

一套吃透 Spring 系列

郝老师



黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌



目录

Contents

- ◆ SpringMVC简介
- ◆ SpringMVC的请求处理
- ◆ SpringMVC的响应处理
- ◆ SpringMVC的拦截器
- ◆ SpringMVC的全注解开发
- ◆ SpringMVC的组件原理剖析
- ◆ SpringMVC的异常处理机制



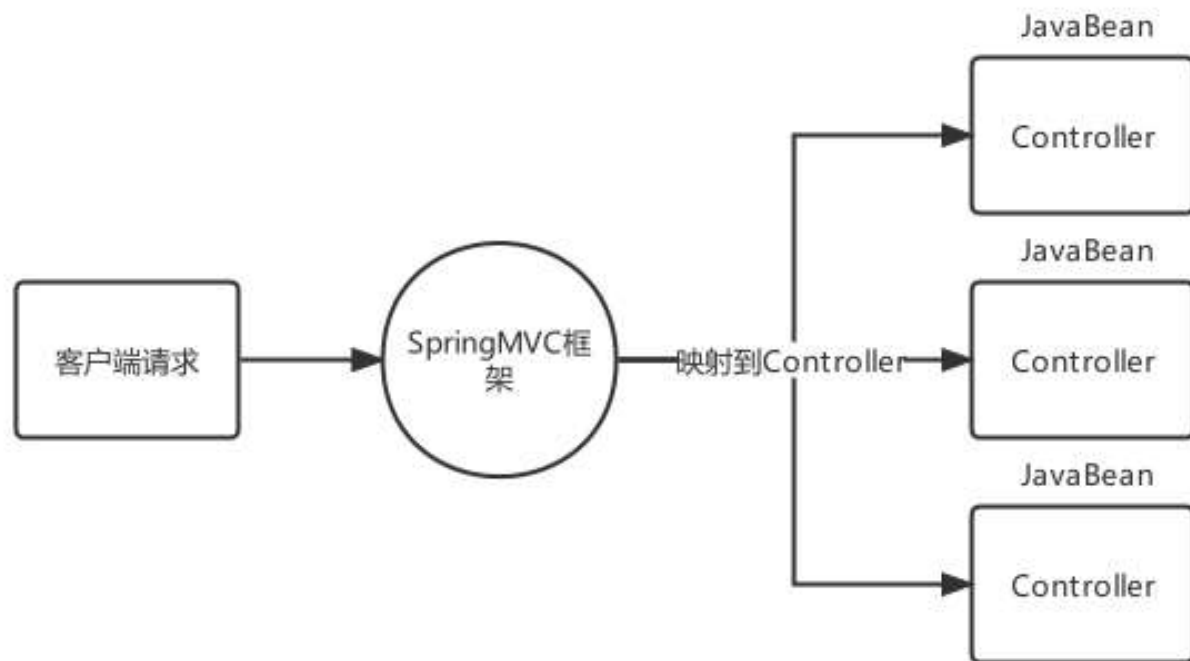
SpringMVC简介

- SpringMVC概述
- SpringMVC快速入门
- Controller中访问容器中的Bean
- SpringMVC关键组件浅析

SpringMVC简介

- SpringMVC概述

SpringMVC是一个基于Spring开发的MVC轻量级框架，Spring3.0后发布的组件，SpringMVC和Spring可以无缝整合，使用DispatcherServlet作为前端控制器，且内部提供了处理器映射器、处理器适配器、视图解析器等组件，可以简化JavaBean封装，Json转化、文件上传等操作。



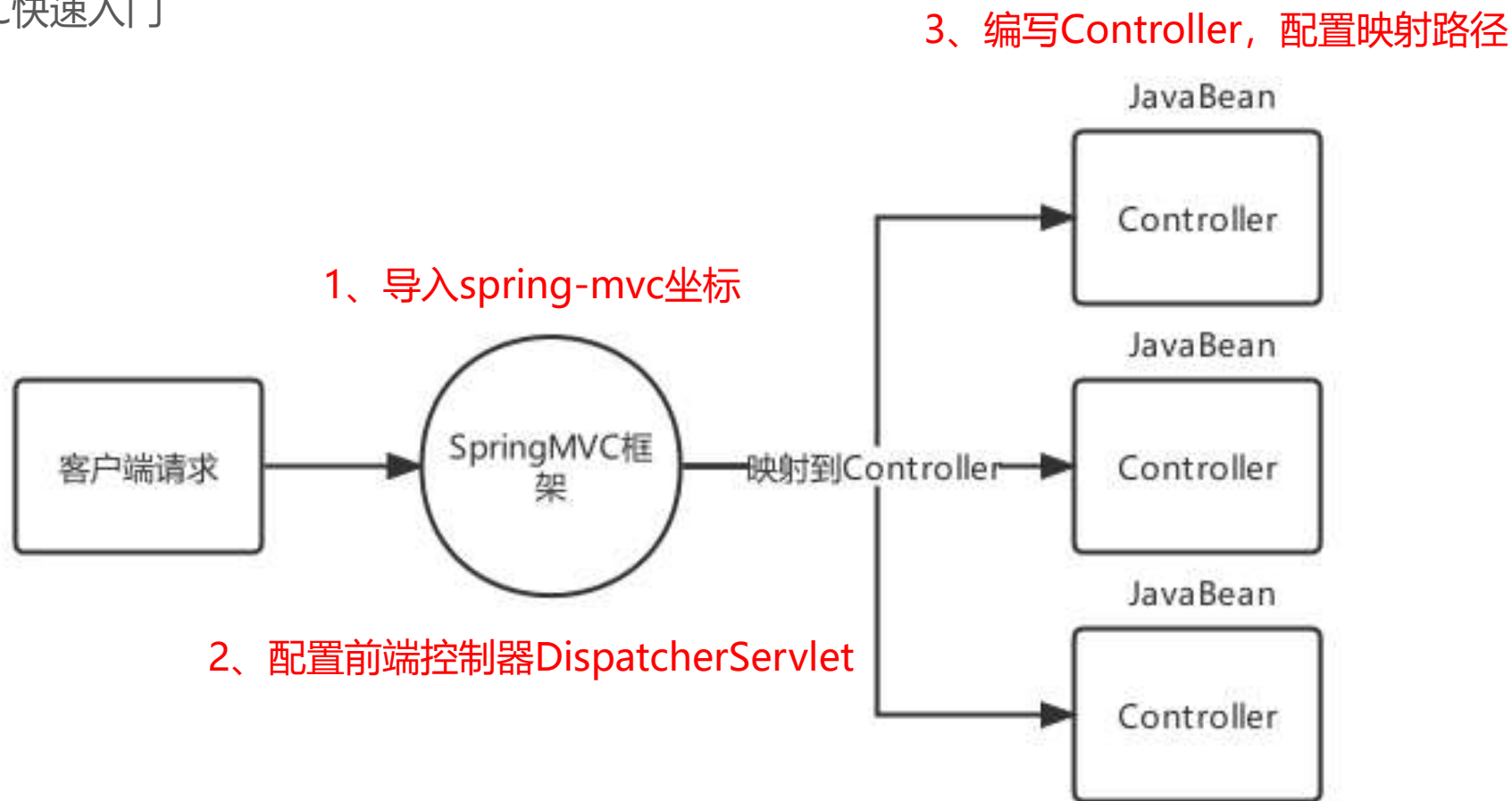


SpringMVC简介

- SpringMVC概述
- SpringMVC快速入门
- Controller中访问容器中的Bean
- SpringMVC关键组件浅析

SpringMVC简介

- SpringMVC快速入门



SpringMVC简介

- SpringMVC快速入门

导入Spring整合SpringMVC的坐标

```
<dependency>  
    <groupId>org.springframework</groupId>  
    <artifactId>spring-webmvc</artifactId>  
    <version>5.3.7</version>  
</dependency>
```

SpringMVC简介

- SpringMVC快速入门

编写一个控制器Controller，配置映射信息

```
@Controller
public class UserController {
    @RequestMapping("/show")
    public String show(){
        System.out.println("show 执行....");
        //视图跳转到index.jsp
        return "/index.jsp";
    }
}
```


SpringMVC简介

- SpringMVC快速入门

在web.xml中配置SpringMVC的前端控制器ServletDispatcher

```
<servlet>
  <servlet-name>DispatcherServlet</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <!--指定springMVC配置文件位置-->
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:spring-mvc.xml</param-value>
  </init-param>
  <!--服务器启动就创建-->
  <load-on-startup>2</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>DispatcherServlet</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
```

SpringMVC简介

- SpringMVC快速入门

创建springMVC的核心配置文件 spring-mvc.xml，并配置组件扫描web层

```
<?xml version="1.0" encoding="UTF-8"?>
  <beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd
      http://www.springframework.org/schema/mvc
      http://www.springframework.org/schema/mvc/spring-mvc.xsd">
    <!-- 组件扫描web层 -->
    <context:component-scan base-package="com.itheima.controller"/>

  </beans>
```



SpringMVC简介

- SpringMVC概述
- SpringMVC快速入门
- Controller中访问容器中的Bean
- SpringMVC关键组件浅析

SpringMVC简介

- Controller中访问容器中的Bean

DispatcherServlet在进行初始化时，加载的spring-mvc.xml配置文件创建的SpringMVC容器，那么web层Controller被扫描进入到了容器中，而之前Spring容器中的Service是否可以获取到呢？下面搭建Spring的web环境进行验证

创建一个applicationContext.xml文件

```
<!-- 组件扫描非web层 -->
<context:component-scan base-package="com.itheima">
    <!--排除com.itheima包下使用@Controller注解的类-->
    <context:exclude-filter type="annotation"
expression="org.springframework.stereotype.Controller"/>
</context:component-scan>
```

SpringMVC简介

- Controller中访问容器中的Bean

在web.xml中配置ContextLoaderListener

```
<!--配置ContextLoaderListener-->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext.xml</param-value>
</context-param>
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

编写UserService和UserServiceImpl

```
public interface UserService {
    public void show();}
@Service("userService")
public class UserServiceImpl implements UserService {
    @Override
    public void show() {
        System.out.println("UserServiceImpl show running ...");
    }
}
```

SpringMVC简介

- Controller中访问容器中的Bean

修改UserController，从Spring容器中匹配Service进行注入

```
@Controller
public class UserController {

    @Autowired
    private UserService userService;

    @RequestMapping("/show")
    public String show(){
        System.out.println("show 执行....");
        //调用userService的show方法
        userService.show();
        //视图跳转到index.jsp
        return "/index.jsp";
    }
}
```



SpringMVC简介

- SpringMVC概述
- SpringMVC快速入门
- Controller中访问容器中的Bean
- SpringMVC关键组件浅析

SpringMVC简介

- SpringMVC关键组件浅析

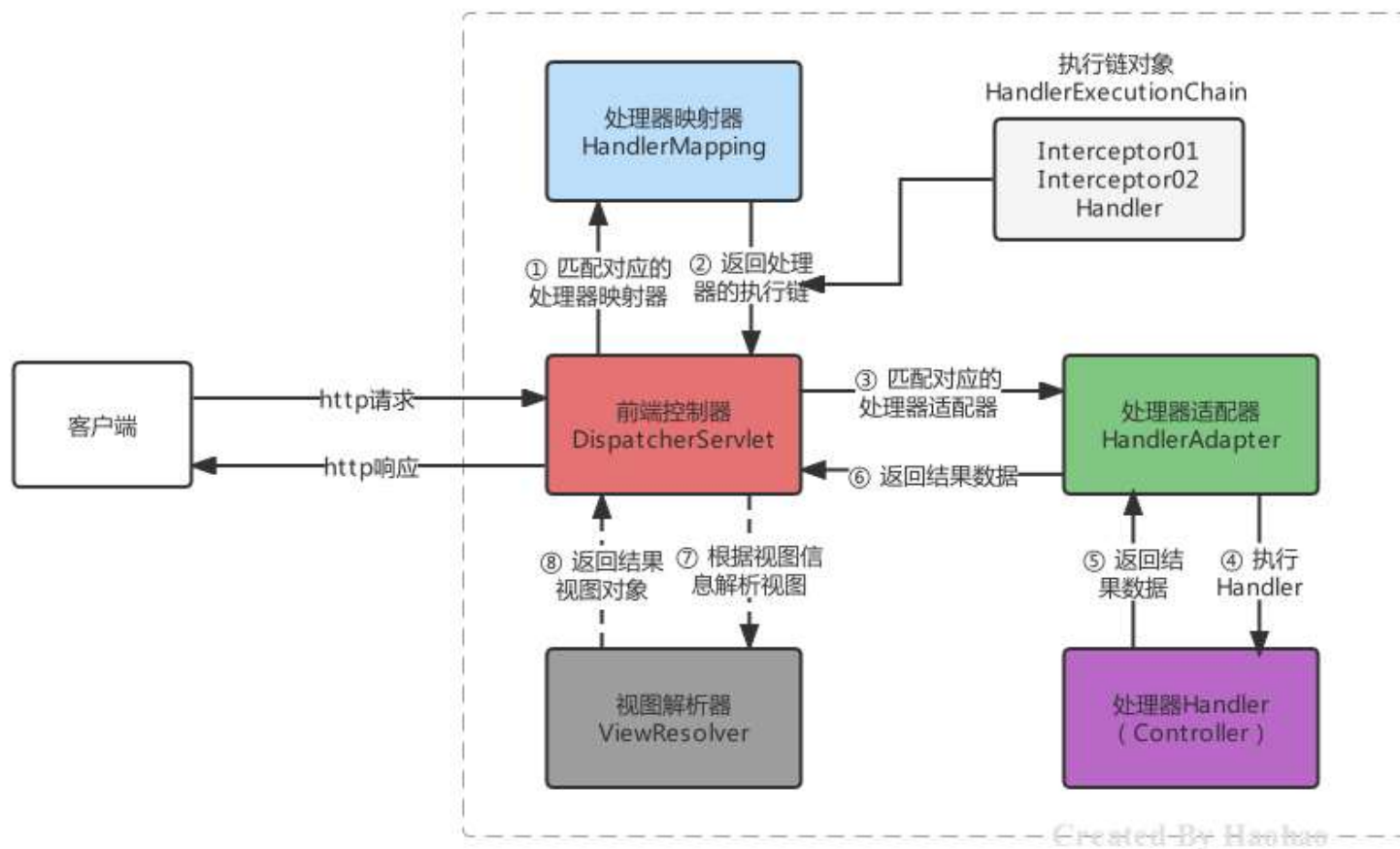
上面已经完成的快速入门的操作，也在不知不觉中完成的Spring和SpringMVC的整合，我们只需要按照规则去定义Controller和业务方法就可以。但是在这个过程中，肯定是很多核心功能类参与到其中，这些核心功能类，一般称为组件。当请求到达服务器时，是哪个组件接收的请求，是哪个组件帮我们找到的Controller，是哪个组件帮我们调用的方法，又是哪个组件最终解析的视图？

组件	描述	常用组件
处理器映射器：HandlerMapping	匹配映射路径对应的Handler，返回可执行的处理器链对象HandlerExecutionChain对象	RequestMappingHandlerMapping
处理器适配器：HandlerAdapter	匹配HandlerExecutionChain对应的适配器进行处理器调用，返回视图模型对象	RequestMappingHandlerAdapter
视图解析器：ViewResolver	对视图模型对象进行解析	InternalResourceViewResolver

SpringMVC简介

- SpringMVC关键组件浅析

先简单了解一下以上三个重要组件的关系



SpringMVC简介

- SpringMVC关键组件浅析

SpringMVC的默认组件，SpringMVC 在前端控制器 DispatcherServlet加载时，就会进行初始化操作，在进行初始化时，就会加载SpringMVC默认指定的一些组件，这些默认组件配置在 DispatcherServlet.properties 文件中，该文件存在与spring-webmvc-5.3.7.jar包下的 org\springframework\web\servlet\DispatcherServlet.properties

