第三阶段 SpringMVC--01-结构讲解以及基本应用

回顾：

```
1. SpEL的使用
2. Spring JDBC的使用
3. Spring中事务的处理
```

今天任务

```
1. SpringMVC简介
2. SpringMVC配置详解
3. Spring常用注解
4. Spring的其他常用操作
```

教学目标

```
1. 掌握SpringMVC核心理论
2. 掌握SpringMVC配置
3. 掌握Spring常用注解
4. 掌握Spring的其他常用操作
```

一.Spring MVC 简介

> 大部分Java应用都是Web应用,展现层是WEB应用不可忽略的重要环节.Spring为了展现层提供了一个优秀的WEB框架-Spring MVC . 和众多的其他WEB框架一样,它基于MVC的设计理念. 此外,它采用了松散耦合,可插拔的组件结构,比其他的MVC框架更具有扩展性和灵活性,Spring MVC通过一套MVC注解,让POJO成为成为处理请求的处理器,无须实现任何接口.同时,Spring MVC还支持REST风格的URL请求:注解驱动及REST风格的Spring MVC是Spring的出色功能之一.
>
> 此外,Spring MVC在数据绑定,视图解析,本地化处理及静态资源处理上都有许多不俗的表现,它在框架设计,可扩展,灵活性等方面全面超越了Struts,WebWork等MVC框架,从原来的追赶者一跃成为了MVC的领跑者.

1.1 MVC模式简介

MVC全名是Model View Controller，是模型(model)－视图(view)－控制器(controller)的缩写，一种软件设计典范，用一种业务逻辑、数据、界面显示分离的方法组织代码，将业务逻辑聚集到一个部件里面，在改进和个性化定制界面及用户交互的同时，不需要重新编写业务逻辑。MVC被独特的发展起来用于映射传统的输入、处理和输出功能在一个逻辑的图形化用户界面的结构中。

MVC 是一种使用 MVC（Model View Controller 模型-视图-控制器）设计创建 Web 应用程序的模式：[1][ ]()

- Model（模型）表示应用程序核心（比如数据库记录列表）。
- View（视图）显示数据（数据库记录）。
- Controller（控制器）处理输入（写入数据库记录）。

MVC 模式同时提供了对 HTML、CSS 和 JavaScript 的完全控制。

Model（模型）是应用程序中用于处理应用程序数据逻辑的部分。 通常模型对象负责在数据库中存取数据。

View（视图）是应用程序中处理数据显示的部分。 通常视图是依据模型数据创建的。

Controller（控制器）是应用程序中处理用户交互的部分。 通常控制器负责从视图读取数据，控制用户输入，并向模型发送数据。

MVC 分层有助于管理复杂的应用程序，因为您可以在一个时间内专门关注一个方面。例如，您可以在不依赖业务逻辑的情况下专注于视图设计。同时也让应用程序的测试更加容易。

