# 前言

在学习SpringMVC前，建议先配置Tomcat热部署，与PHP不同，java每次修改代码，都需要重启Tomcat，重新部署项目才能生效，而热部署技术帮助我们省去这一麻烦，但是需要注意的是**修改Spring配置文件一定要重启的，tomcat热部署只支持java方法内的代码变更**，IDEA关于Tomcat热部署的方法请自行百度

# 概述

## 1.1什么是MVC

先来看看什么是MVC

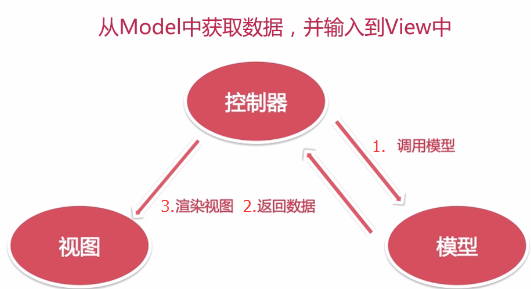
MVC是模型(model)－视图(view)－控制器(controller)的缩写，是一种表现层的框架(传统的**三层架构可分为表现层、中间层和数据层**，MVC正是表现层的框架)

**model：**程序中传递数据的部分，是数据的载体，直接操作数据库的代码应集成到其中；

**view：**程序中负责显示数据的部分，如渲染页面等；

**controller：**是处理视图层与用户交互的部分，对输入和输出做出反应，将http请求的参数交付给下一层进行处理，获取到处理结果的数据后将数据转发给view进行页面渲染(在当下很流行的前后端分离的项目中，渲染页面的工作是交付前端框架进行的，因此前后端分离的项目Controller只需将获取到的数据通过网络传输给客户端，由前端框架在客户端进行渲染)

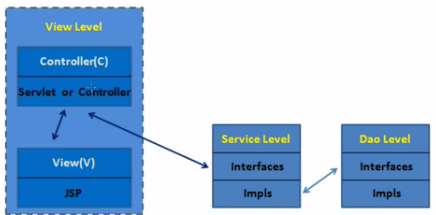
**注：渲染是指js效果，css样式等等显示成页面的过程，web服务器端进行的渲染(只是数据上的渲染)一般是将数据通过模板以占位符的形式将数据填充进页面，随着前后端分离技术的出现，这部分渲染的过程被从服务器后端拿到了前端页面进行**



**整个流程是：**

1.url绑定控制器，访问一个url就是调用一个控制器

2.控制器将本次访问携带的参数传递给下一层。如果业务不复杂，可能直接访问数据库就行了；如果业务复杂，model层可能还会分成三层架构中的业务层(Service层)和持久(Dao层)，这样一来，view和controller就组成了视图层(View层)，model就分成了业务层(Service层)和持久(Dao层)。



3.下一层向controller返回处理完成的数据，controller调起view进行页面渲染

## 1.2 SpringMVC简介

SpringMVC就是一款非常流行的、基于Java的轻量级MVC框架。

轻量级是一个相对的概念，主要是对应用框架使用方便性和所提供服务特性等方面做比较的。简单地来说：**轻量级一般是指的换掉这个框架对项目影响不大，对项目的侵入性较小，重量级一般是指如果你要换掉这个框架，你整个项目的改动将会非常之大，并且重量级集成的功能也非常之多**，现在流行的趋势是尽量使用轻量级的框架，避免使用重量级的框架

在spring3.0以后，springMVC已经超越struts2成为java web中最优秀的MVC框架，当然，还是有很多老项目使用ssh构件的，因此对struts2依然有很大的需求

## 1.3 SpringMVC环境搭建

### 1.3.1.手动搭建(不推荐)

(1)下载jar包

打开链接<http://repo.spring.io/simple/libs-release-local/org/springframework/spring/>

选择版本即可下载

(2)自定义Library

选择window->preferences->搜索User Libraries->new一个自己的library->将jar包导入新建的library中->通过add external jars添加jar包

(3)导入自定义的Library

右键项目，build path，add user library，至此，JAR包导入完成

### 1.3.2.maven自动构建(推荐)

maven具体使用方式请参照《maven学习笔记》

1.首先创建一个maven项目，创建过程参见maven笔记

2.打开pom.xml

在project父节点下添加：

<properties>

<springVersion>5.0.2.RELEASE</springVersion>

<comlangVersion>3.1</comlangVersion>

<slf4jVersion>1.7.25</slf4jVersion>

</properties>

然后在project节点的<dependencies>下添加：

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>javax</groupId>

<artifactId>javaee-api</artifactId>

<version>7.0</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>${springVersion}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${springVersion}</version>

<type>jar</type>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-beans</artifactId>

<version>${springVersion}</version>

<type>jar</type>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

<version>${springVersion}</version>

</dependency>

<dependency>  
 <groupId>javax.servlet</groupId>  
 <artifactId>jstl</artifactId>  
 <version>1.2</version>  
 <scope>runtime</scope>  
</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${springVersion}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-expression</artifactId>

<version>${springVersion}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jcl</artifactId>

<version>${springVersion}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>${springVersion}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jms</artifactId>

<version>${springVersion}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-messaging</artifactId>

<version>${springVersion}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-orm</artifactId>

<version>${springVersion}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-oxm</artifactId>

<version>${springVersion}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

<version>${springVersion}</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context-support</artifactId>

<version>${springVersion}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-instrument</artifactId>

<version>${springVersion}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aspects</artifactId>

<version>${springVersion}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-tx</artifactId>

<version>${springVersion}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webflux</artifactId>

<version>${springVersion}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-websocket</artifactId>

<version>${springVersion}</version>

</dependency>

<dependency>

<groupId>org.apache.commons</groupId>

<artifactId>commons-lang3</artifactId>

<version>${comlangVersion}</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>${slf4jVersion}</version>

</dependency>

<dependency>

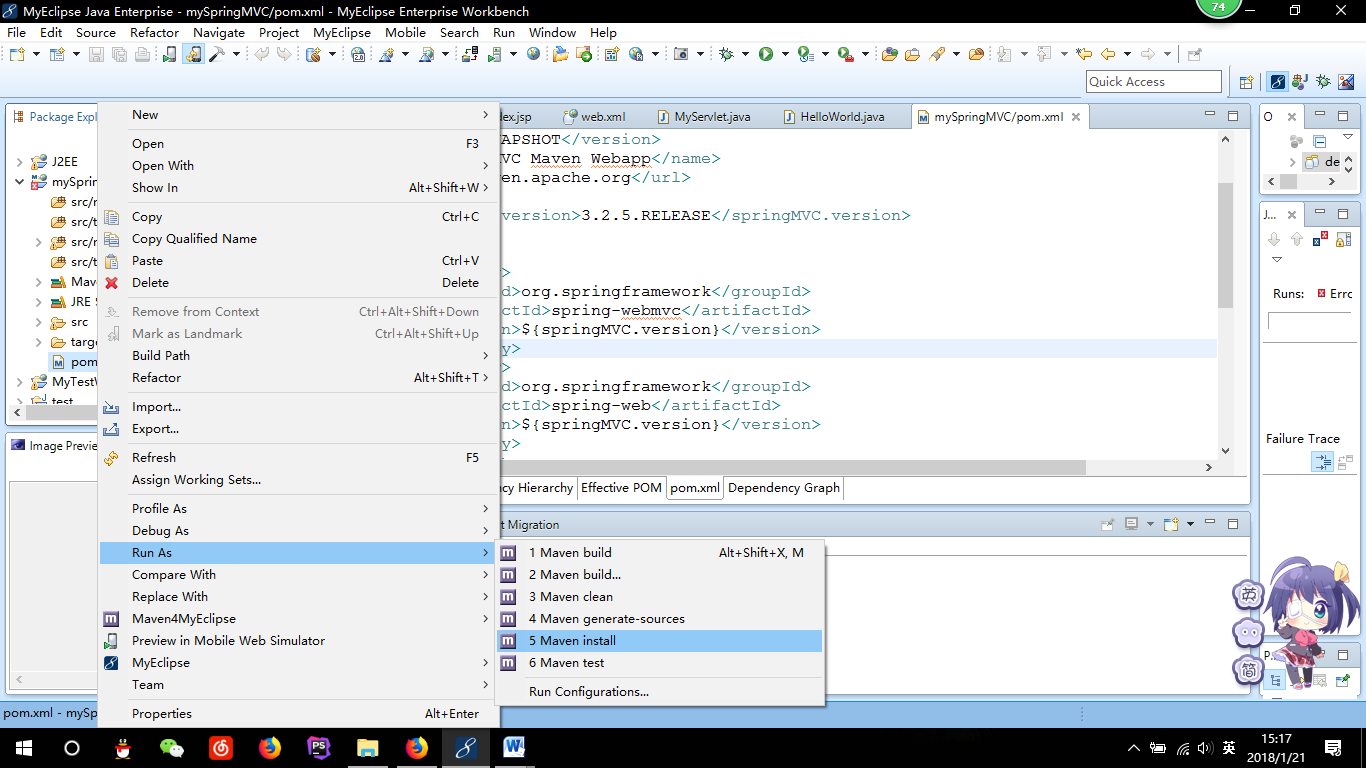
<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

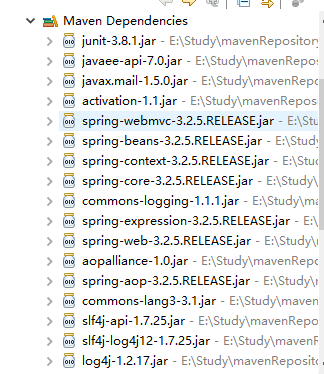
<version>${slf4jVersion}</version>

</dependency>

然后保存，运行pom.xml文件下载响应依赖(注意：占位符中不要出现空格)



这时打开我们的maven依赖库，即可看到对应的依赖包



# SpringMVC的相关概念

## 2.1 前端控制器DispatcherServlet

DispatcherServlet是一个极为重要的servlet，它的主要职责有：

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。

简单地来说，DispatcherServlet就是MVC框架中常说的”路由”，它的原理是**通过spring框架中的前置控制器拦截http请求，把拦截下来的请求，依据相应的规则分发到目标Controller来处理，**这样就实现了MVC框架中路由分发的功能。同时，它还负责渲染视图等功能

