# SpringMVC学习笔记

# 三层架构

我们的开发架构一般都是基于两种形式，一种是 C/S 架构，也就是客户端/服务器，另一种是 B/S 架构，也就是浏览器服务器。在 JavaEE 开发中，几乎全都是基于 B/S架构的开发。那么在 B/S架构中，系统标准的三层架构包括：表现层、业务层、持久层。三层架构在我们的实际开发中使用的非常多，所以我们课程中的案例也都是基于三层架构设计的。

三层架构中，每一层各司其职，接下来我们就说说每层都负责哪些方面：

表现层：

也就是我们常说的web层。它负责接收客户端请求，向客户端响应结果，通常客户端使用http协议请求web 层，web 需要接收 http 请求，完成 http 响应。

表现层包括展示层和控制层：控制层负责接收请求，展示层负责结果的展示。

表现层依赖业务层，接收到客户端请求一般会调用业务层进行业务处理，并将处理结果响应给客户端。

表现层的设计一般都使用 MVC 模型。（MVC 是表现层的设计模型，和其他层没有关系）

业务层：

也就是我们常说的 service 层。它负责业务逻辑处理，和我们开发项目的需求息息相关。web 层依赖业务层，但是业务层不依赖 web 层。

业务层在业务处理时可能会依赖持久层，如果要对数据持久化需要保证事务一致性。（也就是我们说的，事务应该放到业务层来控制）

持久层：

也就是我们是常说的 dao 层。负责数据持久化，包括数据层即数据库和数据访问层，数据库是对数据进行持久化的载体，数据访问层是业务层和持久层交互的接口，业务层需要通过数据访问层将数据持久化到数据库中。通俗的讲，持久层就是和数据库交互，对数据库表进行曾删改查的。

# MVC模型

MVC 全名是 Model View Controller，是模型(model)－视图(view)－控制器(controller)的缩写，是一种用于设计创建 Web 应用程序表现层的模式。MVC 中每个部分各司其职：

Model （模型） ：

通常指的就是我们的数据模型。作用一般情况下用于封装数据。

View （视图） ：

通常指的就是我们的 jsp 或者 html。作用一般就是展示数据的。

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。作用一般就是处理程序逻辑的。

它相对于前两个不是很好理解，这里举个例子：

例如：

我们要保存一个用户的信息，该用户信息中包含了姓名，性别，年龄等等。

这时候表单输入要求年龄必须是 1~100 之间的整数。姓名和性别不能为空。并且把数据填充到模型之中。此时除了 js 的校验之外，服务器端也应该有数据准确性的校验，那么校验就是控制器的该做的。

当校验失败后，由控制器负责把错误页面展示给使用者。

如果校验成功，也是控制器负责把数据填充到模型，并且调用业务层实现完整的业务需求。

# SpringMVC概述

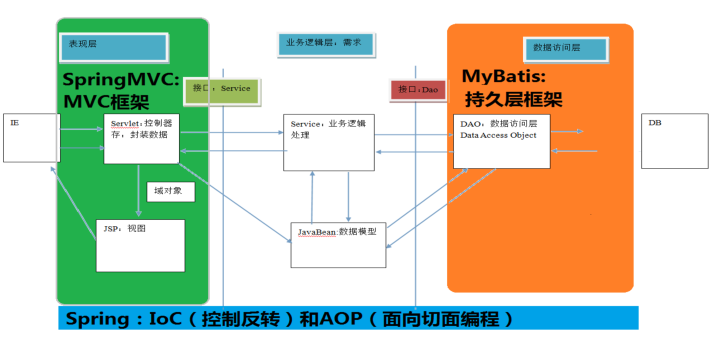
## 什么是springMVC

SpringMVC 是一种基于 Java 的实现 MVC 设计模型的请求驱动类型的轻量级 Web 框架，属于 SpringFrameWork 的后续产品，已经融合在 Spring Web Flow 里面。Spring 框架提供了构建 Web 应用程序的全功能 MVC 模块。使用 Spring 可插入的 MVC 架构，从而在使用 Spring 进行 WEB 开发时，可以选择使用 Spring的 Spring MVC 框架或集成其他 MVC 开发框架，如 Struts1(现在一般不用)，Struts2 等。

SpringMVC 已经成为目前最主流的 MVC 框架之一，并且随着 Spring3.0 的发布，全面超越 Struts2，成为最优秀的 MVC 框架。

它通过一套注解，让一个简单的 Java 类成为处理请求的控制器，而无须实现任何接口。同时它还支持RESTful 编程风格的请求。

## SpringMVC 在三层架构的位置



## SpringMVC的优势

1、清晰的角色划分：

前端控制器（DispatcherServlet）

请求到处理器映射（HandlerMapping）

处理器适配器（HandlerAdapter）

视图解析器（ViewResolver）

处理器或页面控制器（Controller）

验证器（ Validator）

命令对象（Command 请求参数绑定到的对象就叫命令对象）

表单对象（Form Object 提供给表单展示和提交到的对象就叫表单对象）。

2、分工明确，而且扩展点相当灵活，可以很容易扩展，虽然几乎不需要。

3、由于命令对象就是一个 POJO，无需继承框架特定 API，可以使用命令对象直接作为业务对象。

4、和 Spring 其他框架无缝集成，是其它 Web 框架所不具备的。

5、可适配，通过 HandlerAdapter 可以支持任意的类作为处理器。

6、可定制性，HandlerMapping、ViewResolver 等能够非常简单的定制。

7、功能强大的数据验证、格式化、绑定机制。

8、利用 Spring 提供的 Mock 对象能够非常简单的进行 Web 层单元测试。

9、本地化、主题的解析的支持，使我们更容易进行国际化和主题的切换。

10、强大的 JSP 标签库，使 JSP 编写更容易。

………………还有比如RESTful风格的支持、简单的文件上传、约定大于配置的契约式编程支持、基于注解的零配

置支持等等。

## SpringMVC 和 和 Struts2的优略分析

共同点：

它们都是表现层框架，都是基于 MVC 模型编写的。

它们的底层都离不开原始 ServletAPI。

它们处理请求的机制都是一个核心控制器。

区别：

Spring MVC 的入口是 Servlet, 而 Struts2 是 FilterSpring MVC 是基于方法设计的，而 Struts2 是基于类，Struts2 每次执行都会创建一个动作类。所以 Spring MVC 会稍微比 Struts2 快些。

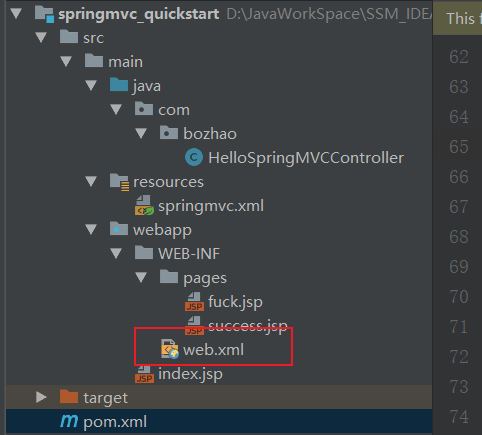
Spring MVC 使用更加简洁,同时还支持 JSR303, 处理 ajax 的请求更方便

(JSR303 是一套 JavaBean 参数校验的标准，它定义了很多常用的校验注解，我们可以直接将这些注解加在我们 JavaBean 的属性上面，就可以在需要校验的时候进行校验了。)

Struts2 的 OGNL 表达式使页面的开发效率相比 Spring MVC 更高些，但执行效率并没有比 JSTL 提升，尤其是 struts2 的表单标签，远没有 html 执行效率高。

# SpringMVC入门程序

创建一个maven的web工程



导入MVC的jar包

<!--spring-webmvc，jar包中包含了spring的五个核心包，好包含了spring-web-->  
<!--所以你会发现，现在做简单的springMVC入门只需要引入spring-webmvc包就够了-->  
<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-webmvc</artifactId>  
 <version>${spring.version}</version>  
</dependency>

在web.xml配置文件中配置核心控制器，配置inti-param来加载springMVC的配置文件

<!--配置servlet核心 控制器-->  
<servlet>  
 <servlet-name>dispatcherServlet</servlet-name>  
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  
 <!--配置全局参数，用来加载配置文件-->  
 <init-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>classpath:springmvc.xml</param-value>  
 </init-param>  
 <load-on-startup>1</load-on-startup>  
</servlet>  
<servlet-mapping>  
 <servlet-name>dispatcherServlet</servlet-name>  
 <url-pattern>/</url-pattern>  
</servlet-mapping>

提供springMVC配置文件

<!--开启包扫描-->  
<context:component-scan base-package="com.bozhao"/>  
  
<!--配置视图解析器-->  
<bean id="internalResourceViewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">  
 <property name="prefix" value="/WEB-INF/pages/"></property>  
 <property name="suffix" value=".jsp"></property>  
</bean>

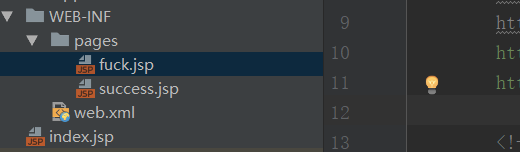
在配置文件中，需要提供相关的命名空间和约束

<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:mvc="http://www.springframework.org/schema/mvc"  
 xmlns:context="http://www.springframework.org/schema/context"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans.xsd  
 http://www.springframework.org/schema/mvc  
 http://www.springframework.org/schema/mvc/spring-mvc.xsd  
 http://www.springframework.org/schema/context  
 http://www.springframework.org/schema/context/spring-context.xsd">

