index.html</welcome-file>

<welcome-file>index.htm</welcome-file>

<welcome-file>index.jsp</welcome-file>

<welcome-file>default.html</welcome-file>

<welcome-file>default.htm</welcome-file>

<welcome-file>default.jsp</welcome-file>

</welcome-file-list>

</web-app>

到此SpringMVC+Spring+Mybatis的整合全部结束！

## 1.7 SSM框架调试

整合工作完成后，下面来开发用户查询功能模块来进行测试。

### 1.7.1 dao开发

接口：

package com.langsin.dao;

import java.util.List;

import com.langsin.dto.\*;

public interface UserDao {

// 通过id来查询用户

public User getUserById(String id) throws Exception;

// 通过用户名来查询用户

public List<User> getUserByName(String name) throws Exception;

// 根据id删除用户

public int deleteUserById(String id) throws Exception;

// 更新用户信息

public int updateUser(User user) throws Exception;

}

Xml:

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Config 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<!-- 命名空间的作用是对SQL进行分类管理，起到隔离SQL的作用，如果使用mapper代理，namespace有特殊重要作用 -->

<mapper namespace="com.langsin.dao.UserDao">

<!-- 通过select元素配置查询语句，id属性值，可以理解为标识映射文件中的SQL，因为最终将SQL语句封装到mappedStatement对象中去，所以id为statement的id ，在此SQL中，使用#{}表示占位符，花括号中的值为userId表示要接收的参数，参数的名称为userId，如果输入的参数是基本数据类型，#{}中的参数可以为任意名称，但是一般都遵循开发规范写形参名，parameterType是SQL要接收的参数的数据类型，而resultType是select进行查询后，返回的数据类型，因为是单条查询，所以映射为一个Java Bean对象-->

<select id="getUserById" parameterType="String" resultType="User">

select \* from t\_user where userId=#{id}

</select>

<select id="getUserByName" parameterType="String" resultType="User">

select \* from t\_user where name like concat('%',#{value},'%')

</select>

<delete id="deleteUserById" parameterType="String">

delete from t\_user where userId=#{id}

</delete>

<update id="updateUser" parameterType="User">

update t\_user set cname=#{user.name} where user\_id=#{user.user\_id}

</update>

</mapper>

### 1.7.2 service层开发

接口：

package com.langsin.service;

import java.util.List;

import com.langsin.dto.User;

public interface UserService {

List<User> getUsers();

}

实现类：

package com.langsin.service.impl;

import java.util.List;

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

import org.springframework.stereotype.Service;

import com.langsin.dao.UserDao;

import com.langsin.dto.User;

import com.langsin.service.OrderService;

import com.langsin.service.UserService;

@Service

public class UserServiceImpl implements UserService {

@Autowired

private UserDao userDao;

public UserServiceImpl(){

System.out.println("UserService 初始化");

}

public List<User> getUsers(){

try {

return userDao.getUserByName("");

} catch (Exception e) {

// TODO Auto-generated catch block

e.printStackTrace();

return null;

}

}

}

### 1.7.3 Controller层开发

下面来编写Controller层（Handler层），如下所示：

package com.langsin.controller;

import java.util.List;

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

import org.springframework.stereotype.Controller;

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

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

import org.springframework.web.servlet.ModelAndView;

import com.langsin.dto.User;

import com.langsin.service.UserService;

@Controller

@RequestMapping("/users")

public class UserController {

public UserController(){

System.out.println("UserController 初始化");

}

@Autowired

private UserService userService = null;

@RequestMapping(value="/query.action",method={RequestMethod.GET})

public ModelAndView query(){

ModelAndView view = new ModelAndView();

List<User> users = userService.getUsers();

view.addObject("models", users);

view.setViewName("User");

return view;

}

}

在Controller类中，使用注解@Controller进行修饰，由Spring框架完成自动加载，用户在访问该

url：users/query.action时，SpringMVC根据此处理器映射器找到该类，然后再将该类交由SpirngMVC的处理器适配器与执行该类，最后返回一个view。

## 1.8 请求与响应

在上面的示例中仅仅是完成最简单的操作，如果需要完成稍微复杂一点的开发，比如限制请求方式、往后台传入参数等，这就需要利用SpringMVC框架提供的注解@RequestMapping，以及处理器Controller层中方法的返回值以及传入参数来完成。

### 1.8.1 @RequestMapping

通过RequestMapping注解可以定义不同的处理器映射规则。

* **URL路径映射：**@RequestMapping(value="/user")或@RequestMapping("/user"），value的值是数组，可以将多个url映射到同一个方法。
* **窄化请求映射：**在class上添加@RequestMapping(url)指定通用请求前缀， 限制此类下的所有方法请求url必须以请求前缀开头，通过此方法对url进行分类管理。

例如：

@RequestMapping放在类名上边，设置请求前缀

@Controller

@RequestMapping("/user")

方法名上边设置请求映射url：

@RequestMapping放在方法名上边，如下：

@RequestMapping("/query.action ")

访问地址为：/user/query.action

* **请求方法限定：**

**限定GET方法**

@RequestMapping(method = RequestMethod.GET)

如果通过Post访问则报错：

HTTP Status 405 - Request method 'POST' not supported

例如：

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

**限定POST方法**

@RequestMapping(method = RequestMethod.POST)

如果通过Post访问则报错：

HTTP Status 405 - Request method 'GET' not supported

GET和POST都可以

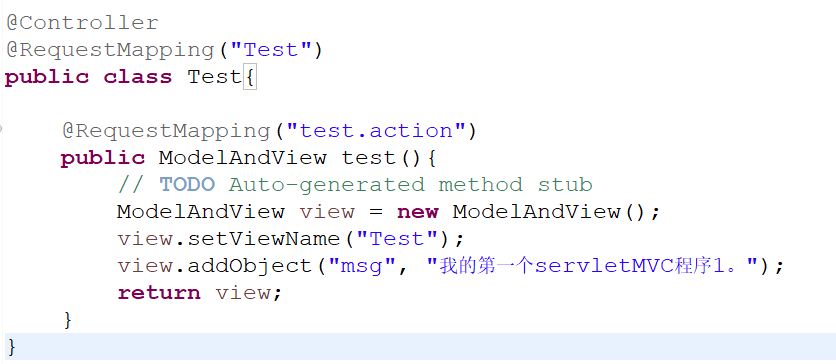
@RequestMapping(method={RequestMethod.GET,RequestMethod.POST})

### 1.8.2 Controller方法的返回值

在Controller层中，响应处理请求的方法，可以定义如下几种类型，返回一个ModelAndView或者返回一个String字符串，或者使用void修饰。

#### 返回ModelAndView

controller方法中定义ModelAndView对象并返回，对象中可添加model数据、指定view。如下所示：



#### 返回String字符串

使用字符串作为返回值类型时，Controler层中的方法可以实现如下三种功能：

1. 返回逻辑视图名
2. 实现重定向
3. 实现请求转发

**1、返回逻辑视图名**

**用法1：**通过 HttpServletRequest 实现数据传递。

@RequestMapping("test1.action")

public String test1(HttpServletRequest request){

// TODO Auto-generated method stub

request.setAttribute("msg", "我的第一个servletMVC程序 返回逻辑视图名测试1。");

return "Test";

}

**用法2：**使用 Model类型的参数实现数据传递。

@RequestMapping("test2.action")

public String test2(Model model){

// TODO Auto-generated method stub

model.addAttribute("msg", "我的第一个servletMVC程序 返回逻辑视图名测试2。");

return "Test";

}

controller方法返回字符串可以指定逻辑视图名，通过视图解析器解析为物理视图地址。

**物理视图地址=视图解析器前缀+逻辑视图名+视图解析器后缀**

重点：使用逻辑视图名作为返回值时，就不需要使用ModelAndView这个数据模型对象来封装要显示的数据及显示的JSP页面了，但是String仅仅是表示转向的JSP页面而已，所以数据的封装可以使用形参来进行封装。Controller处理器默认可以使用的形参在后面会讲到。

**2、实现重定向**

springMVC中实现重定向，使用redirect关键字。

案例：

@RequestMapping("test2.action")

public String test2(Model model){

// TODO Auto-generated method stub

model.addAttribute("msg", "我的第一个servletMVC程序 返回逻辑视图名测试2。");

return "Test";

}

@RequestMapping("test3.action")

public String test3(){

// TODO Auto-generated method stub

return "redirect:test2.action";

}

**3、实现请求转发**

请求转发与Servlet中的请求转发一致，服务器的多次处理在一个请求范围内，所以数据模型model可以实现共享。如下所示：

@RequestMapping("test2.action")

public String test2(Model model){

return "Test";

}

@RequestMapping("test3.action")

public String test3(Model model){

// TODO Auto-generated method stub

model.addAttribute("msg", "我是test3.action,但是我是通过 test2.action进行的跳转界面");

return "forward:test2.action";

}

**注意：重定向和请求转发，在进行处理时，后面要跟的url是一个handler的url而不能是一个逻辑视图：即不能是一个普通的jsp的路径！**

总结：

请求重定向可有效避免表单重复提交，但是无法使用request作用域。请求转发可以使用request作用域，但是无法避免表单重复提交。所以，需要在设计层面去考虑使用请求重定向和请求转发。

经典案例：

信息编辑成功后刷新列表页。

例如：

1. 新增或修改完商品后，刷新商品列表页面。
2. 新增或修改完学生信息后，刷新学生列表页面。

#### 返回void

所谓的返回void实际上是利用了Controller方法的形参，request和response进行开发，变为原始的servlet开发方式了。

1、使用request进行请求转发

request.getRequestDispatcher(“url”).forwart(request,response);