```
org.springframework.web.servlet.HandlerMapping=org.springframework.web.servlet.handler.BeanNameUrl  
HandlerMapping,\n    org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping,\n    org.springframework.web.servlet.function.support.RouterFunctionMapping
```

```
org.springframework.web.servlet.HandlerAdapter=org.springframework.web.servlet.mvc.HttpRequestHand  
lerAdapter,\n    org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,\n    org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter,\n    org.springframework.web.servlet.function.support.HandlerFunctionAdapter
```

```
org.springframework.web.servlet.ViewResolver=org.springframework.web.servlet.view.InternalResource  
ViewResolver
```

#省略其他代码

SpringMVC简介

- SpringMVC关键组件浅析

这些默认的组件是在DispatcherServlet中进行初始化加载的，在DispatcherServlet中存在集合存储着这些组件，SpringMVC的默认组件会在 DispatcherServlet 中进行维护，但是并没有存储在与SpringMVC的容器中

```
public class DispatcherServlet extends FrameworkServlet {  
    // 存储处理器映射器  
    private List<HandlerMapping> handlerMappings;  
    // 存储处理器适配器  
    private List<HandlerAdapter> handlerAdapters;  
    // 存储视图解析器  
    private List<ViewResolver> viewResolvers;  
  
    // ... 省略其他代码 ...  
}
```

SpringMVC简介

- SpringMVC关键组件浅析

配置组件代替默认组件，如果不想使用默认组件，可以将替代方案使用Spring Bean的方式进行配置，例如，在spring-mvc.xml中配置RequestMappingHandlerMapping

```
<bean  
class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping"/>
```

当我们在Spring容器中配置了HandlerMapping，则就不会在加载默认的HandlerMapping策略了，原理比较简单，DispatcherServlet 在进行HandlerMapping初始化时，先从SpringMVC容器中找是否存在HandlerMapping，如果存在直接取出容器中的HandlerMapping，在存储到 DispatcherServlet 中的handlerMappings集合中去。



SpringMVC简介

- SpringMVC概述
- SpringMVC快速入门
- Controller中访问容器中的Bean
- SpringMVC关键组件浅析



目录

Contents

- ◆ SpringMVC简介
- ◆ SpringMVC的请求处理
- ◆ SpringMVC的响应处理
- ◆ SpringMVC的拦截器
- ◆ SpringMVC的全注解开发
- ◆ SpringMVC的组件原理剖析
- ◆ SpringMVC的异常处理机制



SpringMVC的请求处理

- 请求映射路径的配置
- 请求数据的接收
- Javaweb常用对象获取
- 请求静态资源
- 注解驱动 <mvc:annotation-driven> 标签

SpringMVC的请求处理

- 请求映射路径的配置

配置映射路径，映射器处理器才能找到Controller的方法资源，目前主流映射路径配置方式就是@RequestMapping

相关注解	作用	使用位置
@RequestMapping	设置控制器方法的访问资源路径，可以接收任何请求	方法和类上
@GetMapping	设置控制器方法的访问资源路径，可以接收GET请求	方法和类上
@PostMapping	设置控制器方法的访问资源路径，可以接收POST请求	方法和类上

SpringMVC的请求处理

- 请求映射路径的配置

@RequestMapping注解，主要使用在控制器的方法上，用于标识客户端访问资源路径，常用的属性有value、path、method、headers、params等。当@RequestMapping只有一个访问路径需要指定时，使用value属性、path属性或省略value和path，当有多个属性时，value和path不能省略

```
@RequestMapping(value = "/show") //使用value属性指定一个访问路径
public String show() {}

@RequestMapping(value = {"/show", "/haohao", "/abc"}) //使用value属性指定多个访问路径
public String show() {}

@RequestMapping(path = "/show") //使用path属性指定一个访问路径
public String show() {}

@RequestMapping(path = {"/show", "/haohao", "/abc"}) //使用path属性指定多个访问路径
public String show() {}

@RequestMapping("/show") //如果只设置访问路径时，value和path可以省略
public String show() {}
@RequestMapping({"/show", "/haohao", "/abc"})
public String show() {}
```

SpringMVC的请求处理

- 请求映射路径的配置

当@RequestMapping 需要限定访问方式时，可以通过method属性设置

```
//请求地址是/show,且请求方式必须是POST才能匹配成功  
@RequestMapping(value = "/show",method = RequestMethod.POST)  
public String show() {}
```

method的属性值是一个枚举类型，源码如下：

```
public enum RequestMethod {  
    GET,  
    HEAD,  
    POST,  
    PUT,  
    PATCH,  
    DELETE,  
    OPTIONS,  
    TRACE;  
    private RequestMethod() {  
    }  
}
```

SpringMVC的请求处理

- 请求映射路径的配置

@GetMapping, 当请求方式是GET时, 我们可以使用@GetMapping替代@RequestMapping

```
@GetMapping("/show")  
public String show() {}
```

@PostMapping, 当请求方式是POST时, 我们可以使用@PostMapping替代@RequestMapping

```
@PostMapping("/show")  
public String show() {}
```

SpringMVC的请求处理

- 请求映射路径的配置

@RequestMapping 在类上使用，@RequestMapping、@GetMapping、@PostMapping还可以使用在Controller类上，使用在类上后，该类所有方法都公用该@RequestMapping设置的属性，访问路径则为类上的映射地址+方法上的映射地址，例如：

```
@Controller
@RequestMapping("/xxx")
public class UserController implements ApplicationContextAware, ServletContextAware {

    @GetMapping("/aaa")
    public ModelAndView aaa(HttpServletResponse response) throws IOException,
        ModelAndViewDefiningException {

        return null;
    }
}
```

此时的访问路径为：/xxx/aaa



SpringMVC的请求处理

- 请求映射路径的配置
- 请求数据的接收
- Javaweb常用对象获取
- 请求静态资源
- 注解驱动 <mvc:annotation-driven> 标签

SpringMVC的请求处理

- 请求数据的接收

接收普通请求数据，当客户端提交的数据是普通键值对形式时，直接使用同名形参接收即可

```
username=haohao&age=35
```

```
@GetMapping("/show")
public String show(String username, int age){
    System.out.println(username+"==" +age);
    return "/index.jsp";
}
```

SpringMVC的请求处理

- 请求数据的接收

接收普通请求数据，当请求参数有特殊格式数据，如日期时

```
username=haohao&age=35&birthday=1986/01/01
```

```
@GetMapping("/show")
public String show(String username, int age, Date birthday) {
    System.out.println(username+"=="+age+"=="+birthday);
    return "/index.jsp";
}
```

Date可以正常接收，因为Spring内置的类型解析器，可以识别的日期格式是 yyyy/MM/dd，但是如果我们提交其他格式，例如：yyyy-MM-dd 时，类型转换会报错，如下：

```
username=haohao&age=35&birthday=1986-01-01
```

HTTP Status 400 – Bad Request

SpringMVC的请求处理

- 请求数据的接收

解决方案，使用@DateTimeFormat 指定日期格式，修改UserController如下：

```
@GetMapping("/show")
public String show(String username, int age, @DateTimeFormat(pattern = "yyyy-MM-dd") Date
birthday) {
    System.out.println(username+"=="+age+"=="+birthday);
    return "/index.jsp";
}
```


SpringMVC的请求处理

- 请求数据的接收

接收普通请求数据，当请求参数的名称与方法参数名不一致时，可以使用@RequestParam注解进行标注

```
username=haohao&age=35
```

```
@GetMapping("/show")
public String show(@RequestParam(name = "username", required = true) String name, int age) {
    System.out.println(name+"==" + age);
    return "/index.jsp";
}
```

SpringMVC的请求处理

- 请求数据的接收

接收实体JavaBean属性数据，单个JavaBean数据：提交的参数名称只要与Java的属性名一致，就可以进行自动封装

```
username=haohao&age=35&hobbies=eat&hobbies=sleep
```

```
public class User {  
    private String username;  
    private Integer age;  
    private String[] hobbies;  
    private Date birthday;  
    private Address address;  
    //... 省略get和set方法 ...  
}
```

```
@GetMapping("/show")  
public String show(User user){  
    System.out.println(user);  
    return "/index.jsp";  
}
```

SpringMVC的请求处理

- 请求数据的接收

接收实体JavaBean属性数据，嵌套JavaBean数据：提交的参数名称用.去描述嵌套对象的属性关系即可

```
username=haohao&address.city=tianjin&address.area=jinghai
```

同上，JavaBean中如果有日期数据，且日期格式不符合yyyy/MM/dd时，需要通过@DateTimeFormat指定日期格式

```
username=haohao&birthday=1986-11-11
```

```
public class User {  
    private String username;  
    private Integer age;  
    private String[] hobbies;  
    @DateTimeFormat(pattern = "yyyy-MM-dd")  
    private Date birthday;  
    private Address address;  
    //... 省略get和set方法 ...  
}
```

SpringMVC的请求处理

- 请求数据的接收

接收数组或集合数据，客户端传递多个同名参数时，可以使用数组接收

```
hobbies=eat&hobbies=sleep
```

```
@GetMapping("/show")
public String show(String[] hobbies) {
    for (String hobby : hobbies) {
        System.out.println(hobby);
    }
    return "/index.jsp";
}
```

接收数组或集合数据，客户端传递多个同名参数时，也可以使用单列集合接收，但是需要使用@RequestParam告知框架传递的参数是要同名设置的，不是对象属性设置的

```
@GetMapping("/show")
public String show(@RequestParam List<String> hobbies) {
    for (String hobby : hobbies) {
        System.out.println(hobby);
    }
    return "/index.jsp";
}
```

SpringMVC的请求处理

- 请求数据的接收

接收数组或集合数据，客户端传递多个不同命参数时，也可以使用Map<String,Object> 进行接收，同样需要用@RequestParam 进行修饰

```
username=haohao&age=18
```

```
@PostMapping("/show")
public String show(@RequestParam Map<String,Object> params){
    params.forEach((key,value)->{
        System.out.println(key+"==" +value);
    });
    return "/index.jsp";
}
```

SpringMVC的请求处理

- 请求数据的接收

接收Json数据格式数据，Json数据都是以请求体的方式提交的，且不是原始的键值对格式的，所以我们要使用@RequestBody注解整体接收该数据。

```
{
  "username": "haohao",
  "age": 18,
  "hobbies": ["eat", "sleep"],
  "birthday": "1986-01-01",
  "address": {
    "city": "tj",
    "area": "binhai"
  }
}
```

```
@PostMapping("/show6")
public String show6(@RequestBody String body) {
    System.out.println(body);
    return "/index.jsp";
}
```

SpringMVC的请求处理

- 请求数据的接收

使用Json工具（jackson）将Json格式的字符串转化为JavaBean进行操作

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.9.0</version>
</dependency>
```

```
@PostMapping("/show")
public String show(@RequestBody String body) throws IOException {
    System.out.println(body);
    //获得ObjectMapper
    ObjectMapper objectMapper = new ObjectMapper();
    //将json格式字符串转化成指定的User
    User user = objectMapper.readValue(body, User.class);
    System.out.println(user);
    return "/index.jsp";
}
```

SpringMVC的请求处理

- 请求数据的接收

配置RequestMappingHandlerAdapter，指定消息转换器，就不用手动转换json格式字符串了

```
<bean
class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">
    <property name="messageConverters">
        <list>
            <bean
class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter"/>
        </list>
    </property>
</bean>
```

```
@PostMapping("/show")
public String show(@RequestBody User user) {
    System.out.println(user);
    return "/index.jsp";
}
```


SpringMVC的请求处理

- 请求数据的接收

接收Json数据格式数据，使用Map接收json格式字符串

```
@PostMapping("/show")
public String show(@RequestBody Map map) {
    System.out.println(map);
    return "/index.jsp";
}
```

SpringMVC的请求处理

- 请求数据的接收

接收Restful风格数据

什么是Rest风格?