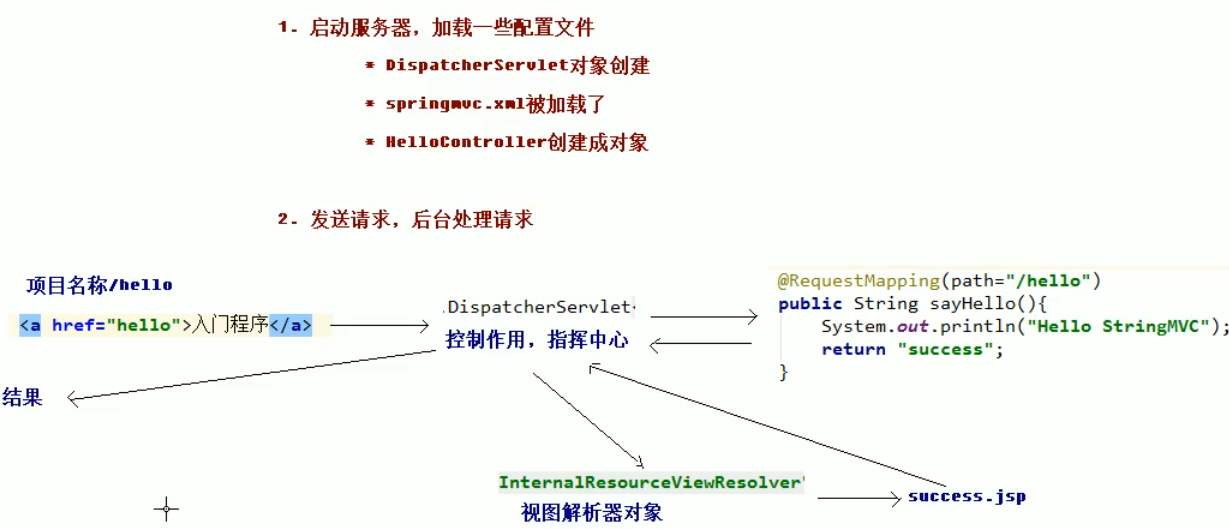
编写controller



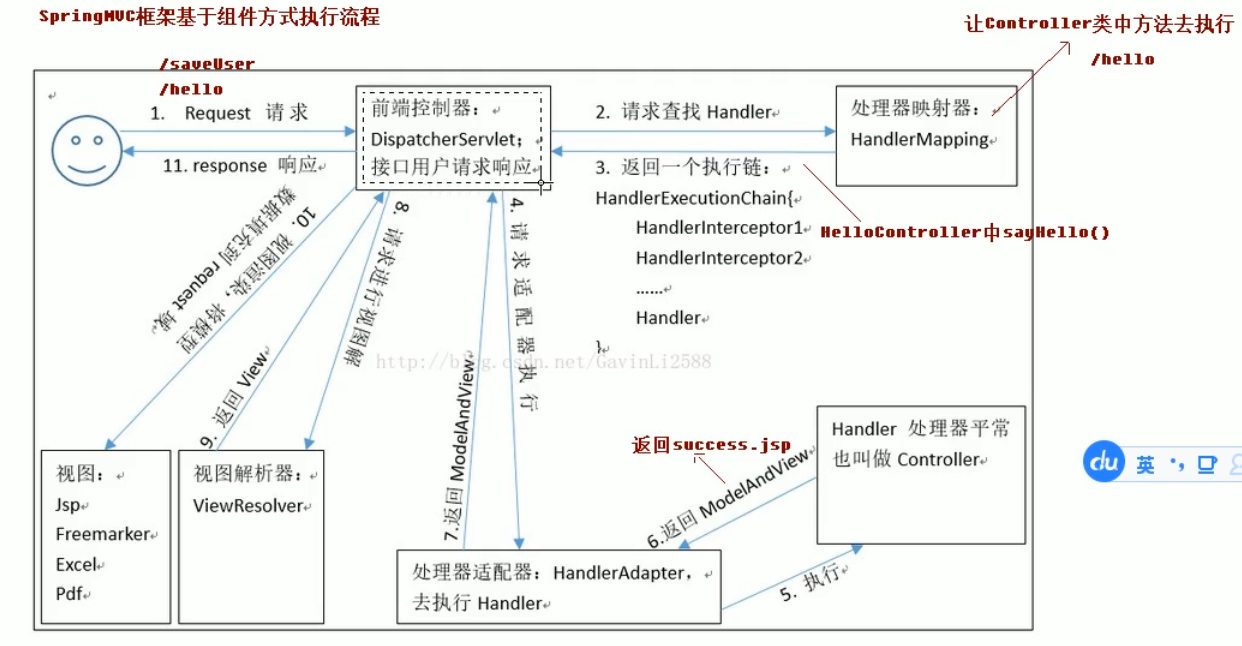
根据试图视图解析器的配置，提供跳转的试图，试图解析器决定了试图存放的位置，以及试图文件的格式



## 入门程序流程分析



## SpringMVC原理图



## SpringMVC中的组件

### DispatcherServlet ：前端控制器

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。

### HandlerMapping ：处理器映射器

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。

### Handler ：处理器

它就是我们开发中要编写的具体业务控制器。由 DispatcherServlet 把用户请求转发到 Handler。由Handler 对具体的用户请求进行处理。

### HandlAdapter ：处理器适配器

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。

这里用的其实是类的适配器模式，类的适配器模式就是希望把一个原有的类转换为适配另一个新的接口，是通过创建一个类继承原有的类，然后实现这个新的接口来实现的

### View Resolver ：视图解析器

View Resolver 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。

### View ：视图

SpringMVC 框架提供了很多的 View 视图类型的支持，包括：jstlView、freemarkerView、pdfView等。我们最常用的视图就是 jsp。

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，需要由程序员根据业务需求开发具体的页面

### <mvc:annotation-driven>

在 SpringMVC 的各个组件中，处理器映射器、处理器适配器、视图解析器称为 SpringMVC 的三大组件。使 用 <mvc:annotation-driven> 自 动加载 RequestMappingHandlerMapping （处理映射器） 和RequestMappingHandlerAdapter （ 处 理 适 配 器 ） ， 可 用 在 SpringMVC.xml 配 置 文 件 中 使 用<mvc:annotation-driven>替代注解处理器和适配器的配置。

它就相当于在 xml 中配置了：

<!-- 上面的标签相当于 如下配置-->

<!-- Begin -->

<!-- HandlerMapping -->

<bean

class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerM

apping"></bean>

<bean

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

<!-- HandlerAdapter -->

<bean

class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerA

dapter"></bean>

<bean

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

<bean

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

<!-- HadnlerExceptionResolvers -->

<bean

class="org.springframework.web.servlet.mvc.method.annotation.ExceptionHandlerExcept

ionResolver"></bean>

<bean

class="org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionResolv

er"></bean>

<bean

class="org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver"

></bean>

<!-- End -->

注意：一般开发中，我们都需要写上此标签（虽然从入门案例中看，我们不写也行，随着课程的深入，该标签还有具体的使用场景）。

## SpringMVC中的注解

### RequestMapping 注解

源码：

@Target({ElementType.METHOD, ElementType.TYPE})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
@Mapping  
public @interface RequestMapping {}

作用：

用于建立请求 URL 和处理请求方法之间的对应关系。

**出现 位置：**

类上：

请求 URL 的第一级访问目录。此处不写的话，就相当于应用的根目录。写的话需要以/开头。

它出现的目的是为了使我们的 URL 可以按照模块化管理:

例如：

账户模块：

/account/add

/account/update

/account/delete

...

订单模块：

/order/add

/order/update

/order/delete

红色的部分就是把 RequsetMappding 写在类上，使我们的 URL 更加精细。

方法上：

请求 URL 的第二级访问目录。

**属性：**

value：用于指定请求的 URL。它和 path 属性的作用是一样的。

method：用于指定请求的方式。(超链接的请求方式是固定的，固定为get)

params：用于指定限制请求参数的条件。它支持简单的表达式。要求请求参数的 key 和 value 必须和配置的一模一样。

例如：

params = {"accountName"}，表示请求参数必须有 accountName

params = {"moeny!100"}，表示请求参数中 money 不能是 100。

headers：用于指定限制请求消息头的条件。

注意：

以上四个属性只要出现 2 个或以上时，他们的关系是与的关系。

### RequestBody注解

作用：

用于获取请求体内容。直接使用得到是 key=value&key=value...结构的数据。

get 请求方式不适用。

属性：

required：是否必须有请求体。默认值是:true。当取值为 true 时,get 请求方式会报错。如果取值为 false，get 请求得到是 null。

示例：

Post请求

<h2>@RequestBody注解练习，获取请求体</h2>  
<form action="requestBody" method="post"><br/>  
 姓名：<input type="text" name="username"/><br/>  
 性别：<input type="text" name="age"/><br/>  
 生日：<input type="text" name="birthday"/><br/>  
 <input type="submit" value="提交"/>  
</form>

Get请求：不适用，所以输出null

<a href="useRequestBody?body=test">requestBody 注解 get 请求</a>



***注意：以前一直以为，requestBody注解是用来封装对象的，这是错误的，该注解是用来获取请求体的，用来封装对象，是当前台传递来的数据是json格式的数据时候，才使用该数据来封装对象，也就是post请求获取请求体，而请求体，大多数时候使用的是json格式,如果不是获取请求体，不是json类型的数据，封装对象，属于参数绑定的范畴，比如form表单提交，将表单内容封装到实体类，直接在方法中写上实体类类型参数即可，详见请求参数绑定***

***使用Ajax发异步请求的时候，当传递的参数是json类型的数据（也相当于请求体），这时后，可以直接使用RequestBody注解来将json类型的参数封装到对应的javaBean中，此时还需要导入jackSon的相关jar包，具体转换，是springMVC框架内部已经做好了***

### PathVariable注解

作用：

用于绑定 url 中的占位符。例如：请求 url 中 /delete/{id}，这个{id}就是 url 占位符。

url 支持占位符是 spring3.0 之后加入的。是 springmvc 支持 rest 风格 URL 的一个重要标志。

属性：

value：用于指定 url 中占位符名称。

required：是否必须提供占位符。

使用示例：

<!-- PathVariable 注解 -->  
<a href="annotation/usePathVariable/100">pathVariable 注解</a>

@RequestMapping("/usePathVariable/{id}")  
public String usePathVariable(@PathVariable(name = "id") Integer money) {  
 logger.info("PathVariable注解练习：" + money);  
 return "success";  
}

#### Rest风格

什么是 rest ：

REST（英文：Representational State Transfer，简称 REST）描述了一个架构样式的网络系统，比如 web 应用程序。它首次出现在 2000 年 Roy Fielding 的博士论文中，他是 HTTP 规范的主要编写者之一。在目前主流的三种 Web 服务交互方案中，REST 相比于 SOAP（Simple Object Access protocol，简单对象访问协议）以及 XML-RPC 更加简单明了，无论是对 URL 的处理还是对 Payload 的编码，REST 都倾向于用更加简单轻量的方法设计和实现。值得注意的是 REST 并没有一个明确的标准，而更像是一种设计的风格。

它本身并没有什么实用性，其核心价值在于如何设计出符合 REST 风格的网络接口。

restful 的优点它结构清晰、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。

restful 的特性：

资源（ Resources）：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存在。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 即为每一个资源的独一无二的识别符。

表现层（ Representation）：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。状态转化（ State Transfer）：每 发出一个请求，就代表了客户端和服务器的一次交互过程。HTTP 协议，是一个无状态协议，即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 Tran sfer）。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 ”。具体说，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， PUT 用来更新资源， DELETE 用来删除资源。

restful 的示例：

/account/1 HTTP GET ： 得到 id = 1 的 account

/account/1 HTTP DELETE： 删除 id = 1 的 account

/account/1 HTTP PUT： 更新 id = 1 的 account

#### 基于HiddentHttpMethodFilter的示例

作用：

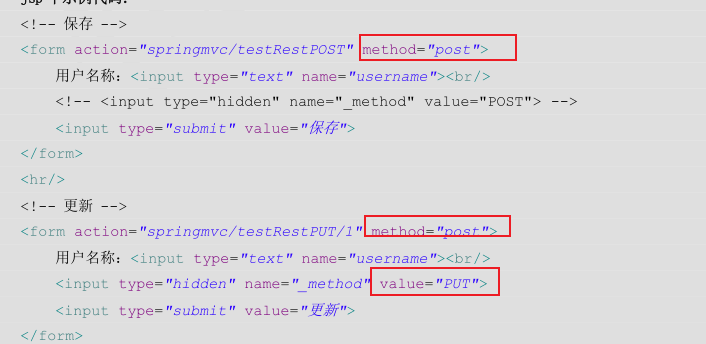
由于浏览器 form 表单只支持 GET 与 POST 请求，而 DELETE、PUT 等 method 并不支持，Spring3.0 添加了一个过滤器，可以将浏览器请求改为指定的请求方式，发送给我们的控制器方法，使得支持 GET、POST、PUT与 DELETE 请求。