MVC 分层同时也简化了分组开发。不同的开发人员可同时开发视图、控制器逻辑和业务逻辑。

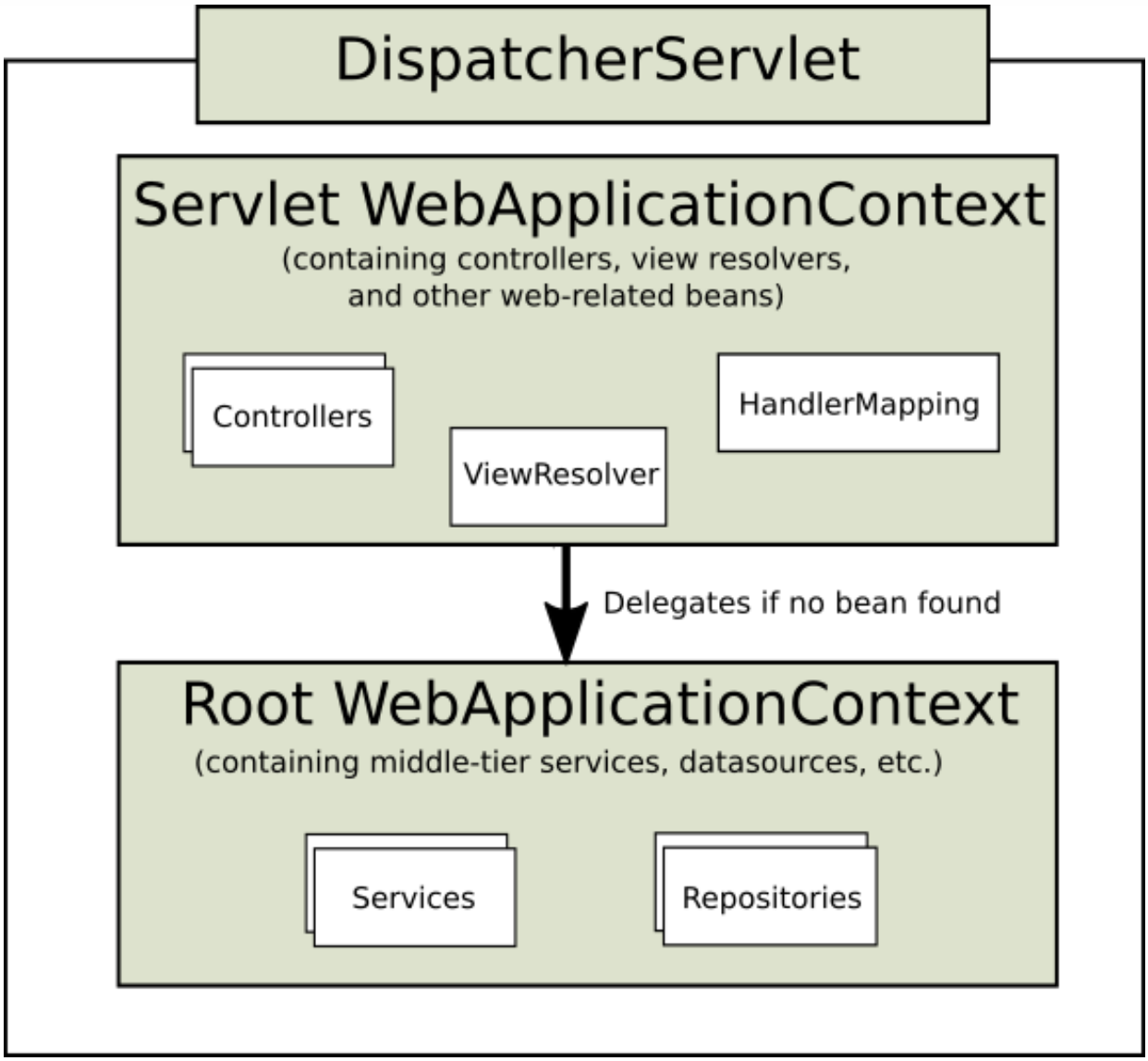
1.2 Spring MVC 体系结构

Spring MVC是基于 Model 2实现的技术框架,Model 2是经典的MVC(model,view,control)模型在WEB应用中的变体.这个改变主要源于HTTP协议的无状态性,Model 2 的目的和MVC一样,也是利用处理器分离模型,视图和控制,达到不同技术层级间松散层耦合的效果,提高系统灵活性,复用性和可维护性.大多情况下,可以将Model 2 与 MVC等同起来.

Spring MVC体系概述

Spring MVC框架围绕DispatcherServlet这个核心展开,DispatcherServlet是Spring MVC的总导演,总策划.它负责截获请求并将其分派给相应的处理器处理.Spring MVC框架包括注解驱动控制器,请求及响应的信息处理,视图解析,本地化解析,上传文件解析,异常处理及表单标签绑定内容等...

Spring核心组件



组件介绍

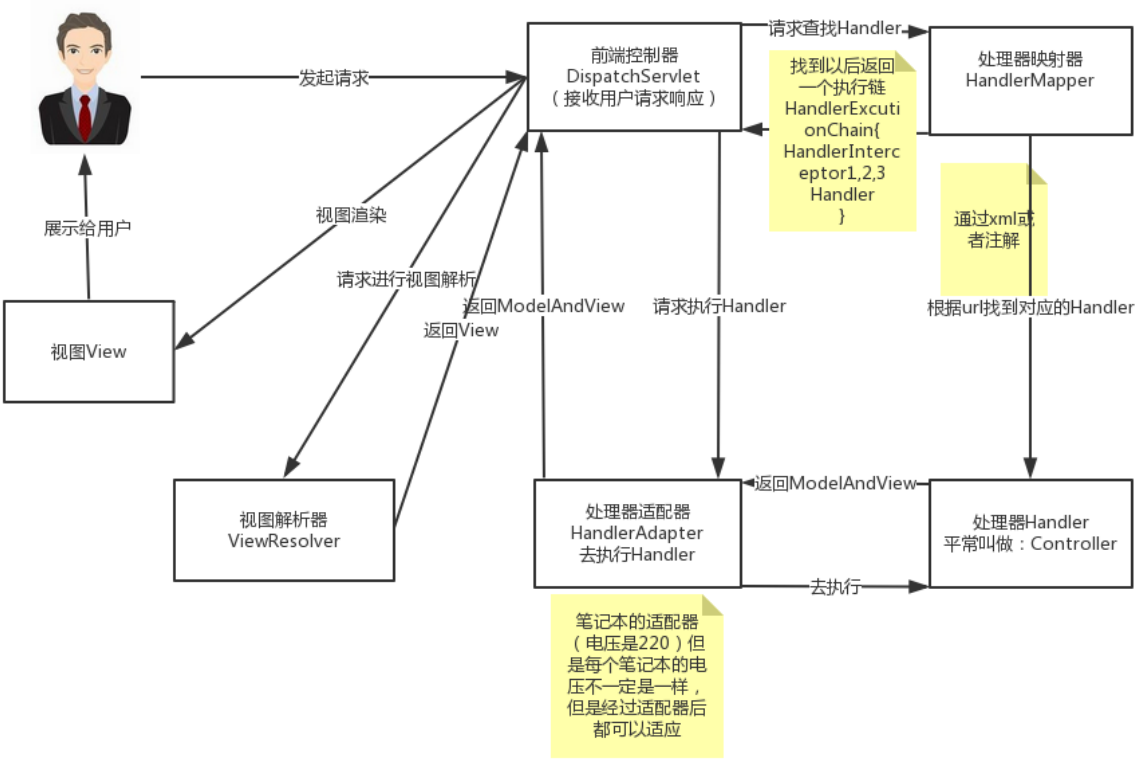
DispatcherServlet：作为前端控制器，整个流程控制的中心，控制其它组件执行，统一调度，降低组件之间的耦合性，提高每个组件的扩展性。