Rest (Representational State Transfer) 表象化状态转变 (表述性状态转变), 在2000年被提出, 基于HTTP、URI、xml、JSON等标准和协议, 支持轻量级、跨平台、跨语言的架构设计。是Web服务的一种新网络应用程序的设计风格 and 开发方式。

SpringMVC的请求处理

- 请求数据的接收

接收Restful风格数据

Restful风格的请求，常见的规则有如下三点：

- 用URI表示某个模块资源，资源名称为名词；

模块	URI资源
用户模块 user	http://localhost/user
商品模块 product	http://localhost/product
账户模块 account	http://localhost/account
日志模块 log	http://localhost/log

SpringMVC的请求处理

- 请求数据的接收

- 用请求方式表示模块具体业务动作，例如：GET表示查询、POST表示插入、PUT表示更新、DELETE表示删除

URI资源	请求方式	参数	解释
http://localhost/user/100	GET	存在URL地址中：100	查询id=100的User数据
http://localhost/user	POST	存在请求体中Json： {"username":"haohao","age":18}	插入User数据
http://localhost/user	PUT	存在请求体中Json： {"id":100,"username":"haohao","age":18}	修改id=100的User数据
http://localhost/user/100	DELETE	存在URL地址中：100	删除id=100的User数据
http://localhost/product/5	GET	存在URL地址中：5	查询id=5的Product数据
http://localhost/product	POST	存在请求体中Json：{"proName":"小米手机", "price":1299}	插入Product数据
http://localhost/product	PUT	存在请求体中Json：{"id":5,"proName":"小米手机", "price":1299}	修改id=5的Product数据
http://localhost/product/5	DELETE	存在URL地址中：5	删除id=5的Product数据

SpringMVC的请求处理

- 请求数据的接收

- 用HTTP响应状态码表示结果，国内常用的响应包括三部分：状态码、状态信息、响应数据

```
{
  "code":200,
  "message":"成功",
  "data":{
    "username":"haohao",
    "age":18
  }
}

{
  "code":300,
  "message":"执行错误",
  "data":"","
}
```

SpringMVC的请求处理

- 请求数据的接收

接收Restful风格数据，Restful请求数据一般会在URL地址上携带，可以使用注解 @PathVariable(占位符参数名称)

```
http://localhost/user/100
```

```
@PostMapping("/user/{id}")
public String findUserById(@PathVariable("id") Integer id){
    System.out.println(id);
    return "/index.jsp";
}
```

请求URL资源地址包含多个参数情况

```
http://localhost/user/haohao/18
```

```
@PostMapping("/user/{username}/{age}")
public String findUserByUsernameAndAge(@PathVariable("username") String
username,@PathVariable("age") Integer age){
    System.out.println(username+"==" +age);
    return "/index.jsp";
}
```

SpringMVC的请求处理

- 请求数据的接收

接收文件上传的数据，文件上传的表单需要一定的要求，如下：

- 表单的提交方式必须是POST
- 表单的enctype属性必须是multipart/form-data
- 文件上传项需要有name属性

```
<form action="" enctype="multipart/form-data">  
    <input type="file" name="myFile">  
</form>
```

SpringMVC的请求处理

- 请求数据的接收

服务器端，由于映射器适配器需要文件上传解析器，而该解析器默认未被注册，所以手动注册

```
<!--配置文件上传解析器，注意：id的名字是固定写法-->
<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <property name="defaultEncoding" value="UTF-8"/><!--文件的编码格式 默认是ISO8859-1-->
    <property name="maxUploadSizePerFile" value="1048576"/><!--上传的每个文件限制的大小 单位字节-->
    <property name="maxUploadSize" value="3145728"/><!--上传文件的总大小-->
    <property name="maxInMemorySize" value="1048576"/><!--上传文件的缓存大小-->
</bean>
```

而CommonsMultipartResolver底层使用的Apache的是Common-fileupload等工具API进行的文件上传

```
<dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
    <version>1.4</version>
</dependency>
```

```
<dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.7</version>
</dependency>
```


SpringMVC的请求处理

- 请求数据的接收

使用MultipartFile类型接收上传文件

```
@PostMapping("/fileUpload")
public String fileUpload(@RequestBody MultipartFile myFile) throws IOException {
    System.out.println(myFile);
    //获得上传的文件的流对象
    InputStream inputStream = myFile.getInputStream();
    //使用commons-io存储到C:\haohao\abc.txt位置
    FileOutputStream outputStream = new
FileOutputStream("C:\\Users\\haohao\\"+myFile.getOriginalFilename());
    IOUtils.copy(inputStream,outputStream);
    //关闭资源
    inputStream.close();
    outputStream.close();

    return "/index.jsp";
}
```

如果进行多文件上传的话，则使用MultipartFile数组即可

SpringMVC的请求处理

- 请求数据的接收

接收HttpRequest头数据，接收指定名称的请求头

```
@GetMapping("/headers")
public String headers(@RequestHeader("Accept-Encoding") String acceptEncoding) {
    System.out.println("Accept-Encoding:" + acceptEncoding);
    return "/index.jsp";
}
```

接收所有的请求头信息

```
@GetMapping("/headersMap")
public String headersMap(@RequestHeader Map<String, String> map) {
    map.forEach((k, v) -> {
        System.out.println(k + ":" + v);
    });
    return "/index.jsp";
}
```

SpringMVC的请求处理

- 请求数据的接收

获得客户端携带的Cookie数据

```
@GetMapping("/cookies")  
public String cookies(@CookieValue(value = "JSESSIONID",defaultValue = "") String jsessionid){  
    System.out.println(jsessionid);  
    return "/index.jsp";  
}
```

SpringMVC的请求处理

- 请求数据的接收

获得转发Request域中数据，在进行资源之间转发时，有时需要将一些参数存储到request域中携带给下一个资源

```
@GetMapping("/request1")
public String request1(HttpServletRequest request) {
    // 存储数据
    request.setAttribute("username", "haohao");
    return "forward:/request2";
}

@GetMapping("/request2")
public String request2(@RequestAttribute("username") String username) {
    System.out.println(username);
    return "/index.jsp";
}
```

SpringMVC的请求处理

- 请求数据的接收

请求参数乱码的解决方案，Spring已经提供好的CharacterEncodingFilter来进行编码过滤

```
<!--配置全局的编码过滤器-->
<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>
</filter>
<filter-mapping>
    <filter-name>CharacterEncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```



SpringMVC的请求处理

- 请求映射路径的配置
- 请求数据的接收
- Javaweb常用对象获取
- 请求静态资源
- 注解驱动 <mvc:annotation-driven> 标签

SpringMVC的请求处理

- Javaweb常用对象获取

获得Javaweb常见原生对象，有时在我们的Controller方法中需要用到Javaweb的原生对象，例如：Request、Response等，我们只需要将需要的对象以形参的形式写在方法上，SpringMVC框架在调用Controller方法时，会自动传递实参：

```
@GetMapping("/javawebObject")
public String javawebObject(HttpServletRequest request, HttpServletResponse response,
    HttpSession session) {
    System.out.println(request);
    System.out.println(response);
    System.out.println(session);
    return "/index.jsp";
}
```



SpringMVC的请求处理

- 请求映射路径的配置
- 请求数据的接收
- Javaweb常用对象获取
- 请求静态资源
- 注解驱动 <mvc:annotation-driven> 标签

SpringMVC的请求处理

- 请求静态资源

静态资源请求失效的原因，当DispatcherServlet的映射路径配置为 / 的时候，那么就覆盖的Tomcat容器默认的缺省Servlet，在Tomcat的config目录下有一个web.xml 是对所有的web项目的全局配置，其中有如下配置：

```
<servlet>
  <servlet-name>default</servlet-name>
  <servlet-class>org.apache.catalina.servlets.DefaultServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>default</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
```

url-pattern配置为 / 的Servlet我们称其为缺省的Servlet，作用是当其他Servlet都匹配不成功时，就找缺省的Servlet，静态资源由于没有匹配成功的Servlet，所以会找缺省的DefaultServlet，该DefaultServlet具备二次去匹配静态资源的功能。但是我们配置DispatcherServlet后就将其覆盖掉了，而DispatcherServlet会将请求的静态资源的名称当成Controller的映射路径去匹配，即静态资源访问不成功了！

SpringMVC的请求处理

- 请求静态资源

静态资源请求的三种解决方案：

第一种方案，可以再次激活Tomcat的DefaultServlet，Servlet的url-pattern的匹配优先级是：精确匹配>目录匹配>扩展名匹配>缺省匹配，所以可以指定某个目录下或某个扩展名的资源使用DefaultServlet进行解析：

```
<servlet-mapping>
    <servlet-name>default</servlet-name>
    <url-pattern>/img/*</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>default</servlet-name>
    <url-pattern>*.html</url-pattern>
</servlet-mapping>
```

SpringMVC的请求处理

- 请求静态资源

静态资源请求的三种解决方案：

第二种方式，在spring-mvc.xml中去配置静态资源映射，匹配映射路径的请求到指定的位置去匹配资源

```
<!-- mapping是映射资源路径，location是对应资源所在的位置 -->
<mvc:resources mapping="/img/*" location="/img/" />
<mvc:resources mapping="/css/*" location="/css/" />
<mvc:resources mapping="/js/*" location="/js/" />
<mvc:resources mapping="/html/*" location="/html/" />
```

第三种方式，在spring-mvc.xml中去配置< mvc:default-servlet-handler >，该方式是注册了一个DefaultServletHttpRequestHandler 处理器，静态资源的访问都由该处理器去处理，这也是开发中使用最多的

```
<mvc:default-servlet-handler />
```



SpringMVC的请求处理

- 请求映射路径的配置
- 请求数据的接收
- Javaweb常用对象获取
- 请求静态资源
- 注解驱动 <mvc:annotation-driven> 标签

SpringMVC的请求处理

- 注解驱动 <mvc:annotation-driven> 标签

静态资源配置的第二第三种方式我们可以正常访问静态资源了，但是Controller又无法访问了，报错404，即找不到对应的资源



SpringMVC的请求处理

- 注解驱动 <mvc:annotation-driven> 标签

第二种方式是通过SpringMVC去解析mvc命名空间下的resources标签完成的静态资源解析，第三种方式通过SpringMVC去解析mvc命名空间下的default-servlet-handler标签完成的静态资源解析，根据前面所学习的自定义命名空间的解析的知识，可以发现不管是以上哪种方式，最终都会注册SimpleUrlHandlerMapping

```
public BeanDefinition parse(Element element, ParserContext context) {  
    //创建SimpleUrlHandlerMapping类型的BeanDefinition  
    RootBeanDefinition handlerMappingDef =  
        new RootBeanDefinition(SimpleUrlHandlerMapping.class);  
    //注册SimpleUrlHandlerMapping的BeanDefinition  
    context.getRegistry().registerBeanDefinition(beanName, handlerMappingDef);  
}
```

又结合组件浅析知识点，一旦SpringMVC容器中存在 HandlerMapping 类型的组件时，前端控制器 DispatcherServlet在进行初始化时，就会从容器中获得HandlerMapping，不在加载 dispatcherServlet.properties 中默认处理器映射器策略，那也就意味着RequestMappingHandlerMapping不会被加载到了。

SpringMVC的请求处理

- 注解驱动 <mvc:annotation-driven> 标签

手动将RequestMappingHandlerMapping也注册到SpringMVC容器中就可以了，这样DispatcherServlet在进行初始化时，就会从容器中同时获得RequestMappingHandlerMapping存储到DispatcherServlet中名为handlerMappings的List集合中，对@RequestMapping 注解进行解析。

```
<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping"/>
```

SpringMVC的请求处理

- 注解驱动 <mvc:annotation-driven> 标签

根据上面的讲解，可以总结一下，要想使用@RequestMapping正常映射到资源方法，同时静态资源还能正常访问，还可以将请求json格式字符串和JavaBean之间自由转换，我们就需要在spring-mvc.xml中尽心如下配置：

```
<!-- 显示配置RequestMappingHandlerMapping -->
<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping"/>

<!-- 显示配置RequestMappingHandlerAdapter -->
<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">
    <property name="messageConverters">
        <list>
            <bean class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter"/>
        </list>
    </property>
</bean>
<!--配置DefaultServletHttpRequestHandler-->
<mvc:default-servlet-handler/>
```


SpringMVC的请求处理

- 注解驱动 <mvc:annotation-driven> 标签

这么复杂繁琐的配置，是不是看上去有点头大？Spring是个"暖男"，将上述配置浓缩成了一个简单的配置标签，那就是mvc的注解驱动，该标签内部会帮我们注册RequestMappingHandlerMapping、注册RequestMappingHandlerAdapter并注入Json消息转换器等，上述配置就可以简化成如下：

```
<!--mvc注解驱动-->
<mvc:annotation-driven/>
<!--配置DefaultServletHttpRequestHandler-->
<mvc:default-servlet-handler/>
```

PS：<mvc:annotation-driven> 标签在不同的版本中，帮我们注册的组件不同，Spring 3.0.X 版本注册是DefaultAnnotationHandlerMapping 和 AnnotationMethodHandlerAdapter，由于框架的发展，从Spring 3.1.X 开始注册组件变为 RequestMappingHandlerMapping和RequestMappingHandlerAdapter



SpringMVC的请求处理

- 请求映射路径的配置
- 请求数据的接收
- Javaweb常用对象获取
- 请求静态资源
- 注解驱动 <mvc:annotation-driven> 标签



目录

Contents

- ◆ SpringMVC简介
- ◆ SpringMVC的请求处理
- ◆ SpringMVC的响应处理
- ◆ SpringMVC的拦截器
- ◆ SpringMVC的全注解开发
- ◆ SpringMVC的组件原理剖析
- ◆ SpringMVC的异常处理机制



SpringMVC的响应处理

- 传统同步业务数据响应
- 前后端分离异步业务数据响应

SpringMVC的响应处理

- 传统同步业务数据响应

Spring的接收请求的部分我们讲完了，下面在看一下Spring怎么给客户端响应数据，响应数据主要分为两大部分：

- 传统同步方式：准备好模型数据，在跳转到执行页面进行展示，此方式使用越来越少了，基于历史原因，一些旧项目还在使用；
- 前后端分离异步方式：前端使用Ajax技术+ Restful风格与服务端进行Json格式为主的数据交互，目前市场上几乎都是此种方式了。

SpringMVC的响应处理

- 传统同步业务数据响应

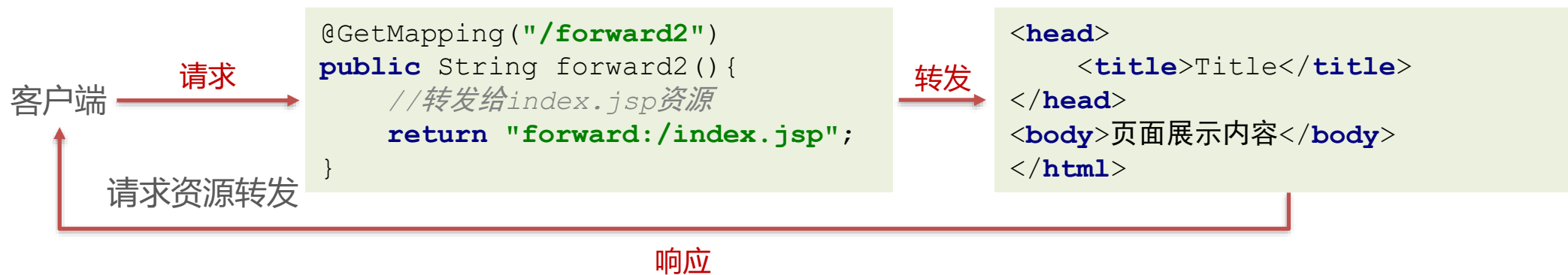
传统同步业务在数据响应时，SpringMVC又涉及如下四种形式：

- 请求资源转发；
- 请求资源重定向；
- 响应模型数据；
- 直接回写数据给客户端；

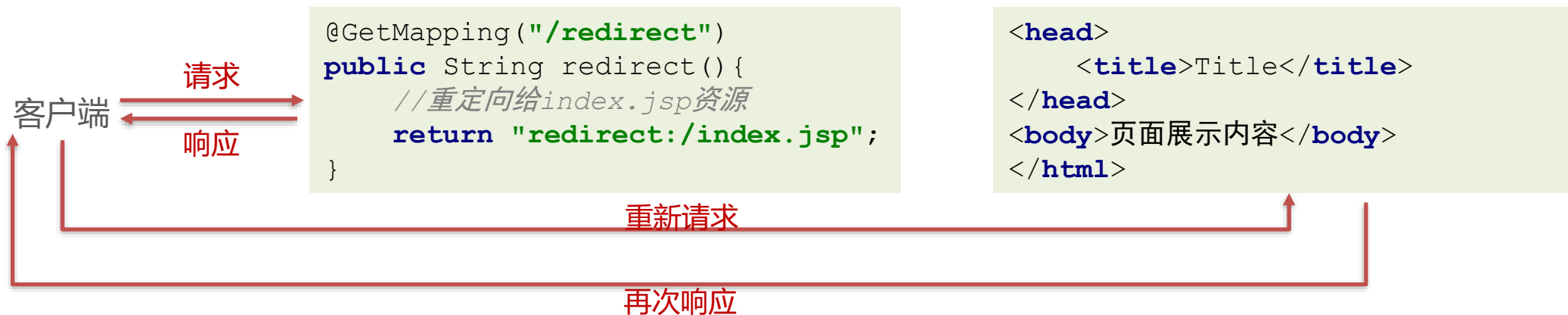
SpringMVC的响应处理

- 传统同步业务数据响应

请求资源转发



请求资源重定向



SpringMVC的响应处理

- 传统同步业务数据响应

响应模型数据，响应模型数据本质也是转发，在转发时可以准备模型数据