使用方法：

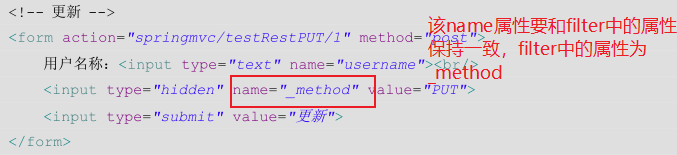
第一步：在 web.xml 中配置该过滤器。

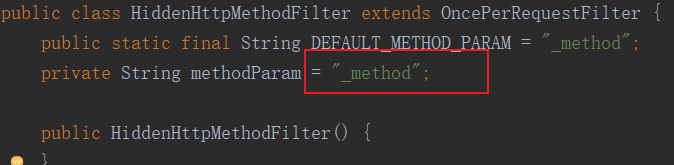
<!--  
 配置 org.springframework.web.filter.HiddenHttpMethodFilter  
 -->  
<filter>  
 <filter-name>HiddenHttpMethodFilter</filter-name>  
 <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>  
</filter>  
<filter-mapping>  
 <filter-name>HiddenHttpMethodFilter</filter-name>  
 <url-pattern>/\*</url-pattern>  
</filter-mapping>

第二步：请求方式必须使用 post 请求。



第三步：按照要求提供\_method 请求参数，该参数的取值就是我们需要的请求方式。





<https://www.cnblogs.com/xiximayou/p/12177670.html>

代码示例：

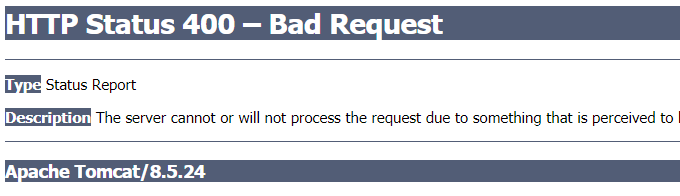
<form action="mockPutByFilter/10" method="post">  
 <input type="hidden" name="\_method" value="DELETE"/>  
 姓名：<input type="text" name="username"/><br/>  
 年龄：<input type="text" name="age"/><br/>  
 生日：<input type="text" name="birthday"/><br/>  
 <input type="submit" value="PUT"/>  
</form>

@RequestMapping(value = "/mockPutByFilter/{id}",method = RequestMethod.*PUT*)

public String mockPutByFilter(@PathVariable("id") Integer id,User user) {  
 logger.info("模拟PUT请求的方法执行，进行更新操作，id: + {" + id + "}" );  
 logger.info("模拟PUT请求的方法执行，进行更新操作，user: + {" + user + "}" );  
 return "success";  
}

#### 遇到的问题★★★★★

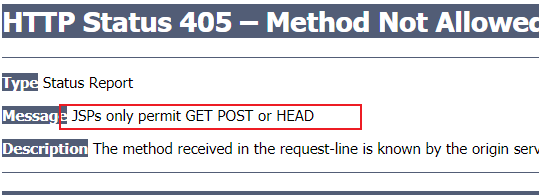
1. 在web.xml中HiddentHttpMethodFilter过滤器已经正确配置，jsp,控制器也书写正确，但是报错如下：后台没有打印日志，说名方法里面没有执行，都没有进入方法



网上查资料说应把过滤器配置在servlet前，然后改了，发现并没有卵用，也有人说这是Tomcat8的问题，可以改用Tomcat7试试，但是说这也没有根本解决问题

说明：出现如上错误，与配置无关，也不是因为没有使用@ResponseBody注解的原因，这里就算没使用@ResponseBody注解，还是能够进入方法了，执行方法，打印日志了，一开始出现如上错误的原因是，进行参数绑定的时候出现了异常，因为User类中是年龄age属性，而我传了一个性别属性“女”,这是参数绑定时出了异常

只是没有使用@ResponseBody注解的时候会出现如下异常，也就是说进入方法执行方法和有没有使用@ResponseBody注解无关



1. 针对如上出现的错误：JSPs only permit GET POST or HEAD的错误：网上的说法是在方法上使用@ResponseBody注解，只是使用@ResponseBody注解后，返回字符串就无法跳转视图，因为使用该注解后，返回的字符串，就会在页面显示该字符串
2. 针对2中使用了@ResponseBody注解后返回字符串无法跳转视图，显然使用@ResponseBody有些时候显得不太合适，因为不能跳转到想要的视图了，这里有如下解决方法
3. 添加一个控制器名为tosuccess，返回类型为string，RequestMapping value值为"/tosuccess"不要添加包名，否则404，return “success”；

@Controller  
public class ToSuccessController {  
 @RequestMapping("/toSuccess")  
 public String toSuccess() {  
 return "success";  
 }  
}

1. 在处理Delete请求 和 Put请求的控制器，返回值改为“redirect:/tosuccess”

@RequestMapping(value = "/mockRestByFilter/{id}",method = RequestMethod.*PUT*)  
public String mockPutByFilter(@PathVariable("id") Integer id,User user) {  
 logger.info("模拟PUT请求的方法执行，进行更新操作，id: + {" + id + "}, user:" + user );  
 return "redirect:/toSuccess";  
}

说明：redirect也就是重定向，后面会升入学习请求重定向和请求转发

参考连接：<https://blog.csdn.net/weixin_45165669/article/details/104617304>

### RequestHeader注解

作用：

用于获取请求消息头。

属性：

value：提供消息头名称

required：是否必须有此消息头

注：

在实际开发中一般不怎么用。

使用示例：

<!-- RequestHeader 注解 -->  
<a href="useRequestHeader">获取请求消息头</a>

@RequestMapping("/useRequestHeader")  
public String useRequestHeader(@RequestHeader(value="Accept-Language",  
 required=false) String requestHeader) {  
 logger.info("获取到的请求头信息 Accept-Language: " + requestHeader);  
 return "success";  
}

### CookieValue注解

作用：

用于把指定 cookie 名称的值传入控制器方法参数。

属性：

value：指定 cookie 的名称。

required：是否必须有此 cookie

使用示例：

<!-- CookieValue 注解 -->  
<a href="useCookieValue">绑定 cookie 的值</a>

@RequestMapping("/useCookieValue")  
public String useCookieValue(@CookieValue(value="JSESSIONID",required=false)  
 String cookieValue){  
 logger.info("获取到的cookie信息 cookie: " + cookieValue);  
 return "success";  
}

运行结果：



### ModelAttribute注解

作用：

该注解是 SpringMVC4.3 版本以后新加入的。它可以用于修饰方法和参数。

出现在方法上，表示当前方法会在控制器的方法执行之前，先执行。它可以修饰没有返回值的方法，也可以修饰有具体返回值的方法。出现在参数上，获取指定的数据给参数赋值。

属性：

value：用于获取数据的 key。key 可以是 POJO 的属性名称，也可以是 map 结构的 key。

应用场景：

当表单提交数据不是完整的实体类数据时，保证没有提交数据的字段使用数据库对象原来的数据。

例如：

我们在编辑一个用户时，用户有一个创建信息字段，该字段的值是不允许被修改的。在提交表单数据是肯定没有此字段的内容，一旦更新会把该字段内容置为 null，此时就可以使用此注解解决问题。

使用示例：

#### 基于 POJO 属性的基本使用,ModelAttribute修饰方法带返回值

<h3>ModelAttribute注解的使用</h3>  
<form action="testModelAttribute" method="post">  
 姓名： <input type="text" name="username"><br/>  
 年龄：<input type="text" name="age"><br/>  
 <input type="submit" value="ModelAttribute注解的使用">  
</form>

@ModelAttribute  
 public User showModel(String username) {  
 System.*out*.println("执行了 showModel 方法,username: "+ username);  
 User user = new User();  
 user.setAge(25);  
 user.setId(12);  
 user.setBirthday(new Date());  
 user.setUsername(username);  
 return user;  
 }

@RequestMapping(value = "/testModelAttribute",method = RequestMethod.*POST*)  
public String modelAttributeTest(User user) {  
 System.*out*.println("执行了控制器的方法,username: " + user);  
 return "success";  
}

说明：这里当form表单提交的时候，会请求testModelAttribute，但是showModel方法上有ModelAttribute注解，所以showModel方法会在请求的控制器中的方法执行前先执行，这时就可以进行查数据库，封装数据等操作，防止表单提交信息不全，出现封装对象时有属性为null的情况，比如这里，user对象有四个属性，但是form表单只提交了两个属性，如果不执行加了ModelAttribute的方法，form表单中没提交的属性将为null值，比如表单提交了username属性，在showModel方法中就可以根据username属性查询数据库，给表单没有提交的其它属性赋值，并返回一个User对象，这时候在执行请求控制器的方法时，form表单没有提交的属性也赋值了

#### 基于Map的应用场景示例ModelAttribute修饰方法不带返回值

被@ModelAttribute注解修饰的方法不带返回值，这时后在方法中需要使用一个参数为Map类型来封装数据，Map的key类型可以自己定义，一般使用String类型，value为要封装的类型如User，然后在请求控制器的方法中参数上使用@ModelAttribute注解，name属性，或者value属性，给为Map中的存的key示例如下：

<form action="testModelAttribute" method="post">  
 <input type="hidden" name="\_method" value="PUT">  
 姓名： <input type="text" name="username"><br/>  
 年龄：<input type="text" name="age"><br/>  
 <input type="submit" value="ModelAttribute注解的使用PUT">  
</form>

@ModelAttribute  
 public void showModel(String username, Map<String,User> map) {  
 System.*out*.println("执行了 showModel 方法,username: "+ username);  
 User user = new User();  
 user.setAge(25);  
 user.setId(12);  
 user.setBirthday(new Date());  
 user.setUsername(username);  
 map.put("user", user);  
 }

@RequestMapping(value = "/testModelAttribute",method = RequestMethod.*PUT*)  
public String modelAttributeTest2(@ModelAttribute("user") User user) {  
 System.*out*.println("执行了控制器的方法,: " + user);  
 return "redirect:/toSuccess";  
}

### SessionAttribute注解

作用：

用于多次执行控制器方法间的参数共享。

属性：

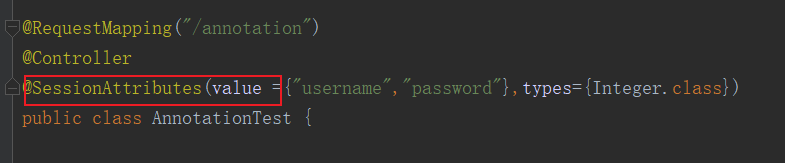
value：用于指定存入的属性名称

type：用于指定存入的数据类型。

使用示例：

<!-- SessionAttribute 注解的使用 -->  
<a href="testPut">存入 SessionAttribute</a>  
<hr/>  
<a href="testGet">取出 SessionAttribute</a>  
<hr/>  
<a href="testClean">清除 SessionAttribute</a>

