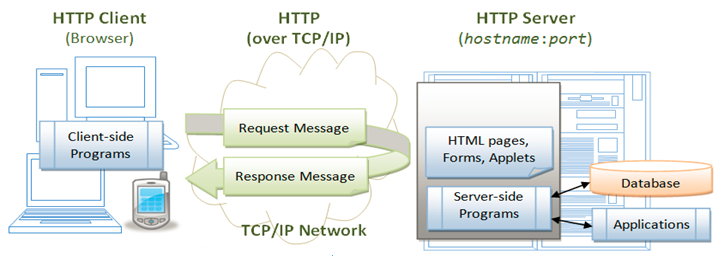
# Spring MVC简介

## Web开发请求响应模型

在web的世界里都是客户端发起请求，服务端处理请求，并返回响应给客户端。



以上图形描述的就是典型的B/S架构,是一种特殊的C/S机构,就是客户端

服务端/架构.

回顾:

1. 客户端技术: html/css/javascript,jquery
2. 服务端技术: java/servlet/jsp
3. 数据传输协议:HTTP协议(超文本传输协议)

## MVC 架构模式概述

### MVC 架构模式定义

MVC全名是Model View Controller，是模型(model)－视图(view)－控制器(controller)的缩写，一种软件设计典范。其中：

M-Model 模型：职责是负责业务逻辑。

V-View 视图 ：职责是负责显示界面和用户交互（收集用户信息）。

C-Controller 控制器：是模型层M和视图层V之间的桥梁，用于控制流程。

MVC 应用场景:应用相对复杂的场景.(电商系统,企业ERP系统,…)

MVC 应用优势:

1. 可以更好实现应用解耦,提到代码可扩展性.
2. 提高代码的可重用性,可读性以及可维护性.

### 生活中的MVC

分析正规大饭店中菜单(view)，服务员(controller)，厨师(model)的角色

菜单:呈现数据(有哪些美味食品)

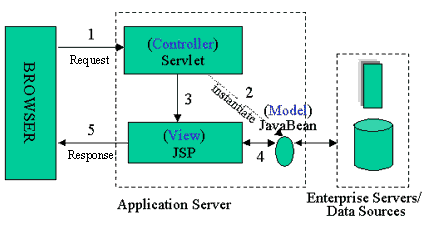
服务员:服务于客户,将菜单信息传递给厨师.

厨师:数据加工(洗菜,切菜,做菜)

记住:程序中所有的逻辑都是来自于生活

### 程序中的MVC

Java中最典型的MVC架构设计就是JSP + servlet + javabean的模式。



其中:

1. JSP 通常会充当视图层角色(借助html呈现数据)
2. Servlet 通常会充当控制层角色(主要负责流程控制)
3. JavaBean 通常会用于处理业务逻辑,数据逻辑(充当业务层对象)

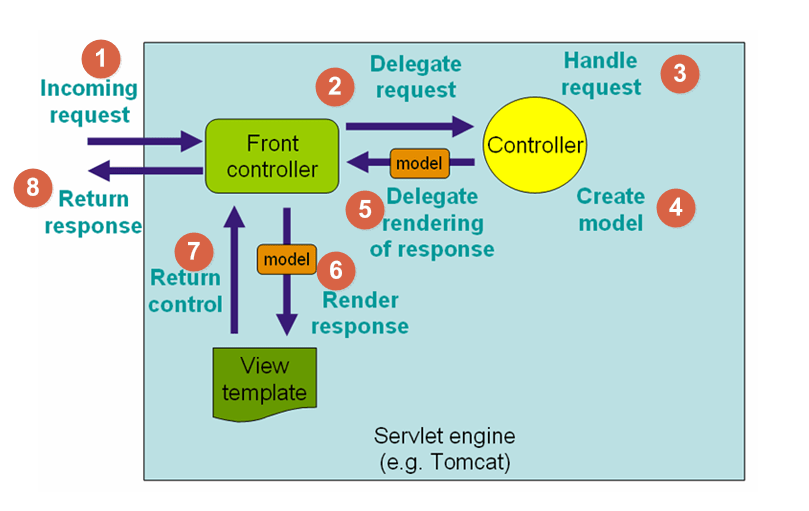
目前市场上基于这样的一种架构有一些对应的框架(FrameWork):