```
@GetMapping("/forward5")
public ModelAndView forward5(ModelAndView modelAndView) {
    //准备JavaBean模型数据
    User user = new User();
    user.setUsername("haohao");
    //设置模型
    modelAndView.addObject("user", user);
    //设置视图
    modelAndView.setViewName("/index.jsp");
    return modelAndView;
}
```


SpringMVC的响应处理

- 传统同步业务数据响应

直接回写数据，直接通过方法的返回值返回给客户端的字符串，但是SpringMVC默认的方法返回值是视图，可以通过 `@ResponseBody` 注解显示的告知此处的返回值不要进行视图处理，是要以响应体的方式处理的

```
@GetMapping("/response2")
@ResponseBody
public String response2() throws IOException {
    return "Hello haohao!";
}
```



SpringMVC的响应处理

- 传统同步业务数据响应
- 前后端分离异步业务数据响应

SpringMVC的响应处理

- 前后端分离异步业务数据响应

其实此处的会写数据，跟上面回写数据给客户端的语法方式一样，只不过有如下一些区别：

- 同步方式回写数据，是将数据响应给浏览器进行页面展示的，而异步方式回写数据一般是回写给Ajax引擎的，即谁访问服务器端，服务器端就将数据响应给谁
- 同步方式回写的数据，一般就是一些无特定格式的字符串，而异步方式回写的数据大多是Json格式字符串

SpringMVC的响应处理

- 前后端分离异步业务数据响应

回写普通数据使用@ResponseBody标注方法，直接返回字符串即可，此处不在说明；

回写Json格式的字符串，即将直接拼接Json格式的字符串或使用工具将JavaBean转换成Json格式的字符串回写

```
@GetMapping("/response3")
@ResponseBody
public String response3(HttpServletResponse response) {
    return "{\"username\":\"haohao\",\"age\":18}";
}

@GetMapping("/response4")
@ResponseBody
public String response4() throws JsonProcessingException {
    //创建JavaBean
    User user = new User();
    user.setUsername("haohao");
    user.setAge(18);
    //使用Jackson转换成json格式的字符串
    String json = new ObjectMapper().writeValueAsString(user);
    return json;
}
```

SpringMVC的响应处理

- 前后端分离异步业务数据响应

在讲解SpringMVC接收请求数据时，客户端提交的Json格式的字符串，也是使用Jackson进行的手动转换成JavaBean，可以当我们使用了@RequestBody时，直接用JavaBean就接收了Json格式的数据，原理其实就是SpringMVC底层帮我们做了转换，此处@ResponseBody也可以将JavaBean自动给我们转换成Json格式字符串回响应

```
@GetMapping("/response5")
@ResponseBody
public User response5() throws JsonProcessingException {
    //创建JavaBean
    User user = new User();
    user.setUsername("haohao");
    user.setAge(18);
    //直接返回User对象
    return user;
}
```

SpringMVC的响应处理

- 前后端分离异步业务数据响应

@ResponseBody注解使用优化，在进行前后端分离开发时，Controller的每个方法都是直接回写数据的，所以每个方法上都得写@ResponseBody，可以将@ResponseBody写到Controller上，那么该Controller中的所有方法都具备了返回响应体数据的功能了

```
@Controller
@ResponseBody
public class UserController{
    @GetMapping("/response7")
    public ResultInfo response7() {
        //省略其他代码
        return info;
    }
    @GetMapping("/response5")
    public User response5() throws JsonProcessingException {
        //省略其他代码
        return user;
    }
    // ... 省略其他方法 ...
}
```

SpringMVC的响应处理

- 前后端分离异步业务数据响应

进一步优化，可以使用@RestController替代@Controller和@ResponseBody，@RestController内部具备的这两个注解的功能

```
@RestController
public class UserController{
    @GetMapping("/response7")
    public ResultInfo response7() {
        //省略其他代码
        return info;
    }
    @GetMapping("/response5")
    public User response5() throws JsonProcessingException {
        //省略其他代码
        return user;
    }
    // ... 省略其他方法 ...
}
```



SpringMVC的响应处理

- 传统同步业务数据响应
- 前后端分离异步业务数据响应



目录

Contents

- ◆ SpringMVC简介
- ◆ SpringMVC的请求处理
- ◆ SpringMVC的响应处理
- ◆ SpringMVC的拦截器
- ◆ SpringMVC的全注解开发
- ◆ SpringMVC的组件原理剖析
- ◆ SpringMVC的异常处理机制



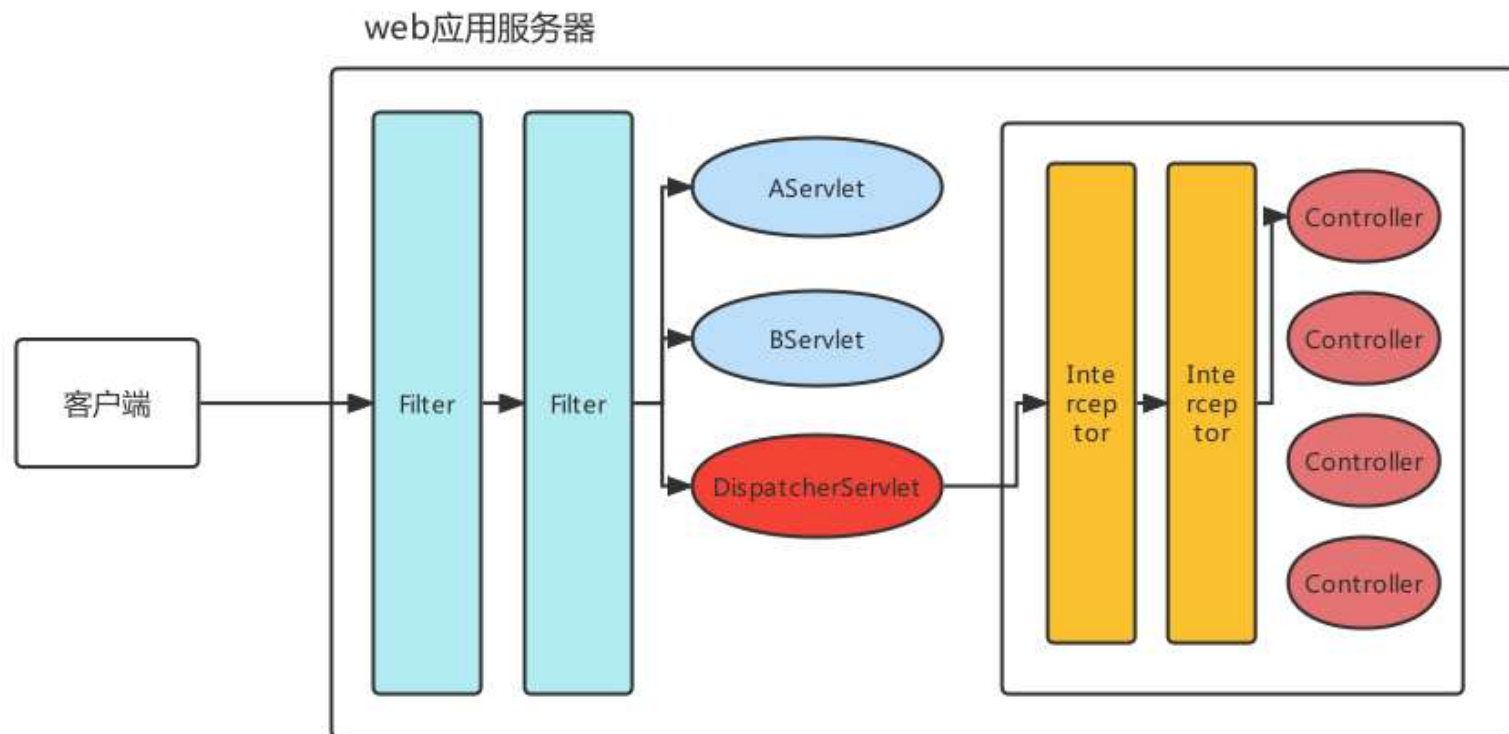
SpringMVC的拦截器

- 拦截器 Interceptor 简介
- 拦截器快速入门
- 拦截器执行顺序
- 拦截器执行原理

SpringMVC的拦截器

- 拦截器 Interceptor 简介

SpringMVC的拦截器Interceptor规范，主要是对Controller资源访问时进行拦截操作的技术，当然拦截后可以权限控制，功能增强等都是可以的。拦截器有点类似 Javaweb 开发中的Filter，拦截器与Filter的区别如下图：



SpringMVC的拦截器

- 拦截器 Interceptor 简介

由上图，对Filter 和 Interceptor 做个对比：

	Filter技术	Interceptor技术
技术范畴	Javaweb原生技术	SpringMVC框架技术
拦截/过滤资源	可以对所有请求都过滤，包括任何Servlet、Jsp、其他资源等	只对进入了SpringMVC管辖范围的才拦截，主要拦截Controller请求
执行时机	早于任何Servlet执行	晚于DispatcherServlet执行

SpringMVC的拦截器

- 拦截器 Interceptor 简介

实现了HandlerInterceptor接口，且被Spring管理的Bean都是拦截器，接口定义如下：

```
public interface HandlerInterceptor {  
  
    default boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {  
        return true;  
    }  
  
    default void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, @Nullable ModelAndView modelAndView) throws Exception {  
    }  
  
    default void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, @Nullable Exception ex) throws Exception {  
    }  
}
```

SpringMVC的拦截器

- 拦截器 Interceptor 简介

HandlerInterceptor接口方法的作用及其参数、返回值详解如下:

	作用	参数	返回值
preHandle	对拦截到的请求进行预处理, 返回true放行执行处理器方法, false不放行	Handler是拦截到的Controller方法处理器	一旦返回false, 代表终止向后执行, 所有后置方法都不执行, 最终方法只执行对应preHandle返回了true的
postHandle	在处理器的方法执行后, 对拦截到的请求进行后处理, 可以在方法中对模型数据和视图进行修改	Handler是拦截到的Controller方法处理器; modelAndView是返回的模型视图对象	无
afterCompletion	视图渲染完成后(整个流程结束之后), 进行最后的处理, 如果请求流程中有异常, 可以处理异常对象	Handler是拦截到的Controller方法处理器; ex是异常对象	



SpringMVC的拦截器

- 拦截器 Interceptor 简介
- 拦截器快速入门
- 拦截器执行顺序
- 拦截器执行原理

SpringMVC的拦截器

- 拦截器快速入门

编写MyInterceptor01实现HandlerInterceptor接口：

```
public class MyInterceptor01 implements HandlerInterceptor {  
    @Override  
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object  
handler) throws Exception {  
        System.out.println("Controller方法执行之前...");  
        return true; //放行  
    }  
    @Override  
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,  
ModelAndView modelAndView) throws Exception {  
        System.out.println("Controller方法执行之后...");  
    }  
    @Override  
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object  
handler, Exception ex) throws Exception {  
        System.out.println("渲染视图结束，整个流程完毕...");  
    }  
}
```


SpringMVC的拦截器

- 拦截器快速入门

配置Interceptor

```
<!--配置拦截器-->
<mvc:interceptors>
    <mvc:interceptor>
        <!--配置对哪些资源进行拦截操作-->
        <mvc:mapping path="/*" />
        <bean class="com.itheima.interceptor.MyInterceptor01"></bean>
    </mvc:interceptor>
</mvc:interceptors>
```



SpringMVC的拦截器

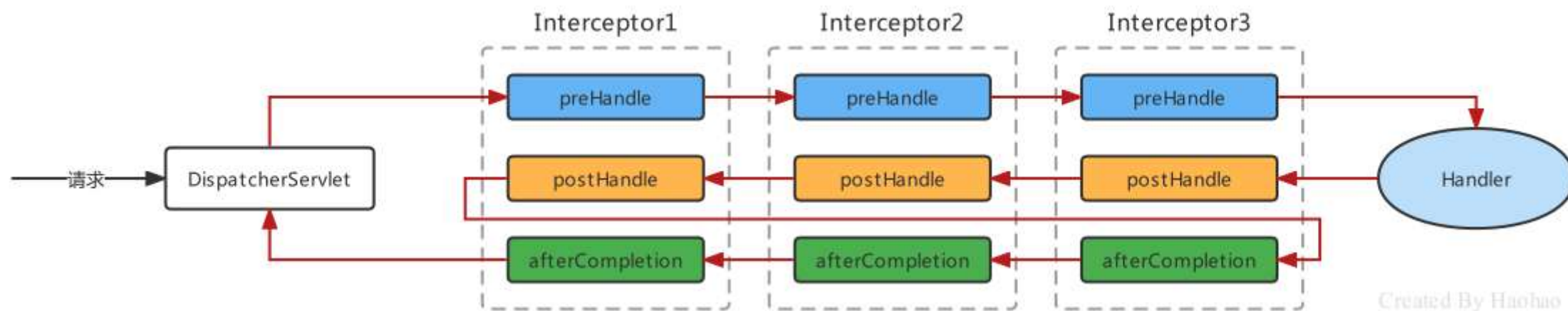
- 拦截器 Interceptor 简介
- 拦截器快速入门
- 拦截器执行顺序
- 拦截器执行原理

SpringMVC的拦截器

- 拦截器执行顺序

拦截器三个方法的执行顺序

当每个拦截器都是放行状态时，三个方法的执行顺序如下：

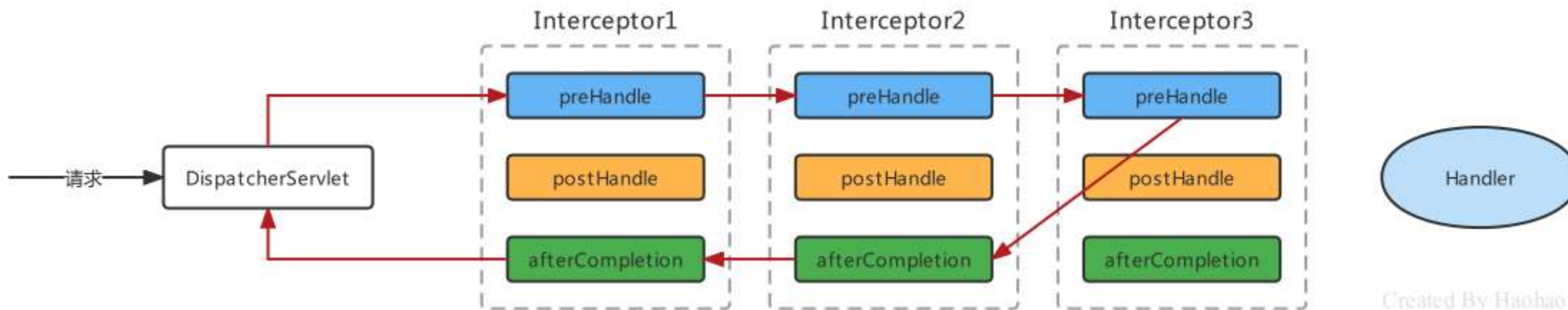


SpringMVC的拦截器

- 拦截器执行顺序

拦截器三个方法的执行顺序

当Interceptor1和Interceptor2处于放行，Interceptor3处于不放行时，三个方法的执行顺序如下：



SpringMVC的拦截器

- 拦截器执行顺序

拦截器执行顺序取决于 interceptor 的配置顺序