2、使用response进行重定向

response.sendRedirect(“url”);

3、也可以通过response指定响应结果，例如响应json数据

response.setContentType(“application/json”);

response.getWriter().write(“json串”);

## 1.9 参数绑定

在上一节的内容中，我们看到Controller方法默认支持的形参中包括了request对象，可以通过request对象来获取客户端请求的key/value对的数据，把key/value对绑定到request对象中的这步操作是由SpringMVC框架中的参数绑定组件来帮我们完成的，这步操作就称为参数绑定。

### 1.9.1 参数绑定过程

参数绑定的实际过程如下：

* 处理器适配器调用Springmvc提供参数绑定组件将key/value数据转成controller方法的形参
* 参数绑定组件：在spirngmvc早期版本使用PropertyEditor（只能将字符串转成java对象），后期使用converter（进行任意类型的转换），spirngmvc提供了很多converter（转换器），在特殊情况下需要自定义converter。

例如：对日期数据绑定有的时候需要自定义converter

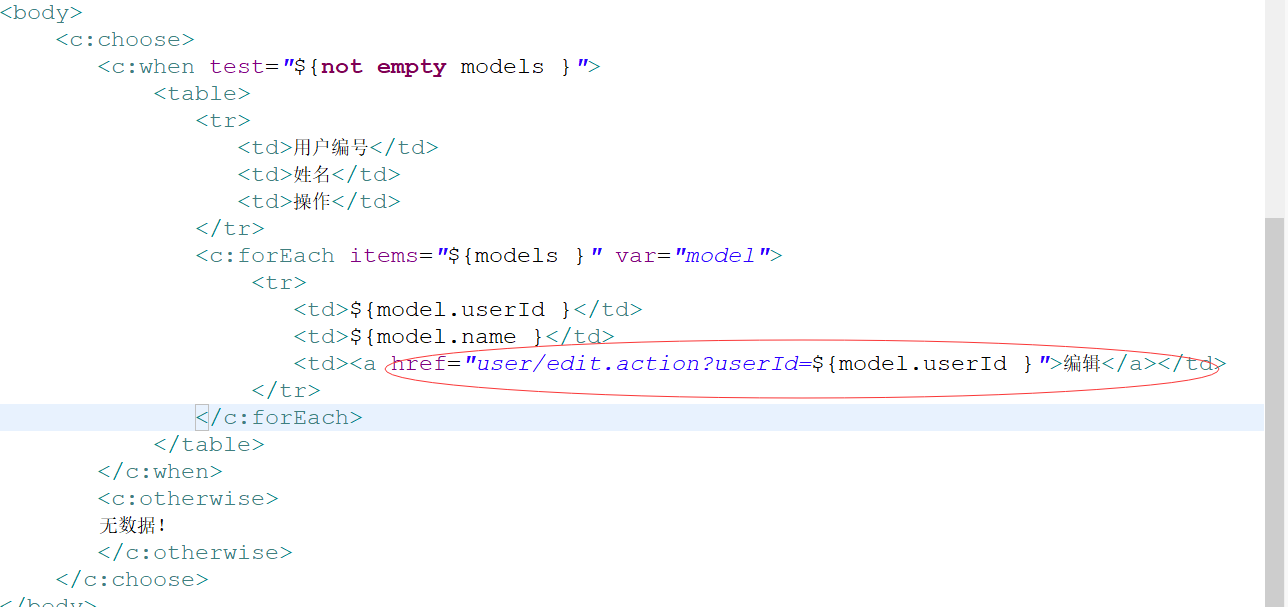
### 1.9.2 SpringMVC支持的简单类型

当请求的参数名称和处理器形参名称一致时会将请求参数与形参进行绑定。默认支持的简单类型为：

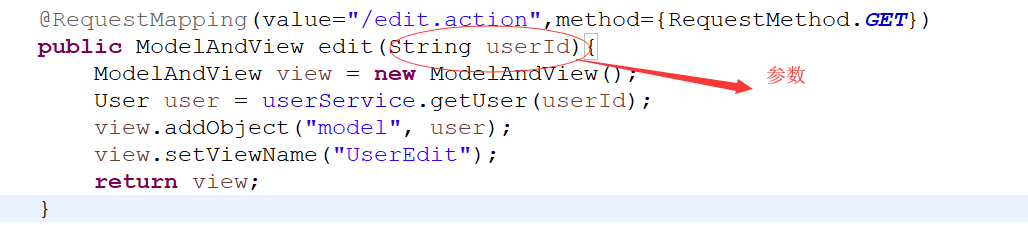
* 整型
* 字符串
* 单精度/双精度
* 布尔类型

以用户信息修改为例：

用户列表页面，增加一“编辑”列：



控制器里面增加edit.action 如下图所示:

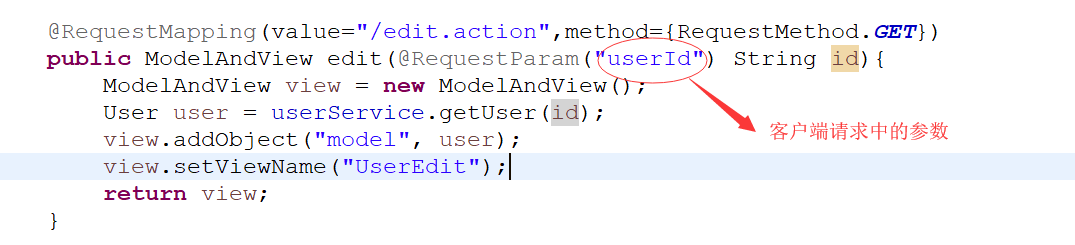


控制器里面能够取到get请求中 userId 的值。这种方式要求请求的参数名与控制器中方法的参数名必须完全相同。

### 1.9.3 @RequestParam

通过@RequestParam可以对简单类型的参数进行绑定。但是不用限制request传入参数名称和controller方法的形参名称一致。

上面示例中，控制器的代码可以修改为下面的代码：

这种情况下，SpringMVC就会将前面传过来的userId绑定到参数id上面。

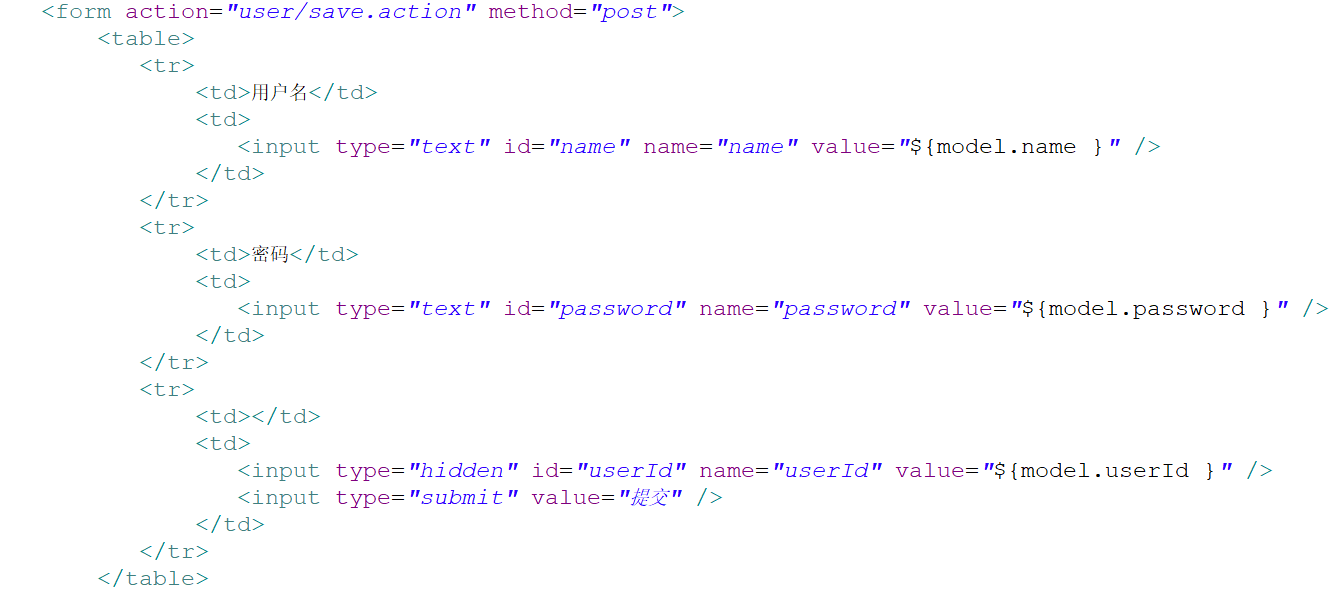
### 1.9.4 绑定POJO类型（重点）

* **普通pojo**

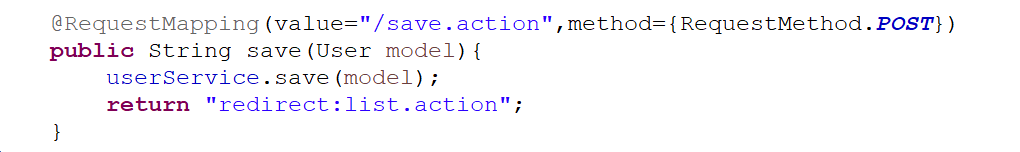
如果在controller的方法中直接定义pojo的形参，如果想要将表单中的数据绑定到pojo对象里面去，需要页面中input的name和controller的pojo形参中的属性名称一致，这样页面中数据将直接绑定到pojo对象上，不需要额外的注解。

