第5章 Spring Boot实现Web MVC

广东财经大学信息学院
罗 东 俊 博士
ZSUJONE@126.COM

（内部资料，请勿外传）

目的和要求

- 熟悉Thymeleaf模板引擎基本语法。
- 掌握Spring Boot整合Thymeleaf模板引擎的使用。
- 掌握Spring Boot整合Spring MVC的使用。
- 掌握拦截器的作用和使用方法。
- 掌握Spring Boot中MVC功能的定制。

主要内容

- 5.1 MVC设计概述
- 5.2 使用视图技术Thymeleaf
- 5.3 使用控制器
- 5.4 使用拦截器
- 5.5 自定义Web MVC配置

5.4 使用拦截器

- 5.4.1 拦截器概述
- 5.4.2 实现用户登录权限验证

5.4.1 拦截器概述

- 1.什么是拦截器？
- 2.拦截器的声明和注册
- 3.拦截器的执行流程

1.什么是拦截器？

- 拦截器（Interceptor）类似于Servlet中的过滤器（Filter），它主要用于拦截用户请求并作相应的处理。
  - ✧例如通过拦截器可以进行权限验证、记录请求信息的日志、判断用户是否登录等。

2.拦截器的声明和注册

- 可通过实现HandlerInterceptor接口来声明一个拦截器类，重写以下3个方法：
  - ✧ **preHandle()**：该方法会在控制器方法前执行，其返回值表示是否中断后续操作。
    - 当其返回值为true时，表示继续向下执行；
    - 当其返回值为false时，会中断后续的所有操作。
  - ✧ **postHandle()**：该方法会在控制器方法调用之后，且解析视图之前执行。
    - 可以通过此方法对请求域中的模型和视图做出进一步的修改。
  - ✧ **afterCompletion()**：该方法会在整个请求完成，即视图渲染结束之后执行。
    - 可以通过此方法实现一些资源清理、记录日志信息等工作。