```
<mvc:interceptors>
  <mvc:interceptor>
    <mvc:mapping path="/target"/>
    <bean class="com.itheima.interceptor.MyInterceptor02"></bean>
  </mvc:interceptor>
  <mvc:interceptor>
    <mvc:mapping path="/*"/>
    <bean class="com.itheima.interceptor.MyInterceptor01"></bean>
  </mvc:interceptor>
</mvc:interceptors>
```



SpringMVC的拦截器

- 拦截器 Interceptor 简介
- 拦截器快速入门
- 拦截器执行顺序
- 拦截器执行原理

SpringMVC的拦截器

- 拦截器执行原理

请求到来时先会使用组件HandlerMapping去匹配Controller的方法（Handler）和符合拦截路径的Interceptor，Handler和多个Interceptor被封装成一个HandlerExecutionChain的对象

HandlerExecutionChain 定义如下：

```
public class HandlerExecutionChain {  
    //映射的Controller的方法  
    private final Object handler;  
    //当前Handler匹配的拦截器集合  
    private final List<HandlerInterceptor> interceptorList;  
    // ... 省略其他代码 ...  
}
```

SpringMVC的拦截器

- 拦截器执行原理

在DispatcherServlet的doDispatch方法中执行拦截器

```
protected void doDispatch(HttpServletRequest request, HttpServletResponse response){
    //根据请求信息获得HandlerExecutionChain
    HandlerExecutionChain mappedHandler = this.getHandler(request);
    //获得处理器适配器
    HandlerAdapter ha = this.getHandlerAdapter(mappedHandler.getHandler());
    //执行Interceptor的前置方法，前置方法如果返回false，则该流程结束
    if (!mappedHandler.applyPreHandle(request, response)) {
        return;
    }
    //执行handler，一般是HandlerMethod
    ModelAndView mv = ha.handle(processedRequest, response, mappedHandler.getHandler());
    //执行后置方法
    mappedHandler.applyPostHandle(processedRequest, response, mv);
    //执行最终方法
    this.triggerAfterCompletion(processedRequest, response, mappedHandler, e);
}
```


SpringMVC的拦截器

- 拦截器执行原理

跟踪 HandlerExecutionChain的applyPreHandle方法源码：

```
boolean applyPreHandle(HttpServletRequest request, HttpServletResponse response) throws
Exception {
    //对interceptorList进行遍历,正向遍历,与此同时使用interceptorIndex进行计数
    for(int i = 0; i < this.interceptorList.size(); this.interceptorIndex = i++) {
        //取出每一个Interceptor对象
        HandlerInterceptor interceptor = (HandlerInterceptor) this.interceptorList.get(i);
        //调用Interceptor的preHandle方法, 如果返回false, 则直接执行Interceptor的最终方法
        if (!interceptor.preHandle(request, response, this.handler)) {
            //执行Interceptor的最终方法
            this.triggerAfterCompletion(request, response, (Exception) null);
            return false;
        }
    }
    return true;
}
```

SpringMVC的拦截器

- 拦截器执行原理

跟踪 HandlerExecutionChain的applyPostHandle方法源码:

```
void applyPostHandle(HttpServletRequest request, HttpServletResponse response, @Nullable
ModelAndView mv) throws Exception {
    //对interceptorList进行遍历, 逆向遍历
    for(int i = this.interceptorList.size() - 1; i >= 0; --i) {
        //取出每一个Interceptor
        HandlerInterceptor interceptor = (HandlerInterceptor) this.interceptorList.get(i);
        //执行Interceptor的postHandle方法
        interceptor.postHandle(request, response, this.handler, mv);
    }
}
```

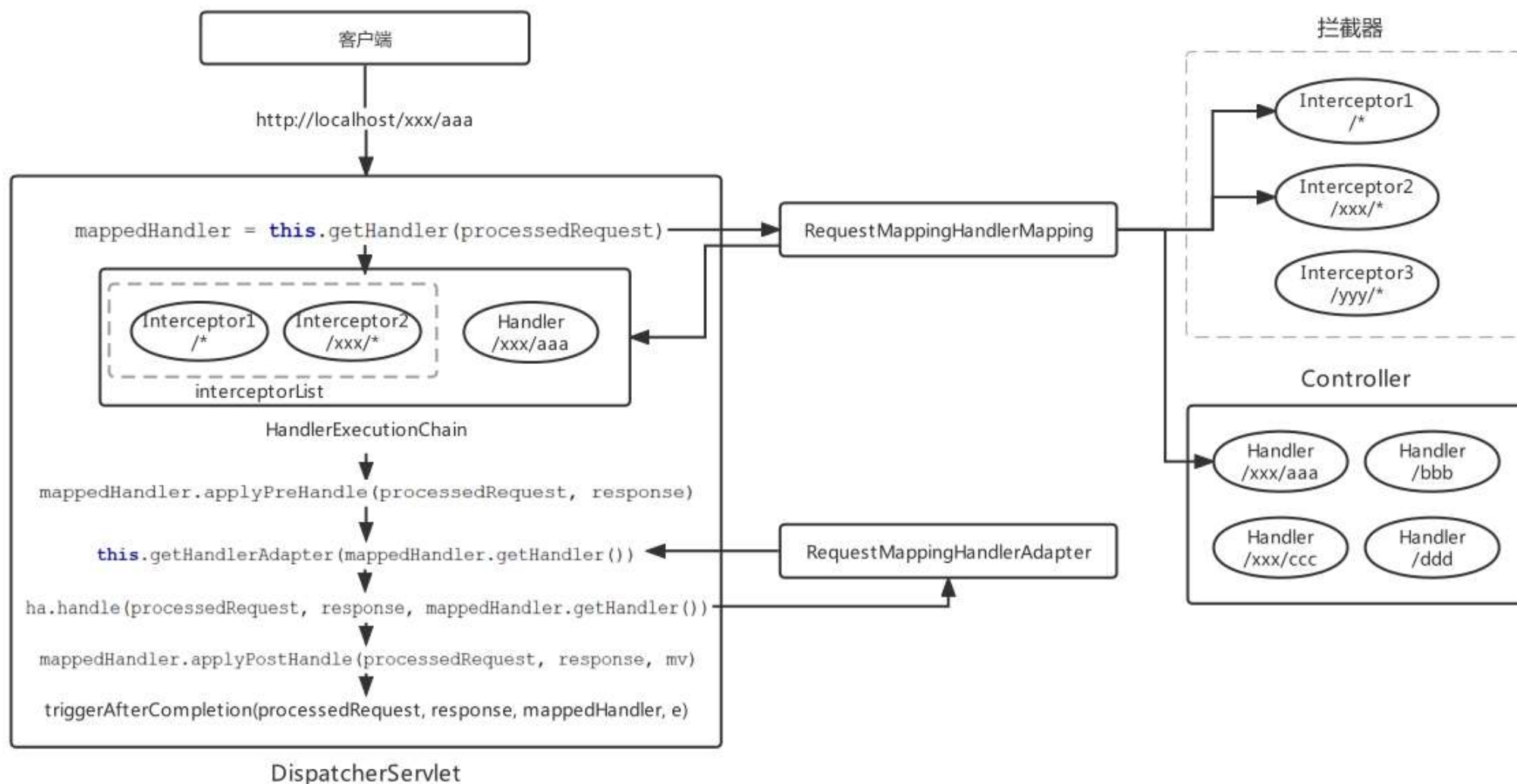
SpringMVC的拦截器

- 拦截器执行原理

跟踪HandlerExecutionChain的triggerAfterCompletion方法源码：

```
void triggerAfterCompletion(HttpServletRequest request, HttpServletResponse response, @Nullable
Exception ex) {
    //逆向遍历interceptorList, 遍历的个数为执行的applyPreHandle次数-1
    for(int i = this.interceptorIndex; i >= 0; --i) {
        //取出每一个Interceptor
        HandlerInterceptor interceptor = (HandlerInterceptor) this.interceptorList.get(i);
        try {
            //执行Interceptor的afterCompletion方法
            interceptor.afterCompletion(request, response, this.handler, ex);
        } catch (Throwable var7) {
            logger.error("HandlerInterceptor.afterCompletion threw exception", var7);
        }
    }
}
```

SpringMVC的拦截器





SpringMVC的拦截器

- 拦截器 Interceptor 简介
- 拦截器快速入门
- 拦截器执行顺序
- 拦截器执行原理



目录

Contents

- ◆ SpringMVC简介
- ◆ SpringMVC的请求处理
- ◆ SpringMVC的响应处理
- ◆ SpringMVC的拦截器
- ◆ SpringMVC的全注解开发
- ◆ SpringMVC的组件原理剖析
- ◆ SpringMVC的异常处理机制



SpringMVC的全注解开发

- spring-mvc.xml 中组件转化为注解形式
- DispatcherServlet加载核心配置类
- 消除web.xml

SpringMVC的全注解开发

- spring-mvc.xml 中组件转化为注解形式

跟之前全注解开发思路一致，xml配置文件使用核心配置类替代，xml中的标签使用对应的注解替代

```
<!-- 组件扫描web层 -->
<context:component-scan base-package="com.ithema.controller"/>
<!--注解驱动-->
<mvc:annotation-driven/>
<!--配置文件上传解析器-->
<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver"/>
<!--配置拦截器-->
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="/*"/>
        <bean class="com.ithema.interceptor.MyInterceptor01"></bean>
    </mvc:interceptor>
</mvc:interceptors>
<!--配置DefaultServletHttpRequestHandler-->
<mvc:default-servlet-handler/>
```


SpringMVC的全注解开发

- spring-mvc.xml 中组件转化为注解形式

- 组件扫描，可以通过@ComponentScan注解完成；
- 文件上传解析器multipartResolver可以通过非自定义Bean的注解配置方式，即@Bean注解完成

```
@Configuration
@ComponentScan("com.itheima.controller")
public class SpringMVConfig {
    @Bean
    public CommonsMultipartResolver multipartResolver() {
        CommonsMultipartResolver multipartResolver = new CommonsMultipartResolver();
        multipartResolver.setDefaultEncoding("UTF-8");
        multipartResolver.setMaxUploadSize(3145728);
        multipartResolver.setMaxUploadSizePerFile(1048576);
        multipartResolver.setMaxInMemorySize(1048576);
        return multipartResolver;
    }
}
```

SpringMVC的全注解开发

- spring-mvc.xml 中组件转化为注解形式

<mvc:default-servlet-handler /> 和 <mvc:interceptor > 怎么办呢？SpringMVC 提供了一个注解叫做 @EnableWebMvc，我们看一下源码，内部通过 @Import 导入了 DelegatingWebMvcConfiguration 类

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
@Documented
@Import({DelegatingWebMvcConfiguration.class})
public @interface EnableWebMvc {}
```

```
@Configuration(proxyBeanMethods = false)
public class DelegatingWebMvcConfiguration extends WebMvcConfigurationSupport {
    private final WebMvcConfigurerComposite configurers = new WebMvcConfigurerComposite();
    //从容器中注入WebMvcConfigurer类型的Bean
    @Autowired(required = false)
    public void setConfigurers(List<WebMvcConfigurer> configurers) {
        if (!CollectionUtils.isEmpty(configurers)) {
            this.configurers.addWebMvcConfigurers(configurers);
        }
    }
    //省略其他代码}
```

SpringMVC的全注解开发

- spring-mvc.xml 中组件转化为注解形式

WebMvcConfigurer类型的Bean会被注入进来，然后被自动调用，所以可以实现WebMvcConfigurer接口，完成一些解析器、默认Servlet等的指定，WebMvcConfigurer接口定义如下：

```
public interface WebMvcConfigurer {  
    //配置默认Servlet处理器  
    default void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurator) { }  
    //添加拦截器  
    default void addInterceptors(InterceptorRegistry registry) { }  
    //添加资源处理器  
    default void addResourceHandlers(ResourceHandlerRegistry registry) { }  
    //添加视图控制器  
    default void addViewControllers(ViewControllerRegistry registry) { }  
    //配置视图解析器  
    default void configureViewResolvers(ViewResolverRegistry registry) { }  
    //添加参数解析器  
    default void addArgumentResolvers(List<HandlerMethodArgumentResolver> resolvers) { }  
    //... 省略其他代码 ...  
}
```

SpringMVC的全注解开发

- spring-mvc.xml 中组件转化为注解形式

创建MyWebMvcConfigurer实现WebMvcConfigurer接口，实现addInterceptors 和 configureDefaultServletHandling方法

```
@Component
public class MyWebMvcConfigurer implements WebMvcConfigurer {
    @Override
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {
        //开启DefaultServlet， 可以处理静态资源了
        configurer.enable();
    }
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        //创建拦截器对象， 进行注册
        //Interceptor的执行顺序也取决于添加顺序
        registry.addInterceptor(new MyInterceptor01()).addPathPatterns("/*");
    }
}
```

SpringMVC的全注解开发

- spring-mvc.xml 中组件转化为注解形式

最后，在SpringMVC核心配置类上添加@EnableWebMvc注解

```
@Configuration
@ComponentScan("com.itheima.controller")
@EnableWebMvc
public class SpringMVConfig {
    @Bean
    public CommonsMultipartResolver multipartResolver(){
        CommonsMultipartResolver multipartResolver = new CommonsMultipartResolver();
        multipartResolver.setDefaultEncoding("UTF-8");
        multipartResolver.setMaxUploadSize(3145728);
        multipartResolver.setMaxUploadSizePerFile(1048576);
        multipartResolver.setMaxInMemorySize(1048576);
        return multipartResolver;
    }
}
```