示例：

界面：



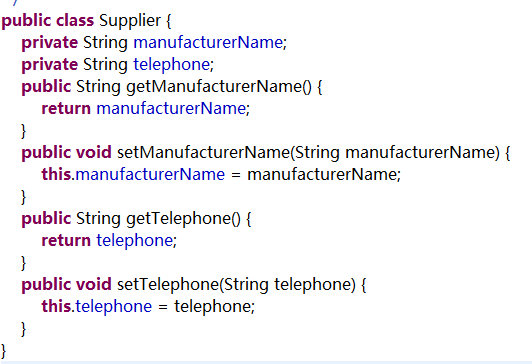
控制器：



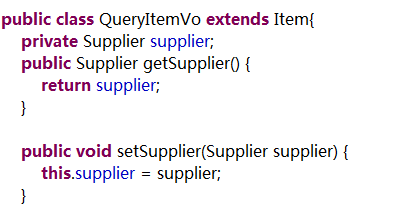
* **包装pojo**

对于包装类型的pojo，在进行参数绑定时按照如下规范即可。

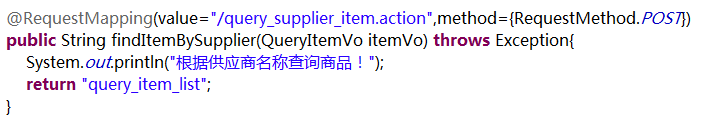
代码1：商品的供应商类



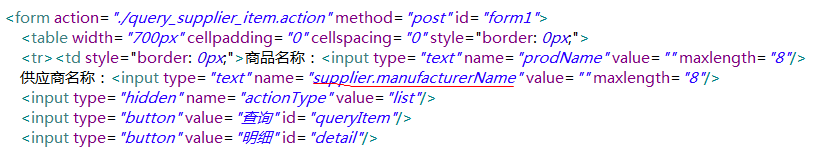
代码2：商品的扩展类



代码3：处理请求的Controller方法



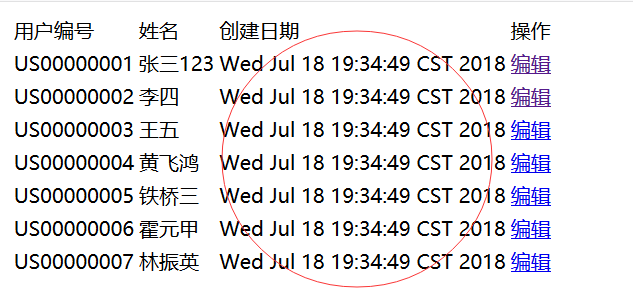
前台页面代码：OGNL（对象导航图语句）

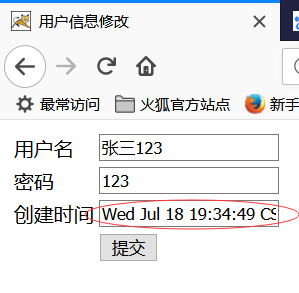


### 1.9.5 自定义类型绑定

**日期问题：**

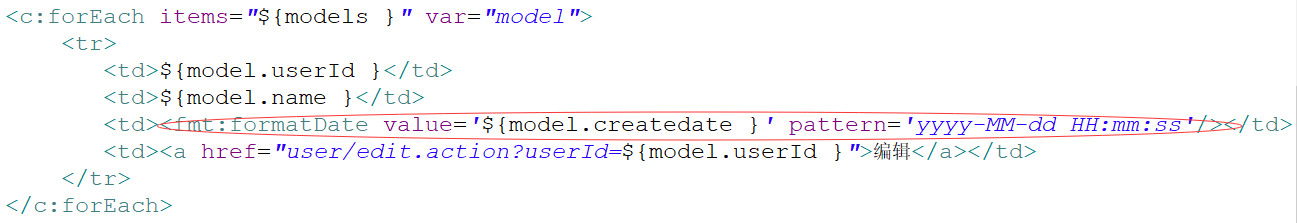
后台的DTO对象中日期字段全部是 Date类型的。如果不做任何处理直接展示的话，前台展示的数据如下图所示：





**解决方法：**

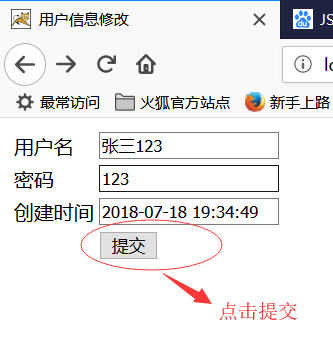
数据展示：



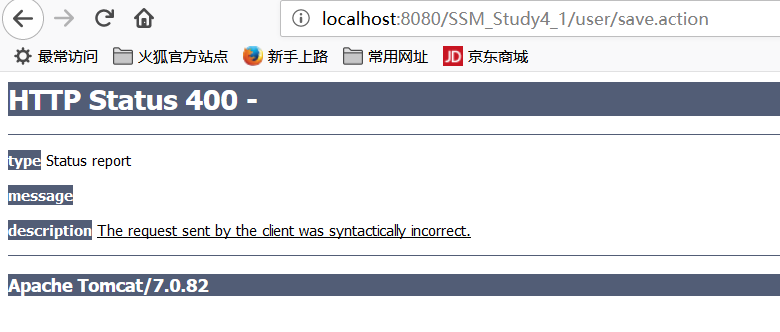
输入框中：

<input type="text" id="createdate" name="createdate" value="<fmt:formatDate value='${model.createdate }' pattern='yyyy-MM-dd HH:mm:ss'/>"/>

修改用户信息，点击提交：



结果：



**400错误说明：**

在ajax请求后台数据时有时会报 HTTP 400 错误 - 请求无效 (Bad request);出现这个请求无效报错说明请求没有进入到后台服务里；

原因：

1）前端提交数据的字段名称或者是字段类型和后台的实体类不一致，导致无法封装；

 2）前端提交的到后台的数据应该是json字符串类型，而前端没有将对象转化为字符串类型；

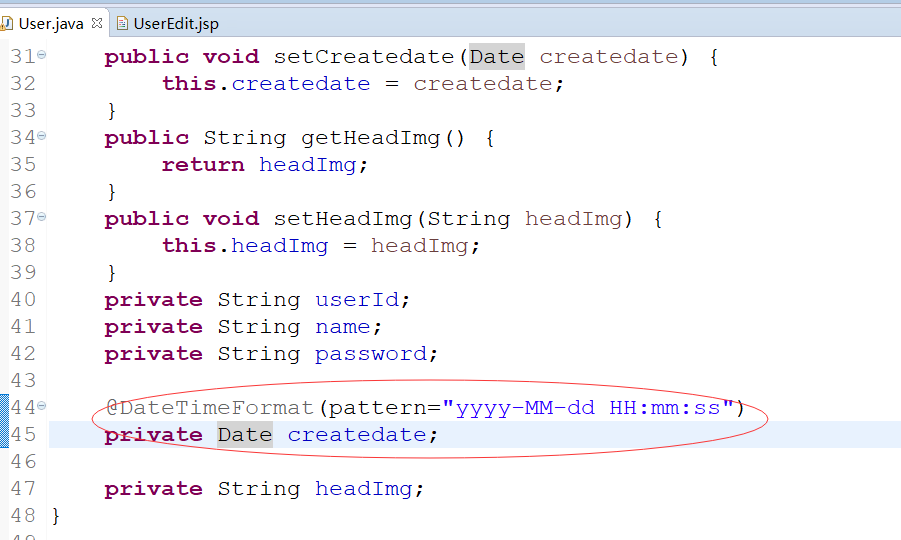
**解决方法：**

1. **对pojo类的日期字段添加注解（@DateTimeFormat）**

@DateTimeFormat(pattern="yyyy-MM-dd HH:mm:ss")

pattern指的是符合要求的字符串格式。

示例：



1. **添加自定义类型绑定**

对于controller形参中pojo对象，如果属性中有日期类型，需要自定义参数绑定。将请求日期字符串转换成日期类型，要转换的日期类型和pojo中日期属性的类型保持一致。

转换成java.util.Date类型。要转成此种类型，需要向处理器适配器中注入自定义的参数绑定组件。如下所示：

<!-- 向处理器适配器中注入自定义的参数绑定组件 -->

<mvc:annotation-driven conversion-service="conversionService" />

在SpringMVC的处理器适配器中指定conversion-service属性，conversionService是SpringMVC提供的接收自定义的类型转换的服务Bean。此Bean为FormattingConversionServiceFactoryBean类，配置该类时需要为该Bean的converters属性指定一个list集合，集合中存储的就是实现自定转换的服务Bean。如下所示：

<!-- 自定义类型绑定 -->

<bean id="conversionService" class="org.springframework.format.support.FormattingConversionServiceFactoryBean">

<property name="converters">

<list>

<bean class="com.langsin.converter.CustomDateConverter" />

</list>

</property>

</bean>

在SpringMVC中一个自定义类型转换Bean需要实现Convert接口，如下所示：CustomDateConverter类实现自定义日期格式转换，代码如下：

CustomDateConverter 类：

package com.langsin.converter;

import java.text.SimpleDateFormat;

import java.util.\*;

import org.springframework.core.convert.converter.Converter;

public class CustomDateConverter implements Converter<String, Date> {

@Override

public Date convert(String source) {

SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd hh:mm:ss");

try {

return dateFormat.parse(source);

} catch (java.text.ParseException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

return null;