示例代码

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;
//实现了HandlerInterceptor接口的自定义拦截器类
@Component
public class CustomInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        System.out.println("CustomInterceptor...preHandle");
        return true;                                //对拦截的请求进行放行处理
    }
    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {
        System.out.println("CustomInterceptor...postHandle");
    }
    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws Exception {
        System.out.println("CustomInterceptor...afterCompletion");
    }
}
```

拦截器的注册

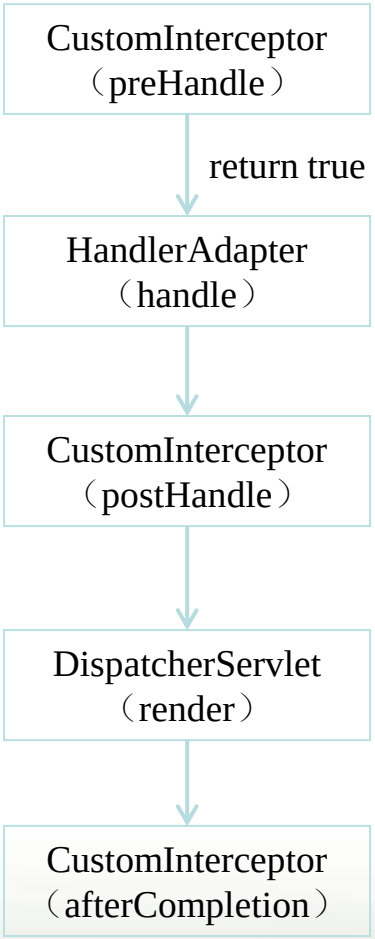
```
import com.itheima.interceptor.CustomInterceptor;
import com.itheima.interceptor.LoginInterceptor;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
@Configuration
public class InterceptorConfig implements WebMvcConfigurer {
    @Autowired
    private LoginInterceptor loginInterceptor;
    @Autowired
    private CustomInterceptor customInterceptor;
    @Override
    // 注册拦截器
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(loginInterceptor);
        registry.addInterceptor(customInterceptor);
    }
}
```

注册顺序会影响执行顺序

3.拦截器的执行流程

- 1）单个拦截器的执行流程
- 2）多个拦截器的执行流程

1）单个拦截器的执行流程



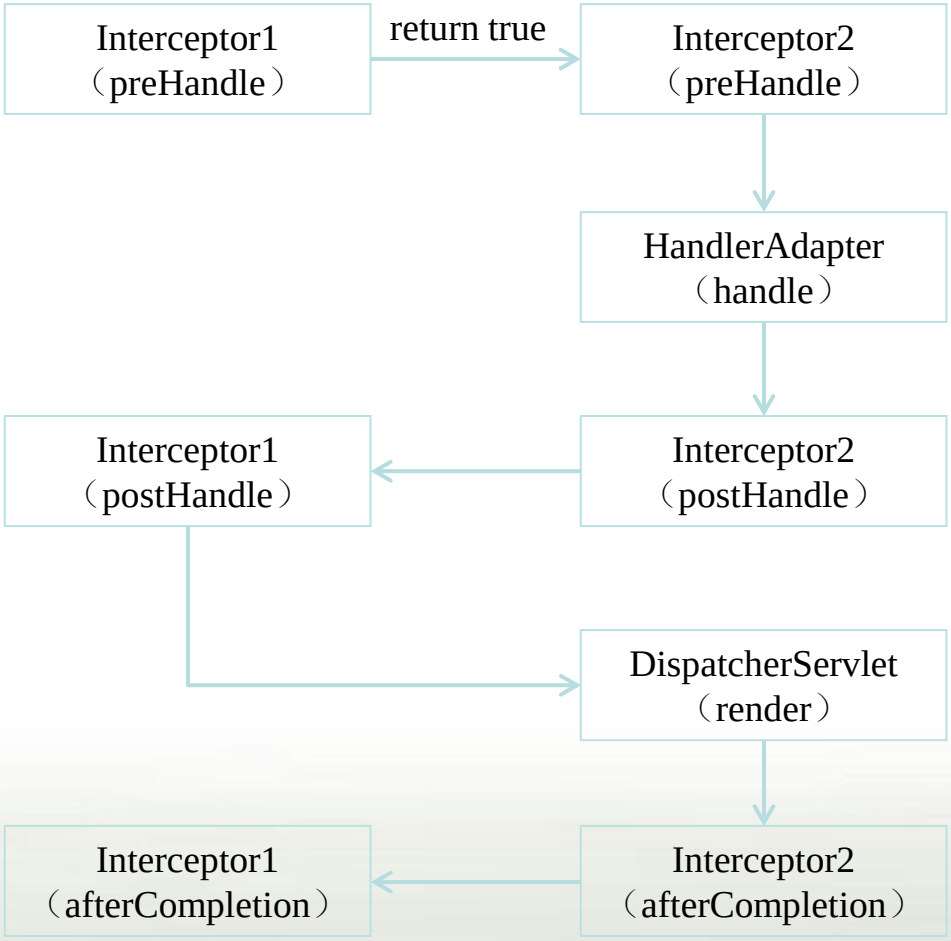
程序先执行preHandle()方法，如果该方法的返回值为true，则程序会继续向下执行处理器中的方法，否则将不再向下执行

在业务处理器（即控制器Controller类）处理完请求后，会执行postHandle()方法，然后会通过DispatcherServlet向客户端返回响应

在DispatcherServlet处理完请求后，才会执行afterCompletion()方法

2）多个拦截器的执行流程

多个拦截器（假设有两个拦截器Interceptor1和Interceptor2，并且在注册时，Interceptor1拦截器注册在前），在程序中的执行流程如下图所示：



从图可以看出，当有多个拦截器同时工作时，它们的preHandle()方法会按照拦截器的注册顺序执行，而它们的postHandle()方法和afterCompletion()方法则会按照注册顺序的反序执行。

5.4.2 实现用户登录权限验证

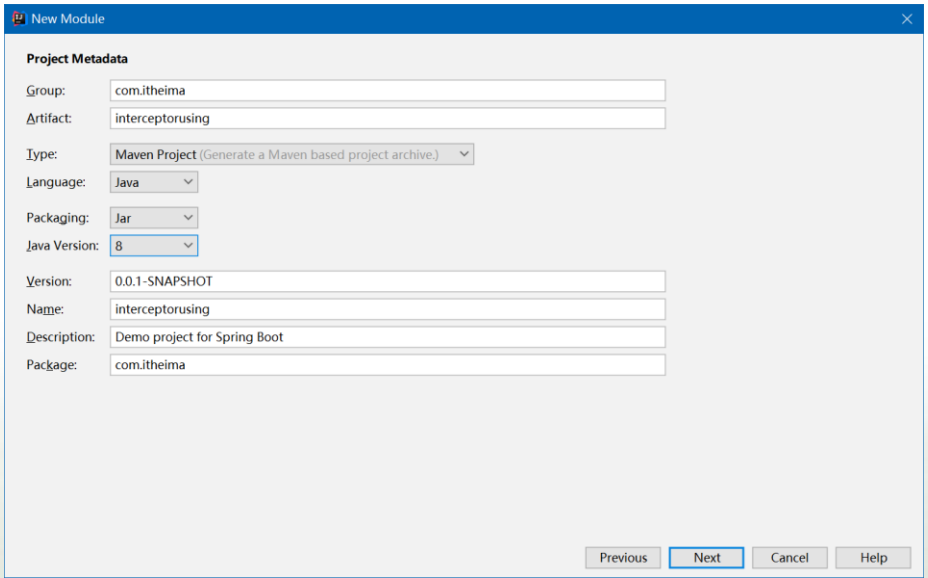
- 本案例通过拦截器实现只有登录后的用户才能访问系统中的主页面。
  - ✧如果没有登录系统而直接访问主页面，则拦截器会将请求拦截，并转发到登录页面，同时在登录页面中给出提示信息。
  - ✧如果用户名或密码错误，也会在登录页面给出相应提示信息。
  - ✧当已登录的用户在系统主页中单击“退出”链接时，系统同样会回到登录页面。

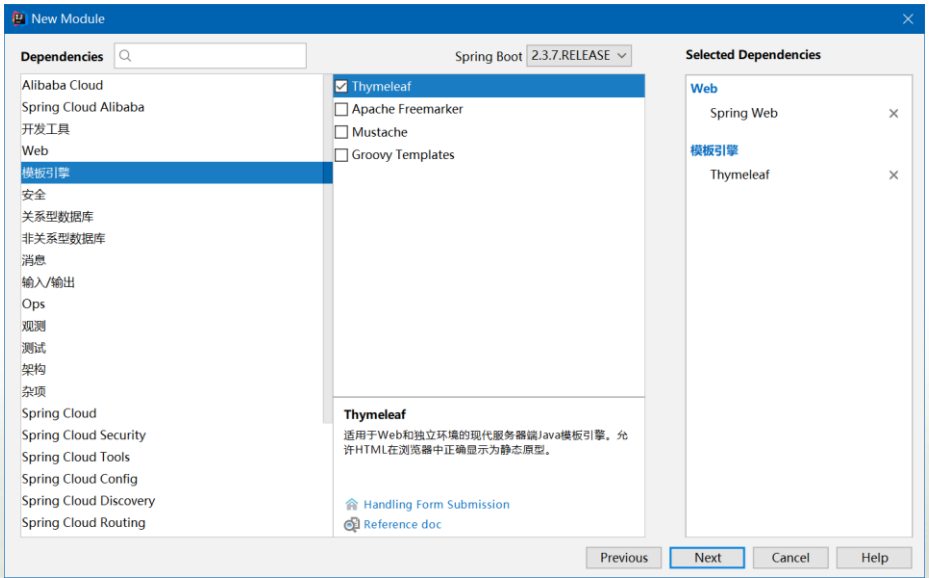
实现步骤

- 搭建步骤：
  - ①创建Spring Boot模块
  - ②创建持久化类
  - ③创建Web控制类
  - ④创建拦截器类
  - ⑤配置拦截器
  - ⑥创建登录页面
  - ⑦创建主页面
  - ⑧效果测试

①创建Spring Boot模块

- 在chapter05项目中，使用Spring Initializr方式创建一个Spring Boot模块interceptorusing，在Dependencies依赖选择中选择Web模块中的Spring Web依赖和Template Engines模块中的Thymeleaf依赖





②创建持久化类

- 在interceptorusing模块中新建一个com.itheima.po包，并在包中新建一个实体类User。

```
public class User {
    private Integer id;
    private String username;
    private String password;
    //省略属性的getXX()和setXX()方法
}
```

③创建Web控制类

- 在interceptorusing模块中新建一个com.itheima.controller包，并在包中新建一个Web控制类UserController，并在该类中定义向主页跳转、向登录页跳转、执行用户登录等操作的方法。

UserController类

```
import javax.servlet.http.HttpSession;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import com.itheima.po.User;

@Controller
public class UserController {
    @GetMapping("/login")                    //向用户登录页面跳转
    public String toLogin() {
        return "login";
    }
    @GetMapping("/main")                     //向用户主页面跳转
    public String toMain() {
        return "main";
    }
```

UserController类

