SpringMVC从入门到精通之第二章_慕课手记

这一章原本我是想写一个入门程序的，但是后来仔细想了一下，先从下面的图中的组件用代码来介绍，可能更效果会更加好一点。

第一节：开发准备

介绍之前先说下我的开发调试环境：

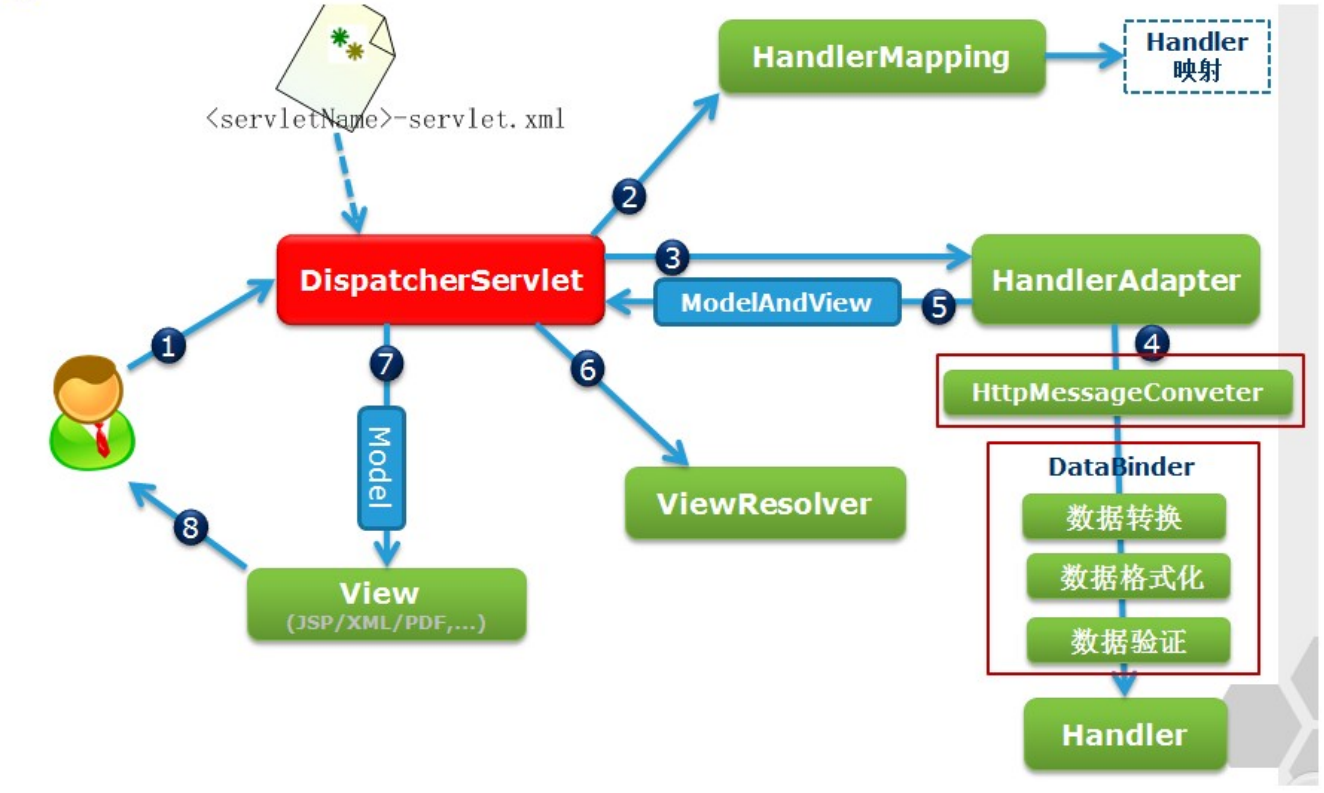
JDK 1.7的64位 、Eclipse Kepler（J2EE） 64位的、Tomcat 7.0.42 mysql 5.1、SQLyog（这是我的标配）