打个比方吧，一个病人**(http请求)**要去医院看病，那么他该去哪个科室**(控制器)**看病呢？我们并不清楚，哪个科室能给我返回我**(http请求)** 需要的药方**(要渲染进视图的数据，下图第5,6步中的Model)**？不知道没关系，我们先去问问医院前台**(DispatcherServlet)**我**(http请求)**该去哪个科室**(控制器)**，拿到药方**(要渲染的数据)**后，由前台**(DispatcherServlet)**将药方**(model)**传递到药房**(MVC中的view)**开药**(渲染视图)**



**上图的前端控制器就是DispatcherServlet**

那么，在项目中我们如何配置DispatcherServlet呢？

1.在WEB-INFO文件夹下新建web.xml文件

<web-app>

<servlet>

<servlet-name>example</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

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

</servlet>

<servlet-mapping>

<servlet-name>example</servlet-name>

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

</servlet-mapping>

</web-app>

**注意：**

**1.** **<servlet-name>example</servlet-name>**

**这个Servlet的名字是example，可以有多个DispatcherServlet，是通过名字来区分的。每一个DispatcherServlet有自己的WebApplicationContext上下文对象。同时保存在ServletContext中和Request对象中**

**2.<load-on-startup>1</load-on-startup>**

**启动顺序，让这个Servlet随Servletp容器一起启动**

**3.**  **<url-pattern>\*.form</url-pattern>**

**拦截\*.form结尾的请求。**

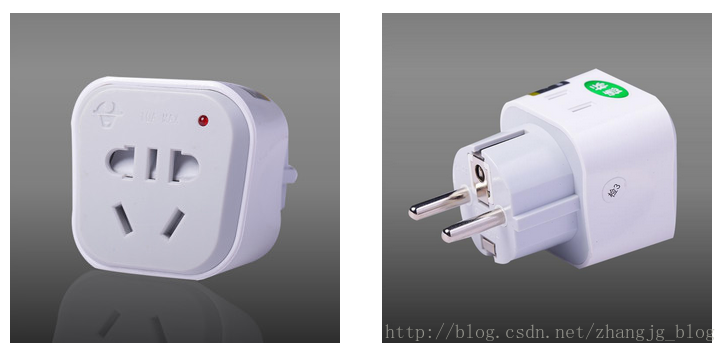
## 2.2 HandleMapping路由映射

前面我们说到，DispatcherServlet是”医院的前台”，它将http请求分发到各个控制器进行处理，那么它是如何知道要将http请求交给哪个对应的控制器呢？这就需要咨询一个Bean——这个Bean的名字为“HandlerMapping”。 handlerMapping是把一个URL指定到一个Controller上，就像应用系统的web.xml文件使用<servlet-mapping>将URL映射到servlet一样，这样一来，访问该URL的http请求都会被路由导向该controller

## 2.3 设计模式之适配器模式adapter

适配器模式是23中设计模式之一，它的主要作用是在新接口和老接口之间进行适配。它非常像我们出国旅行时带的电源转换器。我们国家的电器使用普通的扁平两项或三项插头，而去外国的话，使用的标准就不一样了，比如德国，使用的是德国标准，是两项圆头的插头。如果去德国旅游，就意味着我们无法正常充电，那么如何解决呢？只要使用一个电源转化器就行了，这个电源转换器就是我们的适配器

**适配器模式：就是将一个类的接口适配成用户所期待的样子，主要解决接口兼容问题**



比如，我们有一个德国的插座和中国的充电器，德国的插座使用德标(上右图)，中国的充电器使用国标三角插座(上左图)，那么如何让中国的充电器在德国的插座上充电呢？

**interface** IGermanySocket{

//在德国插座充电

**public** **void** powerInGermanySocket();

}

**interface** IChinaSocket{

//在中国插座充电

**public** **void** powerInChinaSocket();

}

//中国充电器，必须能够在中国插座上充电，因此实现国标

**class** ChinaCharger **implements** IChinaSocket{

**public** **void** powerInChinaSocket() {

System.*out*.println("在中国插座上充电，可以充进去");

}

}

//中国到德国的适配器,必须能在德国插座上充电，因此必须实现德国标准

**class** SocketAdapter **implements** IGermanySocket{

//适配器上中国插头的接入点

**private** IChinaSocket chinaSocket;

**public** SocketAdapter(IChinaSocket chinaSocket){

**this**.chinaSocket=chinaSocket;

}

//关键一步，兼容插座

@Override

**public** **void** powerInGermanySocket() {

chinaSocket.powerInChinaSocket();

}

}

**public** **class** Run {

**public** **static** **void** main(String[] args) {

//把中国充电器插上适配器

SocketAdapter socket=**new** SocketAdapter(**new** ChinaCharger());

//在德国插座上充电

socket.powerInGermanySocket();

}

}

以上就是适配器模式，如果想问为什么不能直接在中国插头ChinaCharger上implements一个德国接口IGermanySocket，那你真是太天真了，你什么时候见过中国插头能够两用，既能插德国插座又能插中国插座，在当前需求场景下可能需要你同时能插两个插座，但如果是在代码的别的地方只需要你实现一个国标就可以了呢？

## 2.4 HandlerAdapter适配器

从adapter这个单词中，我们可以知道，这明显是一个适配器模式， HandlerAdapter实际上是**前端控制器(DispatcherServlet)**用于调用控制器controller的一种适配器(adapter)，DispatcherServelet调用HandlerAdapter来调起控制器，由控制器controller处理http请求

## 2.5 ModelAndView数据和视图

ModelAndView就如其名称所示，是代表了MVC中Model与View的对象，不过它只是方便我们一次返回这两个对象，是二者的集合，Model与View两者仍是分离的概念。model是数据，view就是视图，也就是页面。

为保证逻辑清晰和低耦合，spring mvc搭建的后台一般分为处理分发的Controller层，业务逻辑处理的Service层，数据库交互的Dao层。**ModelAndView 一般在Controller层作为结果类型返回，其内部的view部分是返回的页面(未渲染前只有页面骨架没有数据)，model部分是Dao层查询的数据经由service层处理后得到的数据**

## 2.6 viewResolver

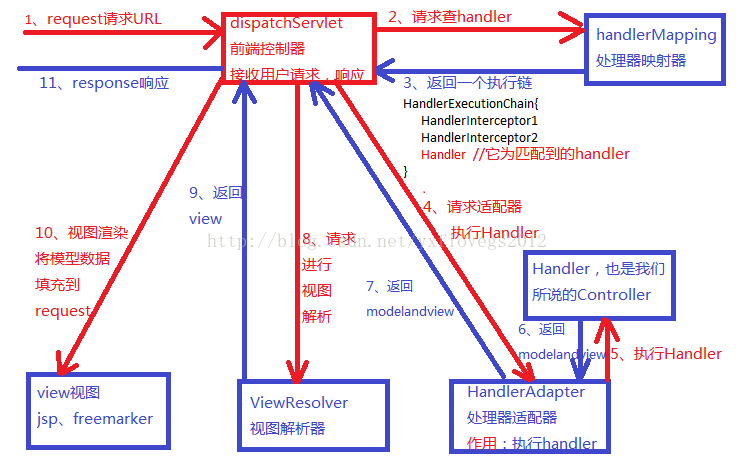
SpringMVC用于处理视图最重要的两个接口是ViewResolver和View。ViewResolver的主要作用是把一个逻辑上的视图名称解析为一个真正的视图，也就ViewResolver是说通过给定的viewName名**(String viewName)**找到对应的View对象，然后使用该view对象的render(渲染)方法将本身的内容写到response中。

## 2.7 一次http请求处理的全过程

了解了上面的主要概念后，就可以了解springMVC的工作周期：

注意：controller既可以叫Handler(处理者)又可以叫action(动作)

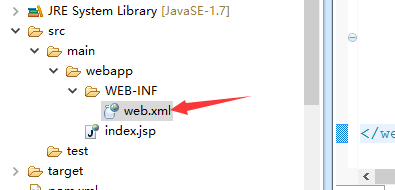
**1.客户端发起http请求到前端控制器(DispatcherServlet)  
2.前端控制器请求HandleMapping查找Handler(controller)   
3.处理器映射器HandleMapping向前端控制器返回Handler(controller)执行链  
4.前端控制器 调用 处理器适配器(HandlerAdapter)执行Handler(controller)  
5.处理器适配器执行Handler  
6.Handler执行完给处理器适配器(HandlerAdapter)返回ModelAndView   
7.处理器适配器(HandlerAdapter)向前端控制器(DispatcherServlet)返回ModelAndView   
8.前端控制器请求视图解析器(viewResolver)进行视图解析  
9.视图解析器(viewResolver)向前端控制器(DispatcherServlet)返回view  
10.前端控制器进行视图渲染  
11.前端控制器向客户端返回http响应**



# 使用SpringMVC

## 3.1 HelloWorld

在配置好springMVC的依赖后，我们需要配置web.xml文件，配制出一个前端控制器DispatcherServlet



1.就像配置一个普通的servlet一样，url将通过servlet-mapping节点找到对应的servlet(DispatcherServlet是http请求的第一站)，这也即是DispatcherServlet拦截http请求的过程

2.然后DispatcherServlet会查询一个或多个处理器映射（handler mapping）并根据请求所携带的URL信息进行决策，将请求发送给哪个SpringMVC控制器（controller）；

3.控制器做两件事：一是将数据打包，二是定义逻辑视图名，然后返回给DispatcherServlet；

4.DispatcherServlet通过视图解析器（view resolver）来将逻辑视图名匹配为一个特定的视图实现，它可能是也可能不是JSP；

5.渲染视图，返回http响应

web.xml配置如下：

<?xml version="1.0" encoding="UTF-8"?>  
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd">

<servlet>

<servlet-name>MyDispatcher</servlet-name>

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

</servlet>

<servlet-mapping>

<servlet-name>MyDispatcher</servlet-name>

<!-- url设为'/'意味着这个servlet将拦截所有的请求 -->

<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app>

上述配置意味着springMVC将默认从WEB-INF下面*(和web.xml同级目录)*读取MyDispatcher-servlet.xml配置文件，该文件的名字由DispatcherServlet的servlet名决定，也就是说，如果在web.xml中将DispatcherServlet的<servlet-name>节点配置成<servlet-name>TestDispatcher</servlet-name>，那本次要读取的配置文件就叫TestDispatcher-servlet.xml。