}

}

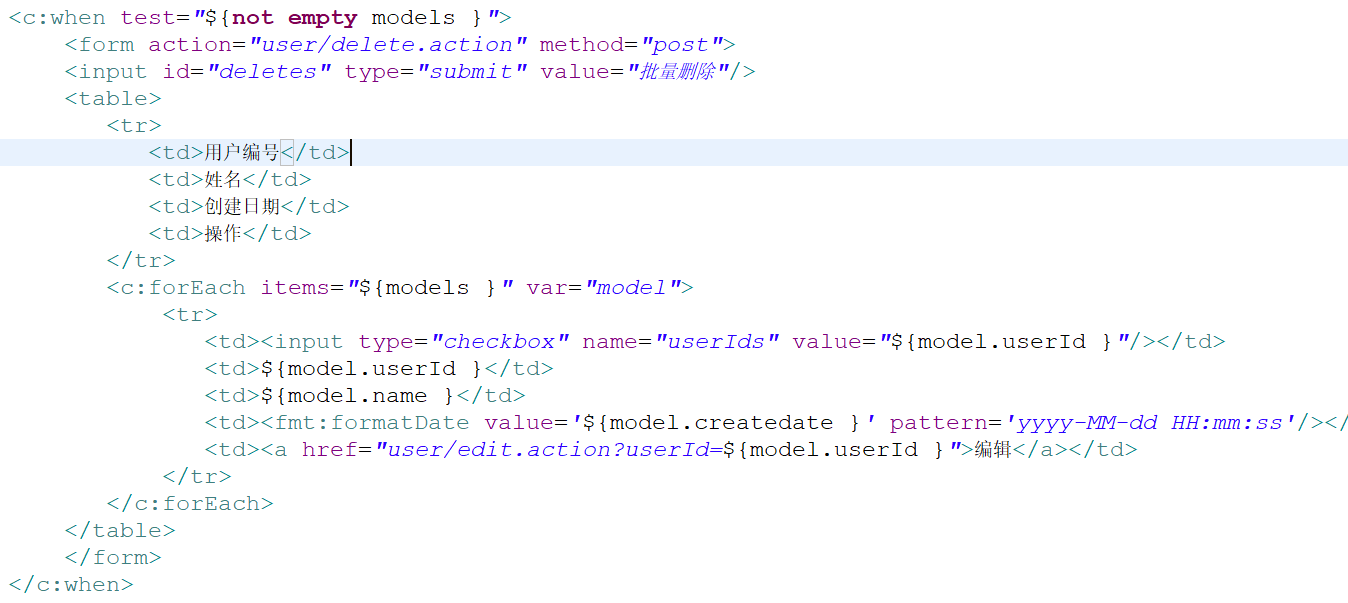
### 1.9.6 数组、List、Map参数绑定

* **数组绑定**

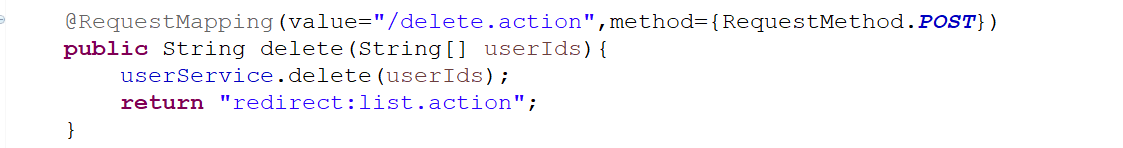
对于数组类型的参数绑定，只需要Controller处理方法中参数名与前台页面中的name属性保持一致即可，例如：批量删除。

批量删除关键实现代码：

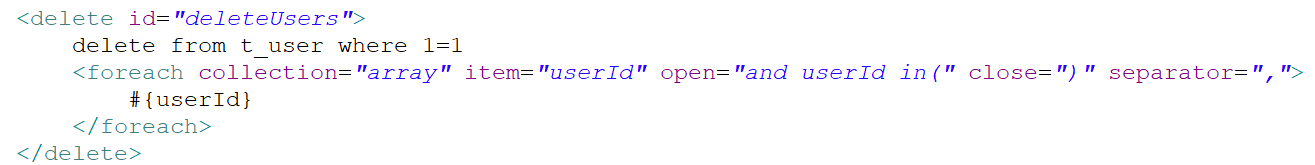
界面：



控制器：



Mapper:

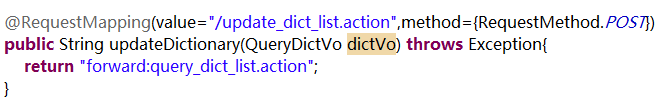


* **List参数绑定**

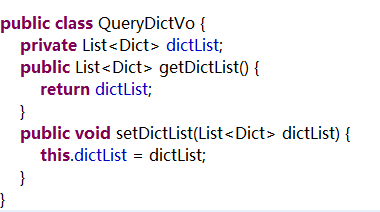
对于List集合的参数绑定，需要将List集合作为一个属性封装到一个包装类pojo中才可以。如下所示：

例如：批量保存。假设需要对字典表中的数据进行更新，对于字典表来说就三、五个字段，完全可以放在一个页面中进行整条记录的维护，而且可以一次性维护多条记录，那么在调用Controller的方法时，传入的参数为List集合对象。

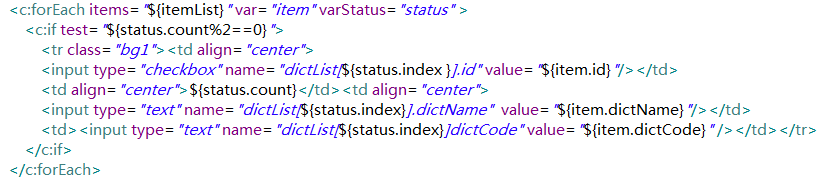
Controller方法中的形参列表格式如下：



对于dictVo这个包装类，代码如下：

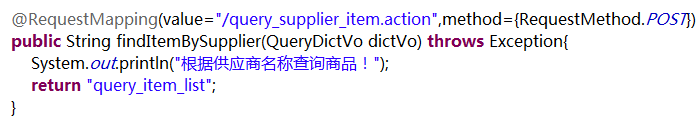


此类中包含的一个dictList集合属性，前台页面中table表格中每条记录对应数据表中一条记录，就三个字段，对应的每个文本框的name属性值为：dictList[${status.index}].dictName，dictList对应形参名，[]中括号对应的索引值，后面dictName集合中每个对象中的dictName属性。

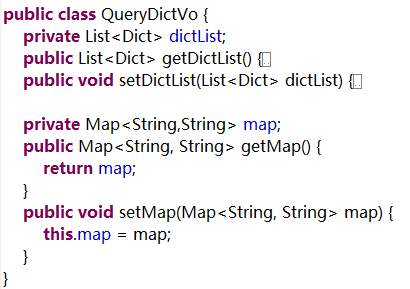


* Map集合参数绑定

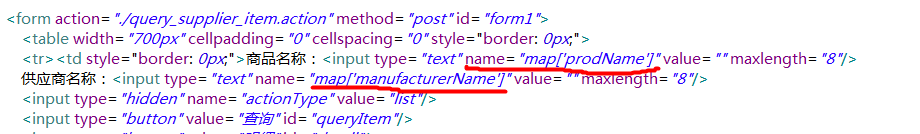
原理同List集合绑定一样，也需要一个包装类型的pojo对象，对于Controller方法如下：



包装pojo类代码如下：



前台页面代码如下：



## 1.10 Spring中文乱码解决

配置一个Spring的Filter过滤器即可，如下所示：

<!-- 配置Spring的过滤器 -->

<filter>

<filter-name>CharacterFilter</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>CharacterFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

# 2、SpringMVC高级应用

本节讲解SpringMVC框架的高级使用部分。

## 2.1 SpringMVC的校验

B/S结构的系统对http请求的数据的校验多数在客户端进行，这也是出于简单以及用户体验上考虑，但是在一些安全性要求高的系统，服务端的校验也是不可缺少的。另外目前大部分系统除了客户端是浏览器之外，还有手机客户端、WebService接口调用等。

SpringMVC提供了一种Bean Validation的校验机制，采用了Hibernate Vaildator的校验，但是与Hibernate持久层框架没有任何关系。

### 2.1.1 理解校验

服务端校验一般分以下三种情况：

控制层controller：校验页面请求的参数的合法性。在服务端控制层conroller进行校验，不区分客户端类型（浏览器、手机客户端、远程调用）

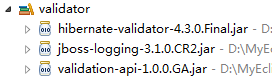
业务层service（使用较多）：主要校验关键业务参数。

持久层dao：一般是不校验的。

对于SpringMVC的校验来说，并不是要完成业务的逻辑校验，仅仅是校验参数的合法性，因此SpringMVC的校验是在controller这层完成的。

### 2.1.2 配置环境

Hibernate的Validator的校验框架，所需要的JAR包为，如下图所示：



版本号不要出现错误，不然容器出现异常信息。

### 2.1.3 配置校验器

Spring框架提供的Context包下，包含了一个LocalValidatorFactoryBean，本地校验器工厂bean，该类可以将已经实现javax.validation.spi.ValidationProvider接口的类当做校验器组件进行管理。使用属性providerClass进行接收，而hibernate-validator包下的org.hibernate.validator.HibernateValidator类已经实现该接口，所以可以将该类注入给providerClass属性：如下所示：

在springmvc.xml中增加：

<!-- 配置SpringMVC的校验器 -->

<bean id="validator" class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean">

<property name="providerClass" value="org.hibernate.validator.HibernateValidator" />

<!-- 指定校验使用的资源文件，在文件中配置错误信息。如果不指定则默认使用classpath下面的ValidationMessages.properties -->

<property name="validationMessageSource" ref="messageSource" />

</bean>

属性validationMessageSource属性用于接收错误消息资源，所以下面配置messageSource错误消息Bean

<bean id="messageSource" class="org.springframework.context.support.ReloadableResourceBundleMessageSource">