```
    @PostMapping("/login")                          //用户登录
    public String login(User user, Model model, HttpSession session) {
        String username = user.getUsername();
        String password = user.getPassword();
        // 此处模拟从数据库中获取用户名和密码后进行判断
        if (username != null && username.equals("xiaoxue")
                && password != null && password.equals("123456")) {
            session.setAttribute("user", user);       // 将用户对象添加到Session
            return "redirect:/main";                  // 重定向到主页面的跳转方法
        }
        model.addAttribute("msg", "用户名或密码错误，请重新登录！");
        return "login";
    }
    @GetMapping("/logout")                          //退出登录
    public String logout(HttpSession session) {
        session.invalidate();                         // 清除Session
        return "redirect:/login";                     //重定向到登录页面
    }
}
```

④创建拦截器类

- 在interceptorusing模块中新建一个com.itheima.interceptor包，并在包中新建一个拦截器类LoginInterceptor，实现对URL请求进行拦截控制。

LoginInterceptor类

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;
import com.itheima.po.User;
@Component
public class LoginInterceptor implements HandlerInterceptor {
    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,
                        ModelAndView modelAndView) throws Exception {
        System.out.println("LoginInterceptor...postHandle");
    }
    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)
            throws Exception {
        System.out.println("LoginInterceptor...afterCompletion");
    }
```

LoginInterceptor类

```
  @Override
   public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception
{
       System.out.println("LoginInterceptor...preHandle");
      HttpSession session = request.getSession();      //获取Session
       User user = (User) session.getAttribute("user");
       if (user != null) {                                    //判断Session中是否有用户数据，如果有，则返回true,继续向下执行
          return true;
       }
       request.setAttribute("msg", "您还没有登录，请先登录！");
       // 不符合条件的给出提示信息，并转发到登录页面
       request.getRequestDispatcher("/login").forward(request, response);
       return false;
   }
}
```

⑤配置拦截器

- 在interceptorusing模块中新建一个com.itheima.config包，并在包中新建一个配置类InterceptorConfig，对拦截器LoginInterceptor注册配置，实现除了/login和静态资源是可以公开访问的，其他的URL都进行拦截控制。

InterceptorConfig类

```
import com.itheima.interceptor.LoginInterceptor;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
@Configuration
public class InterceptorConfig implements WebMvcConfigurer {
    @Autowired
    private LoginInterceptor loginInterceptor;
   @Override
    // 注册拦截器
    public void addInterceptors(InterceptorRegistry registry) {
        List<String> patterns=new ArrayList<>();
        patterns.add("/login");
        patterns.add("/login/**");
        //除了“/login”和静态资源是可以公开访问的，其它的URL都进行拦截控制
        registry.addInterceptor(loginInterceptor).excludePathPatterns(patterns);
    }
}
```

⑥创建登录页面

- 在interceptorusing模块resources的templates目录下，创建一个登录模板页面login.html，在页面中编写一个用于实现登录操作的form表单。

登录页面login.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1,shrink-to-fit=no">
    <title>用户登录界面</title>
    <link th:href="@{/login/css/bootstrap.min.css}" rel="stylesheet">
    <link th:href="@{/login/css/signin.css}" rel="stylesheet">
</head>
<body class="text-center">
<style type="text/css">
    .warn {
        color: red
    }
</style>
```

登录页面login.html

```
<!-- 用户登录form表单 -->
<form class="form-signin" th:action="@{/login}" method="POST">
    <img class="mb-4" th:src="@{/login/img/login.jpg}" width="72" height="72">
    <h1 class="h3 mb-3 font-weight-normal" th:text="欢迎登录"></h1>
    <input type="text" name="username" class="form-control" th:placeholder="用户名" required="" autofocus="">
    <input type="password" name="password" class="form-control" th:placeholder="密码" required="">
    <div class="checkbox mb-3">
        <label>
            <input type="checkbox" value="remember-me"> 记住我
        </label>
    </div>
    <div class="warn">[[${msg}]]</div>
    <button class="btn btn-lg btn-primary btn-block" type="submit" th:text="登录"></button>
</form>
</body>
</html>
```

⑦创建主页面

- 在interceptorusing模块resources的templates目录下，创建一个主页面main.html，在该页面中使用变量表达式获取用户信息，并且通过一个超链接来实现“退出”功能。

主页面main.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1,shrink-to-fit=no">
    <title>系统主页</title>
    <link th:href="@{/login/css/bootstrap.min.css}" rel="stylesheet">
</head>
<body class="text-center">
当前用户：[[${session.user.username}]]
<a th:href="@{/logout}">退出</a>
</body>
</html>
```

⑧效果测试

- 启动项目进行测试，在浏览器上访问**http://localhost:8080/main**，将出现图1效果；然后分别输入错误和正确的用户名和密码后将出现图2和图3效果。

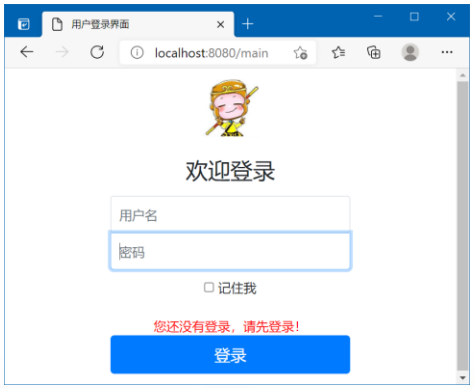


图1

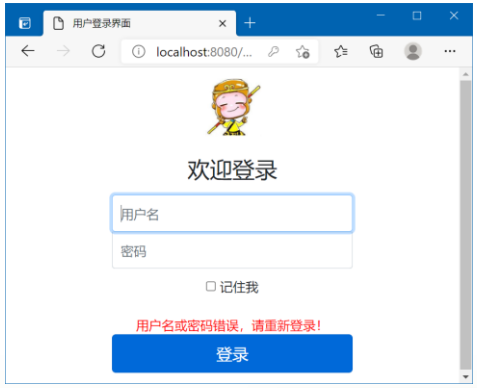


图2

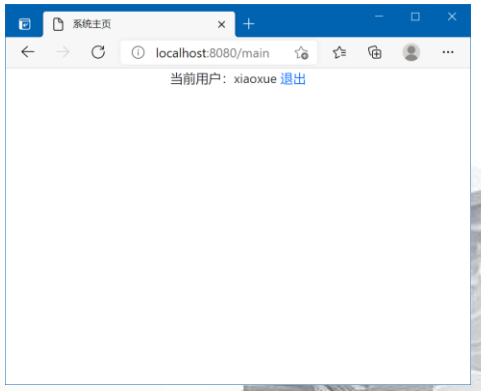


图3

5.5 自定义Web MVC配置

- 5.5.1 Web MVC配置简介
- 5.5.2 异常统一显示和处理
- 5.5.3 自定义Servlet三大组件

5.5.1 Web MVC配置简介

- Spring Boot整合Spring MVC进行Web开发提供了很多自动化配置，但在实际开发中还需要对一些功能进行扩展实现，为此就需要对Web MVC进行自定义配置。
  - ✧可以通过实现Spring Boot提供的WebMvcConfigurer接口来对MVC功能进行定制和扩展。
  - ✧编写一个@Configuration标注的自定义配置类，同时在项目启动类上添加@EnableWebMvc，还可关闭Spring Boot提供的所有关于MVC功能的默认配置，而启用自定义的配置。