上面是默认读取的WEB-INF/MyDispatcher-servlet.xml，如果我们想要手动配置读取某个文件作为DispatcherServlet的配置文件，需要在web.xml中加入节点，如：

<servlet>

<servlet-name>MyDispatcher</servlet-name>

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

<!-- 告诉springMVC不使用默认配置，而是将Dispatcher的配置文件放在了/WEB-INF/config/MyDispatcher-servlet.xml下面 -->

<init-param>

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

<param-value>/WEB-INF/config/MyDispatcher-servlet.xml</param-value>

</init-param>

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

</servlet>

<!—如果需要添加spring的applicationContext配置文件-->

<context-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>/WEB-INF/applicationContext.xml</param-value>  
</context-param>  
<listener>  
 <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  
</listener>

<servlet-mapping>

<servlet-name>MyDispatcher</servlet-name>

<!-- url设为'/'意味着这个servlet将拦截所有的请求 -->

<url-pattern>/</url-pattern>

</servlet-mapping>

从上述配置中我们可以看到，springMVC配置文件的自定义路径是放在servletConfig中的(参见servlet学习笔记 应用上下文 一节)，如果还需要加载spring的配置文件，需要添加spring上下文applicationContext的路径到servletContext中,并添加listener监听

接下来就是DispatcherServlet的配置文件————MyDispatcher-servlet.xml：

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

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

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

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* 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-3.0.xsd"*>

**<!-- 允许使用注解配置bean -->**

<context:annotation-config />

**<!-- 扫描com.test.helloWorld包下的所有@controller注解，并将带有该注解的类注册成bean -->**

<context:component-scan base-package=*"com.test.helloWorld"*>

<context:include-filter type=*"annotation"*

expression=*"org.springframework.stereotype.Controller"* />

</context:component-scan>

**<!-- 允许通过注解配置handle-mapping -->**

<mvc:annotation-driven />

**<!-- viewResolver将会去/WEB-INF/jsp下查找后缀为.jsp的文件，viewClass属性指明要解析的视图类型 -->**

<bean

class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>

**<!--本属性标明了jsp文件存放在***/WEB-INF/jsp/***下-->**

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

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

<property name=*"viewClass"*

value=*"org.springframework.web.servlet.view.JstlView"* />

</bean>

</beans>

再然后就是编写controller来处理http请求

package com.test.helloWorld;  
  
import org.springframework.stereotype.Controller;  
import org.springframework.web.bind.annotation.RequestMapping;  
  
@Controller//告诉springMVC这个类是controller  
@RequestMapping("/hello")//告诉springMVC，这个controller将拦截url为"/hello"的http请求  
public class HelloWorld {  
 @RequestMapping("/sayBySpringMVC")//告诉springMVC，这个方法将拦截url为"/hello/sayBySpringMVC"的http请求  
 public String say(){  
 return "home";  
 }  
}

接着在*/WEB-INF/jsp/*下编写**home.jsp**视图文件

<html>  
<body>  
<h2>Hello World!Welcome to Hangzhou!</h2>  
</body>  
</html>

最后，将web项目部署到tomcat上，打开浏览器，输入<http://localhost:8080/hello/sayBySpringMVC>



## DispatcherServlet

### 3.2.1.层次化的应用上下文

在spring和servlet的学习中，我们有学到过spring上下文(webApplicationContext)，servlet上下文(servletContext)，现在又多了一个springMVC的上下文(mlWebApplicationContext)

在一个SpringMVC的项目中，通常有三个配置文件：

**web.xml** 所有的web项目都要有的最最基本的配置文件。所有框架监听器 servet都在这里配置。一个web.xml对应一个servletContext

**MyDispatcher-servlet.xml** Spring MVC的配置文件，有关DispatcherServlet的相关配置都在这里进行。一个MyDispatcher-servlet.xml对应一个mlWebApplicationContext

**applicationContext.xml** 这个是Spring Bean容器的配置文件，管理着IOC (依赖注入)、AOP (面向切面)相关的配置。一个applicationContext.xml对应一个webApplicationContext

1. **servletContext**

servlet容器需要在应用项目启动时，给应用项目初始化一个ServletContext作为公共环境容器存放公共信息，所有在同一web.xml中配置的servlet都共享同一个ServletContext。ServletContext中的信息都是由servlet容器提供的

**一个web.xml对应一个servletContext**

1. **webApplicationContext**

webApplicationContext是springMVC的上下文，也是DispatcherServlet的配置文件(**MyDispatcher-servlet.xml**)

**一个MyDispatcher-servlet.xml对应一个webApplicationContext**

1. **mlWebApplicationContext**

mlWebApplicationContext是spring的应用上下文，是spring的配置文件(**applicationContext.xml** )

**一个applicationContext.xml** **对应一个mlWebApplicationContext**

为了更好的理解这三个上下文之间的关系，有必要了解springMVC的启动过程:

首先，对于一个web应用，其部署在web容器中，web容器提供其一个全局的上下文环境，这个上下文就是ServletContext，其为后面的spring IoC容器提供宿主环境；

其次，在web.xml配置webApplicationContext时会有一个<listener>节点提供contextLoaderListener。在web容器启动时，会触发容器初始化事件，此时contextLoaderListener会监听到这个事件，然后spring会初始化一个应用上下文，这个上下文被称为根上下文，即WebApplicationContext。这个就是spring的IoC容器

contextLoaderListener监听器初始化并创建IOC容器完毕后，开始初始化web.xml中配置的Servlet，这个servlet可以配置多个，最常见的就是DispatcherServlet ，DispatcherServlet上下文在初始化的时候会建立自己的IoC上下文(mlWebApplicationContext)，用以持有spring mvc相关的bean。在初始化自己的上下文之前，会获取WebApplicationContext作为自己的父(根)上下文。有了这个parent上下文之后，再初始化自己持有的上下文。这样每个servlet就持有自己的上下文，即拥有自己独立的bean空间，同时各个servlet共享相同的bean，即根上下文定义的那些bean。

### 3.2.2. web.xml配置文件

在servlet的学习中，我们知道，web.xml文件是servlet的配置文件，因为DispatcherServlet本质上也是一个servlet(特殊的servlet)，因此需要在web.xml中对DispatcherServlet进行servlet相关配置：

让我们看一看一个典型的springMVC的web.xml:

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

<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd">  
 <!--定义了WEB应用的名字-->  
 <display-name>MySpringMVC</display-name>  
 <!--声明WEB应用的描述信息-->  
 <description>Its my first JAVA MVC</description>  
 <servlet>  
 <servlet-name>MyDispatcher</servlet-name>  
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  
 <!-- 告诉springMVC不使用默认配置，而是将Dispatcher的配置文件放在了/WEB-INF/config/MyDispatcher-servlet.xml下面 -->  
 <init-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>/WEB-INF/config/MyDispatcher-servlet.xml</param-value>  
 </init-param>  
 <load-on-startup>1</load-on-startup>  
 </servlet>

<!—加载spring的应用上下文-->  
 <context-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>/WEB-INF/config/myWebApplicationContext.xml</param-value>  
 </context-param>  
 <listener>

<listener-class>

org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>

<servlet-mapping>  
 <servlet-name>MyDispatcher</servlet-name>  
 <!-- url设为'/'意味着这个servlet将拦截所有的请求 -->  
 <url-pattern>/</url-pattern>  
 </servlet-mapping>  
</web-app>

init-param配置的是servletConfig对象的属性，一个servletConfig对应一个servlet，一个servlet可以有多个init-param属性。init-param节点是<servlet>节点的子节点。这里的init-param是用来配置contextConfigLocation的，这个属性将标明dispatcherServlet配置文件相对根目录的位置

context-param配置的是servletContext对象的属性，一个web.xml文件对应一个servlet上下文(servletContext)，一个web.xml文件可以有多个context-param属性，该属性不单独属于任何一个servlet，而是所有的servlet共享。因此，context-param属性和<servlet>节点同级。context-param在这里被用作加载spring配置文件，他将标明webApplicationContext的配置文件相对根目录的位置

listener在一般的web.xml文件中被用作配置监听，在springMVC的web.xml中，它被用作配置监听容器初始化事件，监听到之后，就去加载context-param中标明的spring的配置文件。因此，这里的listener节点是配合context-param节点使用的

dispatcherServlet中的url-pattern设为**” / ”**意为着这个前端控制器将拦截所有的请求

*有时候也会见到****<param-value>classpath:MyDispatcher-servlet.xml</param-value>****这种写法，classPath即为java文件编译之后的class文件的存放目录，一般为WEB-INF/classes，用maven构建项目时候resource目录就是默认的classpath，也可能被修改*

classpath和classpath\*区别：

*classpath：只会到你的classpath路径中查找找文件。*

*classpath\*：查找范围不仅包含classpath路径，还包括classpath路径中的jar文件和classpath下的子文件夹。*

### 3.2.3. 详解DispatcherServlet.xml配置文件

MyDispatcher-servlet.xml是前端控制器dispatcherServlet的配置文件，一个MyDispatcher-servlet.xml文件对应一个springMVC的上下文(mlWebApplicationContext)

让我们先来看一看一个典型的MyDispatcher-servlet.xml配置文件：

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:context="http://www.springframework.org/schema/context" xmlns:tx="http://www.springframework.org/schema/tx"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" 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-3.0.xsd">  
 <!-- 允许使用注解配置bean -->  
 <context:annotation-config />  
 <!-- 扫描com.test.helloWorld包下的所有@controller注解，并将带有该注解的类注册成bean -->  
 <context:component-scan base-package="com.MVC.helloworld">  
 <context:include-filter type="annotation"  
 expression="org.springframework.stereotype.Controller" />  
 </context:component-scan>  
 <!-- 允许通过注解配置handle-mapping -->  
 <mvc:annotation-driven />

<!-- 静态资源管理,如果访问/resources/下的静态资源都会去/WEB-INF/static下查找 -->  
<mvc:resources mapping="/resources/\*\*" location="/WEB-INF/static/" />

<!-- viewResolver将会去/WEB-INF/jsp下查找后缀为.jsp的文件，viewClass属性指明要解析的视图类型 -->  
 <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">  
 <!--本属性标明了jsp文件存放在/WEB-INF/jsp/下-->  
 <property name="prefix" value="/WEB-INF/jsp/" />  
 <property name="suffix" value=".jsp" />  
 <property name="viewClass"  
 value="org.springframework.web.servlet.view.JstlView" />  
 </bean>  
</beans>