springmvc 版本 3.2（这个大家可以到网上自行下载最好要有源码）

第二节：开发与配置

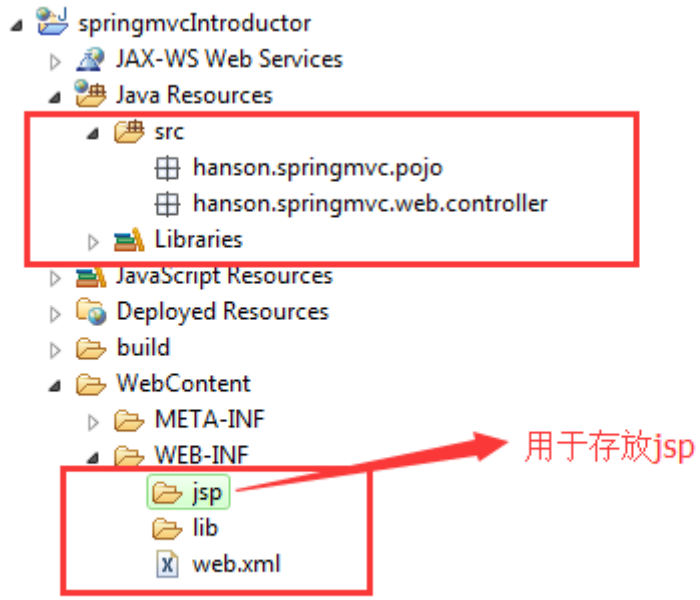
Spring MVC工作流程图

图一



第一步发送request请求，这个一步我就不说了，之前有看到慕课有很多朋友分享了http协议的介绍传送。

开发环境介绍过了，下面来创建一个工程：如下图 整个工程结构



在hanson.springmvc.pojo中新建一个Items类包含属性如下图：

```
public class Items {
    private Integer id;

    private String name;

    private Float price;

    private String pic;

    private Date createtime;

    private String detail;
```

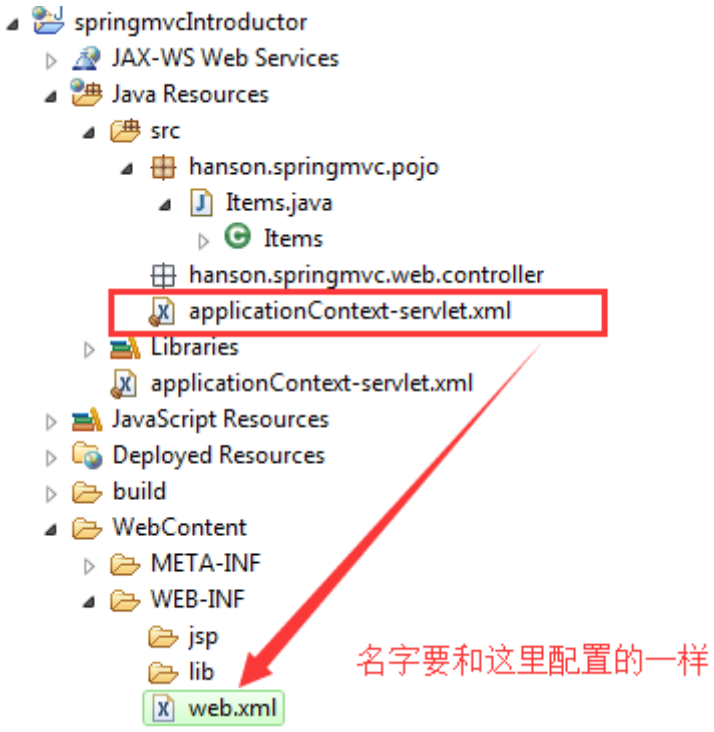
提供get和set方法,另外重写toString方法

用户发送请求到前端控制器。前端控制器需要在web.xml中配置：

```
<!-- 配置前端控制器 -->
  <servlet>
    <servlet-name>springmvc</servlet-name>
    <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
    <!--加载前端控制器配置文件 上下文配置位置-->
    <init-param>
        <!-- 备注：
            contextConfigLocation：指定springmvc配置的加载位置，如果
```

```
不指定则默认加
            载WEB-INF/[DispatcherServlet的Servlet 名字]-
servlet.xml(例如springmvc-servlet.xml)。
         -->
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:applicationContext-
servlet.xml</param-value>
    </init-param>
    <!-- 表示随WEB服务器启动 -->
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>springmvc</servlet-name>
    <!-- 备注：可以拦截三种请求
        第一种：拦截固定后缀的url，比如设置为 *.do、*.action， 例
如：/user/add.action 此方法最简单,不会导致静态资源（jpg,js,css）被拦截.
        第二种：拦截所有,设置为/，例如：/user/add  /user/add.action此
方法可以实现REST风格的url,
                很多互联网类型的应用使用这种方式.但是此方法会导致静态文件
(jpg,js,css)被拦截后不能正常显示.需要特殊处理.
        第三种：拦截所有,设置为/*，此设置方法错误,因为请求到Action,当
action转到jsp时再次被拦截,提示不能根据jsp路径mapping成功.
     -->
    <url-pattern>*.action</url-pattern>
  </servlet-mapping>
```