使用在类上



*/\*\*  
 \* 把数据存入 SessionAttribute  
 \** ***@param*** *model  
 \** ***@return*** *\* Model 是 spring 提供的一个接口，该接口有一个实现类 ExtendedModelMap  
 \* 该类继承了 ModelMap，而 ModelMap 就是 LinkedHashMap 子类  
 \*/*@RequestMapping("/testPut")  
public String testPut(Model model){  
 //Model底层会将信息存储到Request域对象中  
 model.addAttribute("username", "泰斯特");  
 model.addAttribute("password","123456");  
 model.addAttribute("age", 31);  
 //跳转之前将数据保存到 username、password 和 age 中，因为注解@SessionAttribute 中有这几个参数  
 return "success";  
}

*/\*\*  
 \* 从SessionAttribute中获取保存的数据  
 \** ***@param*** *model  
 \** ***@return*** *\*/*@RequestMapping("/testGet")  
public String testGet(ModelMap model){  
 System.*out*.println(model.get("username")+";"+model.get("password")+";"+model.get("age"));  
 return "success";  
}

*/\*\*  
 \* 清空SessionAttribute中保存的数据  
 \** ***@param*** *sessionStatus  
 \** ***@return*** *\*/*@RequestMapping("/testClean")  
public String complete(SessionStatus sessionStatus){  
 sessionStatus.setComplete();  
 return "success";  
}

***其它注解：详见讲义第一天中常用注解***

# 请求参数的绑定

## 请求参数绑定说明

### 绑定机制

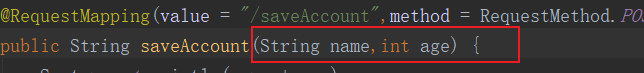
1. 表单提交的数据都是k = v格式的，如：username=haha&password=123
2. SpringMVC的参数绑定过程是把表单提交的请求参数，作为控制器中方法的参数进行绑定的
3. 要求：提交表单的name和参数名称是相同的

### 支持的数据类型

1. 基本数据类型和字符串类型
2. 实体类型（JavaBean）
3. 集合数据类型（List、Map集合等）

## 基本数据类型和字符串数据类型

1. 提交表单的name和参数的名称是相同的
2. 区分大小写

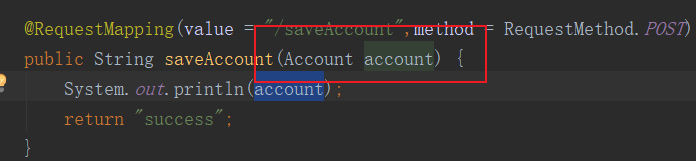


基本类型和string绑定就非常简单，直接在方法中使用参数就行，但是要注意请求中的参数名要和方法中的参数名对应，如果是form表单则表单中标签的name属性，和参数名保持一致

## 实体类型

1. 提交表单的name和JavaBean中的属性名称需要一致
2. 如果JavaBean中包含其它的引用类型，那么表单的name属性需要编写成：对象.属性 例如:user.userName，user指的是JavaBean中引用类型的属性名

<%--请求参数绑定练习，JavaBean的封装--%>  
<form action="account/saveAccount" method="post">  
 <%--input 标签是一个自闭合标签--%>  
 账号：<input type="text" name="account"><br/>  
 密码：<input type="text" name="password"><br/>  
 金额：<input type="text" name="money"><br/>  
 用户名称：<input type="text" name="user.uname"><br/>  
 用户年龄:<input type="text" name="user.age"><br/>  
 <input type="submit" value="提交">  
</form>



## 集合属性数据封装

1. Jsp页面编写方式：list[0].属性

<%--请求参数绑定练习，JavaBean的封装,JavaBean中包含list、map集合类型--%>  
<form action="account/saveAccount" method="post">  
 <%--input 标签是一个自闭合标签--%>  
 账号：<input type="text" name="account"><br/>  
 密码：<input type="text" name="password"><br/>  
 金额：<input type="text" name="money"><br/>  
 <%--封装到list集合中--%>  
 用户名称：<input type="text" name="list[0].uname"><br/>  
 用户年龄:<input type="text" name="list[0].age"><br/>  
 用户名称：<input type="text" name="list[1].uname"><br/>  
 用户年龄:<input type="text" name="list[1].age"><br/>  
 <%--封装到map集合中--%>  
 用户名称：<input type="text" name="map['one'].uname"><br/>  
 用户年龄:<input type="text" name="map['one'].age"><br/>  
 用户名称：<input type="text" name="map['two'].uname"><br/>  
 用户年龄:<input type="text" name="map['two'].age"><br/>  
 <input type="submit" value="提交">  
</form>

public class Account implements Serializable{  
  
 private String account;  
 private String password;  
 private double money;  
 // private User user;  
 private List<User> list;  
 private Map<String,User> map;

## 请求参数中文乱码的解决

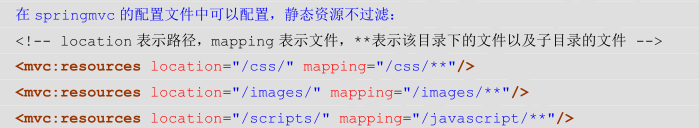
1. 在web.xml中配置spring提供的过滤器

post 请求方式：

在 web.xml 中配置一个过滤器

<!-- 配置 springMVC 编码过滤器 -->

<!--配置post请求中文编码过滤器-->  
<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>  
 <!-- 启动过滤器 -->  
 <init-param>  
 <param-name>forceEncoding</param-name>  
 <param-value>true</param-value>  
 </init-param>  
</filter>  
<!-- 过滤所有请求 -->  
<filter-mapping>  
 <filter-name>characterEncodingFilter</filter-name>  
 <url-pattern>/\*</url-pattern>  
</filter-mapping>

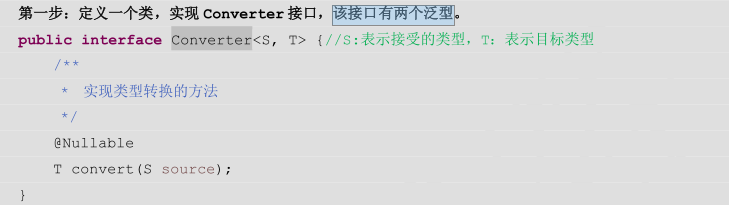


Get请求



## 自定义类型转换器

主要是用来转换时间格式的字符串



public class StringToDateConvert implements Converter<String,Date>{  
 @Override  
 public Date convert(String s) {  
  
 DateFormat format;  
 Date date;  
 try {  
 if(StringUtils.*isEmpty*(s)) {  
 throw new RuntimeException("请输入要转换的日期！");  
 }  
 format = new SimpleDateFormat("yyyy-MM-dd");  
 date = format.parse(s);  
 }catch (Exception e){  
 throw new RuntimeException("输入日期有误！");  
 }  
 return date;  
 }  
}

第二步：在 spring 配置文件中配置类型转换器。

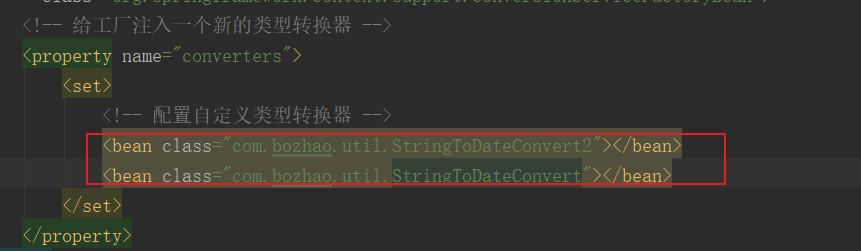
<!-- 配置类型转换器工厂 -->  
<bean id="converterService"  
 class="org.springframework.context.support.ConversionServiceFactoryBean">  
 <!-- 给工厂注入一个新的类型转换器 -->  
 <property name="converters">  
 <array>  
 <!-- 配置自定义类型转换器 -->  
 <bean class="com.bozhao.util.StringToDateConvert"></bean>  
 </array>  
 </property>  
</bean>

第三步：在 annotation-driven 标签中引用配置的类型转换服务

<!--开启springMVC注解的支持-->  
<mvc:annotation-driven conversion-service="converterService"/>

## 遇到的问题★★

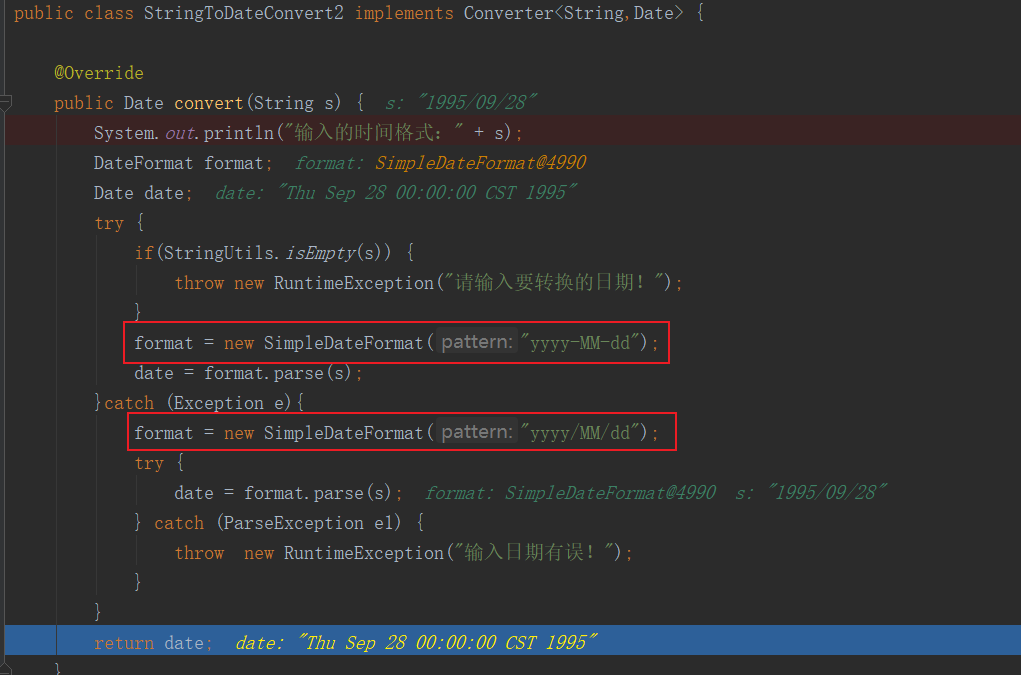
在这里写了个日期类型转换器，因为有可能输入多种时间格式，一开始想着可以多写几个转换器类，因为转换器配置中是arary，以为可以配置多个bean如下:



但是发现这样并不好使，这样，只有配置的后面一个Bean生效了，当输入其它格式的时候仍然不好使，

自定义类型转换器的执行时机，会在请求的方法执行之前先执行，如果转换器报错，请求的方法将不执行，浏览器会看到400bad request的报错信息，

此时，可以对转换器进行处理，让转换器能够匹配多种情况，最好不让转换器异常，导致请求方法无法执行，不能完成请求，如下：其实try，catch不单只有捕获异常，我们还可以用它来做其它情况的处理，比如这里当输入的格式不是第一种，必定出错，捕获异常后，我们就可以在catch里用另一种格式来解析，catch甚至可以有多层，就可以解决多种情况