**1.<context:annotation-config/>**开启spring的注解功能，这一点在spring的学习中就已经学习了

**2.<context:component-scan/>**扫描查找使用构造型（stereotype）注解所标注的类，如@Component(组件)，@Service（服务），@Controller（控制器），@Repository（数据仓库）

其属性**base-package**指明了要扫描的包的位置，这里是要扫描**com.MVC.helloworld**

属性**use-default-filters="false"**指明是否使用默认的过滤器，默认值true，若为true，那么其子节点 过滤器**<context:include-filter/>** 是选中符合条件的内容，如果为false则排除符合过 滤器子节点 条件的内容

**<context:include-filter/>**过滤器是**<context:component-scan/>**的子节点，他可以有多个这样的过滤器子节点

其属性type指明了要扫描的是注解(**annotation**)，还可以扫描类或接口(**type= "assignable"**)，还可以使用正则匹配(**type= "** **regex"**)

属性**expression**是过滤条件，由于父节点的**use-default-filters**属性默认为true，org.springframework.stereotype.Controller代表 **选中** 带有@controller注解的类并将其注册成bean，该属性还可以直接指明某个类或接口并将其注册成bean，甚至支持正则表达式，如：**<context:exclude-filter type="regex" expression=".service.\*"/>**

**3.** **<mvc:annotation-driven />**帮助我们快速配置默认的HandleMapping和HandleAdapter(参见2.7节)，同时，它还提供了数据绑定支持，如果希望手动配置HandleMapping和HandleAdapter，请参见3.2.4

**4.** **<mvc:resources/>**允许静态资源放在任何地方，如WEB-INF目录下、类路径下等，你甚至可以将JavaScript等静态文件打到JAR包中。通过location属性指定静态资源的位置，由于location属性是Resources类型，因此可以使用诸如"classpath:"等的资源前缀指定资源位置。传统Web容器的静态资源只能放在Web容器的根路径下，**<mvc:resources/>**完全打破了这个限制。

其属性**mapping**代表映射的url，**location**代表实际存储的文件夹，也就是说如果配置了**<mvc:resources mapping="/resources/\*\*" location="/WEB-INF/static/" />**，并且编写了**/WEB-INF/static/js/test.js**文件，访问<http://localhost:8080/resources/js/test.js>就可以访问到该文件，**<mvc:resources/>**可以配置多个，也有这种配法：

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

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

**<mvc:resources mapping="/img/\*\*" location="/WEB-INF/static/img/" />**

### 3.2.4. 非注解的HandlerMapping和HandlerAdapter的配置

如果不想使用**<mvc:annotation-driven />**的默认配置，可以自行配置HandlerMapping和HandlerAdapter

#### 3.2.4.1非注解的处理适配器HandlerAdapter

##### 1. 非注解适配器SimpleControllerHandlerAdapter

该适配器要求handler必须实现controller接口，该接口需重写handleRequest方法，详见3.3节第1小节

<!--处理适配器-->  
<bean class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"/>

springMVC中有多种处理适配器，我们ctrl+鼠标左键查看该 适配器 源码，发现该处理适配器实现HandlerAdapter接口，而HandlerAdapter的源码中明确指明需要重写一个名为supports()方法，该方法用于判断HandlerAdapter是否支持该类型的Handler

如：在SimpleControllerHandlerAdapter，support()被重写为：

@Override  
public boolean supports(Object handler) {  
 return (handler instanceof Controller);  
}

由此我们可以知道，SimpleControllerHandlerAdapter这个处理适配器可以执行所有实现了Controller接口的处理器(Handler)

当然，springMVC中还有一些其他的处理器适配器(HandlerAdapter)，依然可以通过上述方法查看其适配哪种类型的Handler

##### 2. 非注解适配器HttpRequestHandlerAdapter

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

要求handler实现HttpRequestHandler接口，该接口需重写handleRequest方法，同样我们可以查看其support方法，其用法同上，唯一不同的是handleRequest方法返回void

上一种适配器要求必须返回ModelAndView，这一种不需要返回值，一切返回通过原生servlet设置，比如我们需要返回json数据的时候，通常使用这种适配器

public class HelloWorld implements HttpRequestHandler{  
 public void handleRequest(HttpServletRequest request, HttpServletResponse response) throws Exception {  
 // 添加模型数据 可以是任意的POJO对象  
 request.setAttribute("param111", "I am DJ");  
 // 设置要跳转到的到具体的url页面

request.getRequestDispatcher("/home").forward(request,response);  
 }  
}

#### 3.2.4.2非注解的处理器映射器HandlerMapping

在dispatcherServlet收到request后，它需要通过request的url查询到该由哪个处理器handler来处理该请求，那它是到哪查询的呢？就是到我们的HandlerMapping中查询的，HandlerMapping给URL和handler之间提供了映射关系，指明了url该由哪个对应的Handler处理，eg：

<bean name="/test" class="com.test.helloWorld.HelloWorld" />

我们可以看到，HandlerMapping实际上就是一个bean，其name属性标明了url，而class属性则是该url对应由哪个handler进行处理

##### 1. 根据bean的name属性映射controller (BeanNameUrlHandlerMapping)

如上例就是通过controller的name名称(com.test.helloWorld.HelloWorld)来寻找controller，即name属性就作为url，**默认是开启的**，如果非要用手动开启，则配置：

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

使用方式如下(*由于是默认开启，第一句可省略*)：

<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"></bean>  
<bean name="/test" class="com.test.helloWorld.HelloWorld" />

访问localhost:8080/test即可访问该controller

##### 2. 根据URL寻找controller

SimpleUrlHandlerMapping处理器映射器允许通过url绑定处理器(handler)，首先声明一个handler的bean(本例中为hello1)

<bean id="hello1" class="com.test.helloWorld.HelloWorld" />

<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">  
 <property name="mappings">  
 <props>  
 <prop key="/hello1.do">hello1</prop>  
 <prop key="/hello2.do">hello1</prop>  
 </props>  
 </property>  
</bean>

访问http://localhost:8080/hello1.do和http://localhost:8080/hello2.do都能访问该handler

## 3.3控制器controller

controller有两种编写方式，一种是基于配置的编写方式，另一种是基于注解的编写方式，早期，SpringMVC都是通过第一种方式编写controller的

#### 1.基于配置的controller

下面，我们先不使用注解和**<mvc:annotation-driven />**的默认配置来编写一个SpringMVC的应用

在3.2.4节我们已经配置好了我们的处理适配器(HandlerAdapter)，这意味着框架已经能正常地通过HandlerAdapter调用Handler来处理请求了，接下来，我们来手动编写Handler

首先，我们的handler类必须实现controller接口并重写其handleRequest方法：

public class HelloWorld implements Controller{

public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) throws Exception {  
 ModelAndView mv = new ModelAndView();  
 // 添加model数据 可以是任意的POJO对象  
 mv.addObject("myParam", "I am DJ");  
 // 设置view，视图解析器会根据该名字解析到具体的视图页面  
 mv.setViewName("home");  
 return mv;  
 }  
}

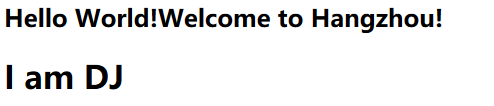
在viewResolver指定的文件夹里编写home.jsp页面

<%@ **page** language="java" contentType="text/html;" 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"%>  
<html>  
<body>  
<h2>Hello World!Welcome to Hangzhou!</h2>  
<h1>**${my**Param**}**</h1>  
<h1></h1>  
</body>  
</html>

在dispatcherServlet.xml中配置该controller对应的url路径(handlerMapping)：

<bean name="/test" class="com.test.helloWorld.HelloWorld" />

部署到tomcat后，在浏览器中输入：http://localhost:8080/test，即可看到：



这样写的一个缺点就是：所有的url都得需要在文件里做配置。当系统庞大页面多的时候，这种效率上以及代码可读性上都大打折扣。于是，基于注解的方法就来了

#### 2.基于注解的controller

请参见3.6.1.@RequestMapping

基于注解的方法虽然提高了编写效率并且增加了可读性，但在配置文件里需要配置使用 DefaultAnnotationHandlerMapping 和 AnnotationMethodHandlerAdapter 作为 url映射控制器 (也可以直接使用<mvc:annotation-driven />的默认配置)。但这样相对较为死板，并且打破了spring的初衷，一旦修改项目就必须修改程序代码(修改注解)，无法通过修改配置文件修改程序

两种方式各有千秋，需要酌情使用

## 3.4 controller的返回值

### 3.4.1.返回ModelAndView

### 3.4.2.返回String

如果controller返回sting，就意味着其返回的是逻辑视图名，那么什么是逻辑视图名呢？简单地来说，逻辑视图名就是视图文件的文件名，有这么一个公式：

视图文件的相对路径(相对于web根目录)=视图解析器中的前缀+逻辑视图名+视图解析器中的后缀，eg：假设有视图解析器如下：

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">  
 <!--本属性标明了jsp文件存放在/WEB-INF/jsp/下-->  
 <property name="prefix" value="/WEB-INF/jsp/" />  
 <property name="suffix" value=".jsp" />  
 <property name="viewClass"  
 value="org.springframework.web.servlet.view.JstlView" />  
</bean>

相对路径=/WEB-INF/jsp/+逻辑视图名+.jsp

@Controller  
@RequestMapping("hello")  
public class HelloWorld{  
 @RequestMapping("say")  
 public String say(Model myModel){  
 myModel.addAttribute("username","丁健");  
 return "say";  
 }  
 @RequestMapping("listen")  
 public String listen(Model myModel){  
 myModel.addAttribute("username","王超");  
 return "listen";  
 }  
}

**注意：当返回值为String逻辑视图时，若想向逻辑视图中传入参数，需要给controller方法添加形参(Model 形参名)，然后通过addAttribute方法传入参数**

### 3.4.3.返回void

返回值为void时可以使用原生的servlet方法进行请求处理和返回response，该方式可以用来返回json数据

### 3.4.3.返回重定向

可以通过"redirect:listen"返回重定向响应，请注意，这里不需要写成"redirect:/hello/listen"，因为他们都是在同一个@RequestMapping("hello")下的