1. Struts1.0
2. Struts2.0 (WebWork)
3. Spring MVC
4. ……

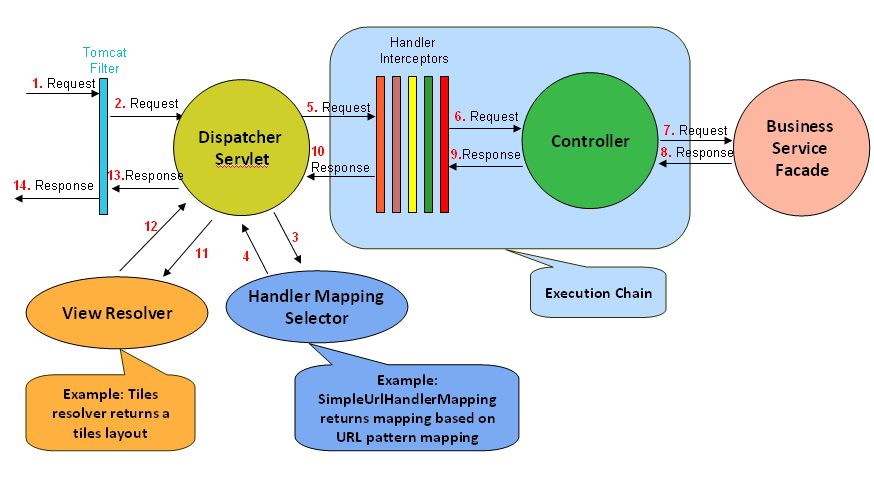
## Spring MVC 概述

Spring MVC是MVC架构模式的一种完美实现，是为了简化Java中MVC的编程过程而推出的一个Spring中的WEB应用模块。

Spring MVC 概要架构图及工作流程如下(先了解)：



Spring MVC 核心架构图及工作流程（先了解，写完项目案例再重点强化）



从Spring MVC 核心架构分析，Spring MVC围绕DispatcherServlet而计，使用HandlerMapping决定由哪个处理器处理请求。

说明：具体流程可根据断点调试法逐步跟踪尝试(暂且可先记住简要流程)

Spring MVC 中的核心组件：

1. DispatcherServlet （控制器, 请求入口）
2. HandlerMapping （控制器, 请求派发）
3. Controller （控制器, 请求处理流程）
4. ModelAndView （模型, 封装业务处理结果和视图）
5. ViewResolver（视图解析器,解析对应的视图关系）

Spring MVC 设计的优势：

1. 简化传统MVC 编程过程.
2. 更好提高代码的编码效率以及程序的可扩展性.

# Spring MVC 基本实现

## 案例分析

### 案例功能分析

浏览器中输入某个地址，直接访问资源访问地址，页面显示helloworld

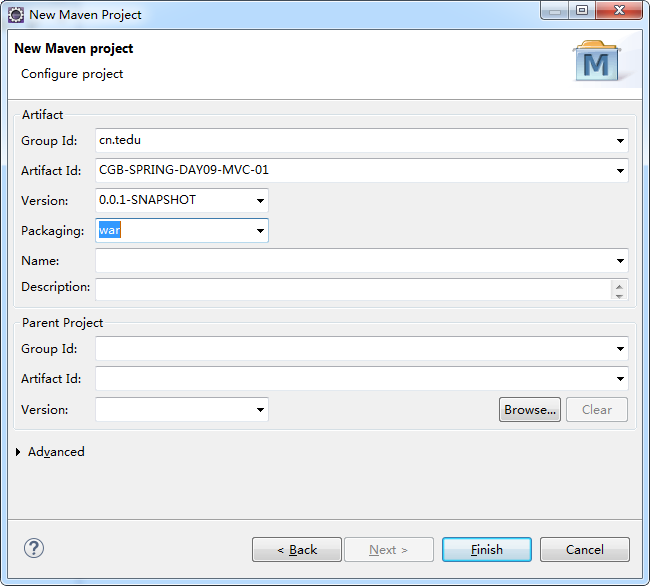
### 基本步骤分析

1. 创建spring maven web项目
2. 添加Spring MVC 依赖库(建议通过maven添加)
3. 添加Spring 配置文件spring-mvc.xml。
4. Web.xml中配置Spring MVC 前端核心处理器（DispatcherServlet）
5. 创建Spring MVC 后端处理器（Controller）
6. spring-mvc.xml文件中配置Spring后端处理器及视图解析器(ViewResolver)
7. 部署运行spring web项目(tomcat)

## 基本配置

### 创建Maven Web项目

具体maven web项目的创建过程可参考doc.tedu.cn.



项目创建完成以后

1. 右键生成部署描述符(web.xml)
2. 右键项目,选择属性,修改project facets,将java版本改成1.8.
3. 右键项目,修改targeted runtimes,选择对应的tomcat服务器.

### pom.xml基本依赖实现

添加spring-webmvc依赖,参考官网说明.

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>4.3.9.RELEASE</version>

</dependency>

假如添加好了依赖,都没有找到对应的jar依赖,先检测网络

是否是通的.假如网络是通的,还没有下载到具体的依赖,此时要

右键项目,选择maven/upate maven project/fore update…

进行maven强制更新操作.