HandlerMapping：通过扩展处理器映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。

HandlAdapter：通过扩展处理器适配器，支持更多类型的处理器,调用处理器传递参数等工作!

ViewResolver：通过扩展视图解析器，支持更多类型的视图解析，例如：jsp、freemarker、pdf、excel等。

Spring MVC执行流程



从接收请求到响应,Spring MVC框架的众多组件通力配合,各司其职,有条不紊的完成分内工作!在整个框架中,DispatchserServlet处于核心的位置,它负责协调和组织不同组件以完成请求处理以及返回影响工作.和大多数Web MVC框架一样,Spring MVC 通过一个前段的Servlet接收所有请求,并将这些工作委托给其他组件进行处理,DispatcherServlet 就是Spring MVC的前段Servlet。下面对Spring MVC处理请求的整体过程进行详解!

1. 整个过程始于客户端发出的一个HTTP请求,WEB应用服务器接收到这个请求，如果匹配DispatcherServlet的映请求映射路径(web.xml中指定),则Web容器将该请求转交给DispatcherServlet处理。
2. 接收到这个请求后,将根据请求的信息(包括 URL,HTTP方法,请求头,请求参数,Cookie等)及HandlerMapping的配置找到处理请求的处理器(Handler)。可将HandlerMapping看做路由控制器,将Handler看做目标主机.值得注意的是,在Spring MVC中并没有定义一个Handler接口,实际上,任何一个Object都可以成为请求处理器。
3. 当DispatcherServlet根据HandlerMapping得到对应当前请求的Handler后,通过HandlerAdapter对Handler的封装,再以统一的适配器接口调用Handler。HandlerAdapter是Spring MVC的框架级接口,顾名思义,HandlerAdapter是一个适配器,它用统一的接口对各种Handler的方法进行调用.
4. 处理器完成业务逻辑的处理后将返回一个ModelAndView给DispatcherServlet,ModelAndView包含了视图逻辑名和模型数据信息。
5. ModelAndView中包含的是"逻辑视图名"而并非真正的视图对象,DispatcherServlet借由ViewResolver完成逻辑视图名到真实视图对象的解析工作。
6. 当得到真实的视图对象View后,DispatcherServlet就使用这个View对象对ModelAndView中的模型数据进行视图渲染。
7. 最终客户端得到的响应信息可能是一个普通的HTML页面,也可能是一个XML或者JSON串,甚至是一张图片或者一个PDF文档等不同的媒体形式。

组件开发实现情况

1、前端控制器DispatcherServlet（不需要工程师开发）,由框架提供 作用：接收请求，响应结果，相当于转发器，中央处理器。有了dispatcherServlet减少了其它组件之间的耦合度。 用户请求到达前端控制器，它就相当于mvc模式中的c，dispatcherServlet是整个流程控制的中心，由它调用其它组件处理用户的请求，dispatcherServlet的存在降低了组件之间的耦合性。

2、处理器映射器HandlerMapping(不需要工程师开发),由框架提供 作用：根据请求的url查找Handler HandlerMapping负责根据用户请求找到Handler即处理器，springmvc提供了不同的映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。

3、处理器适配器HandlerAdapter 作用：按照特定规则（HandlerAdapter要求的规则）去执行Handler 通过HandlerAdapter对处理器进行执行，这是适配器模式的应用，通过扩展适配器可以对更多类型的处理器进行执行。