@Controller  
@RequestMapping("hello")  
public class HelloWorld{  
 @RequestMapping("say")  
 public String say(Model myModel){  
 return "redirect:listen";  
 }  
 @RequestMapping("listen")  
 public String listen(Model myModel){  
 myModel.addAttribute("username","王超");  
 return "listen";  
 }  
}

### 3.4.4.返回页面转发

在servlet中，提到重定向就不得不提到页面转发，重定向是将跳转response发送到客户端，由客户端重新定位url进行访问，无法共享前一个请求的上下文，因为这是两次http请求，而页面转发是在服务器端完成的，是将本次请求转发到另外一个servlet中进行处理，其url是不变的，是一个http请求

@Controller  
@RequestMapping("hello")  
public class HelloWorld{  
 @RequestMapping("say")  
 public String say(Model myModel,HttpServletRequest request){  
 myModel.addAttribute("username","王超");  
 return "forward:listen";  
 }  
 @RequestMapping("listen")  
 public String listen(Model myModel,HttpServletRequest request){  
 return "listen";  
 }  
}

**直接在形参中加入需要共享的参数即可，如访问http://localhost:8080/hello/say，将参数username填入model，通过forward页面转发给控制器listen处理，所以最终页面也会带有参数username**

## 3.5 viewResolver视图解析器

视图解析器的功能这里不再赘述，常用的视图解析器如下：

### 3.5.1. InternalResourceViewResolver解析器

## 3.6注解

java注解是java5引入的特性，不同的注解有不同的功能，一般有三种功能类型：

**1.编写文档：**生成文档，这也是java最早提供的注解，帮助编译器进行代码提示，如：@see @param @return 等

**2.代码分析：**通过注解帮助实现配置，简化对配置文件的编写，现在主流的java框架都提供了对注解的支持来简化配置文件的编写，如springMVC的@controller，@RequestMapping等

**3.编译检查：**这类注解可以帮助编译器检查java语法错误，最典型的如：使用@Override声明方法，如果该方法不是覆盖了父类方法**(重写)**，则编译时就能检查出。

**在springMVC中学习的注解，主要是第二种功能，帮助我们简化框架的配置**

### 3.6.1.@RequestMapping

@RequestMapping是springMVC中最常用的注解，它被用来声明地址映射，简单地说，就是声明这个url由哪个controller进行处理，给URl和handler中间加上映射关系

需要注意的是：该注解既可以声明controller的方法，也可以声明controller类，当用于类上，表示类中的所有响应请求的方法都是以该地址作为父路径。eg:

@Controller  
@RequestMapping("/say ")  
public class HelloWorld{  
 @RequestMapping("/hello")  
 public String sayHello(){  
 return "hello";  
 }  
}

*访问localhost:8080/say/hello*

RequestMapping注解有六个属性，下面我们把她分成三类进行说明。下面的每一个属性，都相当于一个筛选条件

#### 1.value和method属性

value：     指定请求的实际地址，指定的地址可以是URI Template 模式，该模式可以帮助我们实现路由参数；所谓的路由参数就是url的一部分字符串作为参数传入controller

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

eg:

@Controller  
@RequestMapping("/hello ")  
public class HelloWorld{

@RequestMapping(value = "/say/{num}",method = RequestMethod.*GET*)  
public void say(HttpServletResponse response, @PathVariable("num") String myNum) {

response.setContentType("text/html;charset=UTF-8");  
 response.setCharacterEncoding("UTF-8");  
 try {  
 PrintWriter pw = response.getWriter();  
 pw.println(myNum);  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }

}

上例中**{num}**就是常用的路由参数，代码中我们通过注解**@PathVariable**来帮助我们接受路由参数并将参数传入say方法，我们将路由参数打印出来(**pw.println(myNum);**)看一看，部署到tomcat后，输入 **http://localhost:8080/hello/say/你好** ，打印结果为：你好，说明url的{num}确实被传入了controller了

甚至value值还可以是正则表达式，如：

@RequestMapping("/spring-web/{wangchao:[a-z-]+}-{age:[\\d]+}")  
public void test\_3(HttpServletResponse response, @PathVariable("wangchao") String name, @PathVariable("age") int myAge) {  
 response.setContentType("text/html;charset=UTF-8");  
 response.setCharacterEncoding("UTF-8");  
 try {  
 PrintWriter pw = response.getWriter();  
 pw.println("开始");  
 pw.println("姓名"+name);  
 pw.println("年龄"+myAge);  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
}

**/spring-web/{wangchao:[a-z-]+}-{age:[\\d]+}** 意味着将会接受一组a到z的英文字符串+一个”-”+一个数字，其中英文字符串将赋值给**wangchao**，字符将赋值给**age**，输入**http://localhost:8080/hello/spring-web/wangchao-18**，显示结果为

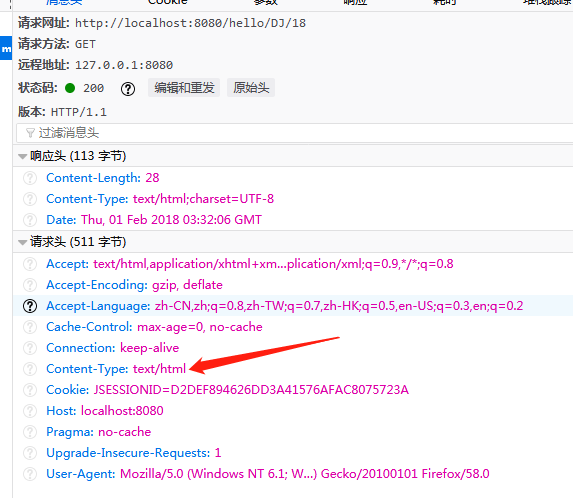
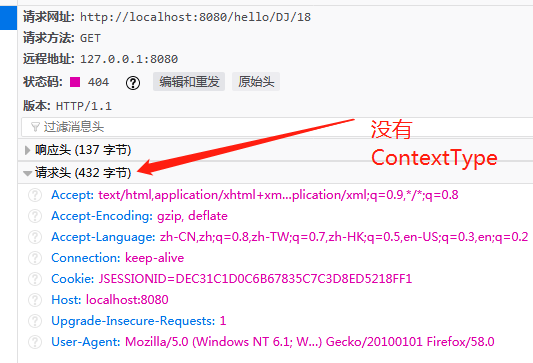


#### 2.consumes和produces属性

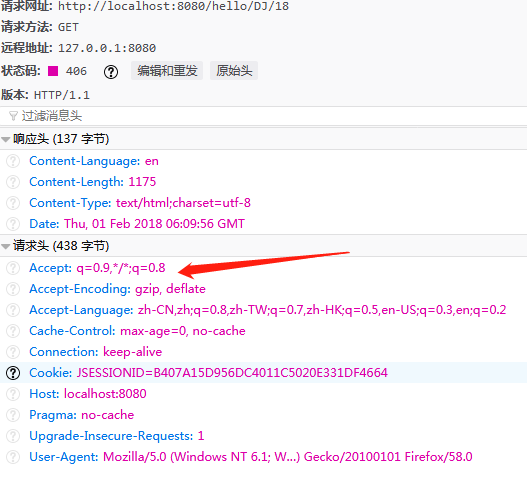
consumes： 指定处理请求的提交内容类型（Content-Type），例如:

@RequestMapping(value="/{name:[a-zA-Z]+}/{age:[\\d]+}",consumes = "text/html")

这意味着**request请求头**中必须带上” Content-Type:text/html”才能被处理(点击”编辑和重发”即可编辑http请求头)



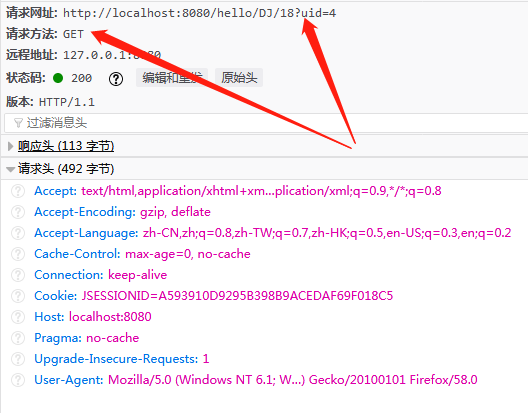
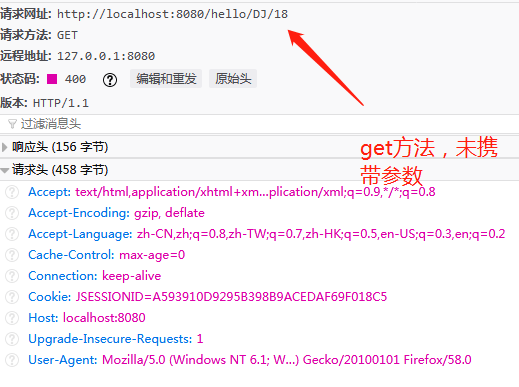
produces:    指定返回的内容类型，仅当request请求头中的(Accept)类型中包含该指定类型才返回；



#### 3. params和headers属性

**params：** 只有当request中携带某些参数值时，才让该方法处理。如

@RequestMapping(value="/{name:[a-zA-Z]+}/{age:[\\d]+}",params = "uid")



***还可以给params指定值，如params = "uid=4"，只有uid=4的请求可以被处理***

**headers：** 指定request的header头中必须包含某些指定的属性值，才能让该方法处理请求。如：

@RequestMapping(value = "/pets", method = RequestMethod.*GET*, headers="Referer=http://www.ifeng.com/")

需要注意的是params和headers支持简单的表达式，如：

**params = "uid"**表示request参数中***必须有***uid参数，**headers**也一样

**params = "!uid"**表示request参数中***不能有***uid参数，**headers**也一样

**params = "uid!=3"**表示request参数中的参数uid不能等于3，如果没有uid参数也不会报404，**headers**也一样

**params = {"username=root","password!=pwd"}**表示request参数中的参数**username**等于root，**password**不能为pwd(或者没有**password**参数)

### 3.6.2.@RequestParam

@RequestParam用于将http请求的参数数据映射到handler的形参上

@RequestParam有三个属性：

value:要绑定的http请求的参数名

required：指定该http参数是否为必须的，如果该值为true，则request必须带有该参数，否则报404

defaultValue：默认值，表示如果请求中没有同名参数时的默认值

eg：

@Controller  
@RequestMapping("hello")  
public class HelloWorld{  
 @RequestMapping("listen")  
 public String listen(Model myModel,@RequestParam(value = "myUid",required = true) Integer uid){  
 myModel.addAttribute("username",uid);  
 return "listen";  
 }  
}

http://localhost:8080/hello/listen?myUid=4

### 3.6.3.@ResponseBody

@responseBody注解的作用是将controller的方法返回的对象通过适当的转换器转换为指定的格式之后，写入到response对象的body区，通常用来返回JSON数据或者是XML，eg：假设User类有两个属性userName和pwd