### spring核心配置文件

项目的资源目录中直接创建spring-mvc.xml核心配置文件

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

<beans default-lazy-init=*"true"*

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

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

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

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

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

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

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

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

xmlns:jpa=*"http://www.springframework.org/schema/data/jpa"*

xsi:schemaLocation=*"*

*http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-4.3.xsd*

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

*http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx-4.3.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-4.3.xsd*

*http://www.springframework.org/schema/util*

*http://www.springframework.org/schema/util/spring-util-4.3.xsd*

*http://www.springframework.org/schema/data/jpa*

*http://www.springframework.org/schema/data/jpa/spring-jpa-1.3.xsd*

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

*http://www.springframework.org/schema/context/spring-context-4.3.xsd"* >

</beans>

### web.xml 基本配置

在web.xml文件中配置spring mvc 前端控制器。

<servlet>

<servlet-name>dispatcherServlet</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<init-param>

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

<param-value>classpath:applicationContext.xml</param-value>

</init-param>

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

</servlet>

<servlet-mapping>

<servlet-name>dispatcherServlet</servlet-name>

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

<servlet-mapping>

## 基于xml方式实现

### 创建后端控制器

通过控制器处理用于请求，例如获取请求数据，对请求数据进行类型转换等。

**public** **class** HelloController **implements** Controller{

**public** ModelAndView handleRequest(HttpServletRequest request,

HttpServletResponse response) **throws** Exception {

ModelAndView mv=**new** ModelAndView("hello");

mv.addObject("message","helloworld");

**return** mv;

}

}

### 创建后端hello.jsp页面

在hello.jsp文件中添加如下内容。

…

<body>

<h1>${message}</h1>

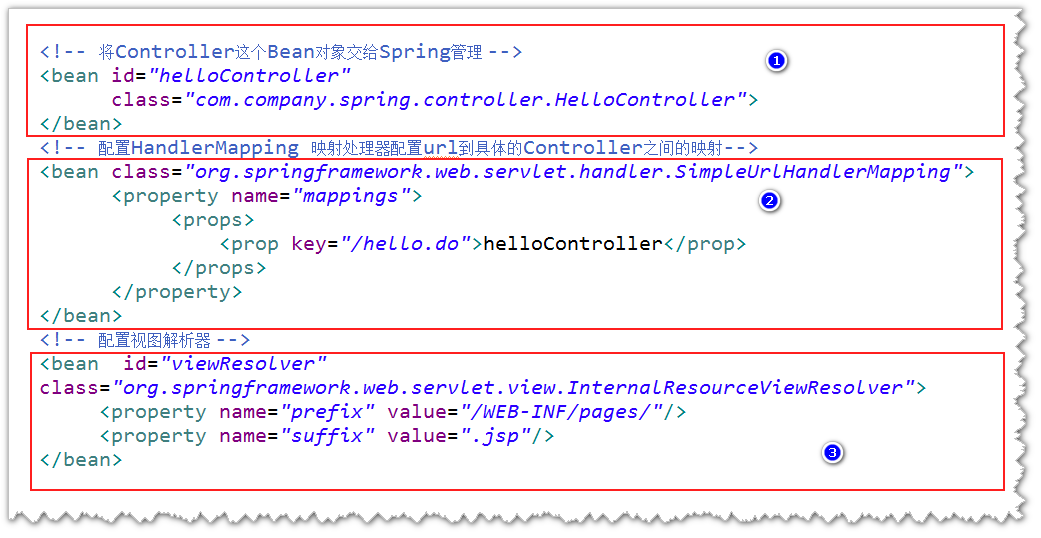
</body>

…

其中${message} 用于显示控制层封装返回的数据

### 配置controller及视图解析

打开spring-mvc配置文件配置控制器，处理映射以及视图解析操作。



具体代码如下

<!-- 将Controller这个Bean对象交给Spring管理 -->

<bean id=*"helloController"*

class=*"com.company.spring.controller.XmlHelloController"*>

</bean>

<!-- 配置HandlerMapping 映射处理器配置url到具体的Controller之间的映射-->

<bean class=

*"org.springframework.web.servlet.handler.SimpleUrlHandlerMapping"*>

<property name=*"mappings"*>

<props>

<prop key=*"/hello.do"*>helloController</prop>

</props>

</property>

</bean>

<!-- 配置视图解析器 -->

<bean id=*"viewResolver"* class=

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

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

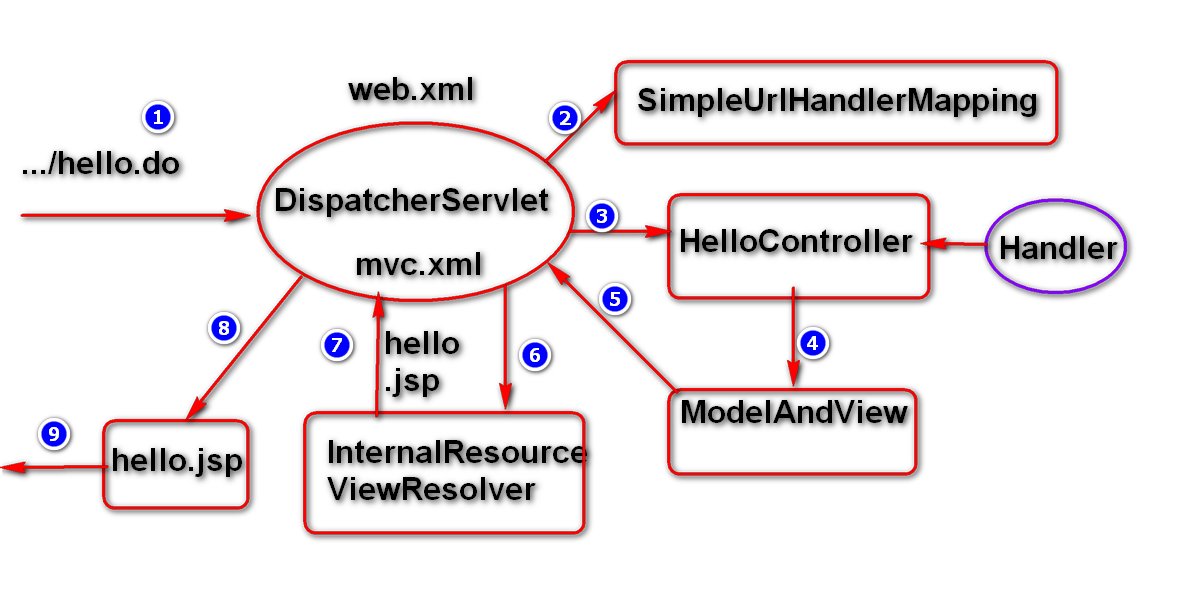
<property name=*"suffix"* value=*".jsp"*/>