可扩展的MVC功能配置

- 静态资源配置
  - 重写addResourceHandlers(ResourceHandlerRegistry registry)
- 拦截器配置
  - 重写addInterceptors(InterceptorRegistry registry)
- 跨域配置
  - 重写addCorsMappings(CorsRegistry registry)
- 视图控制器配置
  - 重写addViewControllers(ViewControllerRegistry registry)
- 视图解析器配置：
  - 重写configureViewResolvers(ViewResolverRegistry registry)
- 数据格式化器配置
  - 重写addFormatters(FormatterRegistry registry)
- 其他配置

示例1：数据格式化器配置

- 本示例通过创建和注册自定义数据格式化器实现将日期格式为“**yyy-MM-dd HH:mm:ss**”的字符串与后台**Date**类型形参进行数据绑定。

示例代码

```
import org.springframework.format.Formatter;
import org.springframework.stereotype.Component;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Locale;
@Component
public class DateFormatter implements Formatter<Date> {                    // 使用Formatter自定义日期转换器
        String datePattern = "yyyy-MM-dd HH:mm:ss";                        // 定义日期格式
        private SimpleDateFormat simpleDateFormat;
        @Override
        public String print(Date date, Locale locale) {
                return new SimpleDateFormat().format(date);
        }
        @Override
        public Date parse(String source, Locale locale) throws ParseException {
                simpleDateFormat = new SimpleDateFormat(datePattern);
                return simpleDateFormat.parse(source);
        }
}
```

创建自定义数据格式化器

示例代码

```
import com.itheima.convert.DateFormatter;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.format.FormatterRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class ConverterConfig  implements WebMvcConfigurer {
    @Autowired
    private DateFormatter myDateFormatter;

    @Override
    public void addFormatters(FormatterRegistry registry) {
        registry.addFormatter(myDateFormatter);
    }
}
```

注册自定义数据格式化器

示例2：拦截器配置

- 本示例通过定制和注册Spring Boot内置的LocaleChangeInterceptor拦截器实现根据前端选择的语言（反映在请求头中的Accept-Language信息上）自动进行语言切换。

示例代码

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.LocaleResolver;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
import org.springframework.web.servlet.i18n.LocaleChangeInterceptor;
import org.springframework.web.servlet.i18n.SessionLocaleResolver;
import java.util.Locale;
@Configuration
public class LocaleConfig implements WebMvcConfigurer {
    @Bean
    // 根据用户本次会话过程中的语义设定语言区域（如用户进入首页时选择的语言种类）
    public LocaleResolver localeResolver() {
        SessionLocaleResolver slr = new SessionLocaleResolver();
        slr.setDefaultLocale(Locale.US);   //设置默认语言区域
        return slr;
    }
```

示例代码

```
@Bean
// 使用SessionLocaleResolver存储语言区域时，须定制LocaleChangeInterceptor拦截器
public LocaleChangeInterceptor localeChangeInterceptor() {
    LocaleChangeInterceptor lci = new LocaleChangeInterceptor();
    lci.setParamName("loc");    //设置选择语言的参数名
    return lci;
}
@Override
// 注册拦截器，拦截器会在业务处理器（即Web控制器中URL映射处理方法）执行前执行
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(localeChangeInterceptor());
}
}
```

定制拦截器

注册拦截器

示例3：跨域配置

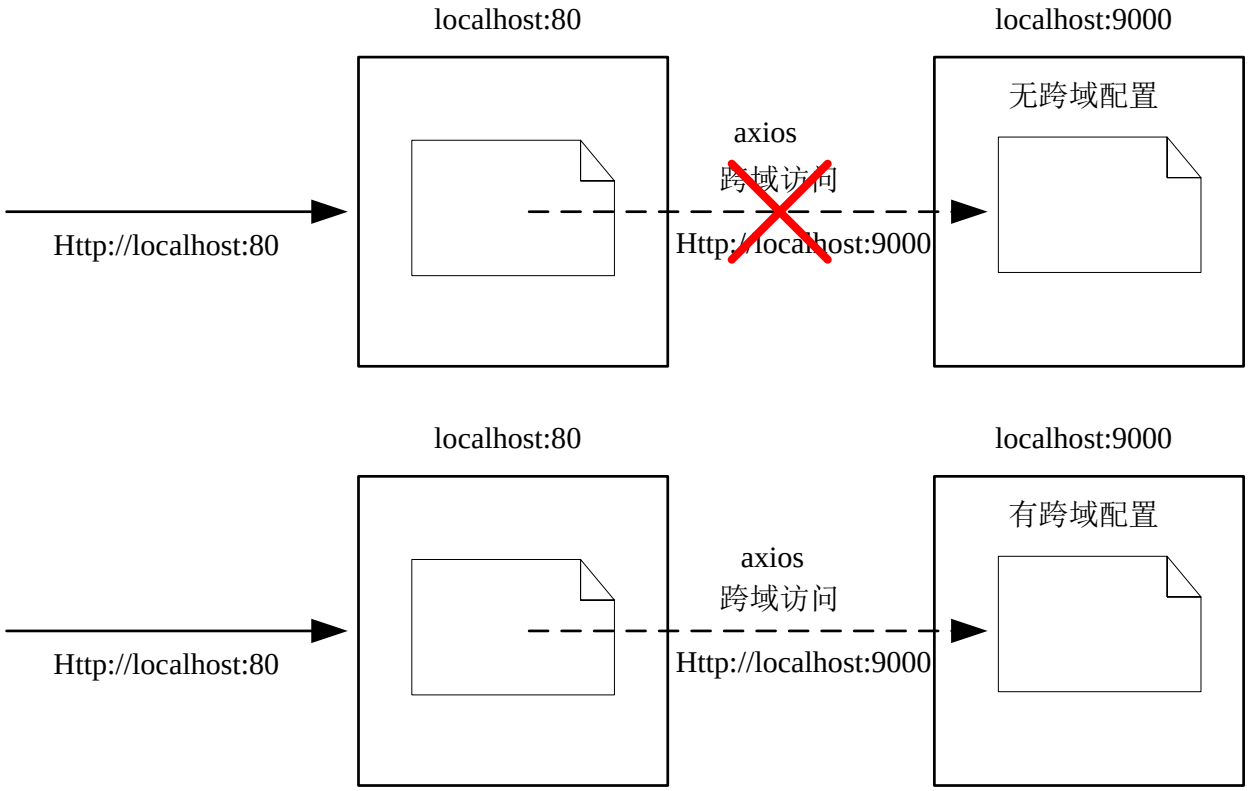
```
@Override
// 配置跨域路径映射
public void addCorsMappings(CorsRegistry registry) {
    registry.addMapping("/**")
            .allowedOrigins("*")
            .allowedMethods("PUT,POST,GET,DELETE,OPTIONS")
            .allowedHeaders("*");
}
```

其中
addMapping：配置允许访问的跨域资源路径；
allowedOrigins：配置允许访问跨域资源的请求源；
allowedMethods：配置允许访问该跨域资源服务器的请求方法，如：PUT、POST、GET、DELETE等；
allowedHeaders：配置允许访问跨域资源的请求header，如：X-TOKEN

什么叫跨域访问

- 跨域访问即通过HTTP请求，在一个域去请求另一个域的资源。
  - 只要协议、域名、端口有任何一个不相同，都会被当作是不同的域。

跨域访问


localhost:80
localhost:9000
无跨域配置
axios
跨域访问
Http://localhost:80
Http://localhost:9000
localhost:80
localhost:9000
有跨域配置
axios
跨域访问
Http://localhost:80
Http://localhost:9000
allowed-origins:”*”
allowed-headers:”*”
allowed-methods:”*”

5.5.2 异常统一显示和处理

- Spring Boot支持将所有类型的异常处理从各层中解耦出来，实现异常的统一显示和处理。
  - ✧1.异常统一显示：
    - 使用error.html页面。
  - ✧2.异常统一处理：
    - 使用@ControllerAdvice和@ExceptionHandler组合。

1.异常统一显示

- 在Spring Boot Web应用的templates目录下添加error.html页面，访问发生错误或异常时，Spring Boot将自动找到该页面作为错误显示页面。
- Spring Boot为错误显示页面提供了以下属性：
  - timestamp：错误发生时间；
  - status：HTTP状态吗；
  - error：错误原因；
  - path：错误发生时请求的URL路径。
  - message：异常消息（如果这个错误是由异常引起的）；
  - trace：异常跟踪信息（如果这个错误是由异常引起的）；
  - exception：异常的类名；
  - errors：BindingResult异常里的各种错误（如果这个错误是由异常引起的）；

error.html页面示例

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>异常统一显示界面</title>
</head>
<body>
<div th:text="'HTTP状态码：'+${status}"></div>
<div th:text="${#dates.format(timestamp,'yyyy-MM-dd HH:mm:ss')}"></div>
<div th:text="${message}"></div>
<div th:text="${error}"></div>
<div th:text="${path}"></div>
</body>
</html>
```