@RequestMapping("/login")  
@ResponseBody  
public User login(User user){  
　　　return user;  
}

那么在前台接收到的数据为：'{"userName":"xxx","pwd":"xxx"}'

以上代码等同于

@RequestMapping("/login")  
public void login(User user, HttpServletResponse response){  
　　　　response.getWriter.write(JSONObject.fromObject(user).toString());  
}

使用该注解请注意：如果返回的值为json类型，需要导入jackson的jar包，jackson的maven代码为：

下述具体使用方式请参见maven笔记

<jacksonVersion>2.9.4</jacksonVersion>

<dependency>  
 <groupId>com.fasterxml.jackson.core</groupId>  
 <artifactId>jackson-core</artifactId>  
 <version>${jacksonVersion}</version>  
</dependency>  
<dependency>  
 <groupId>com.fasterxml.jackson.core</groupId>  
 <artifactId>jackson-databind</artifactId>  
 <version>${jacksonVersion}</version>  
</dependency>  
<dependency>  
 <groupId>com.fasterxml.jackson.core</groupId>  
 <artifactId>jackson-annotations</artifactId>  
 <version>${jacksonVersion}</version>  
</dependency>

### 3.6.4.@RequestBody

### 3.6.5.@InitBinder

@InitBinder常常用于解决类型类型转换问题，由@InitBinder标记的方法，可以对WebDataBinder对象进行初始化*(也就是说WebDataBinder可以放在方法的形参中直接用)*。WebDataBinder是DataBinder的子类，用于完成由表单到JavaBean属性的绑定

**需要注意的是：被@InitBinder标记的方法返回值必须为null，@InitBinder方法的参数通常是WebDataBinder，@InitBinder可以自动对WebDataBinder进行初始化**

那么WebDataBinder该如何使用呢？

这就涉及到自定义参数绑定，请参见3.7.2.3

### 3.6.6.@SessionAttributes

@SessionAttributes只能作用于类上，将指定数据加入session，通过3.8.2节我们可以知道，modelMap中的属性将被放置到request域中，然后用于渲染jsp页面，但这并不能实现跨请求，因此，需要将其放到session中去。modelMap中被@SessionAttributes标记过的属性将被放到session中，eg：

@Controller  
@RequestMapping("/bbtForum.do")  
@SessionAttributes("currUser") //将ModelMap中属性名为currUser的属性放到Session属性列表中，以便这个属性可以跨请求访问   
public class BbtForumController {  
 @RequestMapping(params = "method=listBoardTopic")  
 public String listBoardTopic(@RequestParam("id")int topicId, User user, ModelMap model) {  
 bbtForumService.getBoardTopics(topicId);  
 System.*out*.println("topicId:" + topicId);  
 System.*out*.println("user:" + user);  
 model.addAttribute("currUser",user); //向ModelMap中添加一个属性   
 return "listTopic";  
 }  
}

## 3.7参数绑定

在web开发中，参数绑定是一项极其重要的技术，通过参数绑定，我们可以从客户端通过GET/POST方法传递key/value数据到服务器端进行处理，在springMVC中，就是通过转换器converter这么一个参数绑定组件将key/value数据转换成controller的形参

### 3.7.1.默认支持的参数类型

SpringMVC 有支持的默认参数类型，我们直接在controller方法的形参上给出这些默认类型的声明，就能直接使用了。因为在调用handler方法的时候，遇到以下类型作为controller的形参，springMVC会直接进行参数绑定

#### 3.7.1.1.HttpServletRequest

request请求对象

#### 3.7.1.2.HttpServletResponse

response响应对象

#### 3.7.1.3.HttpSession

过HttpSession对象得到session中存放的数据

#### 3.7.1.4.Model/ModelMap

Model是一个接口，ModelMap是实现了Model接口的一个类，因此，定义这两个中任意一个都行，作用是将Model数据填充到request域

#### 3.7.1.5.简单数据类型

除了以上类型以外，默认参数绑定还支持简单数据类型，如:int，String，float等，比如一种常见的传参：

@RequestMapping("listen")  
public String listen(Model myModel,Integer userId){

myModel.addAttribute("userId", userId);

return "listen";  
}

访问**http://localhost:8080/hello/listen? userId=66**，即可将userId(66)传入controller，如果形参不希望使用userId，还可以通过@RequestParam(详见3.6.1)来指定request的参数和handler的形参间的映射关系，如果不使用@RequestParam注解(如上例)，就要求request参数名和handler形参名一致

#### 3.7.1.6.数组类型绑定

同样的，我们也可以对数组进行参数绑定：

@Controller  
@RequestMapping("hello")  
public class HelloWorld{  
 @RequestMapping(value="say",produces = "text/html;charset=utf-8")  
 @ResponseBody  
 public String listen(String[] name,HttpServletResponse response){  
 StringBuilder strBuilder=new StringBuilder();  
 for (String nameStr:name) {  
 strBuilder.append(nameStr).append("||");  
 }  
 return strBuilder.toString();  
 }  
}

访问http://localhost:8080/hello/say?name=张三&name=赵四&name=jjj

关于**@ResponseBody**请参见3.6.3

**当然，如果不想通过形参这种方式，也可以手动进行参数绑定，通过HttpServletRequest对象的getParameter方法即可，eg：**

@RequestMapping("/hello")  
public String hello(HttpServletRequest request, HttpServletResponse response) {  
 String name = request.getParameter("name");  
 request.setAttribute("username",name);  
 return "hello";  
}

### 3.7.2.POJO绑定

POJO（Plain Ordinary Java Object）简单Java对象，实际就是普通JavaBean

在通过表单提交数据时，SpringMVC是将表单数据作为POJO对象提交，同样，我们也可以通过controller形参(JAVA数据承载bean，也就是MVC中的model)来接收POJO对象，但是要求**controller形参的属性必须和表单的input的name属性一一对应**，这样就可以将页面中的表单数据，当然，POJO绑定并不会影响简单类型的绑定，比如input表单中有一个age属性，POJO绑定的同时也可以对age进行绑定，eg：

@RequestMapping("listen")  
public String listen(MyUserPojo user,@RequestParam(value = "age",required = true) int myAge){  
 return "listen";  
}

**拓展：**

1、PO, Persistence Object, 持久层(dao)对象，对象的属性和数据库表的字段一一对应，不多也不少；PO对象中不应该包含任何对数据库的操作，应该由被spring管理的DAO对象来操作PO对象，对数据库进行增删改查

2、BO, Business Object, 业务层(service)对象，对象的属性和当前业务逻辑所需的数据的名称一一对应，可以包括多个PO对象，通常需要把BO转换成PO才能进行数据的持久化；

eg：一个简历对象(BO)，应当包含一个教育经历的PO，一个工作经历的PO，一个个人能力的PO。我们业务层需要操作的仅仅是一个简历BO对象，将BO转化为多个PO后，使用DAO层方法对PO对象进行操作(增删改查)

3、VO, View Object, 表现层(view)对象,对象的属性和页面展示的数据的名称一一对应，与PO一样仅包含数据；使用VO对象对视图进行数据渲染，通常一张页面的所有数据对应一个VO对象

4、POJO, Plain Ordinary Java Object, 普通JAVA对象，只有属性及其set/get方法。

注意：

（1）PO在向数据库中添加新数据时创建，在删除数据库中数据时删除，只能存活在一个数据库连接中，断开连接即被销毁；且PO对象需要实现序列化借口。

（2）BO、VO和POJO则由new关键字创建，并由GC回收，只要不被回收，其总是存在的

#### 3.7.2.1.简单POJO绑定

简单POJO绑定只有一层，需要传输的POJO对象属性和controller形参的对象属性一一对应，eg：

**前端代码：**

<form action="/hello/say">  
 姓名<input type="text" name="name">  
 年龄<input type="text" name="age">  
 <input type="submit">  
</form>

**后台代码：**

@Controller  
@RequestMapping("hello")  
public class HelloWorld{

//显示表单页面的controller  
 @RequestMapping(value="listen")  
 public String listen(){  
 return "input";  
 }

//处理表单页面请求的controller  
 @RequestMapping(value="say",produces = "application/json;charset=utf-8")  
 @ResponseBody  
 public String say(User user, HttpServletResponse response){  
 return user.toString();  
 }  
}

**User POJO类：**

public class User {  
 private String name;  
 private int age;  
 public String getName() {  
 return name;  
 }  
 public void setName(String name) {  
 this.name = name;  
 }  
 public int getAge() {  
 return age;  
 }  
 public void setAge(int age) {  
 this.age = age;  
 }  
 @Override  
 public String toString() {  
 return new StringBuilder("{\"name\":\"").append(this.name).append("\",\"age\":\"").append(this.age).append("\"}").toString();  
 }  
}

#### 3.7.2.2.多层级POJO绑定

如果POJO对象属性和controller形参对象的属性不是简单类型，那么就会出现多层级POJO绑定，eg：

**前端代码：**

<form action="/hello/say" method="post">  
 姓名<input type="text" name="name">  
 年龄<input type="text" name="age">  
 省<input type="text" name="address.province">  
 市<input type="text" name="address.city">  
 <input type="submit">  
</form>

**后台代码：**

@Controller  
@RequestMapping("hello")  
public class HelloWorld{  
 //显示表单页面  
 @RequestMapping(value="listen")  
 public String listen(){  
 return "input";  
 }  
 @RequestMapping(value="say",produces = "application/json;charset=utf-8")  
 @ResponseBody  
 public String say(User user, HttpServletResponse response){  
 return user.toString();  
 }  
}

**User POJO类：**

public class User {  
 private String name;  
 private int age;  
 private Address address;  
  
 public Address getAddress() {  
 return address;  
 }  
 public void setAddress(Address address) {  
 this.address = address;  
 }  
  
 public String getName() {  
 return name;  
 }  
 public void setName(String name) {  
 this.name = name;  
 }  
 public int getAge() {  
 return age;  
 }  
 public void setAge(int age) {  
 this.age = age;  
 }  
  