</bean>

### 启动tomcat进行测试

部署项目到tomcat，启动tomcat并确定启动OK，然后在地址栏输入对应url，查看运行结果。

分析具体处理流程:



相关组件分析:

1. DispatcherServlet 核心前端处理(负责流程调度)
2. SimpleUrlHandlerMapping(记录url到handler的映射)
3. HelloController(又称之为handler,处理客户端请求)
4. ModelAndView (封装数据以及对应的视图)
5. InternalResourceViewResolver(负责视图解析,基于view找到具体的

view对象,例如hello.jsp)

## 基于注解方式实现

### 创建后端控制器

Sping2.5之前我们都是通过实现Controller接口方式定义我们的处理类，Spring2.5之后可通过@Controller和@RequestMapping注解定义我们的处理类。

@Controller

@RequestMapping("/hello/")

**public** **class** AnnotationHelloController {

@RequestMapping("doSayHello")

**public** String doSayHello(Model model) {

model.addAttribute("message", "helloworld");

**return** "hello";

}

}

说明：

通过@Controller配置这是一个控制器。

通过@RequestMapping实现Url与控制器及方法的映射

### 配置组件扫描及MVC注解应用

在基于xml实现方式上，applictionContext.xml配置文件中添加如下代码

<!-- 配置组件扫描 -->

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

<!-- 配置MVC注解扫描 -->

<mvc:annotation-driven/>

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

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

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

</bean>

# Spring MVC 请求映射

## 请求路径映射

实际项目中我们要借助@RequestMapping注解定义映射路径。其注解应用位置

* 类定义处： 提供初步的请求映射信息。
* 方法定义处： 提供进一步的细分映射信息

### 普通url映射

@RequestMapping(value={"/doSayHello", "/user/doSayWelcome"})：

多个URL路径可以映射到同一个处理器的功能处理方法。

### REST风格URL映射

REST即表述性状态传递（英文：Representational State Transfer，简称REST），是一种软件架构编码风格，是基于网络应用进行设计和开发的编码方式。可以降低开发的复杂度，提高程序的可伸缩性。例如：

@RequestMapping("/msg/{xxx}")

请求的URL可以是“/msg/hello”或“/msg/welcome”

@RequestMapping("/msg/{id}/create")：

请求的URL可以是“/users/1/create”。

@RequestMapping("/msg/{mId}/topics/{tId}")

这样也是可以的，请求的URL可以是“/users/10/topics/12”。

说明：通过@PathVariable可以提取URI模板模式中的{×××}中的×××变量。

<http://localhost:8080/>项目名/doUpdate.do?id=1

<http://localhost:8080/>项目名/doUpdate/1.do

## 请求方式映射

### 请求方式限定

项目中Controller层对象的每个方法默认可以处理任意方式的请求，假如要指定控制层方法只能处理GET或只能处理POST请求，那该如何实现呢。

借助@RequestMapping注解中的method属性指定具体的请求处理方式，例如

@RequestMapping(value=”doSaveObj”,

method=RequestMethod.POST)

public String doSaveObject(Object obj){….}

### 请求方式组合

项目中还可在控制层方法上借助@RequestMapping注解中的method属性指定使用哪几种方式处理请求。

@RequestMapping(value=”doSaveObj”,

method={RequestMethod.POST，

RequestMethod.GET})

public String doSaveObject(Object obj){….}

提示：一般浏览器只支持GET或POST方式。

## 请求参数映射

### 标准Servlet API对象

请求映射方法中可以直接使用ServletAPI 中的对象获取参数数据，例如

HttpServletRequest，HttpSession对象等，例如：

@RequestMapping(value="request",method=RequestMethod.***GET***)

@ResponseBody

**public** String withRequest(HttpServletRequest request){

System.***out***.println(request.getRequestURI());

**return** "Obtainer 'foo' query parameter value

'"+request.getParameter("gid")+"'";