<property name="basenames">

<list>

<value>classpath:config/errorMessage</value>

</list>

</property>

<property name="fileEncodings" value="UTF-8" />

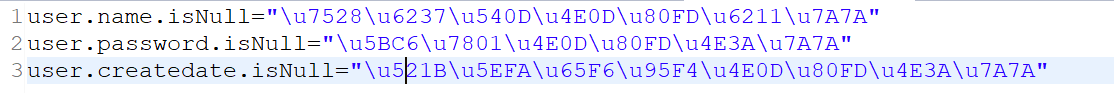
<property name="cacheSeconds" value="120" />

</bean>

配置该Bean时需要为besenames属性指定一个数组类型，表示该Bean需要加载的属性文件的名称，例如classpath:errorMessage表示，加载该类加载路径下的errorMessage.properties属性文件。



properties文件显示：



### 2.1.4 将校验器注入到处理器适配器

将配置好的校验器注入到处理器适配器的驱动Bean中，如下所示：

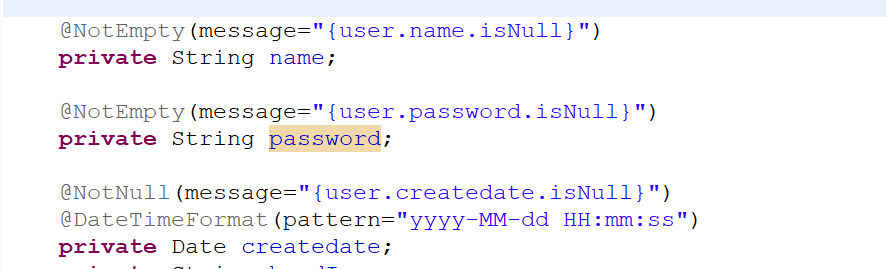
<!-- 将校验器加入适配器处理器中 -->

<mvc:annotation-driven validator="validator" />

### 2.1.5 添加校验规则

如果需要对用户的传入信息做校验，则需要在对应的pojo类的属性上进行校验，如下所示：

dto : User中：



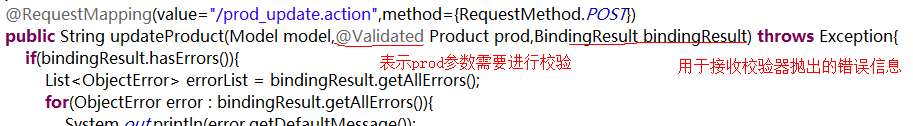
当校验不通过时，校验器则会抛出错误信息，错误信息为errorMessage.properties属性文件中

User.name.isNull 、user.password.isNull、user.createdate.isNull

属性所对应的value值。

### 2.1.6 捕获校验错误信息

如果想获取校验器抛出的错误信息，需要在Controller层的处理方法中加入如下注解：



完整代码：

@RequestMapping("save.action")

public String save(@Validated User user,BindingResult bindingResult){

if(bindingResult.hasErrors()){

for(ObjectError v : bindingResult.getAllErrors()){

System.out.println(v.getDefaultMessage());

}

throw new RuntimeException("sava.action 的参数错误。");

}else{

this.userService.save(user);

return "redirect:list.action";

}

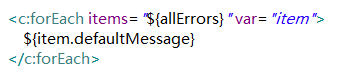
}

注意：

在需要校验的pojo前边添加@Validated，在需要校验的pojo后边添加BindingResult bindingResult接收校验出错信息@Validated和BindingResult bindingResult是配对出现，并且形参顺序是固定的（一前一后）。

完整的B/S系统验证体系包括前端验证和后端验证。前端验证除了数据完整性验证之外，还要考虑验证信息的友好型和美观型。后端验证的意义是前端验证因为某些莫名奇妙的原因失效后的一种保险措施，所以后端验证的重点是数据完整性验证，一旦数据验证失败，抛异常、写日志就可以。至于友好型和美观性则无需考虑。

### 2.1.7 在页面中显示错误信息



通常情况下：后台验证信息无需传给前台。

### 2.1.8 SpringMVC的校验注解

@Null 被注释的元素必须为 null

@NotNull 被注释的元素必须不为 null

@AssertTrue 被注释的元素必须为 true

@AssertFalse 被注释的元素必须为 false

@Min(value) 被注释的元素必须是一个数字，其值必须大于等于指定的最小值

@Max(value) 被注释的元素必须是一个数字，其值必须小于等于指定的最大值

@DecimalMin(value) 被注释的元素必须是一个数字，其值必须大于等于指定的最小值

@DecimalMax(value) 被注释的元素必须是一个数字，其值必须小于等于指定的最大值

@Size(max=, min=) 被注释的元素的大小必须在指定的范围内

@Digits (integer, fraction) 被注释的元素必须是一个数字，其值必须在可接受的范围内

@Past 被注释的元素必须是一个过去的日期

@Future 被注释的元素必须是一个将来的日期

@Pattern(regex=,flag=) 被注释的元素必须符合指定的正则表达式

Hibernate Validator 附加的 constraint

@NotBlank(message =) 验证字符串非null，且长度必须大于0

@Email 被注释的元素必须是电子邮箱地址

@Length(min=,max=) 被注释的字符串的大小必须在指定的范围内

@NotEmpty 被注释的字符串的不能为非空

@Range(min=,max=,message=) 被注释的元素必须在合适的范围内

## 2.2 异常处理器

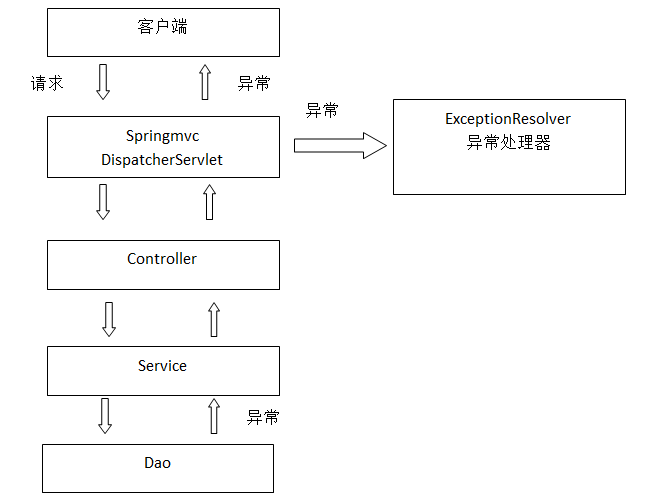
springmvc在处理请求过程中出现异常信息交由异常处理器进行处理，自定义异常处理器可以实现一个统一的异常处理逻辑。

### 2.2.1 异常处理思路

系统中异常包括两类：预期异常和运行时异常RuntimeException，前者通过捕获异常从而获取异常信息，后者主要通过规范代码开发、测试等手段减少运行时异常的发生。

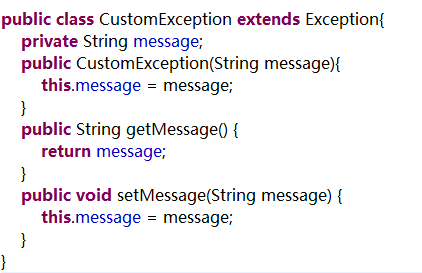
仅管在项目开发完成后，需要进行大量的测试，但仍会存在一些不可预知的错误异常，即使这种错误异常非常少了，但仍不能保证异常一点也不存在，所以针对这种情况，我们就需要由一个统一异常处理机制，来针这种情况（突发异常）进行处理，这就是自定义异常处理机制存在意义。

系统的dao、service、controller出现的异常都通过throws Exception这种方式向上抛出，最后由springmvc前端控制器交由异常处理器进行异常处理，如下图：



### 2.2.2 自定义异常类

为了区别不同的异常，通常根据异常类型自定义一个统一的异常类，这里我们创建一个自定义系统异常，如果controller、service、dao抛出此类异常说明是系统预期处理的异常信息。



代码：

public class CustomerException extends Exception {

private String message;

public String getMessage() {

return message;

}

public void setMessage(String message) {

this.message = message;

}

public CustomerException(String message){

this.message = message;

}

}

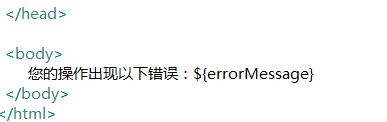
### 2.2.3 自定义异常处理器

自定义异常类编写完成后，需要使用自定义异常处理器来处理系统异常，自定义异常处理器需要实现HandlerExceptionResolver接口。如下所示：



### 2.2.4 错误页面

在jsp页面下创建error文件夹，在此文件夹下创建error.jsp文件



### 2.2.5 异常处理器配置

只需要将异常处理器bean配置在SpringMVC的配置文件中即可，因为该Bean实现了HandlerExceptionResolver接口，SpringMVC框架能自动识别该Bean当成全局的异常处理器。配置如下：

<!-- 配置异常处理器 -->

<bean class="com.langsin.exceptions.CustomerExceptionResolver"/>

### 2.2.6 异常测试

人为抛出异常测试。

@RequestMapping(value="/list.action",method={RequestMethod.GET})

public ModelAndView list() throws Exception{

ModelAndView view = new ModelAndView();

List<User> users = userService.getUsers();

if(users.size() < 1000){

throw new CustomerException("故意报错");

}

view.addObject("models", users);

view.setViewName("User");