SpringMVC的全注解开发

- spring-mvc.xml 中组件转化为注解形式
- DispatcherServlet加载核心配置类
- 消除web.xml

SpringMVC的全注解开发

- DispatcherServlet加载核心配置类

DispatcherServlet在进行SpringMVC配置文件加载时，使用的是以下方式：

```
<!--配置springMVC前端控制器-->
<servlet>
    <servlet-name>DispatcherServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <!--指定springMVC配置文件位置-->
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:spring-mvc.xml</param-value>
    </init-param>
    <!--服务器启动就创建-->
    <load-on-startup>2</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>DispatcherServlet</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

SpringMVC的全注解开发

- DispatcherServlet加载核心配置类

现在是使用SpringMVConfig核心配置类替代的spring-mvc.xml，怎么加载呢？参照Spring的ContextLoaderListener加载核心配置类的做法，定义了一个AnnotationConfigWebApplicationContext，通过代码注册核心配置类

```
public class MyAnnotationConfigWebApplicationContext extends
AnnotationConfigWebApplicationContext {
    public MyAnnotationConfigWebApplicationContext() {
        //注册核心配置类
        super.register(SpringMVConfig.class);
    }
}
```

```
<!--指定springMVC的applicationContext全限定名 -->
<init-param>
    <param-name>contextClass</param-name>
    <param-value>com.itheima.config.MyAnnotationConfigWebApplicationContext</param-value>
</init-param>
```




SpringMVC的全注解开发

- spring-mvc.xml 中组件转化为注解形式
- DispatcherServlet加载核心配置类
- 消除web.xml

SpringMVC的全注解开发

- 消除web.xml

目前，几乎消除了配置文件，但是web工程的入口还是使用的web.xml进行配置的，如下

```
<!--配置springMVC前端控制器-->
<servlet>
    <servlet-name>DispatcherServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <!--指定springMVC的applicationContext全限定名 -->
    <init-param>
        <param-name>contextClass</param-name>
        <param-value>com.itheima.config.MyAnnotationConfigWebApplicationContext</param-value>
    </init-param>
    <!--服务器启动就创建-->
    <load-on-startup>2</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>DispatcherServlet</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

SpringMVC的全注解开发

- 消除web.xml

- Servlet3.0环境中，web容器提供了javax.servlet.ServletContainerInitializer接口，实现了该接口后，在对应的类加载路径的META-INF/services 目录创建一个名为javax.servlet.ServletContainerInitializer的文件，文件内容指定具体的ServletContainerInitializer实现类，那么，当web容器启动时就会运行这个初始化器做一些组件内的初始化工作；
- 基于这个特性，Spring就定义了一个SpringServletContainerInitializer实现了ServletContainerInitializer接口；
- 而SpringServletContainerInitializer会查找实现了WebApplicationInitializer的类，Spring又提供了一个WebApplicationInitializer的基础实现类AbstractAnnotationConfigDispatcherServletInitializer，当我们编写类继承AbstractAnnotationConfigDispatcherServletInitializer时，容器就会自动发现我们自己的类，在该类中我们就可以配置Spring和SpringMVC的入口了。

SpringMVC的全注解开发

- 消除web.xml

按照下面的配置就可以完全省略web.xml

```
public class MyAnnotationConfigDispatcherServletInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {
    //返回的带有@Configuration注解的类用来配置ContextLoaderListener
    protected Class<?>[] getRootConfigClasses() {
        System.out.println("加载核心配置类创建ContextLoaderListener");
        return new Class[] {ApplicationContextConfig.class};
    }
    //返回的带有@Configuration注解的类用来配置DispatcherServlet
    protected Class<?>[] getServletConfigClasses() {
        System.out.println("加载核心配置类创建DispatcherServlet");
        return new Class[] {SpringMVCConfig.class};
    }
    //将一个或多个路径映射到DispatcherServlet上
    protected String[] getServletMappings() {
        return new String[] {"/"};
    }
}
```



SpringMVC的全注解开发

- spring-mvc.xml 中组件转化为注解形式
- DispatcherServlet加载核心配置类
- 消除web.xml



目录

Contents

- ◆ SpringMVC简介
- ◆ SpringMVC的请求处理
- ◆ SpringMVC的响应处理
- ◆ SpringMVC的拦截器
- ◆ SpringMVC的全注解开发
- ◆ SpringMVC的组件原理剖析
- ◆ SpringMVC的异常处理机制



SpringMVC的组件原理剖析

- 前端控制器初始化
- 前端控制器执行主流程

SpringMVC的组件原理剖析

- 前端控制器初始化

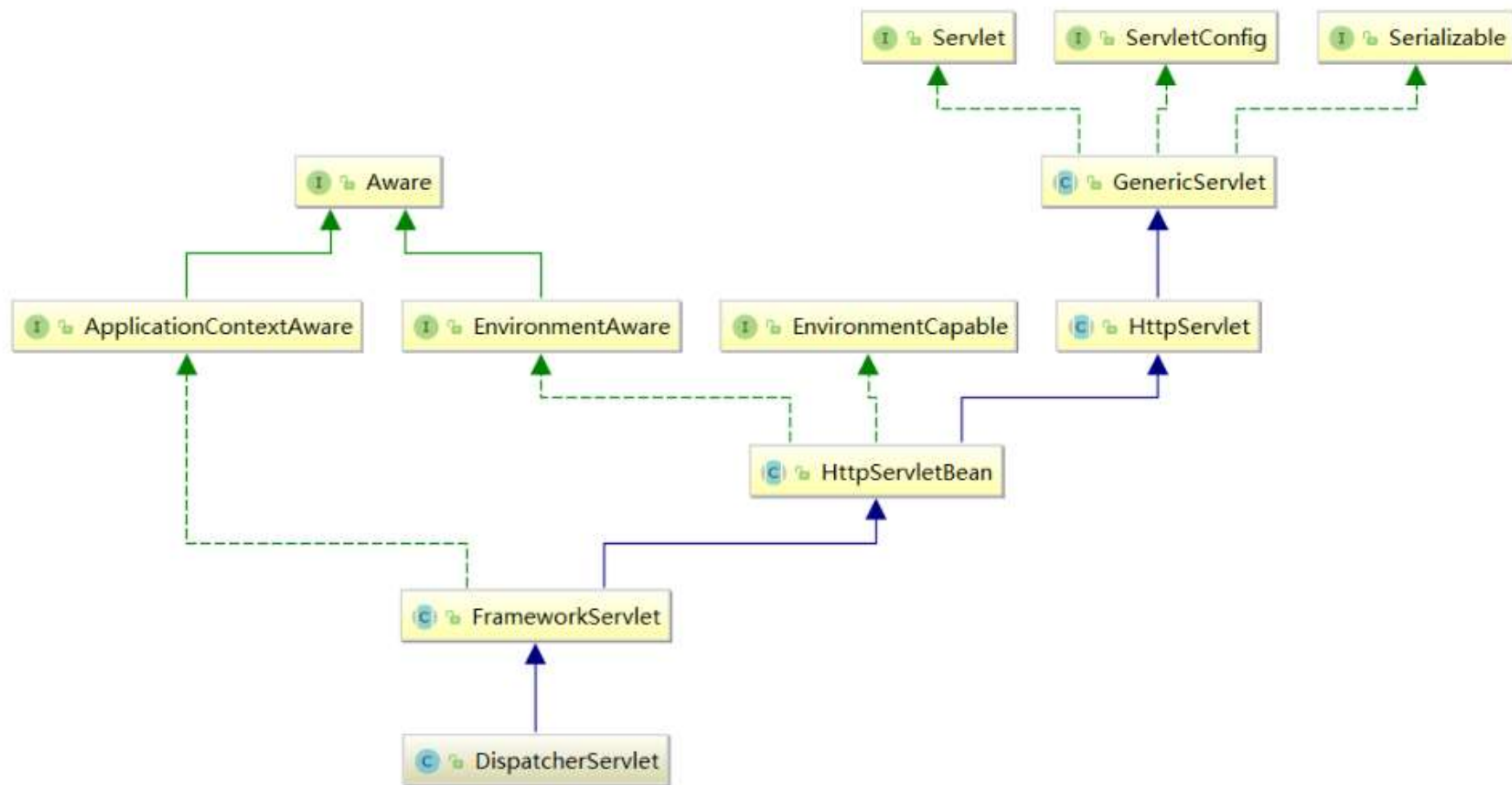
前端控制器DispatcherServlet是SpringMVC的入口，也是SpringMVC的大脑，主流程的工作都是在此完成的，梳理一下DispatcherServlet 代码。DispatcherServlet 本质是个Servlet，当配置了 load-on-startup 时，会在服务器启动时就执行创建和执行初始化init方法，每次请求都会执行service方法

DispatcherServlet 的初始化主要做了两件事：

- 获得了一个 SpringMVC 的 ApplicationContext容器；
- 注册了 SpringMVC的 九大组件。

SpringMVC的组件原理剖析

- 前端控制器初始化



SpringMVC的组件原理剖析

- 前端控制器初始化

SpringMVC 的ApplicationContext容器创建时机，Servlet 规范的 init(ServletConfig config) 方法经过子类重写，最终会调用 FrameworkServlet 抽象类的initWebApplicationContext() 方法，该方法中最终获得 一个根Spring容器（Spring产生的），一个子Spring容器（SpringMVC产生的）

HttpServletBean 的初始化方法

```
public final void init() throws ServletException {  
    this.initServletBean();  
}
```

FrameworkServlet的initServletBean方法

```
protected final void initServletBean() throws ServletException {  
    this.webApplicationContext = this.initWebApplicationContext(); //初始化ApplicationContext  
    this.initFrameworkServlet(); //模板设计模式，供子类覆盖实现，但是子类DispatcherServlet没做使用  
}
```

SpringMVC的组件原理剖析

- 前端控制器初始化

在initWebApplicationContext方法中体现的父子容器的逻辑关系

```
//初始化ApplicationContext是一个及其关键的代码
protected WebApplicationContext initWebApplicationContext() {
    //获得根容器，其实就是通过ContextLoaderListener创建的ApplicationContext
    //如果配置了ContextLoaderListener则获得根容器，没配置获得的是null
    WebApplicationContext rootContext =
        WebApplicationContextUtils.getWebApplicationContext(this.getServletContext());
    //定义SpringMVC产生的ApplicationContext子容器
    WebApplicationContext wac = null;
    if (wac == null) {
        //==>创建SpringMVC的子容器，创建同时将Spring的创建的rootContext传递了过去
        wac = this.createWebApplicationContext(rootContext);
    }
    //将SpringMVC产生的ApplicationContext子容器存储到ServletContext域中
    //key名是: org.springframework.web.servlet.FrameworkServlet.CONTEXT.DispatcherServlet
    if (this.publishContext) {
        String attrName = this.getServletContextAttributeName();
        this.getServletContext().setAttribute(attrName, wac);
    }
}
```

SpringMVC的组件原理剖析

- 前端控制器初始化

跟进创建子容器的源码

```
protected WebApplicationContext createWebApplicationContext(@Nullable ApplicationContext
parent) {
    //实例化子容器ApplicationContext
    ConfigurableWebApplicationContext wac =
    (ConfigurableWebApplicationContext) BeanUtils.instantiateClass(contextClass);
    //设置传递过来的ContextLoaderListener的rootContext为父容器
    wac.setParent(parent);
    //获得web.xml配置的classpath:spring-mvc.xml
    String configLocation = this.getContextConfigLocation();
    if (configLocation != null) {
        //为子容器设置配置加载路径
        wac.setConfigLocation(configLocation);
    }
    //初始化子容器(就是加载spring-mvc.xml配置的Bean)
    this.configureAndRefreshWebApplicationContext(wac);
    return wac;
}
```

SpringMVC的组件原理剖析

- 前端控制器初始化

子容器中的parent维护着父容器的引用

```
> contextClass = {Class@3172} "class org.springframework.web.context.support.XmlWebApplicationContext"
v wac = {XmlWebApplicationContext@4107} "Root WebApplicationContext, started on Thu Jan 01 08:00:00 CST 1970, parent: Root WebAp
  beanFactory = null
  servletContext = null
  servletConfig = null
  namespace = null
  themeSource = null
  configLocations = {String[1]@4147}
  setIdCalled = false
  allowBeanDefinitionOverriding = null
  allowCircularReferences = null
  beanFactory = null
  logger = {LogAdapter$Slf4jLocationAwareLog@4148}
  id = "org.springframework.web.context.support.XmlWebApplicationContext@2974bde1"
  displayName = "Root WebApplicationContext"
  parent = {XmlWebApplicationContext@4090} "Root WebApplicationContext, started on Fri Jul 22 16:13:51 CST 2022"
    beanFactory = {DefaultListableBeanFactory@4169} "org.springframework.beans.factory.support.DefaultListableBeanFactory@458e
    servletContext = {ApplicationContextFacade@4168}
    servletConfig = null
    namespace = null
```

SpringMVC产生的容器

Spring产生的容器，作为SpringMVC产生的容器的父容器存在

SpringMVC的组件原理剖析

- 前端控制器初始化

父容器和子容器概念和关系：

- 父容器：Spring 通过ContextLoaderListener为入口产生的applicationContext容器，内部主要维护的是applicationContext.xml（或相应配置类）配置的Bean信息；
- 子容器：SpringMVC通过DispatcherServlet的init() 方法产生的applicationContext容器，内部主要维护的是spring-mvc.xml（或相应配置类）配置的Bean信息，且内部还通过parent属性维护这父容器的引用。
- Bean的检索顺序：根据上面子父容器的概念，可以知道Controller存在与子容器中，而Controller中要注入Service时，会先从子容器本身去匹配，匹配不成功时在去父容器中去匹配，于是最终从父容器中匹配到的UserService，这样子父容器就可以进行联通了。但是父容器只能从自己容器中进行匹配，不能从子容器中进行匹配。

SpringMVC的组件原理剖析

- 前端控制器初始化

注册 SpringMVC的 九大组件，在初始化容器initWebApplicationContext方法中执行了onRefresh方法，进而执行了初始化策略initStrategies方法，注册了九个解析器组件

```
//DispatcherServlet初始化SpringMVC九大组件
protected void initStrategies(ApplicationContext context) {
    this.initMultipartResolver(context); //1、初始化文件上传解析器
    this.initLocaleResolver(context); //2、初始化国际化解析器
    this.initThemeResolver(context); //3、初始化模板解析器
    this.initHandlerMappings(context); //4、初始化处理器映射器
    this.initHandlerAdapters(context); //5、初始化处理器适配器
    this.initHandlerExceptionResolvers(context); //6、初始化处理器异常解析器
    this.initRequestToViewNameTranslator(context); //7、初始化请求视图转换器
    this.initViewResolvers(context); //8、初始化视图解析器
    this.initFlashMapManager(context); //9、初始化FlashMapManager策略组件
}
```