2.异常统一处理

- 异常统一处理相关注解：
  - ✧@ControllerAdvice：声明一个增强式控制器类，使用该控制器类可以实现全局异常处理、全局数据绑定以及全局数据预处理。
  - ✧@ExceptionHandler：声明一个异常统一处理方法，该方法被声明在增强式控制器类中将对当前Spring Boot应用的所有Web控制器中的URL映射有效。
    - 异常或错误如果不是从Web控制器中在URL映射时抛出，则仍由默认的error.html视图页面显示。

示例代码

```
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.thymeleaf.exceptions.TemplateProcessingException;
import java.sql.SQLException;
@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(value = Exception.class)
    public String handleException(Model model, Exception e) {
        if (e instanceof SQLException) {
            System.out.println("数据库异常处理");
        } else if (e instanceof TemplateProcessingException) {
            System.out.println("模板引擎异常处理");
        } else {
            System.out.println("未知异常处理");
        }
        model.addAttribute("exc", e);
        return "errorhandler";                          //在errorhandler.html视图页面中显示异常e信息
    }
}
```

示例代码

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.thymeleaf.exceptions.TemplateProcessingException;
import java.sql.SQLException;
@Controller
public class ExceptionController{
    @RequestMapping("/db")
    public void db() throws SQLException {
        throw new SQLException("数据库异常");
    }
    @RequestMapping("/template")
    public void template() throws TemplateProcessingException {
        throw new TemplateProcessingException("模板引擎异常");
    }
    @RequestMapping("/unknown")
    public void unknown() throws Exception {
        throw new Exception("未知异常");
    }
}
```

5.5.3 自定义Servlet三大组件

- 在早期的Web MVC应用中，Servlet是Web控制器，进行Servlet开发时，通常首先自定义Servlet、Filter、Listener三大组件，然后在web.xml文件中对其进行注册配置。
- Spring Boot移除了web.xml文件，如果需要自定义Servlet、Filter、Listener，并对其进行注册配置，有两种方式：
  - ✧①注解配置方式：
    - 使用@WebServlet、@WebFilter、@WebListener
  - ✧②Java Config方式
    - 使用RegistrationBean实现注册

Servlet三大组件作用

- **Servlet**
  - ✧ 接收用户请求HttpServletRequest，并向用户生成响应HttpServletResponse
- **Filter**
  - ✧ 在HttpServletRequest到达Servlet前拦截客户的HttpServletRequest，根据需要检查或修改HttpServletRequest请求头和数据；
  - ✧ 在HttpServletResponse到达客户端前拦截HttpServletResponse，根据需要检查或修改HttpServletResponse响应头和数据；
- **Listener**
  - ✧ 初始化和销毁Servlet上下文环境
  - ✧ 随Web应用的启动而启动，随Web应用的终止而销毁
  - ✧ Servlet容器提供了很多Listener接口，如ServletRequestListener、HttpSessionListener、ServletContextListener

1.注解配置方式案例

- 本案例基于注解配置方式，使用@WebServlet、@WebFilter、@WebListener分别自定义并注册Servlet、Filter、Listener三大组件，展示三大组件的执行顺序和作用。
- 搭建步骤：
  - ✧①自定义Servlet类
  - ✧②自定义Filter类
  - ✧③自定义Listener类
  - ✧④开启Servlet组件扫描方式
  - ✧⑤效果测试

①自定义Servlet类

- 在chapter05项目中新建一个com.itheima.servletdev包，并在包中新建一个Servlet类MyServlet，实现接收用户请求“/myServlet”或“/annotationServlet”，并向用户生成相同响应。

MyServlet类

```
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
@WebServlet(value = {"/myServlet","/annotationServlet"})
public class MyServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        this.doPost(req, resp);
    }
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        resp.getWriter().write("hello MyServlet");
    }
}
```

②自定义Filter类

- 在chapter05项目的com.itheima.servletdev包中新建一个Filter类MyFilter，实现对用户请求“/myServlet”或“/annotationServlet”的拦截处理。

MyFilter类

```
import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
import java.io.IOException;
@WebFilter(value = {"/myServlet","/annotationServlet"})
public class MyFilter implements Filter {
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        System.out.println("filterInitialized ...");
    }
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse,
                         FilterChain filterChain) throws IOException, ServletException {
        System.out.println("hello MyFilter");
        filterChain.doFilter(servletRequest,servletResponse);
    }
    @Override
    public void destroy() {
        System.out.println("filterDestroyed ...");
    }
}
```

③自定义Listener类

- 在chapter05项目的com.itheima.servletdev包中新建一个Listener类MyListener，实现Servlet上下文环境的初始化和销毁。

MyListener类