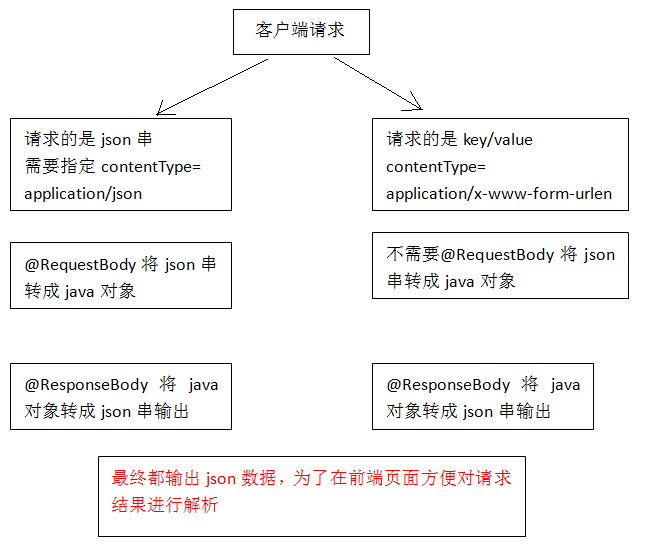
return view;

}

## 2.3 JSON数据交互

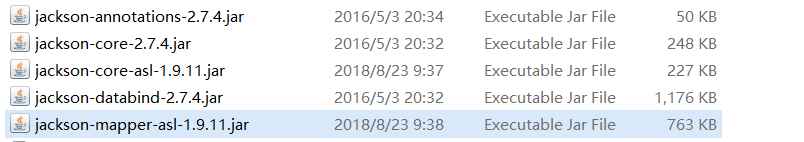
在目前B/S架构的WEB应用开发中，JSON格式的数据类型交互是一种比较常见的应用技术。这是因为JSON格式的数据在接口调用中，html页面中都比较常见。而且JSON格式比较简单，解析还比较方便，所以使用广泛。比如：webService接口、ajax传输json数据等。

### 2.3.1 SpringMVC对JSON的支持



SpringMVC对JSON格式的数据的支持需要用到注解@RequestBody，此注解可以将json格式的字符串转换成相应的java对象，如果需要输出json格式的字符串需要用到注解@ResponseBody，此注解可以将Java对象转成json串输出。

SpringMVC框架使用jackson的JAR包进行json转换（@RequestBody和@ResponseBody），如下所示：



注意SpringMVC框架中上述两种JSON方式的使用比较：

请求json、输出json，要求请求的是json串，所以在前端页面中需要将请求的内容转成json，不太方便。

请求key/value、输出json。此方法比较常用。

### 2.3.2 配置json转换器

在注解处理器适配器中配置json转换器，如下所示：

<!-- json转换器 -->

<bean

class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">

<property name="messageConverters">

<list>

<ref bean="mappingJackson2HttpMessageConverter" />

</list>

</property>

</bean>

<bean id="mappingJackson2HttpMessageConverter"

class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter">

<property name="supportedMediaTypes">

<list>

<value>text/html;charset=UTF-8</value>

<value>text/json;charset=UTF-8</value>

<value>application/json;charset=UTF-8</value>

</list>

</property>

</bean>

**如果使用注解的方式配置处理器映射器，处理器适配置，即：<mvc:annotation-driven />则可以不用定义上面的内容。**

### 2.3.3 请求类型key/value，输出JSON（常用）

下面使用JQuery的ajax方法进行测试，之所以使用ajax方法进行测试，是因为返回的数据为json格式，是纯粹的数据，可以使用JS脚本进行处理，因此更容易来处理数据，所以在测试的JSP页面需要引入相应的jquery库文件，如下所示。

将用户修改功能用Ajax + JSON来实现：

在该JSP页面中通过JQuery的ajax方法进行请求，请求格式更改为JSON格式，如下所示：

$(function(){

$("#btnSubmit").click(function(){

var dataParam = {

userId: $("#userId").val(),

name: $("#name").val(),

password: $("#password").val(),

createdate: $("#createdate").val()

};

var sDataParam = JSON.stringify(dataParam);

$.ajax({

type: "POST",

url: "user/save.action",

dataType:"json",

data: dataParam,

success: function(data){

if(data != null){

if(data.res == "true"){

alert(data.msg);

window.location.href= "user/list.action";

}else{

alert(data.msg);

}

}

}

});

});

});

Controller层代码如下：

@RequestMapping("save.action")

public @ResponseBody Map<String, Object> save(@Validated User user,BindingResult bindingResult){

Map<String, Object> res = new HashMap<String, Object>();

if(bindingResult.hasErrors()){

for(ObjectError v : bindingResult.getAllErrors()){

System.out.println(v.getDefaultMessage());

}

throw new RuntimeException("sava.action 的参数错误。");

}else{

this.userService.save(user);

res.put("res", "true");

res.put("msg", "保存成功！");

return res;

}

}

### 2.3.4 请求JSON、输出JSON（不常用，难用）

### 2.3.5 表单提交最佳实践

脚本代码：

var dataParam = $("#form").serialize();

$.ajax({

type: "POST",

url: "user/save.action",

data: dataParam,

success: function(data){

if(data != null){

if(data.res == "true"){

alert(data.msg);

window.location.href= "user/list.action";

}else{

alert(data.msg);

}

}

}

});

## 2.4 文件上传

如果在开发时需要实现文件上传，则form表单的enctype属性需要指定为multipart/form-data类型，如果使用SpringMVC框架，那么就需要SpringMVC框架来实现对该种数据类型的解析。

### 2.4.1 SpringMVC对多部件数据类型的解析

配置SpringMVC框架对multipart属性类型的解析器。如下所示：

<!-- 配置SpringMVC框架对multipart属性类型的解析器，用于文件上传 -->

<bean id="multipartResolver"

class="org.springframework.web.multipart.commons.CommonsMultipartResolver">

<property name="maxUploadSize" value="5242880" />

</bean>

**虽然该Bean的配置是提供给SpringMVC框架使用，但是与其他提供给框架使用的Bean不同的是，此Bean必须要指定id属性而且id属性值需要写为：multipartResolver，是因为SpringMVC框架需要根据此字符串值来查找该Bean。**

解析器配置完成后还需要添加相应的JAR包，因为该解析器再实现上传时会用到此JAR包的相关类，相关JAR包如下：

1. commons-io-2.2.jar
2. commons-fileupload-1.2.2.jar

### 2.4.2 创建图片的虚拟目录

在普通的web应用中，在项目下创建一个上传图片目录即可，但是如果是个大型的WEB电商平台，则需要

为图片单独指定一个图片服务器，该服务器的配置可以通过修改Tomcat的配置文件来实现，如下所示：

修改Tomcat的config目录下的server.xml文件：

<Context path="/pic" docBase="D:\tomcatUploadFiles" reloadable="false" />

### 2.4.3 上传页面代码

HTML:

<tr>

<td>上传头像</td>

<td>

<input type="file" id="file1" />

<input type="button" id="upload" value="上传" />

<img src="static/imgs/loading.gif" style="display:none" id="imgWait" />

<span id="imageName"></span>

<img id="imageView" style="width:100px;height:100px" />

</td>

</tr>

JS:

//上传附件

$("#upload").click(function () {

$("#imgWait").show();

//创建存储文件的对象

var formData = new FormData();

formData.append("myfile", document.getElementById("file1").files[0]);

$.ajax({

url: "file/upload.action",

type: "POST",

data: formData,

/\*\*

\*必须false才会自动加上正确的Content-Type

\*/

contentType: false,

/\*\*

\* 必须false才会避开jQuery对 formdata 的默认处理

\* XMLHttpRequest会对 formdata 进行正确的处理

\*/

processData: false,

success: function (data) {

if (data.res == "true") {

alert(data.msg);

$("#imageName").text(data.imageName);

//<img id="imageView" />

$("#imageView").attr("src","<%=serverPath %>" + data.imagePath);

}

if (data.res == "false") {

alert(data.msg);

}

$("#imgWait").hide();

},

error: function () {

alert("上传失败！");

$("#imgWait").hide();

}

});

});

### 2.4.4 Controller方法中添加MultipartFile类型参数

在Controller方法中，需要定义MultipartFile类型参数用来接收上传的图片格式类型的数据，如下所示：

@Controller

@RequestMapping("file")

public class FileController {

@RequestMapping("upload.action")

public @ResponseBody Map<String, Object> upload(@RequestParam(value="myfile",required=false) MultipartFile file){

Map<String, Object> res = new HashMap<String, Object>();

if(file != null){

String originalFileName = file.getOriginalFilename();

if(originalFileName != null && originalFileName.length() > 0){

String imgPath = "D:\\tomcatUploadFiles"; //存储图片的物理路径

String newFileName = UUID.randomUUID().toString().trim().replaceAll("-", "");

String fileName = newFileName + originalFileName.substring(originalFileName.lastIndexOf("."));

File newFile = new File(imgPath + "\\" + fileName);

try {

//将内存中的图片数据写入到磁盘中

file.transferTo(newFile);

} catch (IllegalStateException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

res.put("imageName", fileName);

res.put("imagePath", "pic/" + fileName);

}

res.put("res", "true");

res.put("msg", "上传成功！");

}else{

res.put("res", "false");

res.put("msg", "上传失败！");

}

return res;

}

}

## 2.5 RESTful支持

RESTful架构，就是目前最流行的一种互联网软件架构。它结构清晰、符合标准、易于理解、扩展方便，所以正得到越来越多网站的采用。

### 2.5.1 起源

REST这个词，是Roy Thomas Fielding在他2000年的博士论文中提出的。

Fielding是一个非常重要的人，他是HTTP协议（1.0版和1.1版）的主要设计者、Apache服务器软件的作者之一、Apache基金会的第一任主席。所以，他的这篇论文一经发表，就引起了关注，并且立即对互联网开发产生了深远的影响。

Fielding将他对互联网软件的架构原则，定名为REST，即Representational State Transfer的缩写。直译过来是"表现层状态转化"。

如果一个架构符合REST原则，就称它为RESTful架构。

### 2.5.2 名称

要理解RESTful架构，最好的方法就是去理解Representational State Transfer这个词组到底是什么意思，它的每一个词代表了什么涵义。如果你把这个名称搞懂了，也就不难体会REST是一种什么样的设计。