4、处理器Handler(需要工程师开发) 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 Handler 是继DispatcherServlet前端控制器的后端控制器，在DispatcherServlet的控制下Handler对具体的用户请求进行处理。 由于Handler涉及到具体的用户业务请求，所以一般情况需要工程师根据业务需求开发Handler。

5、视图解析器View resolver(不需要工程师开发),由框架提供 作用：进行视图解析，根据逻辑视图名解析成真正的视图（view） ViewResolver负责将处理结果生成View视图，View Resolver首先根据逻辑视图名解析成物理视图名即具体的页面地址，再生成View视图对象，最后对View进行渲染将处理结果通过页面展示给用户。 springmvc框架提供了很多的View视图类型，包括：jstlView、freemarkerView、pdfView等。 一般情况下需要通过页面标签或页面模版技术将模型数据通过页面展示给用户，需要由工程师根据业务需求开发具体的页面。

6、视图View(需要工程师开发jsp...) View是一个接口，实现类支持不同的View类型（jsp、freemarker、pdf...）

1.5 核心分发器DispatcherServlet

> DispatcherServlet是Spring MVC的"灵魂"和"心脏",它负责接受HTTP请求并协调 Spring MVC的各个组件完成请求处理的工作。和任何Servlet一样,用户必须在web.xml中配置好DispatcherServlet。

1.5.1 DispatcherServlet继承关系

继承关系结构图

1.5.2 DispatcherServlet介绍

DispatcherServlet是前端控制器设计模式的实现，提供spring Web MVC的集中访问点，而且负责职责的分派，而且与Spring IoC容器无缝集成，从而可以获得Spring的所有好处。

1.5.3 DispatcherServlet主要职责

DispatcherServlet主要用作职责调度工作，本身主要用于控制流程，主要职责如下：

1. 文件上传解析，如果请求类型是multipart将通过MultipartResolver进行文件上传解析；
2. 通过HandlerMapping，将请求映射到处理器（返回一个HandlerExecutionChain，它包括一个处理器、多个HandlerInterceptor拦截器）；
3. 通过HandlerAdapter支持多种类型的处理器(HandlerExecutionChain中的处理器)；
4. 通过ViewResolver解析逻辑视图名到具体视图实现；
5. 本地化解析；
6. 渲染具体的视图等；
7. 如果执行过程中遇到异常将交给HandlerExceptionResolver来解析。

1.5.4 DispatcherServlet核心代码

```
//前端控制器分派方法
protected void doDispatch(HttpServletRequest request, HttpServletResponse
response)    throws Exception {
        HttpServletRequest processedRequest = request;
        HandlerExecutionChain mappedHandler = null;
        int interceptorIndex = -1;

        try {
            ModelAndView mv;
            boolean errorView = false;

            try {
                //检查是否是请求是否是multipart（如文件上传），如果是将通过
MultipartResolver解析
                processedRequest = checkMultipart(request);
                //步骤2、请求到处理器（页面控制器）的映射，通过HandlerMapping进行映射
                mappedHandler = getHandler(processedRequest, false);
                if (mappedHandler == null || mappedHandler.getHandler() == null) {
                    noHandlerFound(processedRequest, response);
                    return;
                }
                //步骤3、处理器适配，即将我们的处理器包装成相应的适配器（从而支持多种类
型的处理器）
                HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

                // 304 Not Modified缓存支持
                //此处省略具体代码

                // 执行处理器相关的拦截器的预处理（HandlerInterceptor.preHandle）
                //此处省略具体代码

                // 步骤4、由适配器执行处理器（调用处理器相应功能处理方法）
```

```
                mv = ha.handle(processedRequest, response,
mappedHandler.getHandler());

                // Do we need view name translation?
                if (mv != null && !mv.hasView()) {
                    mv.setViewName(getDefaultViewName(request));
                }

                // 执行处理器相关的拦截器的后处理（HandlerInterceptor.postHandle）
                //此处省略具体代码
            }
            catch (ModelAndViewDefiningException ex) {
                logger.debug("ModelAndViewDefiningException encountered", ex);
                mv = ex.getModelAndView();
            }
            catch (Exception ex) {
                Object handler = (mappedHandler != null ?
mappedHandler.getHandler() : null);
                mv = processHandlerException(processedRequest, response, handler,
ex);
                errorView = (mv != null);
            }

            //步骤5 步骤6、解析视图并进行视图的渲染
            //步骤5 由ViewResolver解析View（viewResolver.resolveViewName(viewName,
locale))
            //步骤6 视图在渲染时会把Model传入（view.render(mv.getModelInternal(),
request, response);）
            if (mv != null && !mv.wasCleared()) {
                render(mv, processedRequest, response);
                if (errorView) {
                    WebUtils.clearErrorRequestAttributes(request);
                }
            }
            else {
                if (logger.isDebugEnabled()) {
                    logger.debug("Null ModelAndView returned to DispatcherServlet
with name '" + getServletName() +
                            "': assuming HandlerAdapter completed request
handling");
                }
            }

            // 执行处理器相关的拦截器的完成后处理
 (HandlerInterceptor.afterCompletion)
            //此处省略具体代码

        catch (Exception ex) {
            // Trigger after-completion for thrown exception.
```

```
            triggerAfterCompletion(mappedHandler, interceptorIndex,
processedRequest, response, ex);
            throw ex;
        }
        catch (Error err) {
            ServletException ex = new NestedServletException("Handler processing
failed", err);
            triggerAfterCompletion(mappedHandler, interceptorIndex,
processedRequest, response, ex);
            throw ex;
        }
        finally {
            if (processedRequest != request) {
                cleanupMultipart(processedRequest);
            }
        }
    }
```