## 使用 ServletAPI 对象作为方法参数

SpringMVC 还支持使用原始 ServletAPI 对象作为控制器方法的参数。支持原始 ServletAPI 对象有：

HttpServletRequest

HttpServletResponse

HttpSession

java.security.Principal

Locale

InputStream

OutputStream

Reader

Writer

我们可以把上述对象，直接写在控制的方法参数中使用。

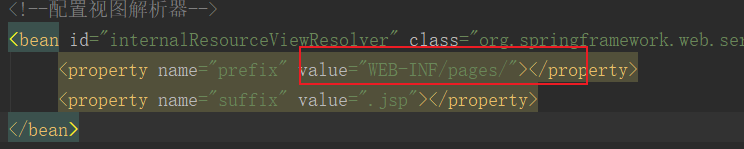
代码示例：

@Controller  
@RequestMapping("/param")  
public class ParameterController {  
 @RequestMapping("/servletAPI")  
 public String testServletAPI(HttpServletRequest request /\*既然想用servlet原生API就需要导入servlet相关依赖\*/) {  
 System.*out*.println("request:" + request);  
 //获取session  
 HttpSession session = request.getSession();  
 System.*out*.println("session:" + session);  
 //获取servletContext与域对象  
 ServletContext servletContext = session.getServletContext();  
 System.*out*.println("servletContext:" + servletContext);  
 return "success";  
 }  
}

# 遇到的问题★★★

这是一个困扰了我许久的问题，

1．在类上没有使用@RequestMapping注解的时候，我返回一个字符串，字符串为要跳转的试图名称，此时发现没配置试图解析器的时候，没有在WEB-INF前加斜杆，也能跳转成功，而且在跳转成功的试图上success.jsp点击超链接，也能跳转到另外一个试图，

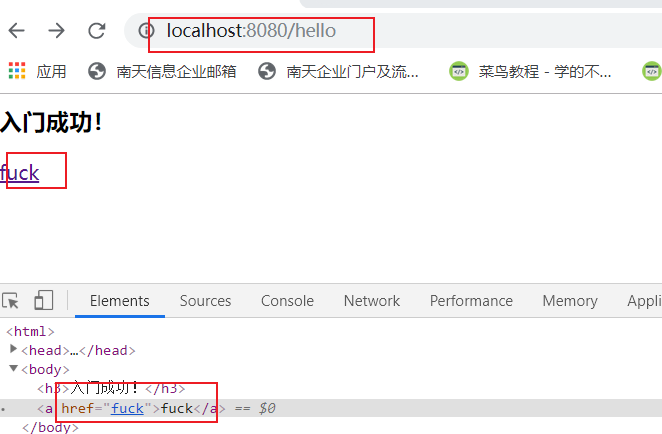


这是因为，类上没加requestMapping注解，项目一开始主页访问路径为<http://localhost:8080/hello>/index.jsp，其中index.jsp可以省略，最终路径为[http://localhost:8080/](http://localhost:8080/hello)，

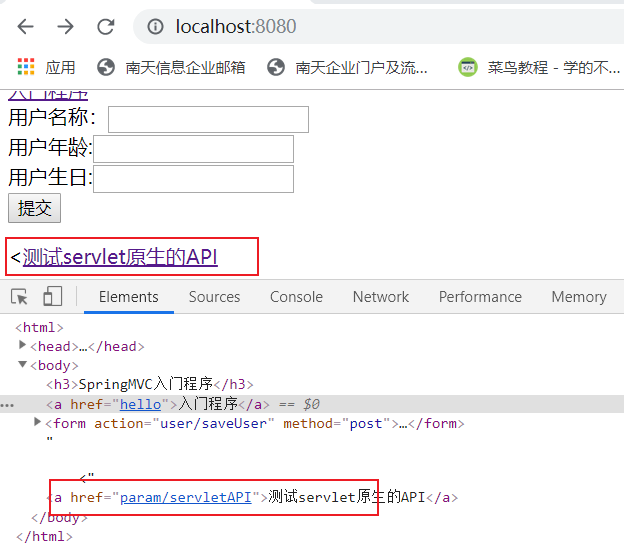
此时点击入门程序按钮，将访问<http://localhost:8080/hello>,能找到路径和响应的控制器，成功访问，也能返回视图，



返回视图success.jsp后，jsp中有一个超链接fuck，请求的路径是工程的相对路径，而工程的路径是[http://localhost:8080/](http://localhost:8080/hello),点击fuck时，请求路径为<http://localhost:8080/fuck>没有问题成功访问

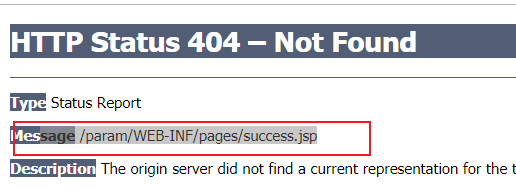


1. [当在类上使用了@RequestMapping](mailto:2.当在类上使用了@RequestMapping)注解后，比如测试原生servletAPI，



当点击测试servlet原生API超链接的时候，请求路径<http://localhost:8080/param/servletAPI>

看后台程序控制输出，可以看到，已经成功访问控制器，就是无法跳转到返回的视图字符串“success



这是因为视图解析器没正确配置，在视图解析器WEB-INF前加斜杆解决，成功跳转到返回的视图“success”

1. 此时问题又来了，success页面中有一个fuck的超链接，如果通过点击入门程序超链跳转到的success视图，在视图中再点击fuck超链接，能成功访问，并且成功跳转到返回的fuck视图，发现此时点击fuck超链接，访问路径为<http://localhost:8080/fuck>，是正确的当然能够正确访问，

但是，通过点击测试原生ServlcetAPI的超链接，进入到success视图后，再点击fuck超链接时，就会报错404，发现当点击fuck超链接时，此时的访问路径为：<http://localhost:8080/param/fuck>，这个路径当然找不到

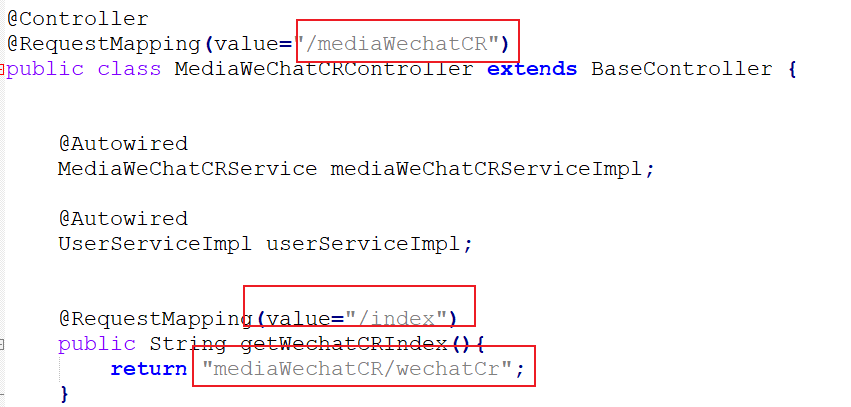
分析其原因：

是因为，入门程序的超链接的控制器中没有在类上使用@RequestMapping注解，此时路径就是一级路径，跳转到succeed视图后，路径为<http://localhost:8080/hello>，而此时，点击fuck超链接，herf属性是相对路径，所以将在当前请求<http://localhost:8080/>下，再加上href属性的相对路径，最终路径是<http://localhost:8080/fuck>这时，能成功访问，

而当我们通过点击测试servlet原生API的超链接，进入到success视图时，servlet原生API的控制器，类上使用了@requestMapping注解这时的访问路径为<http://localhost:8080/param/servletAPI>，此时再点击fuck超链接，herf属性是相对路径，所以将在当前请求<http://localhost:8080/> param下，再加上href属性的相对路径，最终路径是<http://localhost:8080/param/fuck，没有改路径下的控制器，自然报404>了

1. 此时，我再想，那么要是在入门程序的控制器中类上加上ReequestMapping注解，通过入门程序超链接进入到succees视图，是不是fuck超连接也不能访问呢?很明显这种猜测是错误的，在入门程序的控制器类上加上RequestMapping注解后，fuck超链接还是能访问，因为他们访问的是同一个控制器，工程路径相同，比如在类上加个RequestMapping路径为“/testHello”,最终只不过入门程序的路径变为<http://localhost:8080/testHello/hello>，fuck超链接的路劲变为<http://localhost:8080/testHello/fuck>,但是我我没有修改success视图中的fuck超链接的herf属性，仍为“fuck”,这也就说明，这个请求路径真的和请求从哪里过来是有关系的，也就是相对路径
2. 总结，像成功视图，失败视图，这样的视图一般都应该作为最终视图，里面不应该再有其它的操作，比如超链接啥的，因为这种视图的请求可能来自不同路径的请求，如果里面还有个超链接，超链接的属性herf肯定也是写死的，这样该超链接只有特定的请求才能找到控制器，其它都会找不到控制器而报404，

就像在优讯时代做开发的时候，系统左侧菜单每个按钮找到一个控制器，然后首先都会通过调用requestMapping注解为/index的方法返回一个视图，

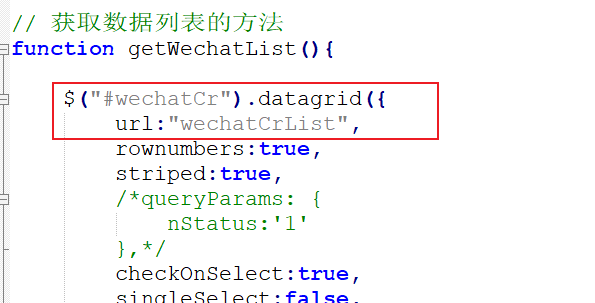


而后面该视图中的所有请求，找的都是该控制器的方法，而且用的都是相对路径，比如上图所示的控制器跳转到mediaWechatCR/wechatCr视图，路径为http://...... / mediaWechatCR/ index,以后在该视图中的所有操作请求，都请求该控制器，而且请求路径都为http://...... / mediaWechatCR/XXX,

比如： http://...... / mediaWechatCR/ wechatCrList,

http://...... / mediaWechatCR/addWechatCr

http://...... / mediaWechatCR/delWeChatCR



***总结就是一个视图对应一个控制器Controller***

# 响应数据和结果视图

## 返回值分类

### 返回字符串

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。

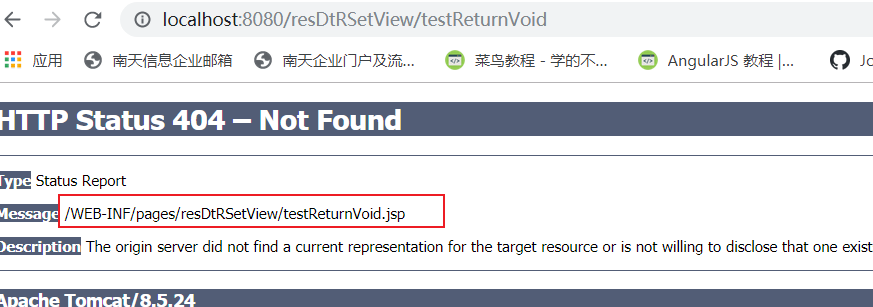
//指定逻辑视图名，经过视图解析器解析为 jsp 物理路径：/WEB-INF/pages/success.jsp