 @Override  
 public String toString() {  
 return new StringBuilder("{\"name\":\"").append(this.name).append("\",\"age\":\"").append(this.age).append("\",\"address\":\"").append(this.address.toString()).append("\"}").toString();  
 }  
}

**Address POJO类：**

public class Address {  
 private String province;  
 private String city;  
  
 public String getProvince() {  
 return province;  
 }  
  
 public void setProvince(String province) {  
 this.province = province;  
 }  
  
 public String getCity() {  
 return city;  
 }  
  
 public void setCity(String city) {  
 this.city = city;  
 }  
  
 @Override  
 public String toString() {  
 return new StringBuilder(this.province).append("省").append(this.city).append("市").toString();  
 }  
}

#### 3.7.2.3.自定义参数绑定

既然controller形参的属性和表单的input的name属性一一对应就可以自动将POJO对象填入controller的形参，那么会不会有这么一种情况，对于一个POJO对象，有两个属性完全相同的形参可以匹配，那么POJO对象到底该填入哪个形参呢？

经过试验，最终发现，POJO对象同时填入了两个形参，这明显不符合业务需求。

再比如，我们在表单中填入一个日期，如1996-10-17 20:32:08，使用Date类型来接收，发现结果并不像我们想象的那样

### 3.7.3.集合框架参数绑定

#### 3.7.3.1.List参数绑定

list参数绑定，需要将list封装成一个POJO对象，eg：

**表单：**

<form action="/hello/say">  
 姓名<input type="text" name="users[0].name">  
 年龄<input type="text" name="users[0].age">  
  
 姓名<input type="text" name="users[1].name">  
 年龄<input type="text" name="users[1].age">  
  
 姓名<input type="text" name="users[2].name">  
 年龄<input type="text" name="users[2].age">  
 <input type="submit">  
</form>

**封装的list对象：**

public class UserList {

private List<User> users;  
  
 public List<User> getUsers() {  
 return users;  
 }  
  
 public void setUsers(List<User> users) {  
 this.users = users;  
 }  
  
 @Override  
 public String toString() {  
 StringBuilder strBuild=new StringBuilder();  
 Iterator it=this.users.iterator();  
 while(it.hasNext()){  
 strBuild.append(it.next().toString()).append("\_\_\_\_\_\_");  
 }  
 return strBuild.toString();  
 }  
}

**后台代码：**

@Controller  
@RequestMapping("hello")  
public class HelloWorld{  
 //显示表单页面  
 @RequestMapping(value="listen")  
 public String listen(HttpServletRequest request){  
 return "input";  
 }  
 @RequestMapping(value="say",produces = "text/html;charset=utf-8")  
 @ResponseBody  
 public String say( UserList users){  
 return users.toString();  
 }  
}

#### 3.7.3.2.Set参数绑定

同list一样，set参数绑定，也需要将set封装，但需要注意的是，在封装时，要在封装的set类的构造函数中对set进行初始化，eg：

public UserSet(){  
 this.users=new HashSet<User>();  
 this.users.add(new User());  
 this.users.add(new User());  
 this.users.add(new User());  
}

请注意上述构造函数，我们在构造函数中对封装的**private Set<User> users;** 进行初始化，给set中添加了三个User，意味着这个POJO对象可以接受3个及以下的User类型的参数绑定，也就是说，访问http://localhost:8080/hello/say?users[0].name=123&users[0].age=123&users[1].name=444&users[1].age=444&users[2].name=555&users[2].age=555正常，访问http://localhost:8080/hello/say?users[0].name=123&users[0].age=123&users[1].name=444&users[1].age=444&users[2].name=555&users[2].age=555&***users[3].name=555&users[3].age=555***则会出现越界，报500异常，封装类如下：

public class UserSet {  
 private Set<User> users;  
  
 public UserSet(){  
 this.users=new HashSet<User>();  
 this.users.add(new User());  
 this.users.add(new User());  
 this.users.add(new User());  
 }  
 public Set<User> getUsers() {  
 return users;  
 }  
  
 public void setUsers(Set<User> users) {  
 this.users = users;  
 }  
  
 @Override  
 public String toString() {  
 StringBuilder strBuild=new StringBuilder();  
 Iterator it=this.users.iterator();  
 while(it.hasNext()){  
 strBuild.append(it.next().toString()).append("\_\_\_\_\_\_");  
 }  
 return strBuild.toString();  
 }  
}

**当然，为了达到set集合排除重复的作用，我们还得在User类中重写equal和hashcode函数，因为set的元素User不是简单数据类型**

#### 3.7.3.2.Map参数绑定

Map参数绑定，有点类似于php中的关联数组，也是通过封装的形式进行绑定，eg：

public class UserMap {  
 private Map<String,User> users;  
  
 public Map<String,User> getUsers() {  
 return users;  
 }  
  
 public void setUsers(Map<String,User> users) {  
 this.users = users;  
 }  
  
 @Override  
 public String toString() {  
 StringBuilder strBuild=new StringBuilder();  
 Iterator it=this.users.entrySet().iterator();  
 while(it.hasNext()){  
 Map.Entry entry = (Map.Entry) it.next();  
 String key =(String) entry.getKey();  
 User value =(User) entry.getValue();  
 strBuild.append("key:").append(key).append(value.toString()).append("\_\_\_\_\_\_");  
 }  
 return strBuild.toString();  
 }  
}

<form action="/hello/say">  
 姓名<input type="text" name="users[a].name">  
 年龄<input type="text" name="users[a].age">  
  
 姓名<input type="text" name="users[b].name">  
 年龄<input type="text" name="users[b].age">  
  
 姓名<input type="text" name="users['c'].name">  
 年龄<input type="text" name="users['c'].age">  
 <input type="submit">  
</form>

@Controller  
@RequestMapping("hello")  
public class HelloWorld{  
 //显示表单页面  
 @RequestMapping(value="listen")  
 public String listen(HttpServletRequest request){  
 return "input";  
 }  
 @RequestMapping(value="say",produces = "text/html;charset=utf-8")  
 @ResponseBody  
 public String say(UserMap users){  
 return users.toString();  
 }  
}

## 3.8页面传参(渲染数据)

页面传参也就是页面数据的渲染，将后台处理后的数据渲染到前端页面上

### 3.8.1.通过ModelAndView对象

通过ModelAndView对象可以帮助我们传参，eg：

@RequestMapping("/hello")  
public ModelAndView hello(HttpServletRequest request, HttpServletResponse response) {  
 Map<String,String> name = new HashMap<String,String>();  
 name.put("myName",request.getParameter("name"));  
   
 Map<String,Integer> age = new HashMap<String,Integer>();  
 age.put("myAge",Integer.*parseInt*(request.getParameter("age")));  
   
 Map<String,Object> data = new HashMap<String,Object>();  
 data.put("name",name);  
 data.put("age",age);

return new ModelAndView("hello",data);  
}

<h1>你好啊!${name.get("myName")}</h1>  
<h1>年龄${age.get("myAge")}</h1>

通过上述方式，可以完成类似php的渲染效果

### 3.8.2.通过Model/ModelMap对象

Model是一个接口，ModelMap是实现了Model接口的一个类，因此，定义这两个中任意一个都行，但需要注意的是，ModelMap支持put和addAttribute方法，而Model只支持addAttribute方法，另外，**这两个对象由框架自动创建实例并作为controller的形参**，用户无需自己创建，eg：

@RequestMapping("/hello")  
public String hello(HttpServletRequest request, HttpServletResponse response,Model data) {  
 data.addAttribute("name",request.getParameter("name"));  
 return "hello";

}

@RequestMapping("/hi")  
public String hi(HttpServletRequest request, HttpServletResponse response,ModelMap data) {  
 data.addAttribute("name",request.getParameter("name"));  
 data.put("age",request.getParameter("age"));  
 return "hello";

}

### 3.8.3.通过@ModelAttribute注解

在Controller方法的参数部分或Bean属性方法上使用，@ModelAttribute数据会利用HttpServletRequest的Attribute将参数传递到视图中

@RequestMapping("/hello")  
public String hello(@ModelAttribute String name,@ModelAttribute int age) {  
 return "hello";  
}

## 3.9拦截器

拦截器可以用于拦截客户端发送到服务器端的请求，也可以拦截服务器发往客户端的响应，对请求和响应中的数据进行相应的更改，比如乱码问题和权限控制，都可以通过拦截器帮助我们处理，拦截器实际上体现了spring中的面向切面编程的思想

### 3.9.1.过滤器

过滤器和拦截器不同，但二者是极其相似的，二者不同的地方有：

**过滤器：** 依赖于servlet容器，是servlet提供的。在实现上基于函数回调，可以对几乎所有请求进行过滤，但是缺点是一个过滤器实例只能在容器初始化时调用一次。使用过滤器的目的是用来做一些过滤操作，获取我们想要获取的数据，比如：在过滤器中修改字符编码；在过滤器中修改HttpServletRequest的一些参数，包括：过滤低俗文字、危险字符等

**拦截器：**依赖于web框架，在SpringMVC中就是依赖于SpringMVC框架，是springMVC为我们提供的。在实现上基于Java的反射机制，属于面向切面编程（AOP）的一种运用。由于拦截器是基于web框架的调用，因此可以使用spring的依赖注入（DI）进行一些业务操作，同时一个拦截器实例在一个controller生命周期之内可以多次调用。但是缺点是只能对controller请求进行拦截，对其他的一些比如直接访问静态资源的请求则没办法进行拦截处理

最常见的例子，比如我们在进行参数绑定的时候，如果前端页面传输了一个中文字符串到后台(get或post方式都行)，handler在形参中接收String参数，将Sring参数打印出来，却发现该中文字符串乱码

SpringMVC提供了一个filter，由于filter是servlet中的东西，并且是web应用级别的，因此部署在web.xml中，eg：

<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>

### 3.9.2.拦截器的实现

编写拦截器类需要实现**HandlerInterceptor**接口，然后再将拦截器注册到SpringMVC中去

1. 编写拦截器类

public class MyInterceptor implements HandlerInterceptor{  
 public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) {  
 System.*out*.println("执行afterCompletion方法");  
 }  
  
 public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) {  
 System.*out*.println("执行postHandle方法");  
 }  
  
 public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) {  
 System.*out*.println("执行preHandle方法");  
 return true;  
 }  
}