### 2.5.3 资源（Resources）

REST的名称"表现层状态转化"中，省略了主语。"表现层"其实指的是"资源"（Resources）的"表现层"。

所谓"资源"，就是网络上的一个实体，或者说是网络上的一个具体信息。它可以是一段文本、一张图片、一首歌曲、一种服务，总之就是一个具体的实在。你可以用一个URI（统一资源定位符）指向它，每种资源对应一个特定的URI。要获取这个资源，访问它的URI就可以，因此URI就成了每一个资源的地址或独一无二的识别符。

所谓"上网"，就是与互联网上一系列的"资源"互动，调用它的URI。

### 2.5.4 表现层（Representation）

"资源"是一种信息实体，它可以有多种外在表现形式。我们把"资源"具体呈现出来的形式，叫做它的"表现层"（Representation）。

比如，文本可以用txt格式表现，也可以用HTML格式、XML格式、JSON格式表现，甚至可以采用二进制格式；图片可以用JPG格式表现，也可以用PNG格式表现。

URI只代表资源的实体，不代表它的形式。严格地说，有些网址最后的".html"后缀名是不必要的，因为这个后缀名表示格式，属于"表现层"范畴，而URI应该只代表"资源"的位置。它的具体表现形式，应该在HTTP请求的头信息中用Accept和Content-Type字段指定，这两个字段才是对"表现层"的描述。

### 2.5.5 状态转化（State Transfer）

访问一个网站，就代表了客户端和服务器的一个互动过程。在这个过程中，势必涉及到数据和状态的变化。

互联网通信协议HTTP协议，是一个无状态协议。这意味着，所有的状态都保存在服务器端。因此，如果客户端想要操作服务器，必须通过某种手段，让服务器端发生"状态转化"（State Transfer）。而这种转化是建立在表现层之上的，所以就是"表现层状态转化"。

客户端用到的手段，只能是HTTP协议。具体来说，就是HTTP协议里面，四个表示操作方式的动词：GET、POST、PUT、DELETE。它们分别对应四种基本操作：GET用来获取资源，POST用来新建资源（也可以用于更新资源），PUT用来更新资源，DELETE用来删除资源。

综合上面的解释，我们总结一下什么是RESTful架构：

（1）每一个URI代表一种资源；

（2）客户端和服务器之间，传递这种资源的某种表现层；

1. 客户端通过四个HTTP动词，对服务器端资源进行操作，实现"表现层状态转化"。

### 2.5.6 REST的示例

以用户信息修改为例：

1、controller层代码如下：

@RequestMapping("edit/{id}")

public ModelAndView edit(@PathVariable("id") String id){

ModelAndView view = new ModelAndView();

User user = this.userService.getModel(id);

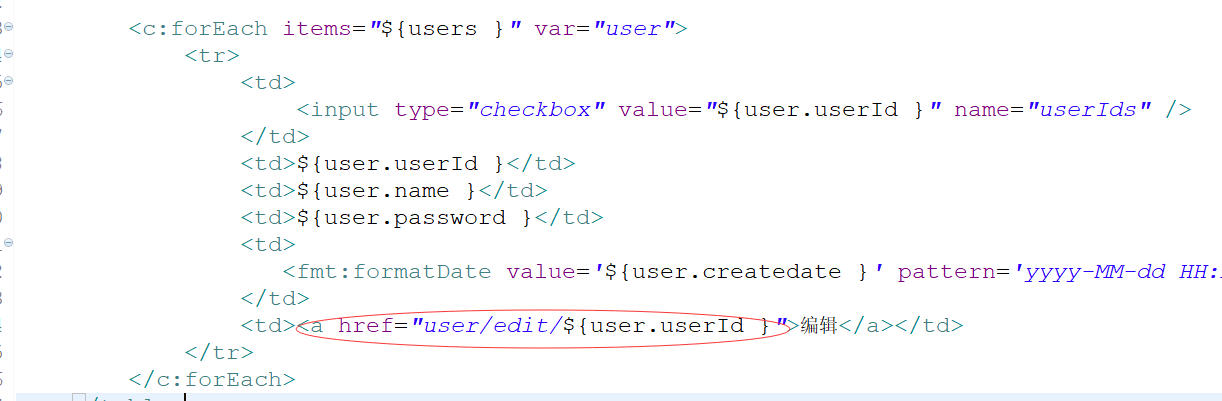
view.addObject("model", user);

view.setViewName("userEdit");

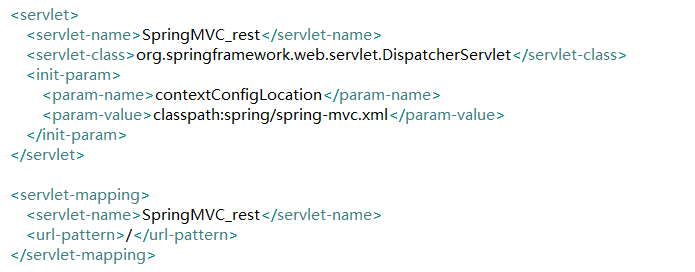
return view;

}

2、界面：



3、REST方式的前端控制器配置：



1. 静态资源的解析：

方法1：

当请求的地址是js或imgs下的资源时，全部映射到应用中的js或imgs下的所有资源。

<!-- 配置SpringMVC的静态资源处理 -->

<mvc:resources location="/static/" mapping="/static/\*\*" />

**注意：不能分开写，下面的写法是错误的。**

<!-- 配置SpringMVC的静态资源处理 -->

<mvc:resources location="/static/js" mapping="/static/js/\*\*"/>

<mvc:resources location="/static/css" mapping="/static/css/\*\*"/>

<mvc:resources location="/static/imgs" mapping="/static/imgs/\*\*"/>

<mvc:resources/> 表示由Spring MVC框架自己处理静态资源。

方法2：

<mvc:default-servlet-handler />

在springMVC-servlet.xml中配置<mvc:default-servlet-handler />后，会在Spring MVC上下文中定义一个org.springframework.web.servlet.resource.DefaultServletHttpRequestHandler，它会像一个检查员，对进入DispatcherServlet的URL进行筛查，如果发现是静态资源的请求，就将该请求转由Web应用服务器默认的Servlet处理，如果不是静态资源的请求，才由DispatcherServlet继续处理。

一般Web应用服务器默认的Servlet名称是"default"，因此DefaultServletHttpRequestHandler可以找到它。如果你所有的Web应用服务器的默认Servlet名称不是"default"，则需要通过default-servlet-name属性显示指定：

<mvc:default-servlet-handler default-servlet-name="所使用的Web服务器默认使用的Servlet名称" />

### 2.5.7 REST的设计误区

RESTful架构有一些典型的设计误区。

最常见的一种设计错误，就是URI包含动词。因为"资源"表示一种实体，所以应该是名词，URI不应该有动词，动词应该放在HTTP协议中。

举例来说，某个URI是/posts/show/1，其中show是动词，这个URI就设计错了，正确的写法应该是/posts/1，然后用GET方法表示show。

如果某些动作是HTTP动词表示不了的，你就应该把动作做成一种资源。比如网上汇款，从账户1向账户2汇款500元，错误的URI是：

POST /accounts/1/transfer/500/to/2

正确的写法是把动词transfer改成名词transaction，资源不能是动词，但是可以是一种服务：

POST /transaction HTTP/1.1

Host: 127.0.0.1

from=1&to=2&amount=500.00

另一个设计误区，就是在URI中加入版本号：

http://www.example.com/app/1.0/foo

http://www.example.com/app/1.1/foo

http://www.example.com/app/2.0/foo

因为不同的版本，可以理解成同一种资源的不同表现形式，所以应该采用同一个URI。版本号可以在HTTP请求头信息的Accept字段中进行区分（参见Versioning REST Services）：

Accept: vnd.example-com.foo+json; version=1.0

Accept: vnd.example-com.foo+json; version=1.1

Accept: vnd.example-com.foo+json; version=2.0

## 2.6 SpringMVC拦截器

SpringMVC的拦截器类似于JSP/Servlet中的Filter过滤器，用于对处理器进行预处理和后处理。

### 2.6.1 拦截器定义

在SpringMVC中提供了一个HandlerInterceptor接口，实现了该接口的实现类即为SpringMVC的拦截器，如下所示：

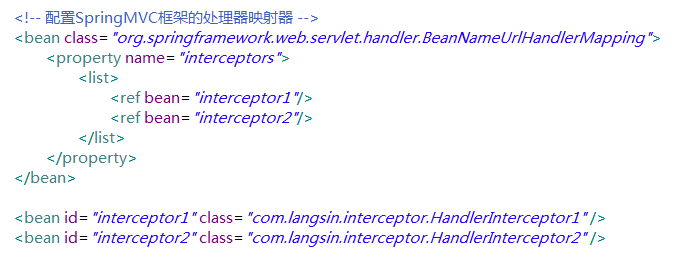




### 2.6.2 拦截器的配置

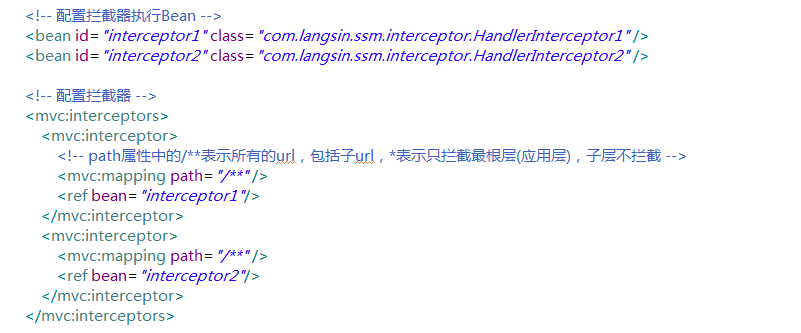
SpringMVC拦截器的配置有两种方式：1、针对单个mapping映射的配置拦截 2、针对所有mapping映射的配置拦截。第一种方式是不推荐的方式，第二种方式推荐使用。