工程结构图：



到这里前端配置器就算配置完了,目前这个配置足够我们学习springmvc了。

你需要知道的是：前端控制器的作用，如何配置，以及拦截什么样的请求。对于restful风格的请求我会在后面讲到。

第三节 Handler的开发

开发之前先说一说处理器适配器。说之前就得说一说适配器模式我们就拿生活中的例子来解释一下适配器模式，现在只有一个三相的插座，但是现在却有一个三相插头的洗衣机和一个二相插头的电视机，那么洗衣机当然没有问题，可以插到三相插座上，但是电视机却没办法插了，于是人们想出了一个适配器，这个适配器一头插在这个三相插座上，另外一端放出一个二相插座，然后电视就插在了这个适配器的二相插座上了，最后洗衣机，电视机都可以使用了。总结起来就是一句：将一个类的接口适配成用户所期待的。

加上之前说的，想要看清3D电影，就必须带上3D眼镜。

springmvc中也是这样要求处理器实现多种接口才能被处理器适配器执行。

下面来介绍几个处理器适配器，以及他们能够执行的处理器。

第一个：

org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter简单的处理器适配器，此适配器能够执行实现org.springframework.web.servlet.mvc.Controller接口的处理器,来看下源码：

```
public class SimpleControllerHandlerAdapter implements
HandlerAdapter {

    public boolean supports(Object handler) {
```

```
        //该类支持Controller类型的处理器
        return (handler instanceof Controller);
    }

    public ModelAndView handle(HttpServletRequest request,
HttpServletResponse response, Object handler)
            throws Exception {

        return ((Controller) handler).handleRequest(request,
response);
    }

    public long getLastModified(HttpServletRequest request,
Object handler) {
        if (handler instanceof LastModified) {
            return ((LastModified)
handler).getLastModified(request);
        }
        return -1L;
    }
}
```

既然要求这个适配器支持的类型是Controller类型的，那我们就实现这个接口
代码如下：

```
package hanson.springmvc.web.controller;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;
/**
 *
 * @ClassName: ItemsController1
 * @Description: TODO(简单的处理器适配器支持执行实现Controller接口的处
理器)
 * @author: Hanson
```

```
 * @date: 2016年1月14日 下午11:59:06
 *
 */
public class ItemsController1 implements Controller {

    @Override
    public ModelAndView handleRequest(HttpServletRequest request,
            HttpServletResponse response) throws Exception {

        return null;
    }

}
```

第四节

ModelAndView对象封装了模型数据和视图对象，有一个组件叫视图解析器，就是用来解析这个对象的，它可以把这个对象解析成两部分一个为Model另一个为View然后将model渲染到View上面（简单点就是将model里面的数据放到页面）,最终返回给用户。
我用静态数据模拟一下这个实现。

```
@Override
    public ModelAndView handleRequest(HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        // 商品列表
        List<Items> itemsList = new ArrayList<Items>();
        Items items_2 = new Items();
        items_2.setName("苹果手机");
        items_2.setPrice(6088f);
        items_2.setDetail("iphone6s苹果手机！");
        itemsList.add(items_2);
        // 创建modelAndView：填充数据、设置视图
        ModelAndView modelAndView = new ModelAndView();
        // 填充数据
        modelAndView.addObject("itemsList", itemsList);// 类似
request.setAttribute("","")
        // 视图:逻辑名称
        modelAndView.setViewName("jsp/itemsList");//
request.getRequestDispatcher("url").forward(request, response);
```