```
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.annotation.WebListener;

@WebListener
public class MyListener implements ServletContextListener {
    @Override
    public void contextInitialized(ServletContextEvent servletContextEvent) {
        System.out.println("contextInitialized ...");
    }
    @Override
    public void contextDestroyed(ServletContextEvent servletContextEvent) {
        System.out.println("contextDestroyed ...");
    }
}
```

④开启Servlet组件扫描方式

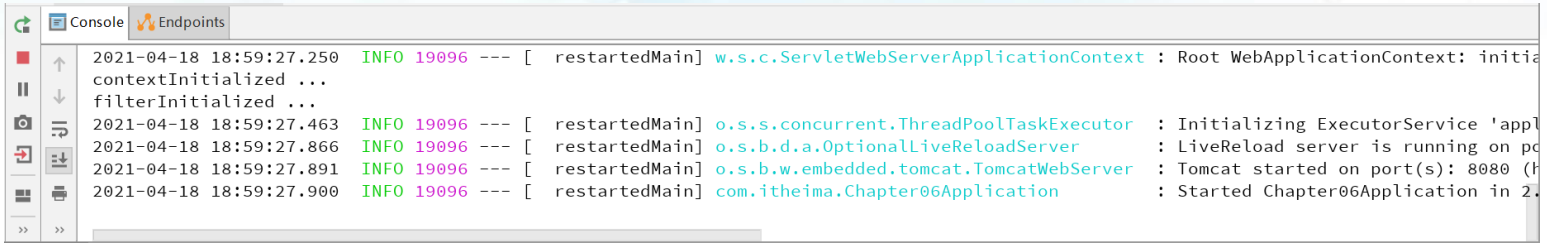
- 在chapter05项目的启动类上添加@ServletComponentScan开启基于注解方式的Servlet组件扫描支持。

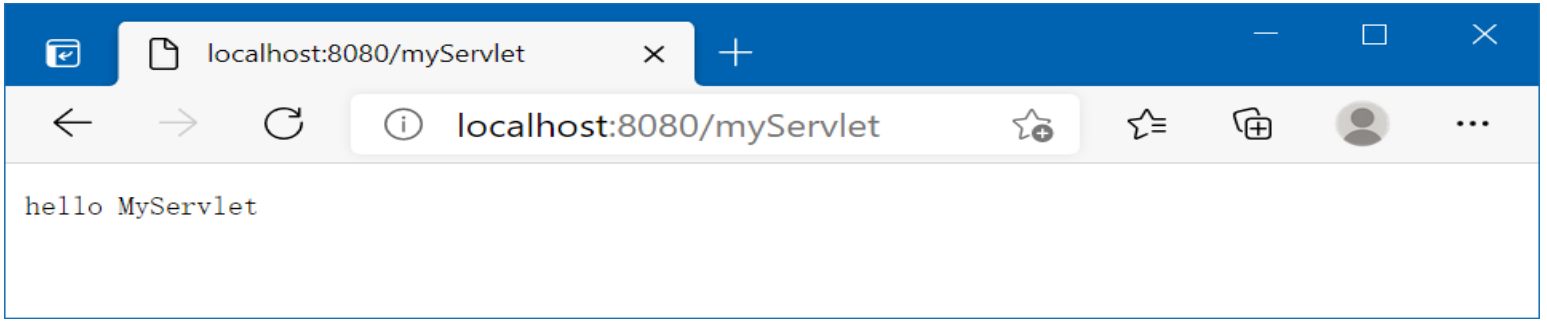
```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.web.servlet.ServletComponentScan;
@ServletComponentScan // 开启基于注解方式的Servlet组件扫描支持
@SpringBootApplication
public class chapter05Application {
    public static void main(String[] args) {
        SpringApplication.run(chapter05Application.class, args);
    }
}
```

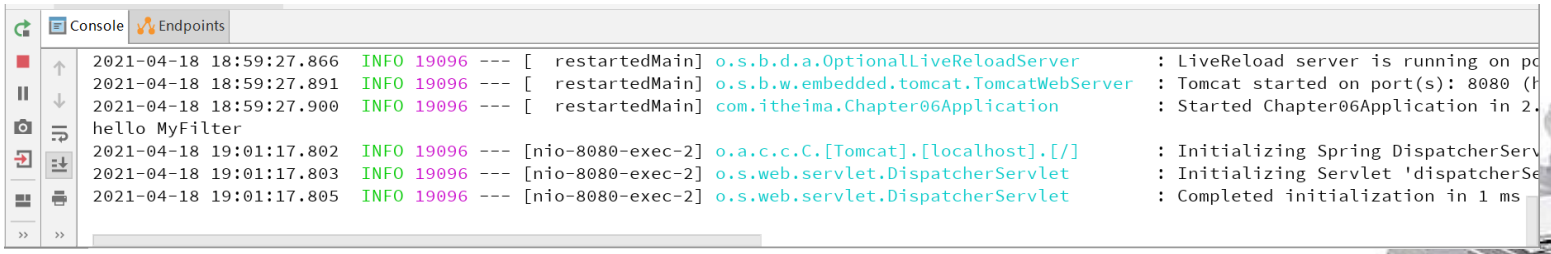
⑤效果测试

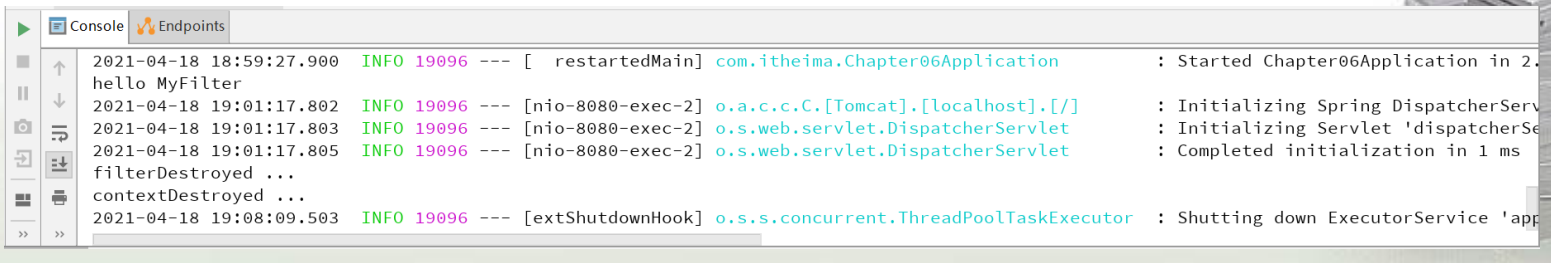
- 启动项目进行测试，在浏览器上访问**http://localhost:8080/myServlet**
- 随后，单击**IDEA**工具控制台左侧的【**Exit**】按钮关闭当前项目。

效果测试


Console
Endpoints
2021-04-18 18:59:27.250 INFO 19096 --- [restartedMain] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initia
contextInitialized ...
filterInitialized ...
2021-04-18 18:59:27.463 INFO 19096 --- [restartedMain] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'appl
2021-04-18 18:59:27.866 INFO 19096 --- [restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on pc
2021-04-18 18:59:27.891 INFO 19096 --- [restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (H
2021-04-18 18:59:27.900 INFO 19096 --- [restartedMain] com.itheima.Chapter06Application : Started Chapter06Application in 2.


localhost:8080/myServlet
localhost:8080/myServlet
hello MyServlet


Console
Endpoints
2021-04-18 18:59:27.866 INFO 19096 --- [restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on pc
2021-04-18 18:59:27.891 INFO 19096 --- [restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (H
2021-04-18 18:59:27.900 INFO 19096 --- [restartedMain] com.itheima.Chapter06Application : Started Chapter06Application in 2.
hello MyFilter
2021-04-18 19:01:17.802 INFO 19096 --- [nio-8080-exec-2] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServ
2021-04-18 19:01:17.803 INFO 19096 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherSe
2021-04-18 19:01:17.805 INFO 19096 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms


Console
Endpoints
2021-04-18 18:59:27.900 INFO 19096 --- [restartedMain] com.itheima.Chapter06Application : Started Chapter06Application in 2.
hello MyFilter
2021-04-18 19:01:17.802 INFO 19096 --- [nio-8080-exec-2] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServ
2021-04-18 19:01:17.803 INFO 19096 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherSe
2021-04-18 19:01:17.805 INFO 19096 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms
filterDestroyed ...
contextDestroyed ...
2021-04-18 19:08:09.503 INFO 19096 --- [extShutdownHook] o.s.s.concurrent.ThreadPoolTaskExecutor : Shutting down ExecutorService 'app

2. Java Config方式案例

- 本案例基于Java Config配置方式，自定义Servlet、Filter、Listener三大组件，并分别使用三大组件的RegistrationBean实现注册，展示三大组件的执行顺序和作用。
- 搭建步骤：
  - ✧①自定义Servlet类
  - ✧②自定义Filter类
  - ✧③自定义Listener类
  - ✧④注册三大组件
  - ✧⑤效果测试

①自定义Servlet类

- 在chapter05项目中新建一个com.itheima.servletdev包，并在包中新建一个Servlet类MyServlet，实现接收用户请求，并向用户生成相同响应。

MyServlet类