}

提示：@ResponseBody注解作用：该注解作用于将Controller的方法返回的对象，通过适当的HttpMessageConverter转换为指定格式后，写入到Response对象的body数据区，使用情况：返回的数据不是Html标签的页面，而是其他数据格式的数据时，（如Json、xml等）使用；

### 基本类型参数

SpringMVC 请求一个控制层资源时，可以在对应方法中直接使用参数变量接收参数数据，但参数变量的类型要为对象类型。

@RequestMapping(value="param",method=RequestMethod.***GET***)

@ResponseBody

**public** String withParam(@RequestParam String foo) {

**return** "Obtained 'foo' query parameter value '" + foo + "'";

}

提示：@RequestParam注解用于接收请求参数中名字为foo的参数值，假如请求参数名与方法中的参数名一致，@RequestParam注解可以省略。假如不一致则可以使用@RequestParam注解定义新的参数名直接接收页面数据，然后传递给方法名，例如

@RequestMapping(value="param", method=RequestMethod.***GET***)

@ResponseBody

**public** String withParam(

@RequestParam(value="param1",required=**false**) String foo) {

**return** "Obtained 'foo' query parameter value '" + foo + "'";

}

提示：required=false表示，参数可以不存在，假如为true，参数不存在时会抛出异常。

### 请求多个参数

当请求中多个参数时可以通过在方法中定义多个参数接收参数数据，也可以利用一个javabean对象接收多个参数数据以简化多个参数变量的定义。

@RequestMapping(value="group",method=RequestMethod.***GET***)

@ResponseBody

**public** String withParamGroup(Message bean) {

**return** "Obtained parameter group " + bean;

}

提示：当使用javabean接收请求参数数据时，bean中需要有与参数名对应的set方法。

### 请求Url参数

SpringMVC请求资源路径的URL可以通过{XXX}形式指定动态的URL，动态URL中的这个可变参数的值可以直接注入到方法对应的参数中。

@RequestMapping(value="path/{var}",method=RequestMethod.***GET***)

@ResponseBody

**public** String withPathVariable(@PathVariable String var) {

**return** "Obtained 'var' path variable value '" + var + "'";

}

通过@PathVariable注解指定参数变量var获取请求url中{var}数据

### 请求头参数

当服务端要获取客户端请求头中数据信息时，可通过@RequestHeader即可将请求头中

的属性值绑定到处理方法的入参中，例如获取请求中Accept属性的值，然后传入到对应方法的参数中。

@RequestMapping(value="header", method=RequestMethod.***GET***)

@ResponseBody

**public** String withHeader(@RequestHeader String Accept) {

**return** "Obtained 'Accept' header '" + Accept + "'";

}

提示：方法中的参数名需要与请求头参数中某个某个参数名相同，具体请求头相关信息可以在浏览器控制台查看。

当应用中要获取请求中所有数据时可以在请求方法中定义一个HttpEntity<String>参数，通过此参数获取请求头及请求体中数据，例如

@RequestMapping(value="entity", method=RequestMethod.***POST***)

**public** @ResponseBody String withEntity(HttpEntity<String> entity) {

**return** "Posted request body '" + entity.getBody() + "';

headers = " + entity.getHeaders();

}

# Spring MVC响应处理

## 响应数据封装

### ModelAndView 对象

在对服务端响应数据进行封装时,可以直接在方法参数中定义一个ModelAndView类型

的参数,借助ModelAndView对象封装响应数据.

@RequestMapping("doModelAndView")

**public** ModelAndView doModelAndView(ModelAndView mv) {

//ModelAndView mv=new ModelAndView();

mv.addObject("data", "model and view");

mv.setViewName("back");

**return** mv;

}

提示:ModelAndView 对象由Spring创建,并可以将数据存储到ModelAndView对象的ModalMap类型的属性中(可参考源代码).

### Model 对象

@RequestMapping("doModel")

**public** String doModel(Model model) {

System.***out***.println("model="+model);

model.addAttribute("data", "modal");

**return** "back";

}

### Map 对象

@RequestMapping("doMap")

**public** String doMap(Map<String,Object> map) {

map.put("data", "map..");

**return** "back";

}

说明： 通过map接收页面参数时，需要使用@RequestParam注解声明

@RequestMapping("doMap")

**public** String doMap(@RequestParam Map<String,Object> map) {

map.put("data", "map..");

**return** "back";

}

## 响应数据转换

### 转换为json格式

JSON(JavaScript Object Notation):一种轻量级数据交换格式,通常用于实现客户端与服务端之间的数据传输.

例如

JSON格式的JavaScript对象:

var o1={id:10,name:’a’,age:20}

var o2=[{id:10,name:’a’,age:20},{id:20,name:’B’,age:30}]

访问json格式的javascript对象中的数据

var id1=o1.id;

var id2=o2[0].id

在实际的项目中,通常会将服务端返回的数据直接转换JSON格式字符串,在客户端

将这个字符串再转换为javascript对象,然后从这个对象直接取数据.