@RequestMapping("/testReturnString")  
public String testReturnString(Model model) {  
 logger.info("testReturnString 方法执行了。。。。");  
 //模拟从数据库中查询一个user对象  
 User user = new User();  
 user.setUsername("美美");  
 user.setPassword("529413");  
 user.setAge(18);  
 //将模拟查询出来的数据保存到Request域对象中，在页面就可以将它取出来  
 model.addAttribute("user", user);  
 return "success";  
}



### 返回值Void

@RequestMapping("/testReturnVoid")  
public void testReturnVoid() {  
 logger.info("testReturnVoid 方法执行了。。。");  
}



可以看到如上所示：当请求一个无返回值类型的控制器的方法的时候，当控制器的方法执行后，默认会访问请求路径的jsp文件，没有找到问价，就会报404

在前面的学习中，我们知道 Servlet 原始 API 可以作为控制器中方法的参数：所以可以在controller 方法形参上可以定义 request和 response，使用 request 或 response 指定响应结果：

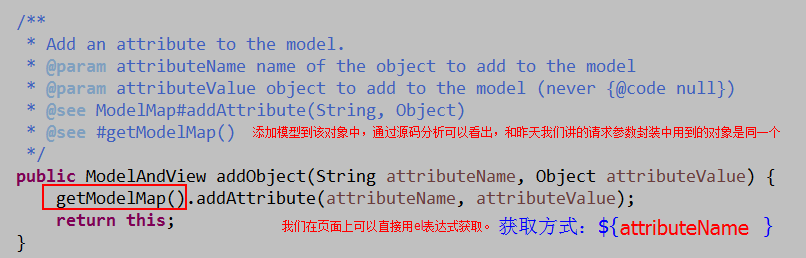
@RequestMapping("/testReturnVoid")  
public void testReturnVoid(HttpServletRequest request, HttpServletResponse response) throws Exception {  
 logger.info("testReturnVoid 方法执行了。。。");  
 //1 、使用 request 转向页面，如下：  
 request.getRequestDispatcher("/WEB-INF/pages/success.jsp").forward(request, response);  
 //2 、也可以通过 response 页面重定向：  
 response.sendRedirect("testReturnString");  
 //3 、也可以通过 response 指定响应结果，例如响应 json 数据：  
 response.setCharacterEncoding("utf-8");  
 response.setContentType("application/json;charset=utf-8");  
 response.getWriter().write("n json 串");  
}

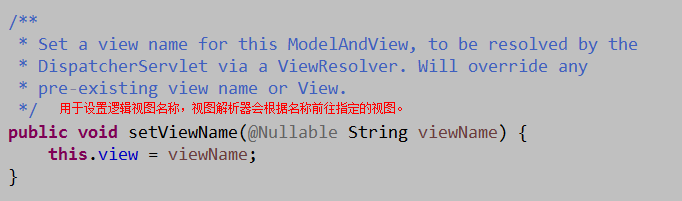
1. 请求转发使用request对象，请求转发是一次请求，不用编写项目名称，手动调用转发的方法，不会再去调用视图解析器，所以在调用getRequestDispatcher方法的时候需要自己提供完整的目录
2. 重定向使用response对象，重定向是重新发了一次新的请求，是不能直接反问到WEB-INF下的内容的，所以可以重定向到不在WEB-INF下的资源，或者其它控制的方法，如上就是重定向到了其它控制器，再有其它控制器来跳转页面，一般控制器里会写一个转门用来跳转页面的控制器方法
3. 使用response拿到writer输出流，直接进行数据响应，响应中文，需要解决中文响应乱码，可以通过setCharacterEncoding()方法设定编码集，还可以设置浏览器解析的编码，使用setContentType()方法

### 返回值ModelAndView

ModelAndView 是 SpringMVC 为我们提供的一个对象，该对象也可以用作控制器方法的返回值。

该对象中有两个方法：





代码示例：

<a href="testReturnModelAndView">测试返回值为ModelAndView类型练习</a>

@RequestMapping("/testReturnModelAndView")  
public ModelAndView testReturnModelAndView(Model model) {  
 logger.info("testReturnModelAndView 方法执行了。。。。");  
 //创建一个ModelAndView对象  
 ModelAndView modelAndView = new ModelAndView();  
 //模拟从数据库中查询一个user对象  
 User user = new User();  
 user.setUsername("ModelAndViewReturning");  
 user.setPassword("529413");  
 user.setAge(18);  
 //把user对象保存到ModelAndView对象中，也会将user对象保存到Request域对象中  
 //ModelAndView底层用的也是ModelMap，实现了Model的接口  
 modelAndView.addObject("user", user); //modelAndView,这就是model  
 modelAndView.setViewName("success"); //modelAndView ，这就是view，modelAndView会选择视图解析器中的配置，来完成页面跳转  
 return modelAndView;  
}

## 转发和重定向

### forward转发

1. controller方法在提供了 String 类型的返回值之后，默认就是请求转发，但是会选择视图解析器，通过我们配置的视图解析器来找到逻辑视图。
2. 我们也可以写成：所以当返回值类型为String的时候，也可以这样来写：return "forward:/WEB-INF/pages/success.jsp";需要注意的是，如果用了 formward ：则路径必须写成实际视图 url，不能写逻辑视图。

它相当于“request.getRequestDispatcher("url").forward(request,response)”。使用请求转发，既可以转发到 jsp，也可以转发到其他的控制器方法

### Redirect重定向

1. contrller 方法提供了一个 String 类型返回值之后，它需要在返回值里使用:redirect: return "redirect:testReturnModelAndView";
2. 它相当于“response.sendRedirect(url)”。需要注意的是，如果是重定向到 jsp 页面，则 jsp 页面不能写在 WEB-INF 目录中，否则无法找到。所以也可以选择重定向到其它控制器方法
3. @RequestMapping("/testForwardOrRedirect")  
   public String testForwardOrRedirect(Integer id,Model model) {  
    logger.info("testForwardOrRedirect 方法执行了。。。");  
    User user = new User();  
    if(id == 1) {  
    user.setUsername("Forward请求");  
    model.addAttribute("user", user);  
    return "forward:/WEB-INF/pages/success.jsp";  
    }  
    user.setUsername("redirect请求");  
    model.addAttribute("user", user);  
    return "redirect:/toSuccess";  
   }

***说明：forward请求，请求转发，是同一次请求，请求参数不会丢失，比如在这里我在Request域中保存了一个user对象，转发到success页面后，能够取到user对象，但是redirect请求重定向，是重新发了一个请求，原请求参数丢失，在重定向前我同样在request域中保存了一个user对像，但是重定向后，将无法从request域中获取到之前保存的user对象了***

## ResponseBody响应Json数据

作用：

该注解用于将 Controller 的方法返回的对象，通过 HttpMessageConverter 接口转换为指定格式的数据如：json,xml 等，通过 Response 响应给客户端

使用示例：

需 求：

使用@ResponseBody 注解实现将 controller 方法返回对象转换为 json 响应给客户端。

前置知识点：

Springmvc 默认用 MappingJacksonHttpMessageConverter 对 json 数据进行转换，需要加入jackson 的包。

***注意：该注解可以使用在方法上，也可以使用在方法返回值前***

### 释放静态资源配置：

1.使用springMVC的，<mvc:resources 进行配置，

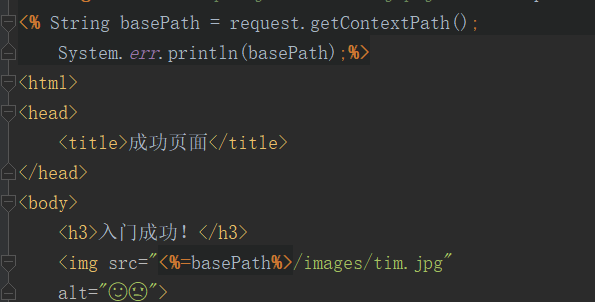
<!--前端控制器哪些静态资源不拦截，释放静态资源-->  
<mvc:resources mapping="/css/\*\*" location="/css/"/>  
<mvc:resources mapping="/js/\*\*" location="/js/"/>  
<mvc:resources mapping="/images/\*\*" location="/images/"/>

2.也可以编写自定义的拦截器来进行过滤

public class CommonInterceptor extends HandlerInterceptorAdapter {  
 private List<String> excludedUrls;  
 public List<String> getExcludedUrls() {  
 return excludedUrls;  
 }  
 public void setExcludedUrls(List<String> excludedUrls) {  
 this.excludedUrls = excludedUrls;  
 }  
 public List<String> getUrlList() {  
 return excludedUrls;  
 }  
  
 // 重写 preHandle()方法，在业务处理器处理请求之前对该请求进行拦截处理 1  
 public boolean preHandle(HttpServletRequest request,



3.觉得./或者../这种写路径的方式不太好确定的时候，可以使用如下方式来写路径，也很好使



### 遇到的问题：★★



看视频的时候，讲师直接如上书写，他的可以正常访问，没问题，我的就不行，问题在于，讲师的页面是在webapp目录下，所以直接写，访问到了，而我自己写的是先从webapp下的index页面，跳转到<http://localhost:8080/resDtRSetView/index>，最终是在WEB-INF下的

resDtRSetView.jsp页面中，这时在如上的引入js路径，真正请求的时候路径为

GET http://localhost:8080/resDtRSetView/js/jquery.min.js 404，

这时比较好的解决方案就是不要这样写，还是用到上面的通过request域对象，出现这种问题的原因就是因为相对路径造成的

jsp中的request.getContextPath()<%=request.getContextPath()%>是为了解决相对路径的问题，可返回站点的根路径。

**<%** String basePath = request.getContextPath();**%>**

<script src="**<%=**basePath**%>**/js/jquery.min.js"></script>

### 代码示例

<button id="btn">提交Ajax请求</button>

<script>  
 //页面加载，绑定一个点击事件  
 $(**function** () {  
 $("#btn").click(**function** () {  
 //发送Ajax请求  
 $.ajax({  
 //编写json格式，设置属性和值  
 url:"testAjax",  
 contentType:"application/json;charset=UTF-8",  
 data:'{"id":"2","username":"张慧","password":"529413","age":"18","birthday":"1995-09-28"}',  
 dataType:"json",  
 type:"post",  
 success:**function** (data, textStatus) {  
 //data为服务器端响应的json数据。进行解析  
 alert(data.username);  
 alert(data.password);  
 }  
 })  
 });  
 });  
</script>

@RequestMapping("/testAjax")  
@ResponseBody  
public User testAjax(@RequestBody User user) {  
 logger.info("testAjax 方法执行了。。。body：" + user);  
 //客户端发送ajax请求，传的是json字符串，后端把json字符串封装到了user对象中，  
 System.*out*.println(user);  
 //做响应，模拟查询数据库  
 user.setUsername("前女友");  
 user.setAge(188);  
 //响应  
 return user;  
}