```
import javax.servlet.ServletException;
import org.springframework.stereotype.Component;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
@Component
public class MyServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        this.doPost(req, resp);
    }
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        resp.getWriter().write("hello MyServlet");
    }
}
```

②自定义Filter类

- 在chapter05项目的com.itheima.servletdev包中新建一个Filter类MyFilter，实现对用户请求的拦截处理。

MyFilter类

```
import javax.servlet.*;
import org.springframework.stereotype.Component;
import java.io.IOException;
@Component
public class MyFilter implements Filter {
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        System.out.println("filterInitialized ...");
    }
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse,
                         FilterChain filterChain) throws IOException, ServletException {
        System.out.println("hello MyFilter");
        filterChain.doFilter(servletRequest,servletResponse);
    }
    @Override
    public void destroy() {
        System.out.println("filterDestroyed ...");
    }
}
```

③自定义Listener类

- 在chapter05项目的com.itheima.servletdev包中新建一个Listener类MyListener，实现Servlet上下文环境的初始化和销毁。

MyListener类

```
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import org.springframework.stereotype.Component;
@Component
public class MyListener implements ServletContextListener {
    @Override
    public void contextInitialized(ServletContextEvent servletContextEvent) {
        System.out.println("contextInitialized ...");
    }
    @Override
    public void contextDestroyed(ServletContextEvent servletContextEvent) {
        System.out.println("contextDestroyed ...");
    }
}
```

④注册三大组件

- 在chapter05项目的com.itheima.config包中新建一个ServletConfig类，使用三大组件的RegistrationBean分别注册配置MyServlet、MyFilter和MyListener，配置对用户请求“/myServlet”或“/jcServlet”的接收和拦截。

ServletConfig类