例如:

@RequestMapping("doJSON")

@ResponseBody

**public** Map<String,Object> doJSON() {

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

map.put("id", 10);

map.put("msg","helloworld");

**return** map;

}

转换为json格式时项目中需要添加json相关API依赖，例如使用jackson第三方库

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.8.5</version>

</dependency>

其他案例

@RequestMapping("doBeanToJson")

@ResponseBody

public Message doBeanToJson(){

Message msg=new Message();

msg.setId(100);

msg.setContent("msg-content-1");

return msg;

}

@RequestMapping("doListToJson")

@ResponseBody

**public** List<Map<String,Object>> doListToJson(){

List<Map<String,Object>> list=

**new** ArrayList<>();

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

map.put("id", 10);

map.put("content", "content-A");

list.add(map);

map=**new** HashMap<>();

map.put("id", 20);

map.put("content", "content-B");

list.add(map);

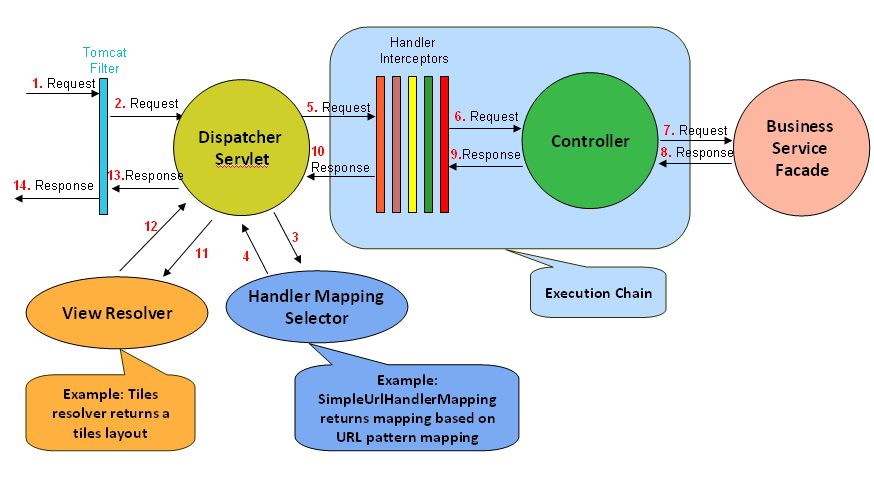
**return** list;