1.5.5 DispatcherServlet辅助类

spring中的DispatcherServlet使用一些特殊的bean来处理request请求和渲染合适的视图。这些bean就是Spring MVC中的一部分。你能够通过在WebApplicationContext中的一个或多个配置来使用这些特殊的bean。但是，你不需要在Spring MVC在维护这些默认要使用的bean时，去把那些没有配置过的bean都去初始化一道。在下一部分中，首先让我们看看在DispatcherServlet依赖的那些特殊bean类型

bean类型	说明
Controlle	处理器/页面控制器，做的是MVC中的C的事情，但控制逻辑转移到前端控制器了，用于对请求进行处理
HandlerMapping	请求到处理器的映射，如果映射成功返回一个HandlerExecutionChain对象（包含一个Handler处理器（页面控制器）对象、多个HandlerInterceptor拦截器）对象；如BeanNameUrlHandlerMapping将URL与Bean名字映射，映射成功的Bean就是此处的处理器
HandlerAdapter	HandlerAdapter将会把处理器包装为适配器，从而支持多种类型的处理器，即适配器设计模式的应用，从而很容易支持很多类型的处理器；如SimpleControllerHandlerAdapter将对实现了Controller接口的Bean进行适配，并且掉处理器的handleRequest方法进行功能处理
HandlerExceptionResolver处理器异常解析器	处理器异常解析，可以将异常映射到相应的统一错误界面，从而显示用户友好的界面（而不是给用户看到具体的错误信息）
ViewResolver视图解析器	ViewResolver将把逻辑视图名解析为具体的View，通过这种策略模式，很容易更换其他视图技术；如InternalResourceViewResolver将逻辑视图名映射为jsp视图
LocaleResolver & LocaleContextResolver地区解析器和地区Context解析器	解析客户端中使用的地区和时区，用来提供不同的国际化的view视图。
ThemeResolver	主题解析器,解析web应用中能够使用的主题，比如提供个性化的网页布局。
MultipartResolver	多部件解析器,主要处理multi-part(多部件)request请求，例如：在HTML表格中处理文件上传。
FlashMapManager	FlashMap管理器储存并检索在"input"和"output"的FlashMap中可以在request请求间(通常是通过重定向)传递属性的FlashMap,

二.基于SpringMVC 的WEB应用!

项目名: spring-mvc-base

maven项目pom配置:

```
<dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
```

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>3.1.0</version>
  <scope>provided</scope>
</dependency>
<!-- 添加Spring包 -->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>4.3.6.RELEASE</version>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>4.3.6.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context-support</artifactId>
  <version>4.3.6.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-web</artifactId>
  <version>4.3.6.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>4.3.6.RELEASE</version>
</dependency>

<!-- 为了方便进行单元测试，添加spring-test包 -->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-test</artifactId>
  <version>4.3.6.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aspects</artifactId>
  <version>4.3.6.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-expression</artifactId>
  <version>4.3.6.RELEASE</version>
```

```
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-orm</artifactId>
      <version>4.3.6.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>org.aspectj</groupId>
      <artifactId>aspectjweaver</artifactId>
      <version>1.8.5</version>
    </dependency>
    <!-- jstl -->
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>jstl</artifactId>
      <version>1.2</version>
    </dependency>
```

普通项目需要jar:

具体实现

1.1 配置web.xml

```
 <servlet>
    //①
    <servlet-name>springMVC</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>

    <!--SpringMVC配置文件的名字  <servlet-name>-servlet.xml
        默认位置:src / resources
        如果放在了 src/resources(maven)
                contextConfigLocation:classpath:文件名即可!
                Web-INF/xx.xml
                contextConfigLocation:/WEB-INF/xx.xml
    -->
    //②
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/springMVC-servlet.xml</param-value>
    </init-param>

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

```
</servlet>
 <!-- 访问DispatcherServlet对应的路径 -->
<servlet-mapping>
   <servlet-name>springMVC</servlet-name>
   <url-pattern>/</url-pattern> <!--/不过滤jsp防止死循环-->
</servlet-mapping>
```

/* 强迫所有的请求及响应都经过该servlet

/ 将使你配置的servlet成为默认的servlet。/不拦截jsp资源!