# 遇到的问题：★★★★

在jsp中想使用C标签，

<%@ **taglib** uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

**<%** String basePath = request.getContextPath();**%>**

在jsp中就能编写代码，需要引入jstl相关依赖

<!-- JSTL -->  
<dependency>  
 <groupId>javax.servlet</groupId>  
 <artifactId>jstl</artifactId>  
 <version>1.2</version>  
</dependency>  
<!-- JSTL实现包 -->  
<dependency>  
 <groupId>org.apache.taglibs</groupId>  
 <artifactId>taglibs-standard-impl</artifactId>  
 <version>1.2.5</version>  
</dependency>

# SpringMVC实现文件上传

## 文件上传的必要前提（三要素）

1. form 表单的 enctype 取值必须是：multipart/form-data(默认值是:application/x-www-form-urlencoded)，enctype:是表单请求正文的类型
2. method 属性取值必须是 Post
3. 提供一个文件选择域<input type=”file” />

## 文件上传的原理分析

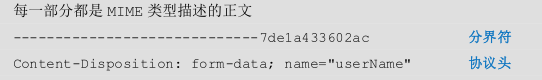
当 form 表单的 enctype 取值不是默认值后，request.getParameter()将失效。

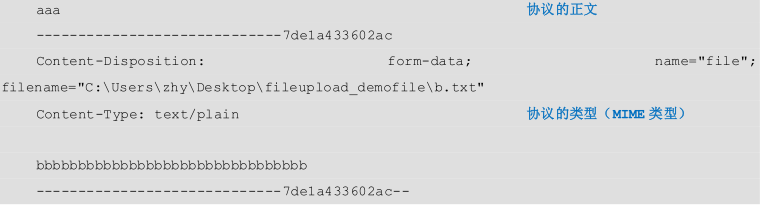
enctype=”application/x-www-form-urlencoded”时，form 表单的正文内容是：

key=value&key=value&key=value

当 form 表单的 enctype 取值为 Mutilpart/form-data 时，请求正文内容就变成：

每一部分都是 MIME 类型描述的正文，使用分隔符的方式把文件分成了若干部分，以二进制字节流进行提交





## 借助第三方组件 实现文件上传

使用 Commons-fileupload 组件实现文件上传，需要导入该组件相应的支撑 jar 包：Commons-fileupload 和commons-io。commons-io 不属于文件上传组件的开发 jar 文件，但Commons-fileupload 组件从 1.1 版本开始，它工作时需要 commons-io 包的支持。



## 不使用springMVC框架传统文件上传

使用组件commons-fileupload、commons-io实现文件上传

### 一、引入commons相关依赖

<!--文件上传相关依赖-->、  
<dependency>  
 <groupId>commons-fileupload</groupId>  
 <artifactId>commons-fileupload</artifactId>  
 <version>1.3.1</version>  
</dependency>  
<!-- https://mvnrepository.com/artifact/commons-io/commons-io -->  
<dependency>  
 <groupId>commons-io</groupId>  
 <artifactId>commons-io</artifactId>  
 <version>2.4</version>  
</dependency>

### 二、编写jsp

<**h3**>使用commons-fileupload组件实现文件上传，不使用SpringMVC框架</**h3**>  
<**form enctype="multipart/form-data" method="post" action="commons"**>  
 <**input type="file" name="commons" value="浏览..."**><**br**/><**br**/>  
 <**input type="submit" value="上传文件"**>  
</**form**>

### 三、编写文件上传控制器

@RequestMapping(**"/commons"**)  
**public** String fileLoadUp(HttpServletRequest request) **throws** Exception {  
  
 *//使用fileupload组件实文件上传  
 //上传的位置：这个路径会将文件上传到项目部署的根路径下* String path = request.getSession().getServletContext().getRealPath(**"/uploads/"**);  
 *//判断路径是否存在* File file = **new** File(path);  
 **if** (!file.exists()) {  
 *//如果路径不存在，创建路径文件夹* file.mkdirs();  
 }  
 *//设置临时目录。 上传文件大于缓冲区则先放于临时目录中  
 // String tempPath = "D:\\tempPath";  
  
 //判断临时目录是否存在（不存在则创建）  
 /\*File f1 = new File(tempPath);  
 if(!f1.isDirectory()){  
 System.out.println("临时文件目录不存在! 创建临时文件目录");  
 f1.mkdir();  
 }\*/  
 //解析request对象，获取上传文件项:这是Commons-fileupload组件中的API* DiskFileItemFactory factory = **new** DiskFileItemFactory();  
 *//设置上传文件的临时目录  
 //factory.setRepository(f1);* ServletFileUpload upload = **new** ServletFileUpload(factory);  
 *//解决上传"文件名"的中文乱码* upload.setHeaderEncoding(**"UTF-8"**);  
 *//3、判断enctype:判断提交上来的数据是否是上传表单的数据  
 /\* if(!ServletFileUpload.isMultipartContent(request)){  
 System.out.println("不是上传文件，终止");  
 //按照传统方式获取数据  
 return "";  
 }\*/  
 //解析request* List<FileItem> fileItems = upload.parseRequest(request);  
 *//遍历* **for** (FileItem item: fileItems) {  
 *//进行判断，当前item对象是否是上传文件项* **if** (item.isFormField()) {  
 *//说明普通表单项* }**else** {  
 *//说明是上传文件项  
 //获取上传文件的名称* String fileName = item.getName();  
 *//完成上传文件* **if** (*logger*.isInfoEnabled()) {  
 *logger*.info(**"文件上传，fileName:{},path:{}"**,fileName,path);  
 }  
 item.write(**new** File(path,fileName));  
 *//删除临时文件* item.delete();  
 }  
 }  
 **return "success"**;  
}

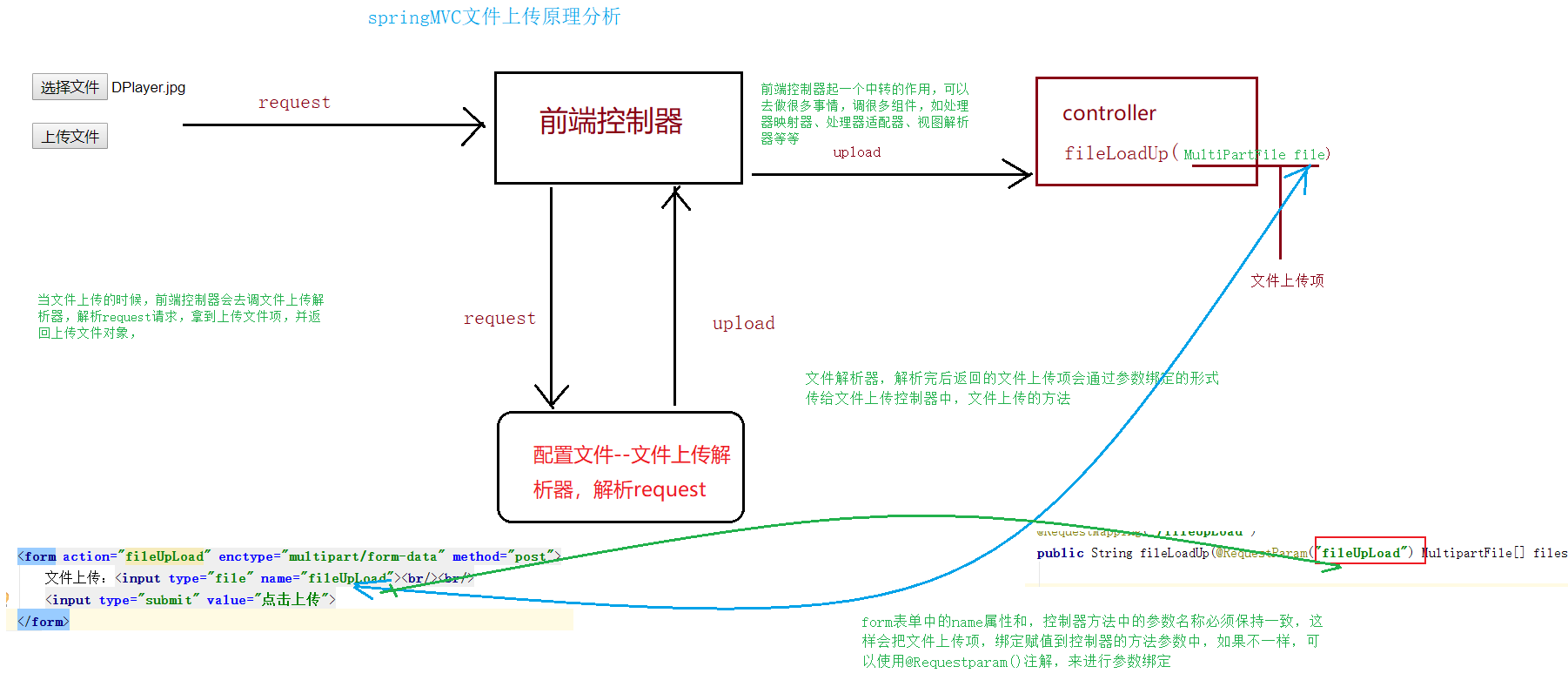
### 遇到的问题：★

使用ServletFileUpload的parseRequest()方法，解析不到request中的数据，获取到的List为空，这是因为这里不使用springMVC方式上传文件，使用commons组件中的API来自己解析数据，但是在sprigMVC的配置文件中，仍然配置了springMVC的文件上传，解决方法，去掉springMVC配置文件中关于支持文件上传的配置，即可解决解析不到数据的问题，详见下面博客文章

<https://www.iteye.com/blog/zzc1684-2258463>

## 使用springmvc传统方式文件上传

### SpringMVC框架文件上传原理分析



### 引入文件上传相关依赖

<!--文件上传相关依赖-->、  
<dependency>  
 <groupId>commons-fileupload</groupId>  
 <artifactId>commons-fileupload</artifactId>  
 <version>1.3.1</version>  
</dependency>  
<!-- https://mvnrepository.com/artifact/commons-io/commons-io -->  
<dependency>  
 <groupId>commons-io</groupId>  
 <artifactId>commons-io</artifactId>  
 <version>2.4</version>  
</dependency>

### 编写 jsp 页面

<**h3**>使用springMVC框架传统方式实现文件上传</**h3**>  
<**form action="fileUpLoad" enctype="multipart/form-data" method="post"**>  
 文件上传：<**input type="file" name="fileUpLoad"**><**br**/><**br**/>  
 <**input type="submit" value="点击上传"**>  
</**form**>

### 编写文件上传控制器

*/\*\*  
 \* 使用springMVC框架传统的文件上传  
 \** ***@param files*** *\** ***@param request*** *\** ***@return*** *\*/*@RequestMapping(**"/fileUpLoad"**)  
**public** String fileLoadUp(@RequestParam(**"fileUpLoad"**) MultipartFile[] files,HttpServletRequest request,String picName){  
  
 *//定义文件名* String fileName = **""**;  
 *//获取文件路径:使用这种路径，文件最终被上传到了项目部署的根路径下的uploads文件夹中  
 //可以看到要想拿到ServletContext域对象，首先获取到session，从session中获取ServletContext域对象* String basePath = request.getSession().getServletContext().getRealPath(**"/uploads"**);  
 *//使用如下路径：文件最终被上传到了D:\uploads\2020-05-17文件夹中，我也不知道为啥子  
 //String basePath = request.getContextPath();  
 //使用如下路径：文件最终被上传到D:\file\fileUpLoad\uploads\2020-05-17文件夹中，可以看到和请求路径好像有关系  
 //String basePath = request.getServletPath();  
 //3.解决同一文件夹中文件过多问题* String datePath = **new** SimpleDateFormat(**"yyyy-MM-dd"**).format(**new** Date());  
 *//判断路径是否存在* File file = **new** File(basePath+**"/"** + datePath);  
 *//File file = new File(basePath+"/uploads/" + datePath);* **if** (!file.exists()) {  
 file.mkdirs();  
 }  
  