}

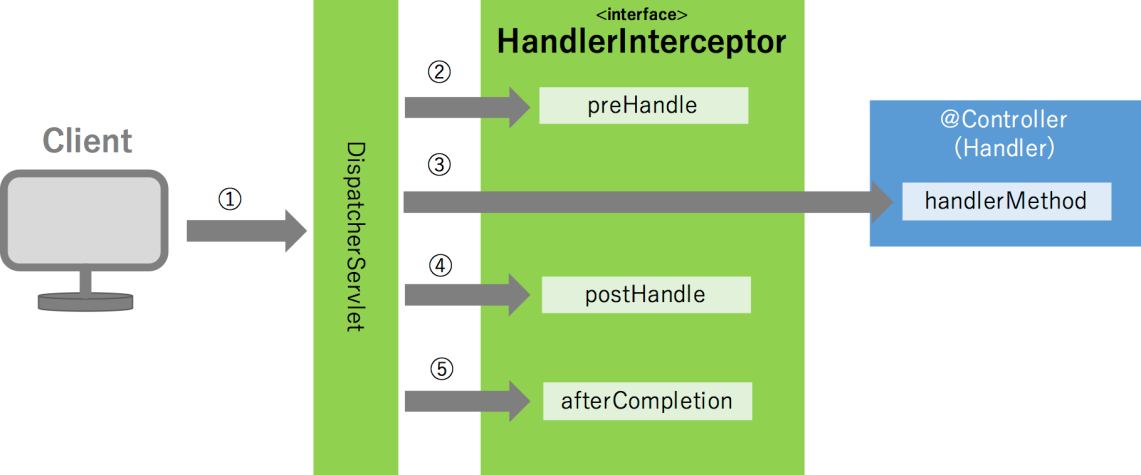
# Spring MVC 拦截器

## Spring MVC拦截器概述

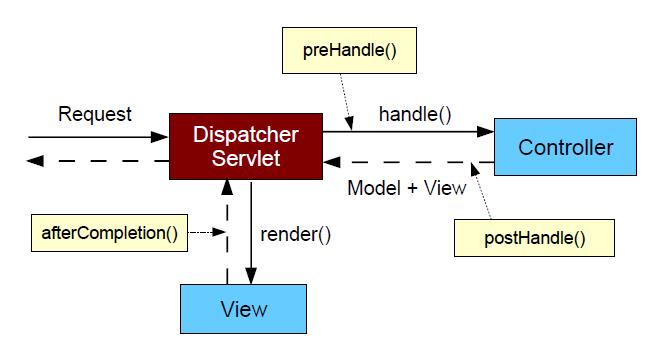
* Handler Interceptors of Sping MVC



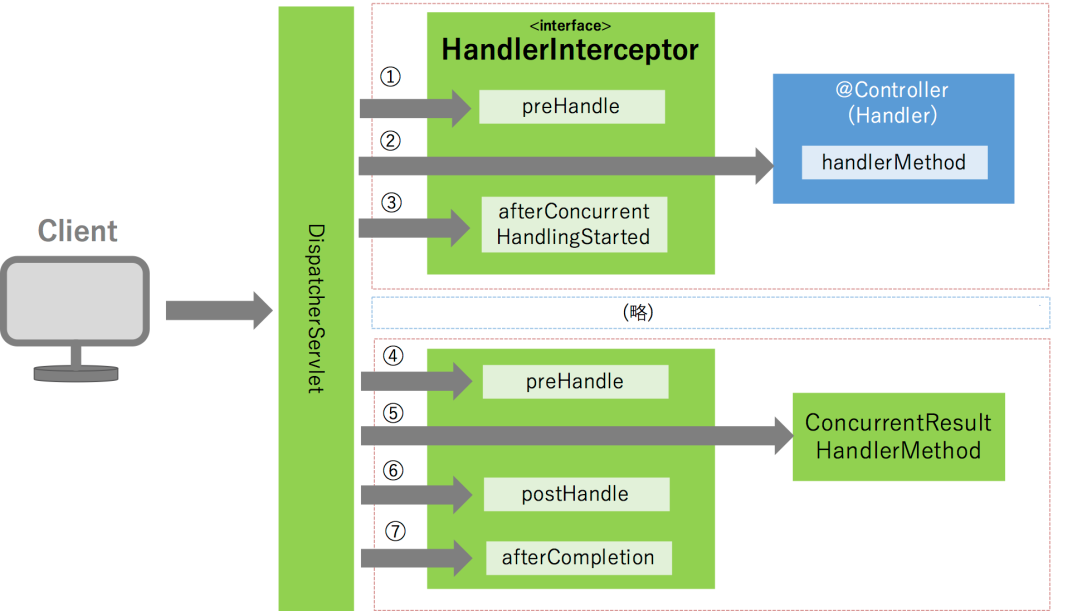
*Interceptors working with the HandlerMapping on the framework*



* *prehandle()* – *called before the actual handler is executed, but the view is not generated yet*
* *postHandle()* – *called after the handler is executed*
* *afterCompletion() – called after the complete request has finished and view was generated*



* Interceptors Chain



## Spring 拦截器应用

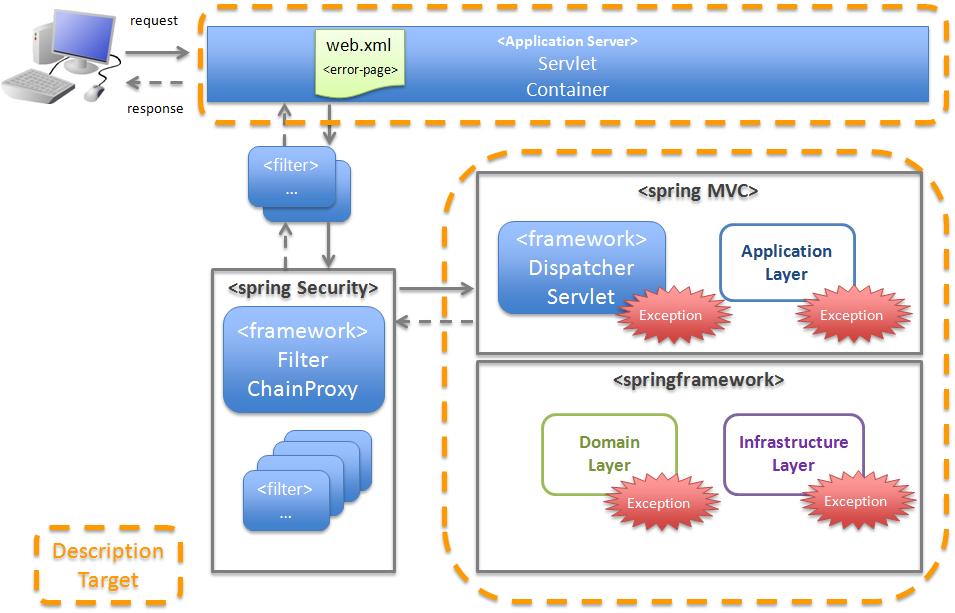
编写自定义的Interceptor实现HandlerInterceptor接口或继承