配置解释:

Spring MVC同样需要创建配置文件,但是配置文件的位置由DispatcherServlet的配置决定!

默认情况下,Spring MVC配置文件的名字为:-servlet.xml 位置:src/resources(maven)。

当然，我们也可以修改默认配置的名字和位置的引用!如果想修改,我们可以直接在声明DispatcherServlet标签中添添加init-param 即可, init-param的参数名固定!具体名称代表含义如下!

param-name	param-value
contextConfigLocation	引入springmvc配置文件,默认classpath:-servlet.xml,如果放在src/resources的下级文件夹,例如:resources/spring/servlet-name-servlet.xml,值可以编写成: classpath:/spring/servlet-name-servlet.xml，如果没有放在src/resources资源的根目录下,放在了WEB项目的WEB-INF/spring/servlet-name-serlvet.xml,值可以编写成:/WEB-INF/spring/servlet-name-servlet.xml。
namespace	修改DispatcherServlet对应的命名空间,默认为-servlet,可以通过namespace修改默认名字!
publishContext	布尔类型的属性,默认值为ture,DispatcherServlet根据该属性决定是否将WebApplicationContext发布到ServletContext的属性列表中,以便调用者可以借用ServletContext找到WebApplicationContext实例,对应的属性名为DispatcherServlet#getServletContextAttributeName()方法返回的值。
publishEvents	布尔类型的属性,当DispatcherServlet处理完一个请求后,是否需要向容器发布一个ServletRequestHandledEvent事件,默认值为true.如果容器中没有任何事件监听器,则可以将该属性设置为false,以便提高运行性能。

注意: Spring4.0已全面支持Servlet3.0,因此,在Servlet3.0环境中,也可以使用编程的方式来配置Servlet容器.具体代码如下:

```
public class AppWebApplicationInit implements WebApplicationInitializer {
    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        ServletRegistration.Dynamic dispatcher =
servletContext.addServlet("dispatcher", new DispatcherServlet());
        dispatcher.setLoadOnStartup(1);
        dispatcher.addMapping("/");
    }
}
```

简单介绍下Servlet3.0的实现原理,在Servlet3.0环境中,容器会在类路径中查找实现javax.servlet.ServletContainerInitializer的类,如果发现已有实现类,就会调用它来配置Servlet容器,在Spring中,org.springframework.web.SpringServletContainerInitializer 类实现了该接口,同时这个类又会查找实现org.springframework.web.WebApplicationInitializer接口的类,并将配置任务交给这些实现类去完成.

1.2 配置Spring MVC配置文件

```
<?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:p="http://www.springframework.org/schema/p"
       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">
     <!-- 扫描controller-->
     <context:component-scan base-package="com.itqf.springmvc.controller"/>


     <!-- 视图解析器 -->
     <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <!-- jsp所在的位置-->
        <property name="prefix" value="/WEB-INF/jsp/" />
        <!-- jsp文件的后缀名-->
        <property name="suffix" value=".jsp" />

    </bean>
</beans>
```

1.3 创建controller包以及控制器类

```
@Controller
public class HelloWorld {

    @RequestMapping("/helloworld")
    public String hellworld(){
        System.out.println("helloworld");
        return "helloworld";
    }

}
```

@Controller 来源于@Component标示为控制层，用于加在类上。

@RequestMapping("/helloworld") 该方法对应的uri;

控制器类的方法返回字符串类型非常常见,返回字符串,代表根据返回的字符串找到对应的视图!

根据springmvc配置文件中视图解析器(InternalResourceViewResolver) 配置的视图文件的前缀和后缀!

helloworld()方法返回 "helloworld" 会找到 WEB-INF/jsp/helloworld.jsp文件!

1.4 测试Spring MVC

发布项目,通过浏览器,访问 当前项目对应地址+ /helloworld即可!

三. 常用注解

1. @RequestMapping注解应用

- 加在类上: 给模块添加根路径
- 加载方法: 方法具体的路径

1.1 @RequestMapping注解value属性

@RequestMapping包含一个value属性!改属性也是默认属性,value属性主要指定的是!方法被访问的具体路径!