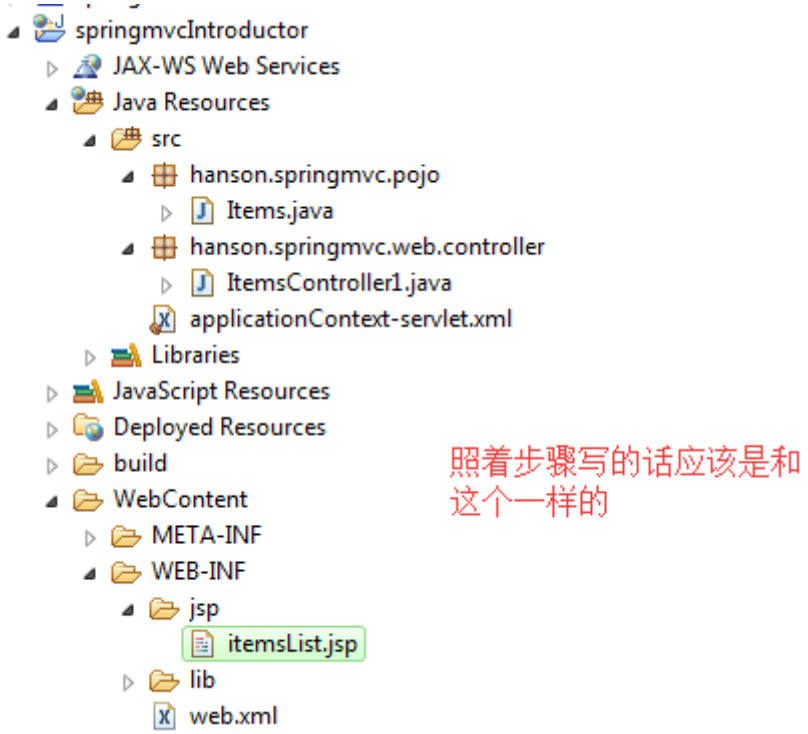
```
        return modelAndView;
    }
```

jsp中代码：

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-
8">
<title>查询商品列表</title>
</head>
<body>
    商品列表：
    <table width="100%" border=1>
        <tr>
            <td>商品名称</td>
            <td>商品价格</td>
            <td>商品描述</td>
        </tr>
        <c:forEach items="${itemsList }" var="item">
            <tr>
                <td>${item.name }</td>
                <td>${item.price }</td>
                <td>${item.detail }</td>
            </tr>
        </c:forEach>

    </table>
</body>
</html>
```

再看下工程结构：



第五节

下面我们来看看之前提到的视图解析器

org.springframework.web.servlet.view.InternalResourceViewResolver类是用来解析jsp的它要求类路径下面有jstl的jar。源码如下：

```
/*
 * Copyright 2002-2009 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the
"License");
 * you may not use this file except in compliance with the
License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing,
software
 * distributed under the License is distributed on an "AS IS"
BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
or implied.
 * See the License for the specific language governing
```

```
permissions and
 * limitations under the License.
 */

package org.springframework.web.servlet.view;

import org.springframework.util.ClassUtils;

/**
 * Convenient subclass of {@link UrlBasedViewResolver} that
supports
 * {@link InternalResourceView} (i.e. Servlets and JSPs) and
subclasses
 * such as {@link JstlView}.
 *
 * <p>The view class for all views generated by this resolver can
be specified
 * via {@link #setViewClass}. See {@link UrlBasedViewResolver}'s
javadoc for details.
 * The default is {@link InternalResourceView}, or {@link
JstlView} if the
 * JSTL API is present.
 *
 * <p>BTW, it's good practice to put JSP files that just serve as
views under
 * WEB-INF, to hide them from direct access (e.g. via a manually
entered URL).
 * Only controllers will be able to access them then.
 *
 * <p><b>Note:</b> When chaining ViewResolvers, an
InternalResourceViewResolver
 * always needs to be last, as it will attempt to resolve any
view name,
 * no matter whether the underlying resource actually exists.
 *
 * @author Juergen Hoeller
 * @since 17.02.2003
 * @see #setViewClass
```

```
 * @see #setPrefix
 * @see #setSuffix
 * @see #setRequestContextAttribute
 * @see InternalResourceView
 * @see JstlView
 */
public class InternalResourceViewResolver extends
UrlBasedViewResolver {

    private static final boolean jstlPresent =
ClassUtils.isPresent(
            "javax.servlet.jsp.jstl.core.Config",
InternalResourceViewResolver.class.getClassLoader());

    private Boolean alwaysInclude;

    private Boolean exposeContextBeansAsAttributes;

    private String[] exposedContextBeanNames;

    /**
     * Sets the default {@link #setViewClass view class} to
{@link #requiredViewClass}:
     * by default {@link InternalResourceView}, or {@link
JstlView} if the JSTL API
     * is present.
     */
    public InternalResourceViewResolver() {
        Class viewClass = requiredViewClass();
        if (viewClass.equals(InternalResourceView.class) &&
jstlPresent) {
            viewClass = JstlView.class;
        }
        setViewClass(viewClass);
    }

    /**
     * This resolver requires {@link InternalResourceView}.
```

```
     */
    @Override
    protected Class requiredViewClass() {
        return InternalResourceView.class;
    }