HandlerInterceptorAdapter，重写相关方法处理具体业务。

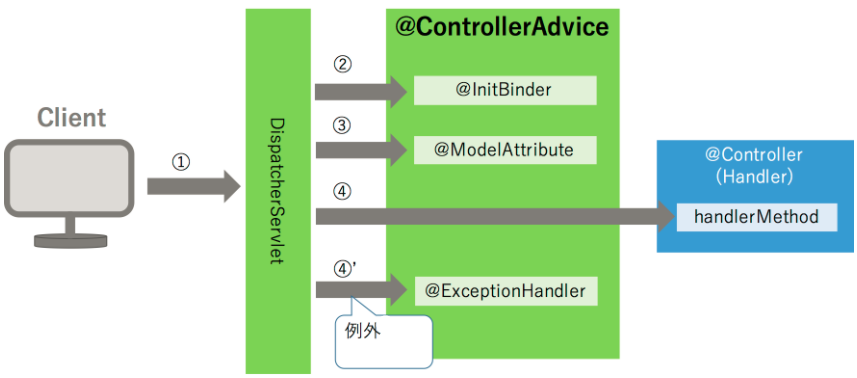
# Spring MVC 异常处理

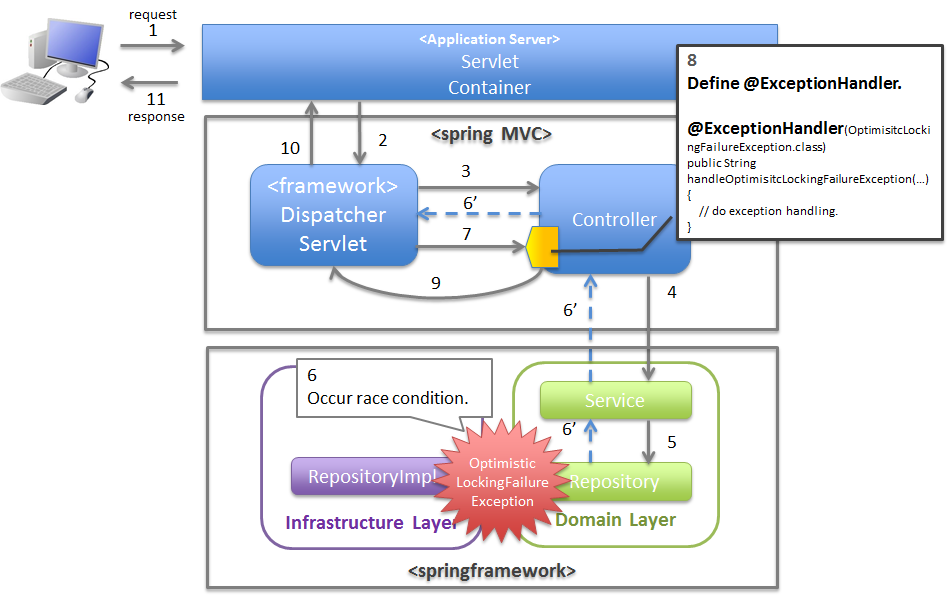
## Spring MVC 异常概述

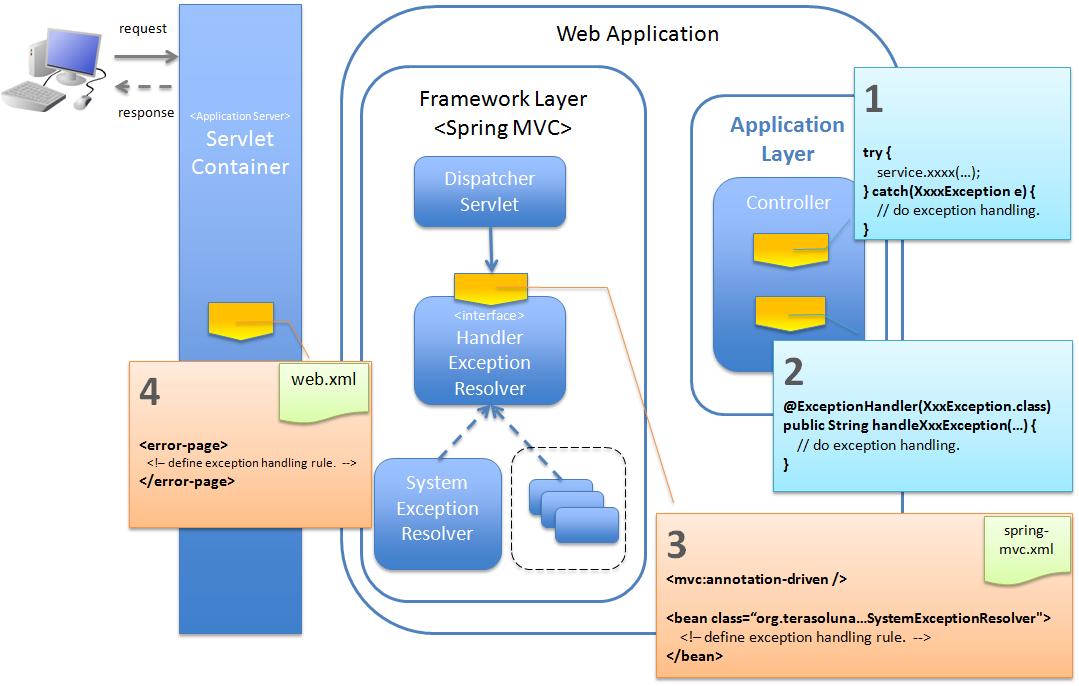
本小节主要是阐述spring mvc 边界出现异常以后的处理方式。



## Spring MVC 异常处理







# 总结

## 重点和难点分析

|  |
| --- |
|  |

## 常见FAQ

|  |
| --- |
|  |