例如:

类 @RequestMapping(value="/user"或者"user");

方法 @RequestMapping(value="/list"或者"list");

那么此方法对应路径为: `协议://主机IP:端口号/项目根路径/user/list`

注意: 类上@RequestMapping可以不写,那么就不需要追加/user即可!

```
@RequestMapping("/xx") / @RequestMapping(value = "/xx")
@Controller
public class HelloWorld {

    @RequestMapping("/helloworld")
    public String hellworld(){
        System.out.println("helloworld");
        return "helloworld";
    }
}
```

1.2 设置@RequestMapping method属性

> 可以指定方法对应的请求方式!如果客户端请求的方式和方法设置的方式不同,请求不成功!

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

@RequestMapping(value = "/helloworld" , method =RequestMethod.POST)

注意: 如果不指定method,可以接收任何类型的请求!如果指定但是访问类型不对会出现405错误!

2. @RequestParam表单参数处理

2.1 获取表单参数

- 创建一个登陆表单

```
<html>
<head>
    <title>Title</title>
</head>
<body>
    <!--action指定controller中对应的方法路径即可!-->
    <form action="/xx/login" method="POST">
        <label for="username">用户名:<input type="text" id="username"
name="username" /></label>
        <label for="password">密码:<input type="text" id="password"
name="password" /></label>
        <input type="submit" value="登陆">
    </form>
</body>

</html>
```

- 获取参数的控制器

```
 @RequestMapping("/xx")
 @Controller
```

```
public class HelloWorld {

    @RequestMapping(value = "/helloworld",method = RequestMethod.GET)
    public String hellworld(){
        System.out.println("helloworld");
        return "helloworld";
    }
   //接收form表单
    @RequestMapping(value = "/login",method = RequestMethod.POST)
    public  String login(String username,String password){
        System.out.println("username = " + username);
        System.out.println("password = " + password);
        return "helloworld";
    }
}
```

获取参数,只需要在对应的方法中添加参数即可,如果参数名与请求传参的name值相同即可直接赋值,注意:对应类型很重要,如果是普通的输入框,使用字符串即可,如果是多选框,可以使用List类型的参数接值!

如果参数名和name值相同,无需使用@RequestParam注解!

注意: 将基本类型转化成包装类型!!

2.2 方法的参数名与传参的name值不同

2.1是指name的值和方法参数名相同,开发中,也会碰到请求参数name的值与方法的参数名不同,我们还需要将指定的name对应参数传给方法的指定参数,这时,就不需要使用@RequestParam注解!

```
//此案例,我们修改了input标签的name值,使得与login方法不同,所以我们需要使用
@RequestParam(value = //"name") String username将其指定到 username参数上!

<label for="username">用户名:<input type="text" id="username" name="name" />
</label>  //name改为 name

 @RequestMapping(value = "/login",method = RequestMethod.POST)
 public  String login(@RequestParam(value = "name") String username, String
password){
        System.out.println("username = " + username);
        System.out.println("password = " + password);
        return "helloworld";
    }
```

2.3 方法参数设置默认值

经过2.1和2.2的学习,不管name值和方法参数是否相同,我们都能讲想要的请求参数赋给对应的方法参数上。

但是，有一种特殊情况，如果客户端没有在请求传参，那么我们将得到null,我们不希望得到null,希望得到一个默认值,这个时候,我们还需要使用@RequestParam的defaultValue属性进行对应的设置。

```
@RequestMapping("/list")
public  String list(@RequestParam(defaultValue = "1") Integer currentPage ,
@RequestParam(defaultValue = "10") Integer pageSize){
    //设置默认值,如果不传递使用参数的默认值
     System.out.println("currentPage = " + currentPage);
     System.out.println("pageSize = " + pageSize);
     return  "list";
 }
```

3. @PathVariable获取路径参数

我们可以通过此注解,获取路径部分的数据!

例如: http://localhost:8080/user/list/1

获取路径/list/后面1的数据!

```
@RequestMapping("/user/list/{id}")

public  String getData(@PathVariable(value = "id") Integer id){

        System.out.println("id = " + id);

        return "list" ;
}
```

代码解释: 将路径中想要获取部分使用 `{标注名}` 标注,在方法对应赋值的参数添加@PathVariable注解即可!value值为标注名!!!

4. @CookieValue

@CookieValue注解可以获取请求中的cookie!!

@RequestMapping("/cookie")

```
public String testCookie(@CookieValue("JSESSIONID")String cookie)
{
    System.out.println("cookie:"+cookie);
    return "result";
}
```

5. @RequestHeader

@RequestHeader注解可以获取请求头中的数据!!