SpringMVC的组件原理剖析

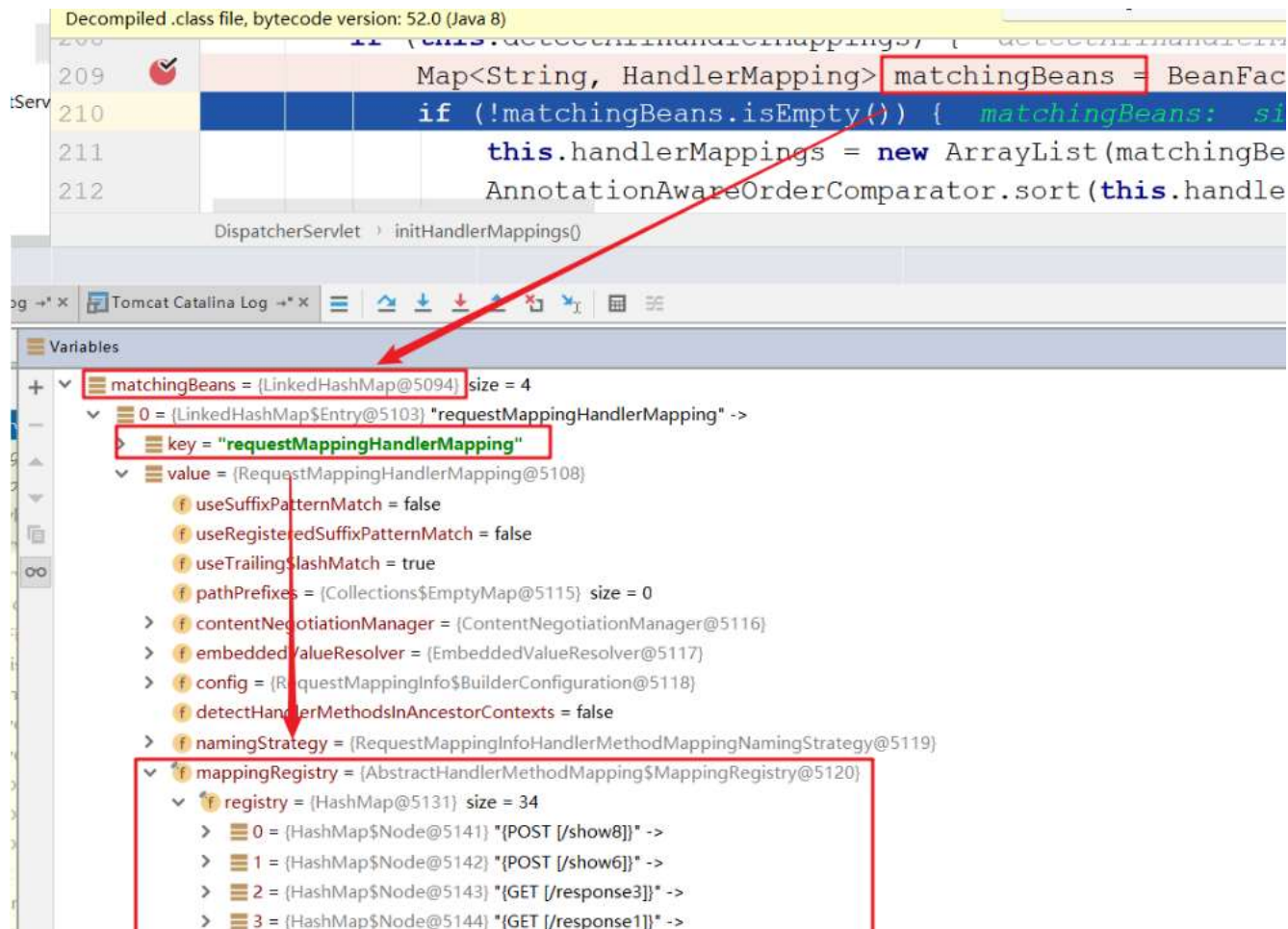
- 前端控制器初始化

以 this.initHandlerMappings(context) 为例，进一步看一下初始化处理器映射器的细节：

```
//定义List容器存储HandlerMapping
private List<HandlerMapping> handlerMappings;
//初始化HandlerMapping的方法
private void initHandlerMappings(ApplicationContext context) {
    this.handlerMappings = null; //初始化集合为null
    //detectAllHandlerMappings默认为true，代表是否从所有容器中(父子容器)检测HandlerMapping
    if (this.detectAllHandlerMappings) {
        //从Spring容器中去匹配HandlerMapping
        Map<String, HandlerMapping> matchingBeans = BeanFactoryUtils.beansOfTypeIncludingAncestors(context,
            HandlerMapping.class, true, false);
        //如果从容器中获取的HandlerMapping不为null就加入到事先定义好的handlerMappings容器中
        if (!matchingBeans.isEmpty()) {
            this.handlerMappings = new ArrayList(matchingBeans.values());
            AnnotationAwareOrderComparator.sort(this.handlerMappings);
        }
        //如果从容器中没有获得HandlerMapping，意味着handlerMappings集合是空的
        if (this.handlerMappings == null) {
            //加载默认的HandlerMapping，就是加载DispatcherServlet.properties文件中的键值对
            this.handlerMappings = this.getDefaultStrategies(context, HandlerMapping.class);
        }
    }
}
```


SpringMVC的组件原理剖析

- 前端控制器初始化



初始化后，映射信息就已经被封装到 `HandlerMapping` 中了，可以在获取 `matchingBeans` 处打断点验证



SpringMVC的组件原理剖析

- 前端控制器初始化
- 前端控制器执行主流程

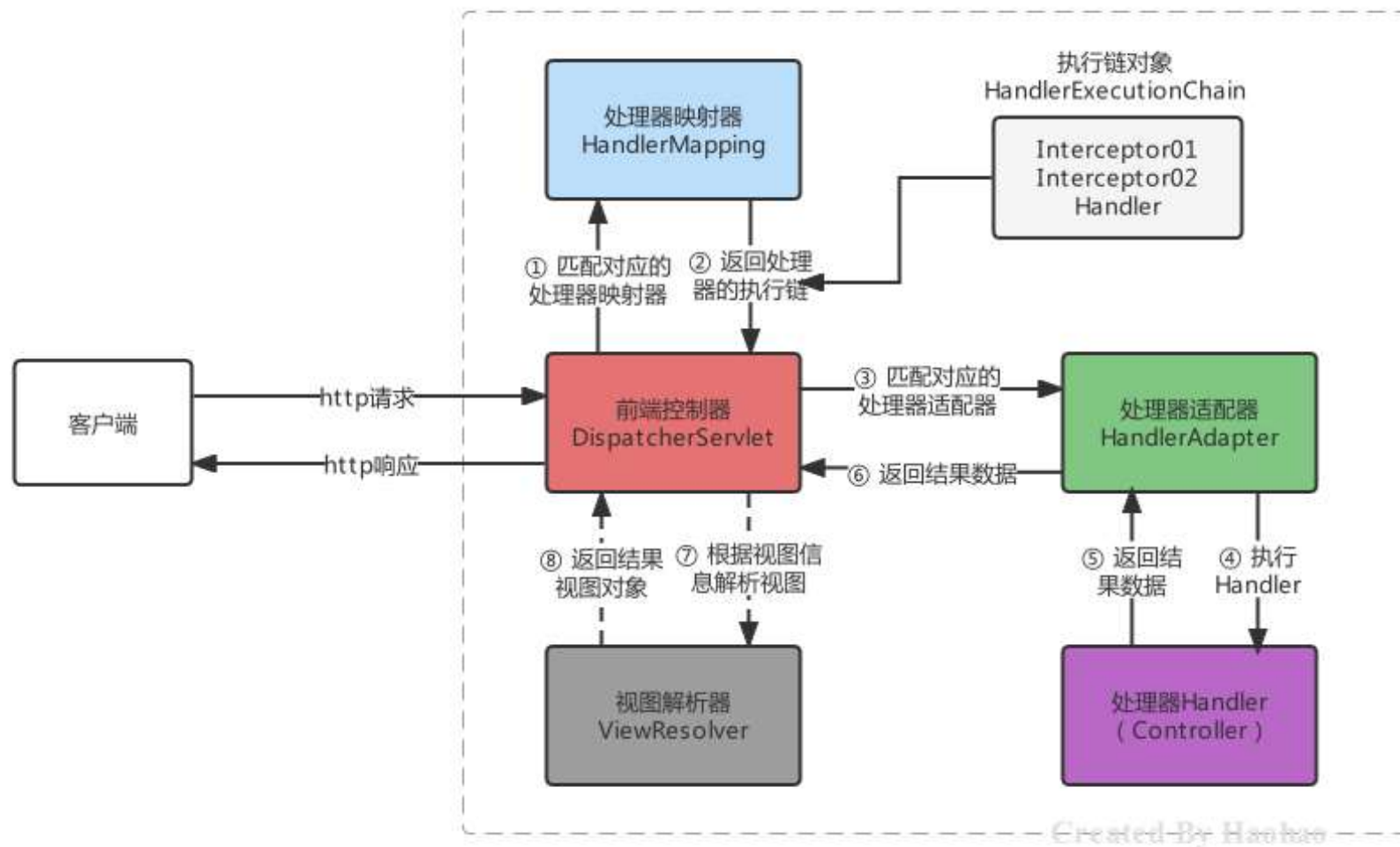
SpringMVC的组件原理剖析

- 前端控制器执行主流程

上面讲解了一下，当服务器启动时，DispatcherServlet 会执行初始化操作，接下来，每次访问都会执行service方法，我们先宏观的看一下执行流程，在去研究源码和组件执行细节

SpringMVC的组件原理剖析

- 前端控制器执行主流程



SpringMVC的组件原理剖析

- 前端控制器执行主流程

FrameworkServlet 复写了service(HttpServletRequest request, HttpServletResponse response)、doGet(HttpServletRequest request, HttpServletResponse response)、doPost(HttpServletRequest request, HttpServletResponse response)等方法，这些方法都会调用processRequest方法

```
protected final void processRequest(HttpServletRequest request, HttpServletResponse response){  
    this.doService(request, response);  
}
```

进一步调用了doService方法，该方法内部又调用了doDispatch方法，而SpringMVC 主流程最核心的方法就是doDispatch 方法

```
protected void doService(HttpServletRequest request, HttpServletResponse response) {  
    this.doDispatch(request, response);  
}
```

SpringMVC的组件原理剖析

- 前端控制器执行主流程

doDispatch方法源码

```
protected void doDispatch(HttpServletRequest request, HttpServletResponse response) {
    HttpServletRequest processedRequest = request;
    HandlerExecutionChain mappedHandler = null; //定义处理器执行链对象
    ModelAndView mv = null; //定义模型视图对象
    //匹配处理器映射器HandlerMapping, 返回处理器执行链对象
    mappedHandler = this.getHandler(processedRequest);
    //匹配处理器适配器HandlerAdapter, 返回处理器适配器对象
    HandlerAdapter ha = this.getHandlerAdapter(mappedHandler.getHandler());
    //执行Interceptor的前置方法preHandle
    mappedHandler.applyPreHandle(processedRequest, response);
    //处理器适配器执行控制器Handler, 返回模型视图对象
    mv = ha.handle(processedRequest, response, mappedHandler.getHandler());
    //执行Interceptor的后置方法postHandle
    mappedHandler.applyPostHandle(processedRequest, response, mv);
    //获取视图渲染视图
    this.processDispatchResult(processedRequest, response, mappedHandler, mv,
        (Exception) dispatchException);
}
```

SpringMVC的组件原理剖析

- 前端控制器执行主流程

通过RequestMappingHandlerMapping 和RequestMappingHandlerAdaptor模拟主流程



SpringMVC的组件原理剖析

- 前端控制器初始化
- 前端控制器执行主流程



目录

Contents

- ◆ SpringMVC简介
- ◆ SpringMVC的请求处理
- ◆ SpringMVC的响应处理
- ◆ SpringMVC的拦截器
- ◆ SpringMVC的全注解开发
- ◆ SpringMVC的组件原理剖析
- ◆ SpringMVC的异常处理机制



SpringMVC的异常处理机制

- SpringMVC 异常的处理流程
- SpringMVC 的异常处理方式
- 异常处理机制原理剖析
- SpringMVC 常用的异常解析器

SpringMVC的异常处理机制

- SpringMVC 异常的处理流程

异常分为编译时异常和运行时异常，编译时异常我们 try-catch 进行捕获，捕获后自行处理，而运行时异常是不可预期的，就需要规范编码来避免，在SpringMVC 中，不管是编译异常还是运行时异常，都可以最终由SpringMVC提供的异常处理器进行统一处理，这样就避免了随时随地捕获处理的繁琐性。

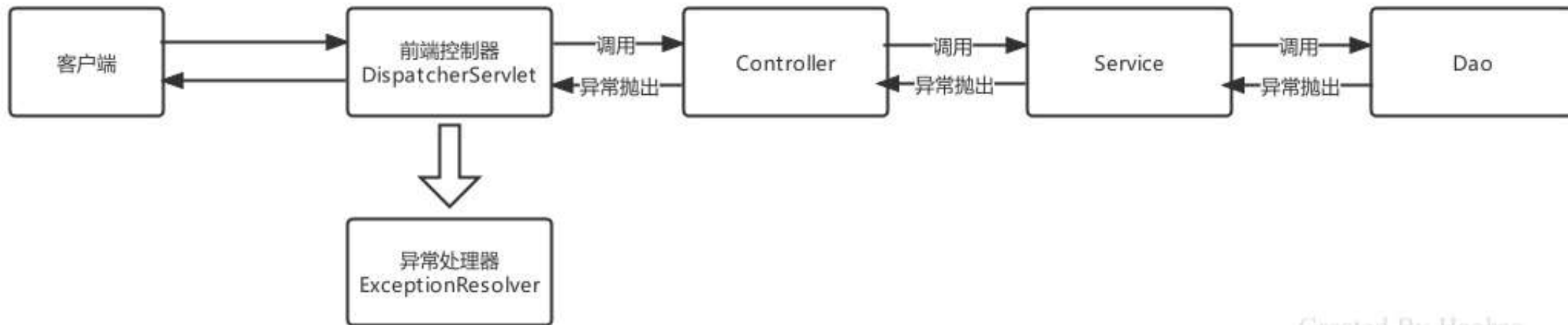
当然除了繁琐之外，我们在进行前后端分离异步开发时，往往返回统一格式的结果给客户端，例如：

`{"code":200,"message":"","data":{"username":"haohao","age":null}}`，即使报异常了，也不能把状态码500直接扔给客户端丢给用户，需要将异常转换成符合上面格式的数据响应给客户端更友好。

SpringMVC的异常处理机制

- SpringMVC 异常的处理流程

SpringMVC 处理异常的思路是，一路向上抛，都抛给前端控制器 DispatcherServlet，DispatcherServlet 在用异常处理器ExceptionHandlerResolver进行处理，如下图：