```
import com.itheima.servletdev.MyFilter;
import com.itheima.servletdev.MyListener;
import com.itheima.servletdev.MyServlet;
import org.springframework.boot.web.servlet.FilterRegistrationBean;
import org.springframework.boot.web.servlet.ServletListenerRegistrationBean;
import org.springframework.boot.web.servlet.ServletRegistrationBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import java.util.Arrays;
@Configuration
public class ServletConfig {
    @Bean
    public ServletRegistrationBean servletRegistrationBean(MyServlet myServlet){
        ServletRegistrationBean registrationBean = new ServletRegistrationBean(myServlet);
        registrationBean.setUrlMappings(Arrays.asList("/myServlet","/jcServlet"));
        return registrationBean;
    }
```

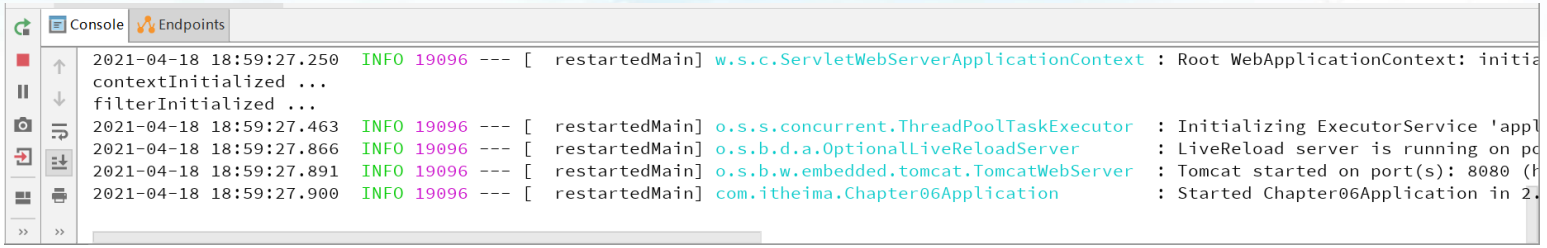
ServletConfig类

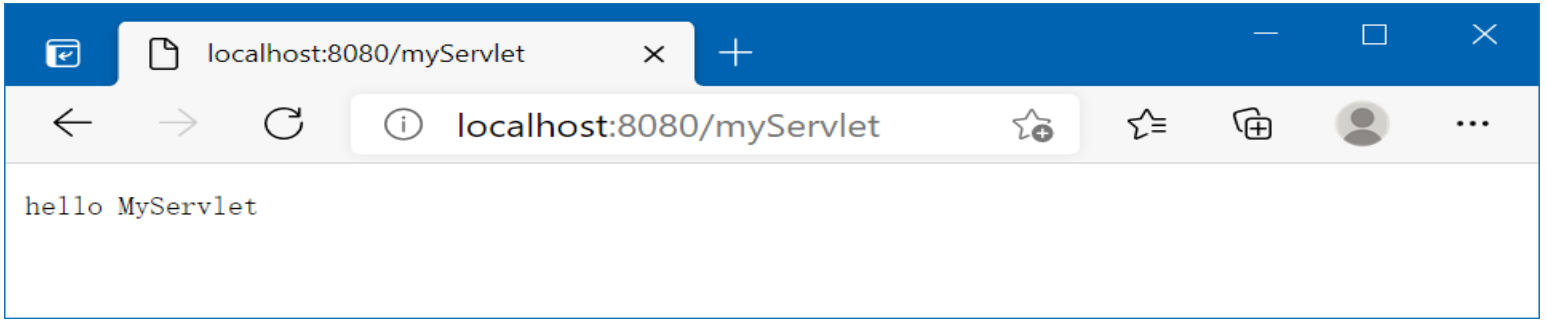
```
  @Bean
   public FilterRegistrationBean filterRegistrationBean(MyFilter filter){
       FilterRegistrationBean registrationBean = new FilterRegistrationBean(filter);
       registrationBean.setUrlPatterns(Arrays.asList("/myServlet","/jcServlet"));
       return registrationBean;
   }
   @Bean
   public ServletListenerRegistrationBean servletListenerRegistrationBean(MyListener myListener){
       ServletListenerRegistrationBean registrationBean = new ServletListenerRegistrationBean(myListener);
       return registrationBean;
   }
}
```

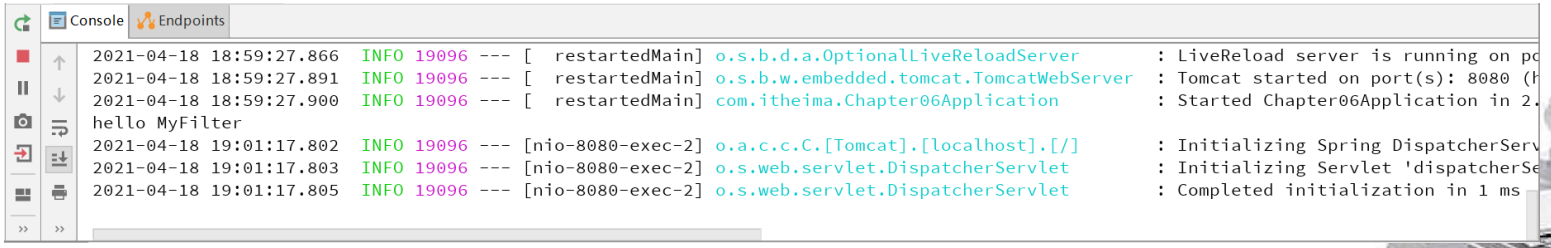
⑤效果测试

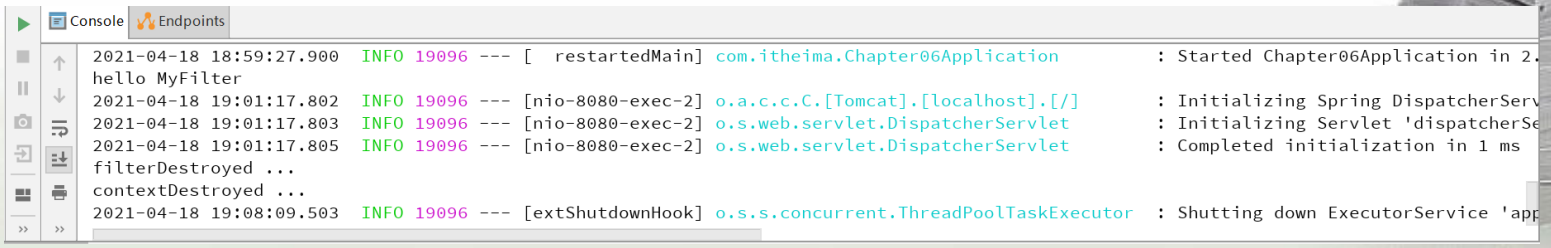
- 启动项目进行测试，在浏览器上访问 **http://localhost:8080/myServlet**
- 随后，单击**IDEA**工具控制台左侧的【**Exit**】按钮关闭当前项目。

效果测试


Console
Endpoints
2021-04-18 18:59:27.250 INFO 19096 --- [restartedMain] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initia
contextInitialized ...
filterInitialized ...
2021-04-18 18:59:27.463 INFO 19096 --- [restartedMain] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'appl
2021-04-18 18:59:27.866 INFO 19096 --- [restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on pc
2021-04-18 18:59:27.891 INFO 19096 --- [restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (H
2021-04-18 18:59:27.900 INFO 19096 --- [restartedMain] com.itheima.Chapter06Application : Started Chapter06Application in 2.


localhost:8080/myServlet
localhost:8080/myServlet
hello MyServlet


Console
Endpoints
2021-04-18 18:59:27.866 INFO 19096 --- [restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on pc
2021-04-18 18:59:27.891 INFO 19096 --- [restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (H
2021-04-18 18:59:27.900 INFO 19096 --- [restartedMain] com.itheima.Chapter06Application : Started Chapter06Application in 2.
hello MyFilter
2021-04-18 19:01:17.802 INFO 19096 --- [nio-8080-exec-2] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServ
2021-04-18 19:01:17.803 INFO 19096 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherSe
2021-04-18 19:01:17.805 INFO 19096 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms


Console
Endpoints
2021-04-18 18:59:27.900 INFO 19096 --- [restartedMain] com.itheima.Chapter06Application : Started Chapter06Application in 2.
hello MyFilter
2021-04-18 19:01:17.802 INFO 19096 --- [nio-8080-exec-2] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServ
2021-04-18 19:01:17.803 INFO 19096 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherSe
2021-04-18 19:01:17.805 INFO 19096 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms
filterDestroyed ...
contextDestroyed ...
2021-04-18 19:08:09.503 INFO 19096 --- [extShutdownHook] o.s.s.concurrent.ThreadPoolTaskExecutor : Shutting down ExecutorService 'app

本章小结

- 本章具体讲解了：
  - ✧5.1 MVC设计概述
  - ✧5.2 使用视图技术Thymeleaf
  - ✧5.3 使用控制器
  - ✧5.4 使用拦截器
  - ✧5.5 自定义Web MVC配置


聪明等于勤奋
努力就是幸福