@RequestMapping("/header")

```
public String testHeader(@CookieValue("JSESSIONID")String cookie,
        @RequestHeader("User-Agent")String header)
{
    System.out.println("cookie:"+cookie);
    System.out.println("header:"+header);
    return "result";
}
```

6. 请求表达式

通过表达式精准映射请求

- params和headers支持简单的表达式
    1. param:表示请求必须包含名为param的请求参数
    2. !param:表示请求中不能包含名为param的参数
    3. param != value:表示请求中包含param的请求参数,但是值不能为value
    4. param == value:表示请求中包含param的请求参数,但是值为value

param

```
@RequestMapping(value = "/param" , params = {"!username","age!=10"})
public String testParam(String usernam , Integer age){
    System.out.println("usernam:"+usernam);
    System.out.println("age:"+age);
    return "result";
}
```

param 和 header

```
@RequestMapping(value = "/param1" ,headers={"Connection!=keep-alive"},params =
{"!username","age!=10"})
public String testParam1(String usernam , Integer age){
     System.out.println("usernam:"+usernam);
     System.out.println("age:"+age);
     return "result";
}
```

7. ant风格的路径

ant风格资源地址支持3种匹配符:

1. ?:匹配文件名中的一个字符
2. *:匹配文件名中的任意字符
3. **:匹配多层路径

代码:

```
 @RequestMapping(value = "/p/*/xx" )
```

四. 其他重要操作

1. 转发和重定向

转发: forward

重定向:redirect

转发语法:

```
@RequestMapping("/user")
@Controller
public class UserContoller {

    @RequestMapping("/index")
    public String index(Integer size){
        System.out.println("index method 被调用!"+size);
        return "forward:/user/result";
    }

    @RequestMapping("/result")
    public String result(){
        return "result";
    }

}
```

重定向语法:

```
@RequestMapping("/user")
@Controller
public class UserContoller {

    @RequestMapping("/index")
    public String index(Integer size){
        System.out.println("index method 被调用!"+size);
        return "redirect:/user/result";
    }

    @RequestMapping("/result")
    public String result(){

        return "result";
    }
}
```

2. 解决参数乱码问题

Spring MVC中 GET方式不会乱码!

在web.xml配置文件中添加spring自带的Filter设置编码格式

```
  <filter>
    <filter-name>characterEncodingFilter</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>
    <init-param>
      <param-name>forceEncoding</param-name>
      <param-value>true</param-value>
    </init-param>
  </filter>

  <filter-mapping>
    <filter-name>characterEncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
```

五.实战练习-向controller传递对象类型数据

> 通过form表单向指定的controller的方法传递对象!

1. 创建Pojo对象

Address.java

```
package com.itqf.springmvc.pojo;

public class Address {

    private String province;
    private String city;
    //toString,getter,setter
}
```

User.java

```
public class User {

    private String username;
    private String password;
    private Integer  age;
    private Address  address;
    //toString,getter,setter
}
```

2. 创建控制器类

```
@RequestMapping("/user")
@Controller
public class UserContoller {
    //跳转到 WEB-INF/user/form.jsp
    @RequestMapping("/form")
    public String from(){
        return "user/form";
    }
    //form表单提交数据到此处!获取在转发到success.jsp
    @RequestMapping("/add")
    public String add(User user){
        System.out.println(user);
        return "success";
    }
}
```

3. 创建form.jsp

文件位置: /WEB-INF/user/form.jsp

```
<form action="/spring-mvc-22/user/add" method="post">
 <label for="username">用户名:<input type="text" id="username" name="username" />
</label><br/>
 <label for="password">密码:<input type="password" id="password" name="password"
/></label><br/>
 <label for="age">年龄:<input type="text" name="age" /></label><br/>
 <label for="address.province">省份:<input type="text" name="address.province" />
</label><br/>
 <label for="address.city">城市:<input type="text" name="address.city" /></label>
<br/>
 <input type="submit"  value="提交" />
</form>
```

注意:name的特殊写法,这里可以直接将表单数据转成User对象,但是User对象内部包含 Address的对象,所以,这里可以调用第一层属性,再点一层属性,如果多层依次类推!

4. 创建success.jsp

文件位置: /WEB-INF/success.jsp

5. 测试

访问跳转路径即可!!!

6. Pojo类汇总包含Date字段处理

实体类中,包含Date类型,需要使用特殊的注解进行转化!

```
public class User {
  private String name;
  private String age;
  @DateTimeFormat(pattern = "yyyy-MM-dd")
  private Date   birthday;'
  //getter setter toString
}
```

课前默写

1.Spring JDBC的配置和使用
2. 在Spring中事务的处理

作业

1. 使用Hibernate第三天作业数据表
2. 使用SpringMVC框架完成CRUD操作
3. 页面使用JSP+EL展示

面试题

1. SpringMVC框架的运行原理和流程
2. 简述SpringMVC的基础配置
3. SpringMVC常用注解的作用
4. SpringMVC中参数的传递
5. SpringMVC中参数类型的转换
6. SpringMVC中页面的跳转
7. SpringMVC中乱码的解决