Created By Haohao



SpringMVC的异常处理机制

- SpringMVC 异常的处理流程
- SpringMVC 的异常处理方式
- 异常处理机制原理剖析
- SpringMVC 常用的异常解析器

SpringMVC的异常处理机制

- SpringMVC 的异常处理方式

SpringMVC 提供了以下三种处理异常的方式：

- 简单异常处理器：使用SpringMVC 内置的异常处理器处理 SimpleMappingExceptionHandler;
- 自定义异常处理器：实现HandlerExceptionHandler接口，自定义异常进行处理;
- 注解方式：使用@ControllerAdvice + @ExceptionHandler 来处理。

SpringMVC的异常处理机制

- SpringMVC 的异常处理方式

使用SimpleMappingExceptionHandler处理一些简单异常，配置开启SimpleMappingExceptionHandler，并指定异常捕获后的处理动作，当发生了异常后，会被 SimpleMappingExceptionHandler 处理，跳转到我们配置的错误页面error.html给用户进行友好展示

```
<!--配置简单异常处理器-->
<bean class="org.springframework.web.servlet.handler.SimpleMappingExceptionHandler">
    <!-- 异常捕获后动作：展示视图 -->
    <property name="defaultErrorView" value="/error.html"/>
</bean>
```

SpringMVC的异常处理机制

- SpringMVC 的异常处理方式

可以在配置SimpleMappingExceptionHandler时，指定一些参数，例如：异常的类型

```
<bean class="org.springframework.web.servlet.handler.SimpleMappingExceptionHandler">
  <property name="defaultErrorView" value="/error.html"/>
  <property name="exceptionMappings">
    <props>
      <!-- 配置异常类型对应的展示视图 -->
      <prop key="java.lang.RuntimeException">/error.html</prop>
      <prop key="java.io.FileNotFoundException">/io.html</prop>
    </props>
  </property>
</bean>
```


SpringMVC的异常处理机制

- SpringMVC 的异常处理方式

注解方式配置简单映射异常处理器

```
@Bean
public SimpleMappingExceptionHandler simpleMappingExceptionHandler() {
    //创建SimpleMappingExceptionHandler
    SimpleMappingExceptionHandler resolver = new SimpleMappingExceptionHandler();
    //设置默认错误展示视图
    resolver.setDefaultErrorView("/error.html");
    //定义Properties设置特殊异常对应的映射视图
    Properties properties = new Properties();
    properties.setProperty("java.lang.RuntimeException", "/error.html");
    properties.setProperty("java.io.FileNotFoundException", "/io.html");
    resolver.setExceptionMappings(properties);
    return resolver;
}
```

SpringMVC的异常处理机制

- SpringMVC 的异常处理方式

自定义异常处理器，实现HandlerExceptionResolver接口自定义异常处理器，可以完成异常逻辑的处理

```
public class MyHandlerExceptionResolver implements HandlerExceptionResolver {  
    @Override  
    //参数Object是当前目标方法处理器对象HandlerMethod  
    public ModelAndView resolveException(HttpServletRequest httpServletRequest,  
    HttpServletResponse httpServletResponse, Object o, Exception e) {  
        ModelAndView modelAndView = new ModelAndView();  
        modelAndView.setViewName("/error.html");  
        return modelAndView;  
    }  
}
```

交给Spring管理异常处理器

```
<bean class="com.ithema.exception.MyHandlerExceptionResolver"></bean>
```

SpringMVC的异常处理机制

- SpringMVC 的异常处理方式

自定义异常处理器，返回Json格式字符串信息

```
@Component
public class MyHandlerExceptionResolver implements HandlerExceptionResolver {
    @Override
    public ModelAndView resolveException(HttpServletRequest httpServletRequest,
        HttpServletResponse httpServletResponse, Object o, Exception e) {
        //编写要返回的json格式的字符串
        String jsonStr = "{\"code\":0,\"message\":\"error\",\"data\":\"\"}";

        try {
            httpServletResponse.getWriter().write(jsonStr);
        } catch (IOException e1) {
            e1.printStackTrace();
        }
        return null;
    }
}
```

SpringMVC的异常处理机制

- SpringMVC 的异常处理方式

使用注解 `@ControllerAdvice` + `@ExceptionHandler` 配置异常，`@ControllerAdvice` 注解本质是一个 `@Component`，也会被扫描到，与此同时，具备AOP功能，默认情况下对所有的Controller都进行拦截操作，拦截后干什么呢？就需要在结合 `@ExceptionHandler`、`@InitBinder`、`@ModelAttribute` 注解一起使用了，此处我们讲解的是异常，所以是 `@ControllerAdvice` + `@ExceptionHandler` 的组合形式。

SpringMVC的异常处理机制

- SpringMVC 的异常处理方式

编写全局异常处理器类，使用@ControllerAdvice标注，且@ExceptionHandler指定异常类型

```
@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(RuntimeException.class)
    public ModelAndView runtimeHandleException(RuntimeException e){
        System.out.println("全局异常处理器执行...."+e);
        ModelAndView modelAndView = new ModelAndView("/error.html");
        return modelAndView;
    }
    @ExceptionHandler(IOException.class)
    @ResponseBody
    public ResultInfo ioHandleException(IOException e){
        //模拟一个ResultInfo
        ResultInfo resultInfo = new ResultInfo(0,"IOException",null);
        return resultInfo;
    }
}
```

SpringMVC的异常处理机制

- SpringMVC 的异常处理方式

如果全局异常处理器响应的数据都是Json格式的字符串的话，可以使用@ControllerAdvice替代@ControllerAdvice 和 @ResponseBody

```
@RestControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(RuntimeException.class)
    public ResultInfo runtimeHandleException(RuntimeException e) {
        //模拟一个ResultInfo
        ResultInfo resultInfo = new ResultInfo(0, "RuntimeException", null);
        return resultInfo;
    }
    @ExceptionHandler(IOException.class)
    public ResultInfo ioHandleException(IOException e) {
        //模拟一个ResultInfo
        ResultInfo resultInfo = new ResultInfo(0, "IOException", null);
        return resultInfo;
    }
}
```



SpringMVC的异常处理机制

- SpringMVC 异常的处理流程
- SpringMVC 的异常处理方式
- 异常处理机制原理剖析
- SpringMVC 常用的异常解析器

SpringMVC的异常处理机制

- 异常处理机制原理剖析

初始化加载的处理器异常解析器，SpringMVC 的前置控制器在进行初始化的时候，会初始化处理器异常解析器
HandlerExceptionResolver

```
//初始化处理器异常解析器
```

```
this.initHandlerExceptionResolvers(context);
```

```
private void initHandlerExceptionResolvers(ApplicationContext context) {  
    //从容器中获得HandlerExceptionResolver的Map集合  
    Map<String, HandlerExceptionResolver> matchingBeans =  
    BeanFactoryUtils.beansOfTypeIncludingAncestors(context, HandlerExceptionResolver.class, true,  
    false);  
    //如果容器中没有HandlerExceptionResolver的话，则加载默认的HandlerExceptionResolver  
    if (this.handlerExceptionResolvers == null) {  
        //从dispatcherServlet.properties中加载  
        this.handlerExceptionResolvers = this.getDefaultStrategies(context,  
        HandlerExceptionResolver.class);  
    }  
}
```


SpringMVC的异常处理机制

- 异常处理机制原理剖析

加载DispatcherServlet.properties中默认的异常处理器

```
org.springframework.web.servlet.HandlerExceptionResolver=org.springframework.web.servlet.mvc.method.annotation.ExceptionHandlerExceptionResolver,\n    org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionHandler,\n    org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver
```

配置了自定义的异常处理器后，默认的异常处理器就不会被加载，当配置<mvc:annotation-driven /> 或配置了注解@EnableWebMvc后，默认异常处理器和自定的处理器异常解析器都会被注册

```
matchingBeans = {LinkedHashMap@4933} size = 2\n  0 = {LinkedHashMap$Entry@4939} "handlerExceptionResolver" ->\n    key = "handlerExceptionResolver"\n    value = {HandlerExceptionResolverComposite@4942}\n      resolvers = {ArrayList@4945} size = 3\n        0 = {ExceptionHandlerExceptionResolver@4947}\n        1 = {ResponseStatusExceptionHandler@4948}\n        2 = {DefaultHandlerExceptionResolver@4949}\n      order = 0\n  1 = {LinkedHashMap$Entry@4940} "myHandlerExceptionResolver" ->\n    key = "myHandlerExceptionResolver"\n    value = {MyHandlerExceptionResolver@4944}
```

SpringMVC的异常处理机制

- 异常处理机制原理剖析

异常处理器加载完毕后，当发生异常时，就会进行处理，跟踪 DispatcherServlet 的 doDispatch() 方法

```
protected void doDispatch(HttpServletRequest request, HttpServletResponse response) {  
    Object dispatchException = null; // 定义异常  
    try {  
        // ... 省略代码 ...  
    } catch (Exception e) {  
        dispatchException = e;  
    } catch (Throwable te) {  
        dispatchException = new NestedServletException("Handler dispatch failed", te);  
    }  
    // 视图处理、拦截器最终方法调用、异常处理都在该方法内  
    this.processDispatchResult(processedRequest, response, mappedHandler, mv,  
        (Exception) dispatchException);  
}
```

SpringMVC的异常处理机制

- 异常处理机制原理剖析

跟踪processDispatchResult方法

```
private void processDispatchResult(HttpServletRequest request, HttpServletResponse response,
@Nullable HandlerExecutionChain mappedHandler, @Nullable ModelAndView mv, @Nullable Exception
exception) throws Exception {
    boolean errorView = false; // 定义错误视图标识，默认为false
    if (exception != null) {
        // 判断当前捕获的异常是否是ModelAndViewDefiningException类型的异常
        if (exception instanceof ModelAndViewDefiningException) {
            // 获得ModelAndViewDefiningException异常对象中的ModelAndView对象
            mv = ((ModelAndViewDefiningException) exception).getModelAndView();
        } else {
            // 捕获到其他异常，获得当前发生异常的Handler对象
            Object handler = mappedHandler != null ? mappedHandler.getHandler() : null;
            // 执行processHandlerException 方法
            mv = this.processHandlerException(request, response, handler, exception);
            // 如果异常处理返回了ModelAndView 则修改错误视图的标识为true
            errorView = mv != null;
        }
    }
    // ... 省略其他代码 ...}
```

SpringMVC的异常处理机制

- 异常处理机制原理剖析

跟踪processHandlerException 方法

```
protected ModelAndView processHandlerException(HttpServletRequest request, HttpServletResponse response,
@Nullable Object handler, Exception ex) throws Exception {
    ModelAndView exMv = null; //定义ModelAndView
    //判断处理器异常解析器集合是否是空的
    if (this.handlerExceptionResolvers != null) {
        //遍历处理器异常解析器List集合
        Iterator var6 = this.handlerExceptionResolvers.iterator();
        while (var6.hasNext()) {
            //取出每一个异常解析器
            HandlerExceptionResolver resolver = (HandlerExceptionResolver)var6.next();
            //执行异常解析器的resolveException方法
            exMv = resolver.resolveException(request, response, handler, ex);
            //只要有一个异常处理器返回ModelAndView 则不在执行后面的异常处理器
            if (exMv != null) {
                break;
            }
        }
    }
    //如果视图解析器不为null
    if (exMv != null) {return exMv;
    } else { throw ex; } }
```



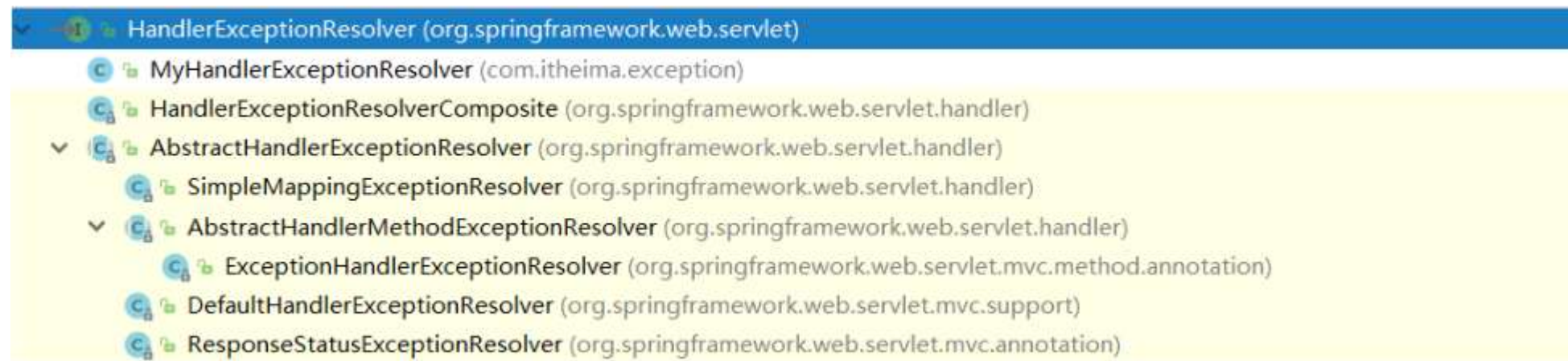
SpringMVC的异常处理机制

- SpringMVC 异常的处理流程
- SpringMVC 的异常处理方式
- 异常处理机制原理剖析
- SpringMVC 常用的异常解析器

SpringMVC的异常处理机制

- SpringMVC 常用的异常解析器

SpringMVC 相关的处理器异常解析器继承体系如下：



SpringMVC的异常处理机制

- SpringMVC 常用的异常解析器

接口或类	说明
HandlerExceptionResolver	异常处理器类的顶级接口，实现了该接口的类都会作为异常处理器类
MyHandlerExceptionResolver	自定义的处理器类，实现了HandlerExceptionResolver接口
HandlerExceptionResolverComposite	异常解析器混合器，内部存在集合存储多种异常解析器
SimpleMappingExceptionResolver	简单映射异常处理器，可以配置异常与对应的错误视图
ExceptionHandlerExceptionResolver	异常处理器异常解析器，默认会被注册到Spring容器中，@ExceptionHandler方式异常处理就是该解析器解析的
DefaultHandlerExceptionResolver	默认处理器异常解析器，所有异常处理器都不匹配时，最后执行的异常处理器
ResponseStatusExceptionResolver	响应状态异常解析器，结合使用@ResponseStatus标注的异常使用



SpringMVC的异常处理机制

- SpringMVC 异常的处理流程
- SpringMVC 的异常处理方式
- 异常处理机制原理剖析
- SpringMVC 常用的异常解析器



目录

Contents

- ◆ SpringMVC简介
- ◆ SpringMVC的请求处理
- ◆ SpringMVC的响应处理
- ◆ SpringMVC的拦截器
- ◆ SpringMVC的全注解开发
- ◆ SpringMVC的组件原理剖析
- ◆ SpringMVC的异常处理机制



传智教育旗下高端IT教育品牌