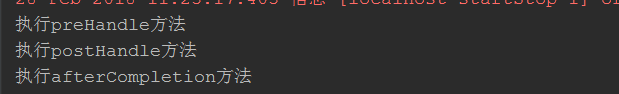
1. 注册拦截器

在dispatcherServlet.xml (SpringMVC的配置文件)中注册拦截器

<mvc:interceptors>  
 <mvc:interceptor>  
 <mvc:mapping path="/hello/\*"/>  
 <bean class="com.MVC.interceptor.MyInterceptor"></bean>  
 </mvc:interceptor>  
 <mvc:interceptor>  
 <mvc:mapping path="/\*/listen"/>  
 <bean class="com.MVC.interceptor.MyInterceptor"></bean>  
 </mvc:interceptor>  
</mvc:interceptors>

以上配置意味着所有符合**/hello/\***和符合**\*/listen**的url都会使用拦截器**MyInterceptor**

任意访问一个controller，如：http://localhost:8080/hello/listen或者http://localhost:8080/want/say (请先编写好controller)发现控制台输出



### 3.9.3. HandlerInterceptor拦截器接口

HandlerInterceptor是springMVC为我们提供的拦截器接口，该接口提供了三个方法：

**preHandle**方法在请求被处理之前调用，该方法有boolean类型返回值，如果返回false，就会将当前请求拦截下来，请求将会终止，不会得到响应，如果返回true，请求将会抛向下一层进行处理

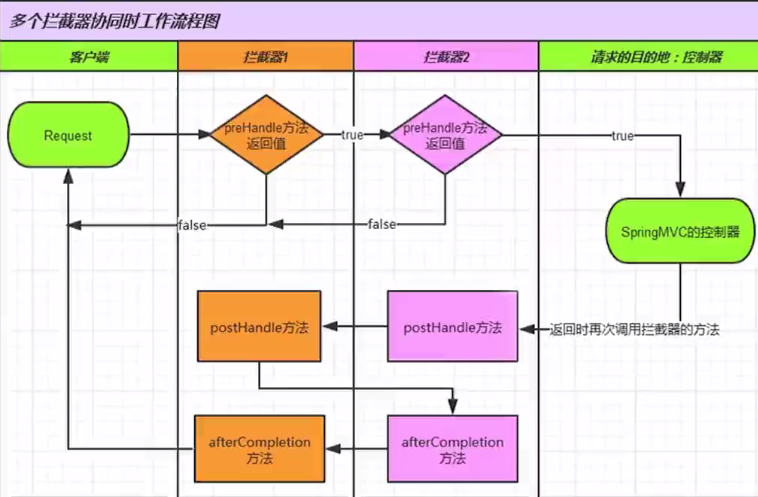
**postHandle**方法在请求被处理之后调用，该方法有一个**ModelAndView modelAndView**的参数，该参数可以修改最终渲染的视图，因为执行到这一步的时候，是请求被处理之后，产生的ModelAndView(在2.7节的第7步)

**afterCompletion**方法在请求结束之后调用(视图被渲染以后)，常用于关闭一些资源，有点类似于析构函数

**上述三个方法，都有一个名为Object handler的形参，该形参代表本次请求要访问的controller，将handler参数打印出来可以看到访问的是HelloWorld的listen方法(控制器)**



### 3.9.4. 多个拦截器的配置



也就是说，如果存在拦截器1，拦截器2，那么执行顺序就是：

**preHandle\_1**

**preHandle\_2**

**postHandle\_2**

**postHandle\_1**

**afterCompletion\_2**

**afterCompletion\_1**

## 3.10文件上传

SpringMVC实现文件上传，需要再添加两个jar包。一个是文件上传的jar包，一个是其所依赖的IO包。

<!--文件上传包-->  
<dependency>  
 <groupId>commons-fileupload</groupId>  
 <artifactId>commons-fileupload</artifactId>  
 <version>${fileupload.version}</version>  
</dependency>  
<dependency>  
 <groupId>commons-io</groupId>  
 <artifactId>commons-io</artifactId>  
 <version>${fileupload.io.version}</version>  
</dependency>

然后在springMVC的配置文件中添加：

<!-- SpringMVC上传文件时，需要配置MultipartResolver处理器 -->  
<bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver">  
 <!-- 指定所上传文件的总大小不能超过200000KB。注意maxUploadSize属性的限制不是针对单个文件，而是所有文件的容量之和 -->  
 <property name="maxUploadSize" value="200000000"/>  
</bean>  
  
<!—(可选)该异常是SpringMVC在检查上传的文件信息时抛出来的，而且此时还没有进入到Controller方法中 -->  
<bean id="exceptionResolver" class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">  
 <property name="exceptionMappings">  
 <props>  
 <!-- 遇到MaxUploadSizeExceededException异常时，自动跳转到/WEB-INF/jsp/error/error.jsp页面 -->  
 <prop key="org.springframework.web.multipart.MaxUploadSizeExceededException">error/error</prop>  
 </props>  
 </property>  
</bean>

# RESTful设计风格

## 4.1 什么是RESTful

RESTful是一种api的设计风格，标准的RESTful接口，每个api的url代表一种资源（resource），所以**url中不能有动词，只能有名词**，而且所用的名词往往与数据库的表格名对应。不在url中使用动词，而使用HTTP协议中的动词来实现资源的增删改查等操作。

**GET 用来获取资源，  
POST 用来新建资源（也可以用于更新资源），  
PUT请求修改URL所在资源的数据元素，未提交的数据项将全部置空。**

**PATCH请求局部修改URL所在资源的数据项，未提交的数据项将不做修改**

**DELETE 用来删除资源**

如：

**非 RESTful风格：**

**url 请求类型 功能**

http://127.0.0.1/user/add POST 新增用户

http://127.0.0.1/user/delete GET 删除用户

http://127.0.0.1/user/modify POST 修改用户信息

http://127.0.0.1/user/getInfo GET 获取用户信息

**RESTful风格：**

**url 请求类型 功能**

http://127.0.0.1/user POST 新增用户

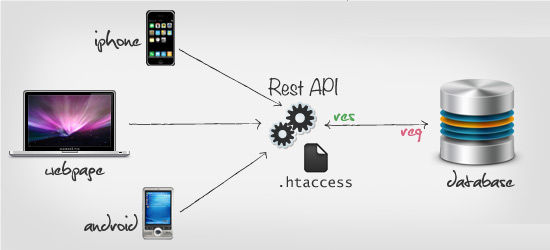
http://127.0.0.1/user DELETE 删除用户

http://127.0.0.1/user PUT 修改用户信息

http://127.0.0.1/user GET 获取用户信息

那么为什么要使用RESTful风格的url呢？

1.由于近些年移动互联网的快速发展，客户端设备层出不穷，RESTful风格的api可以为Browser、Android和IOS等设备提供统一的接口



2.RESTful可以使用HTTP状态码来表示执行状态，如用户需要获取某个资源，RESTful风格通过GET方法访问.../resource/{resourceId} ，如果资源不存在，RESTful风格的api会返回404来表示执行失败的原因。但如果只使用get和post方法，访问.../getResource?resourceId=resourceId，如果资源不存在，则不能使用404作为返回状态，因为url是正确的，是存在的，因此只能返回200和空数据，然后在客户端处理

## 如何设计RESTful风格的api

这里有如下几条建议：

1. 尽量使用HTTPS协议
2. 将api部署在专用的域名下，eg：https://api.demo.com
3. 应将api的版本号放入URL中，eg：https://api.demo.com/v1和https://api.demo.com/v2，分别代表api 1.0和api 2.0
4. 每一个url代表一个资源，因此url只能由名词组成，不能有动词，且推荐使用复数，eg：https://api.demo.com/v1/users

https://api.demo.com/v1/orders

1. 使用HTTP状态码来判断操作是否成功，常用的状态码及含义如下：

200 OK - [GET]：服务器成功返回用户请求的数据，该操作是幂等的（Idempotent）。

201 CREATED - [POST/PUT/PATCH]：用户新建或修改数据成功。

202 Accepted - [\*]：表示一个请求已经进入后台排队（异步任务）

204 NO CONTENT - [DELETE]：用户删除数据成功。

400 INVALID REQUEST - [POST/PUT/PATCH]：用户发出的请求有错误，服务器没有进行新建或修改数据的操作，该操作是幂等的。

401 Unauthorized - [\*]：表示用户没有权限（令牌、用户名、密码错误）。

403 Forbidden - [\*] 表示用户得到授权（与401错误相对），但是访问是被禁止的。

404 NOT FOUND - [\*]：用户发出的请求针对的是不存在的记录，服务器没有进行操作，该操作是幂等的。

406 Not Acceptable - [GET]：用户请求的格式不可得（比如用户请求JSON格式，但是只有XML格式）。

410 Gone -[GET]：用户请求的资源被永久删除，且不会再得到的。

422 Unprocesable entity - [POST/PUT/PATCH] 当创建一个对象时，发生一个验证错误。

500 INTERNAL SERVER ERROR - [\*]：服务器发生错误，用户将无法判断发出的请求是否成功。

1. 如果返回状态码是4\*\*，就代表访问失败，伴随着访问失败的，一般还有错误提示，在api的返回参数中，将error作为键名，出错信息作为键值即可。

eg：

{

error: "Invalid API key"

}

1. 返回的数据格式应尽量使用JSON，尽量避免使用xml

总结：RESTful风格就是：

**看Url就知道要什么  
看http method就知道干什么  
看http status code就知道结果如何**

需要注意的是：由于PATCH是一种比较新的请求方式，tomcat7.0都不支持，导致springMVC无法正常从PATCH请求中拿到请求的参数，比较好的处理方式是在web.xml中加入

<!--使PATCH方式被支持-->  
<filter>  
 <filter-name>HttpPutFormContentFilter</filter-name>  
 <filter-class>org.springframework.web.filter.HttpPutFormContentFilter</filter-class>  
</filter>  
<filter-mapping>  
 <filter-name>HttpPutFormContentFilter</filter-name>  
 <servlet-name>你的dispatcherServlet的<servlet-name></servlet-name>  
</filter-mapping>

Filter的工作是从request body的form data里面读取数据，然后包装成一个ServletRequest，使得ServletRequest.getParameter\*()之类的方法可以读取到数据。

String,String,String,Integer,Integer,Integer

java.lang.Integer, java.lang.Integer, java.lang.Integer