    /**
     * Specify whether to always include the view rather than
forward to it.
     * <p>Default is "false". Switch this flag on to enforce the
use of a
     * Servlet include, even if a forward would be possible.
     * @see InternalResourceView#setAlwaysInclude
     */
    public void setAlwaysInclude(boolean alwaysInclude) {
        this.alwaysInclude = Boolean.valueOf(alwaysInclude);
    }

    /**
     * Set whether to make all Spring beans in the application
context accessible
     * as request attributes, through lazy checking once an
attribute gets accessed.
     * <p>This will make all such beans accessible in plain
<code>${...}</code>
     * expressions in a JSP 2.0 page, as well as in JSTL's
<code>c:out</code>
     * value expressions.
     * <p>Default is "false".
     * @see
InternalResourceView#setExposeContextBeansAsAttributes
     */
    public void setExposeContextBeansAsAttributes(boolean
exposeContextBeansAsAttributes) {
        this.exposeContextBeansAsAttributes =
exposeContextBeansAsAttributes;
    }
```

```
    /**
     * Specify the names of beans in the context which are
supposed to be exposed.
     * If this is non-null, only the specified beans are eligible
for exposure as
     * attributes.
     * @see InternalResourceView#setExposedContextBeanNames
     */
    public void setExposedContextBeanNames(String[]
exposedContextBeanNames) {
        this.exposedContextBeanNames = exposedContextBeanNames;
    }

    @Override
    protected AbstractUrlBasedView buildView(String viewName)
throws Exception {
        InternalResourceView view = (InternalResourceView)
super.buildView(viewName);
        if (this.alwaysInclude != null) {
            view.setAlwaysInclude(this.alwaysInclude);
        }
        if (this.exposeContextBeansAsAttributes != null) {

view.setExposeContextBeansAsAttributes(this.exposeContextBeansAsA
ttributes);
        }
        if (this.exposedContextBeanNames != null) {

view.setExposedContextBeanNames(this.exposedContextBeanNames);
        }
        view.setPreventDispatchLoop(true);
        return view;
    }

}
```

它需要在applicationContext-servlet.xml中配置：

```
<!-- 配置视图解析器 -->
    <!-- InternalResourceViewResolver：支持JSP视图解析 -->
    <!-- viewClass：JstlView表示JSP模板页面需要使用JSTL标签库，所以
classpath中必须包含jstl的相关jar包； -->
    <!-- prefix 和suffix：查找视图页面的前缀和后缀，最终视图的址为： -->
    <!-- 前缀+逻辑视图名+后缀，逻辑视图名需要在controller中返回
ModelAndView指定，比如逻辑视图名为hello，-->
    <!-- 则最终返回的jsp视图地址 "WEB-INF/jsp/hello.jsp" -->
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewR
esolver">
        <property name="viewClass"
value="org.springframework.web.servlet.view.JstlView"/>
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <property name="suffix" value=".jsp"/>
    </bean>
```

重点：到这里你应该知道处理器适配器（简单的处理器适配器、处理器适配器执行处理器，处理器应该具备那些要求才能被执行，视图解析器、以及ModelAndView对象）

第六节

疑问？前端控制器要怎么才能根据URL找到处理器呢？

本节知识点：处理器映射器

前端控制器会去调用处理器映射器找到处理器。

现在来介绍一下处理器映射器：

org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping：

表示将定义的Bean名字作为请求的url，需要将编写的controller在spring容器中进行配置，且指定bean的name为请求的url，且必须以.action结尾（web.xml配置）。

```
<!-- 配置BeanNameUrl处理器映射器 -->
    <!-- BeanNameUrlHandlerMapping：表示将定义的Bean名字作为请求的
url，需要将编写的controller在spring容器中进行配置， -->
    <!-- 且指定bean的name为请求的url，且必须以.action结尾。 -->
    <bean
class="org.springframework.web.servlet.handler.BeanNameUrlHandler
Mapping" />
<!-- controller配置 -->
    <!-- name="/items1.action"：前边配置的处理器映射器为
```

```
BeanNameUrlHandlerMapping， -->
    <!-- 如果请求的URL 为“上下文/items1.action”将会成功映射到ItemList1
控制器。 -->
    <bean name="/items1.action" id="itemList1"
class="hanson.springmvc.web.controller.ItemsController1"/>
```

结束语：这一章要求对上一章理解比较透彻。
有疑问的欢迎留言。

相关标签：