**1、针对单个mapping映射的拦截配置**



在BeanNameUrlHandlerMapping中，对属性interceptors属性进行了设置注入，注入内容为Object[]数组，在数组中引用了两个拦截器bean：interceptor1和interceptor2，拦截器按照引用的顺序依次执行。需要注意的是对于拦截器中的preHandle方法，按引用顺序执行，而其他两个方法postHandle和afterCompletion则反向执行。

**2、针对所有mapping映射的拦截配置（推荐）**



### 2.6.3 拦截器正常执行

拦截器1与拦截器2同时放行，即preHandle方法return为true。测试如下：

interceptor1~~~~~~~preHandle

interceptor2~~~~~~~preHandle

interceptor2~~~~~~~postHandle

interceptor1~~~~~~~postHandle

interceptor2~~~~~~~afterCompletion

interceptor1~~~~~~~afterCompletion

对于拦截器的preHandle方法按配置顺序正常执行，而postHandle、afterCompletion方法则逆向执行

### 2.6.4 拦截器中断执行

拦截器1阻止，拦截器2放行。测试如下：

interceptor1~~~~~~~preHandle

拦截器1的preHandle方法正常执行，其他两个方法都不执行，而且拦截器2的所有方法都不执行，并且Controller也没有被执行。

拦截器1放行，拦截器2阻止。测试如下：

interceptor1~~~~~~~preHandle

interceptor2~~~~~~~preHandle

interceptor1~~~~~~~afterCompletion

拦截器1与拦截器2的preHandle方法正常执行，拦截器2的其他两个方法都不执行，拦截器1的postHandle方法没有调用，只执行了afterCompletion方法。

总结：preHandle方法按拦截器的配置顺序调用

postHandler方法按拦截器的配置顺序逆向调用

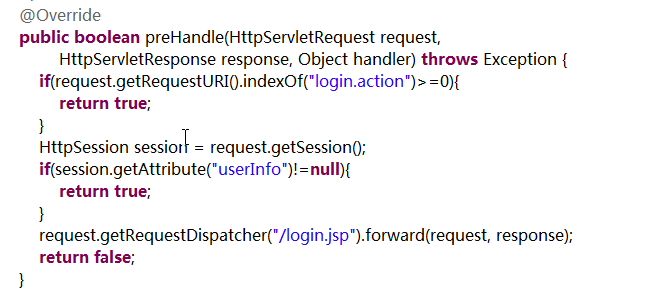
afterCompletion方法按拦截器的配置顺序逆向调用

postHandler方法在拦截器链内的所有拦截器返回为true才调用

afterCompletion方法在拦截器的preHandle方法返回为true才调用

### 2.6.5 拦截器应用

使用拦截器完成一个登陆拦截操作，代码如下所示：



## 2.7 Spring事务

事务控制是信息管理系统质量的重要指标。Spring 框架对事务提供了很方便的实现。

### 2.7.1 基于XML配置的声明式事务管理

Spring的事务管理，提供了一个DataSourceTransactionManager类，用于进行声明式事务管理类，该类在org.springframework.jdbc.datasource包下，该类中包含一个dataSource属性用于配置一个数据源。配置如下：

在spring-dao.xml 中增加：

<!-- 事务管理器 -->

<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource" />

</bean>

事务管理器在配置时引用了dataSource对象，此对象为数据库连接池对象，表示事务管理器对事务的管理是通过连接池对象中的连接进行管理。事务管理器是通过消息通知的方式起作用，所以需要为事务管理来配置一个事务通知消息，即事务管理器在接收到一个消息通知时才会开启事务的操作。

配置通知消息需要使用<tx:advice>标签，此标签包含id、transaction-manager两个重要属性。

id：消息通知的唯一标识ID

transaction-manager：消息通知发出后交由哪个事务管理器去处理。

在<tx:advice>标签下含有<tx:attributes>标签，用于配置在执行什么方法时才会触发消息发送。配置方法使用<tx:method>方法，如下所示：

<!-- 事务消息通知 -->

<tx:advice id="txAdvice" transaction-manager="txManager">

<tx:attributes>

<tx:method name="save" propagation="REQUIRED" />

<tx:method name="get\*" read-only="true" />

</tx:attributes>

</tx:advice>

配置事务切面：

<aop:config>

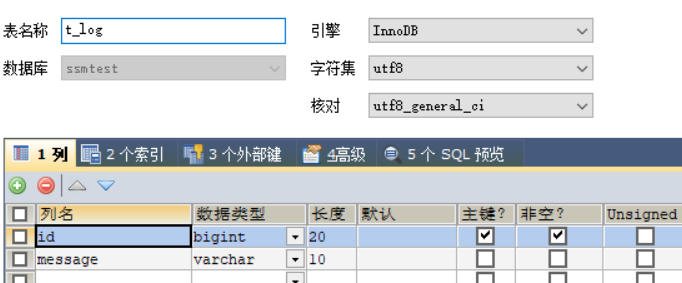
<aop:pointcut expression="execution(\* com.langsin.service.impl.\*.\*(..))" id="pointcut" />

<aop:advisor advice-ref="txAdvice" pointcut-ref="pointcut" />

</aop:config>

简单测试：

1. 新增一日志表。当用户信息修改成功后，写入日志表数据。表结构如下：



1. 并建立该表对应的持久层： LogDao和LogDao.xml
2. 修改UserServiceImpl中的save 方法：

public void save(User model) {

// TODO Auto-generated method stub

try {

userDao.updateUser(model);

logDao.log("测实施实时所所所所所所所");

} catch (Exception e) {

//事务回滚

TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();

// TODO Auto-generated catch block

e.printStackTrace();

}

}

因为数据库中t\_log表的message字段的长度是10。所以logDao.log("测实施实时所所所所所所所");肯定会出错。因为save方法已事务aop监控。所以userDao.updateUser(model);也应该不成功。

**注意：**

**要完成事务。必须将 业务逻辑层的注解从springmvc.xml 中独立出来(spring-content.xml)。**

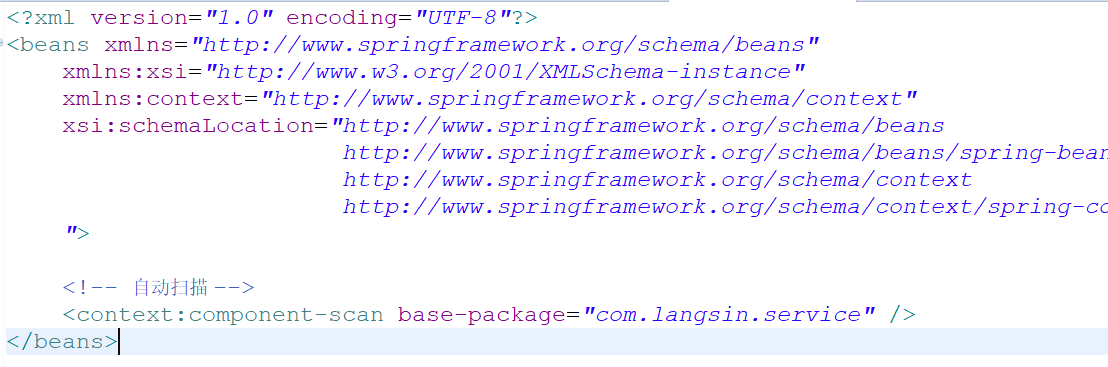
原因：

一般我们在Spring的配置文件spring-content.xml中对Service层代码配置事务管理，可以对Service的方法进行AOP增强或事务处理如事务回滚，但是遇到一个问题，在Controller类中调用Service层方法，配置的事务管理会失效，其实Spring和SpringMVC俩个容器为父子关系，Spring为父容器，而SpringMVC为子容器。

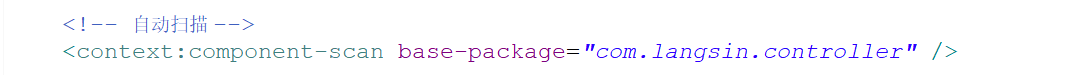
也就是说spring-content.xml中应该负责扫描除@Controller的注解如@Service，

而SpringMVC的配置文件应该只负责扫描@Controller，否则会产生重复扫描导致Spring容器中配置的事务失效。

spring-content.xml 内容如下图：



springmvc.xml 的自动扫描如下：



### 2.7.2 基于注解的声明式事务管理

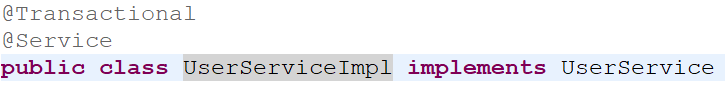
基于注解的事务管理器仍然需要事务管理器。但是与xml配置不同的是，基于注解的方案不需要在spring-dao.xml中配置事务消息通知和事务切面。只需要下面的配置即可。

<!-- 基于注解的事务 -->

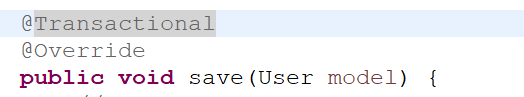
<tx:annotation-driven transaction-manager="txManager"/>

在需要事务的类或者方法上面添加注解 @Transactional

类上面：



方法上面：



测试过程与2.7.1节基于XML配置的声明式事务管理一致。