 **for** (MultipartFile multipartFile:files) {  
 *//获取原始文件名* String originalFilename = multipartFile.getOriginalFilename();  
 *//截取文件扩展名* String extendName = originalFilename.substring(originalFilename.lastIndexOf(**"."**) + 1,originalFilename.length());  
 *//把文件加上随机数，防止文件重复* String uuid = UUID.*randomUUID*().toString().replace(**"-"**, **""**).toUpperCase();  
 *//最终上传文件的文件名* **if** (!StringUtils.*isEmpty*(picName)) {  
 *//判断是否输入了文件名* fileName = uuid + **"\_"** + picName + **"\_"** + extendName;  
 }**else** {  
 fileName = uuid + **"\_"** + originalFilename;  
 }  
 **if**(*logger*.isInfoEnabled()) {  
 *logger*.info(**"开始上传文件：{}"**,fileName);  
 }  
 **try** {  
 multipartFile.transferTo(**new** File(file,fileName));  
 } **catch** (IOException e) {  
 **if** (*logger*.isErrorEnabled()) {  
 *logger*.error(**"文件上传时出现异常，文件：{},error:{}"**,originalFilename,e.getMessage());  
 }  
 }  
 }  
  
 **if** (*logger*.isInfoEnabled()) {  
 *logger*.info(**"文件上传成功：{}"**,basePath + **"/"** + datePath + **"/"** + fileName);  
 }  
 **return "success"**;  
}

### 配置文件上传解析器

*<!--支持文件上传，这里的id必须是multipartResolver-->*<**bean id="multipartResolver"  
 class="org.springframework.web.multipart.commons.CommonsMultipartResolver"**>  
 <**property name="defaultEncoding" value="UTF-8"**/>  
 <**property name="maxUploadSize" value="10240000"** />  
</**bean**>

## 使用springMVC框架实现跨服务器文件上传

### 分服务器的目的

在实际开发中，我们会有很多处理不同功能的服务器。

例如：

应用服务器：负责部署我们的应用

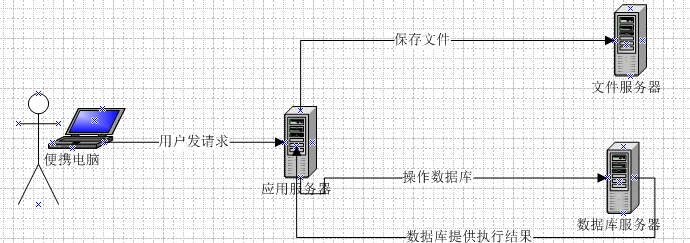
数据库服务器：运行我们的数据库

缓存和消息服务器：负责处理大并发访问的缓存和消息

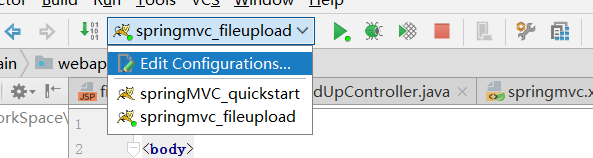
文件服务器：负责存储用户上传文件的服务器。

( 注意：此处说的不是服务器集群）

分服务器处理的目的是让服务器各司其职，从而提高我们项目 的运行效率。



### 一、准备个 两个 tomcat 服务器的 ，并创建一个用于存放图片的 web



### 二、编写jsp

<**h3**>使用springMVC框架跨服务器方式实现文件上传</**h3**>  
<**form action="acrossServer" enctype="multipart/form-data" method="post"**>  
 文件上传：<**input type="file" name="acrossServerFileUpLoad"**><**br**/><**br**/>  
 <**input type="submit" value="点击上传"**>  
</**form**>

### 三、导入支持跨服务器文件上传的相关依赖

*<!-- https://mvnrepository.com/artifact/com.sun.jersey/jersey-client  
支持跨服务器文件上传-->*<**dependency**>  
 <**groupId**>com.sun.jersey</**groupId**>  
 <**artifactId**>jersey-client</**artifactId**>  
 <**version**>1.18.1</**version**>  
</**dependency**>  
*<!-- https://mvnrepository.com/artifact/com.sun.jersey/jersey-core -->*<**dependency**>  
 <**groupId**>com.sun.jersey</**groupId**>  
 <**artifactId**>jersey-core</**artifactId**>  
 <**version**>1.18.1</**version**>  
</**dependency**>

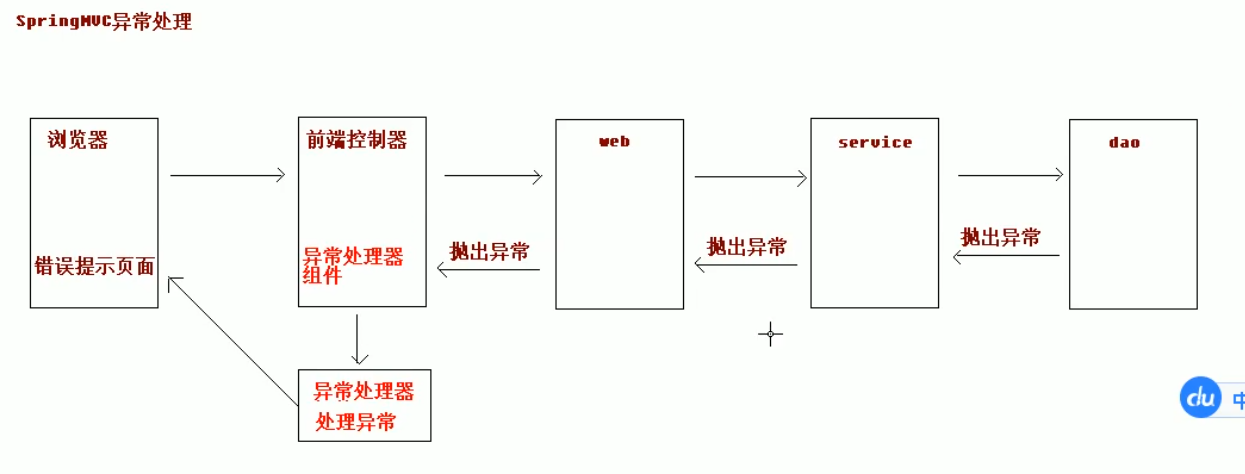
### 四、编写文件上传控制器

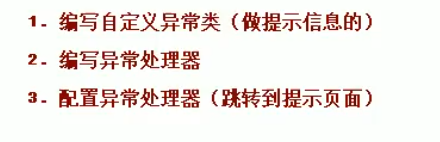
*/\*\*  
 \* 使用springMVC框架跨服务器文件上传  
 \** ***@param files*** *\** ***@param picName*** *\** ***@return*** *\*/*@RequestMapping(**"/acrossServer"**)  
**public** String acrossServerFileUpLoad(@RequestParam(**"acrossServerFileUpLoad"**) MultipartFile[] files,String picName) {  
  
 String fileName = **""**;  
 String serverPath = **"http://localhost:9090/uploads/"**;  
  
 **for** (MultipartFile multipartFile:files) {  
 *//获取原始文件名* String originalFilename = multipartFile.getOriginalFilename();  
 *//截取文件扩展名* String extendName = originalFilename.substring(originalFilename.lastIndexOf(**"."**) + 1,originalFilename.length());  
 *//把文件加上随机数，防止文件重复* String uuid = UUID.*randomUUID*().toString().replace(**"-"**, **""**).toUpperCase();  
 *//最终上传文件的文件名* **if** (!StringUtils.*isEmpty*(picName)) {  
 *//判断是否输入了文件名* fileName = uuid + **"\_"** + picName + **"\_"** + extendName;  
 }**else** {  
 fileName = uuid + **"\_"** + originalFilename;  
 }  
 **if**(*logger*.isInfoEnabled()) {  
 *logger*.info(**"开始上传文件：{}"**,fileName);  
 }  
 **try** {  
 *//跨服务器文件上传  
 //创建客户端对象* Client client = Client.*create*();  
  
 *//和图片服务器进行连接* WebResource webResource = client.resource(serverPath + fileName);  
  
 *//上传文件* webResource.put(multipartFile.getBytes());  
 **if** (*logger*.isInfoEnabled()) {  
 *logger*.info(**"文件上传成功：{}"**,serverPath + fileName);  
 }  
  
 } **catch** (Exception e) {  
 **if** (*logger*.isErrorEnabled()) {  
 *logger*.error(**"文件上传时出现异常，文件：{},error:{}"**,originalFilename,e.getMessage());  
 }  
 **return "error"**;  
 }  
 }  
 **return "success"**;  
}

### 配置文件上传解析器

*<!--支持文件上传，这里的id必须是multipartResolver-->*<**bean id="multipartResolver"  
 class="org.springframework.web.multipart.commons.CommonsMultipartResolver"**>  
 <**property name="defaultEncoding" value="UTF-8"**/>  
 <**property name="maxUploadSize" value="10240000"** />  
</**bean**>

# SpringMVC的异常处理





### 一、编写自定义异常类

**public class** SysException **extends** Exception{  
 **private** String **message**;  
  
 **public** SysException(String message) {  
 **this**.**message** = message;  
 }  
  
 **public** String getMessage() {  
 **return message**;  
 }  
  
 **public void** setMessage(String message) {  
 **this**.**message** = message;  
 }  
}

### 二、编写异常处理器

**public class** SysExceptionResolver **implements** HandlerExceptionResolver{  
  
 */\*\*  
 \* 处理异常的业务逻辑  
 \** ***@param request*** *\** ***@param response*** *\** ***@param handler*** *\** ***@param ex*** *\** ***@return*** *\*/* @Override  
 **public** ModelAndView resolveException(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) {  
  
 *//获取到异常对象* SysException var1 = **null**;  
 **if** (ex **instanceof** SysException) {  
 var1 = (SysException) ex;  
 }**else** {  
 var1 = **new** SysException(**"系统正在维护!"**);  
 }  
  
 *//创建ModelAndView* ModelAndView var2 = **new** ModelAndView();  
 var2.addObject(**"errorMsg"**,var1. getMessage());  
 var2.setViewName(**"error"**);  
  
 **return** var2;  
 }  
}

### 三、配置异常处理器（跳转到提示页面）

*<!--配置自定义异常处理器-->*<**bean id="sysExceptionResolver" class="com.bozhao.exception.handle.SysExceptionResolver"**></**bean**>

### 四、提供错误页面

<**html**>  
<**head**>  
 <**title**>错误信息</**title**>  
</**head**>  
<**body**>  
  
 <**div**>  
 <**img src="<%=**basePath**%>/images/error.jpg"  
 alt="🙂😩"**><**br**/>  
 <**span**>error:**${**errorMsg**}**</**span**>  
 </**div**>  
</**body**>  
</**html**>

# SpringMVC中的拦截器