第三章 项目中最常用的 Web 相关技术

1. Spring Boot2.x 过滤器基础入门&实战项目场景实现

1.1过滤器

过滤器是什么?

它是基于Servlet 技术实现的， 简单的来说，过滤器就是起到过滤的作用，在web项目开发中帮我们过滤一些指定的 url做一些特殊的处理。 **过滤器主要做什么？**

- 过滤掉一些不需要的东西，例如一些错误的请求。
- 也可以修改请求和相应的内容。
- 也可以拿来过滤未登录用户

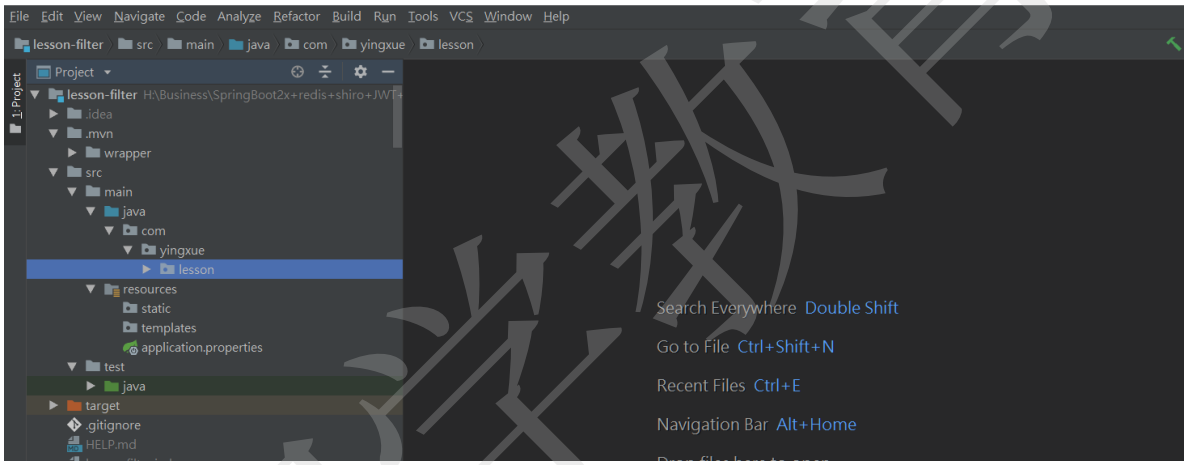
过滤器的代码实现

过滤器(filter)有三个方法，其中初始化（init）和摧毁（destroy）方法一般不会用到，主要用到的是doFilter这个方法。

怎么过滤呢？

如果过滤通过，则在doFilter执行 `filterChain.doFilter(request,response);`。

1.2 创建项目lesson-filter



1.3 Filter 快速入门

那么在springBoot中如何使用过滤器呢？

自定义Filter有两种实现方式，第一种是使用@WebFilter，第二种是使用 FilterRegistrationBean,下面我们分别来实现

1.3.1 @WebFilter 实现

@WebFilter 用于将一个类声明为**过滤器**，该注解将会在部署时被容器处理，容器将根据具体的属性配置将相应的类部署为过滤器。

属性名	类型	描述
filterName	String	指定该Filter的名称
urlPatterns	String	指定该Filter所拦截的URL。
value	String	与 urlPatterns 一致

1. 创建一个MyFilter.java实现Filter接口

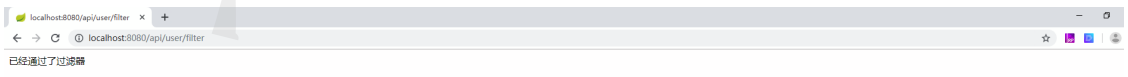
```
package com.yingxue.lesson.filter;

import org.springframework.stereotype.Component;

import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;
import java.io.IOException;
```

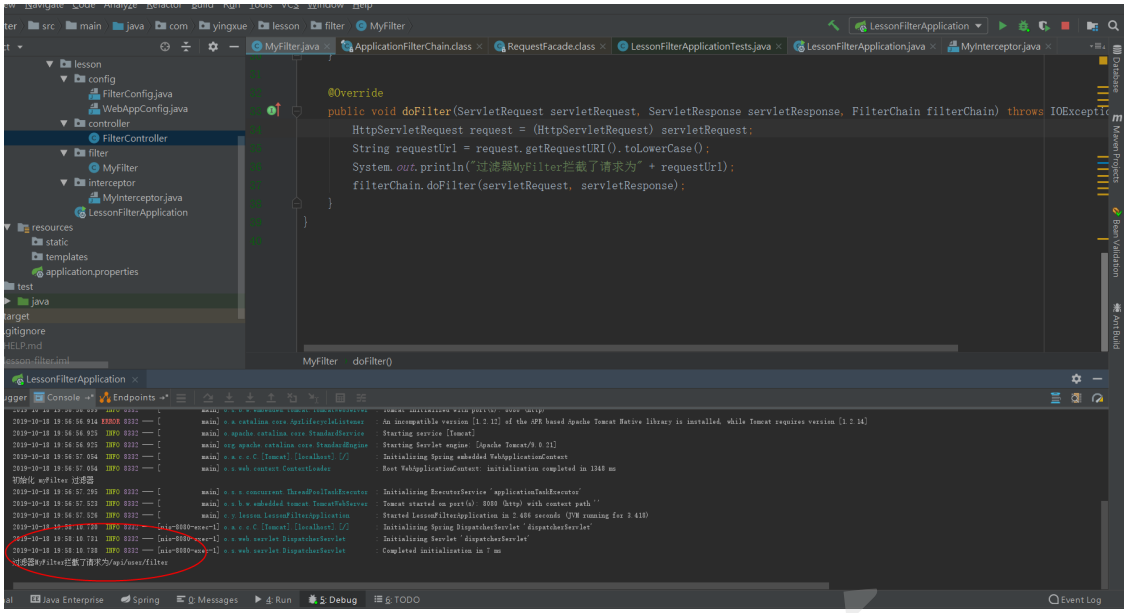
```
@WebFilter(urlPatterns = "/api/*",filterName = "myFilter")
@Order(1)//指定过滤器的执行顺序,值越大越靠后执行
public class MyFilter implements Filter {
  @Override
public void init(FilterConfig filterConfig) throws ServletException {
    System.out.println("初始化过滤器");
}
@Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse,
FilterChain filterChain) throws IOException, ServletException {
        HttpServletRequest request= (HttpServletRequest) servletRequest;
        String uri=request.getRequestURI();
        String method=request.getMethod();
        System.out.println(uri+" "+method+"哈哈我进入了 MyFilter 过滤器了");
        filterChain.doFilter(servletRequest,servletResponse);

    }
}
```

2. 启动类加上 @ServletComponentScan 注解
3. 创建一个 FilterController 接口

```
package com.yingxue.lesson.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

/**

- @ClassName: FilterController
- TODO:类文件简单描述
- @Author: 小霍
- @UpdateUser: 小霍
- @Version: 0.0.1
*/
@RestController
@RequestMapping("/api")
public class HelloController {

    @GetMapping("/user/filter")
    public String hello(){
        return "哈哈我通过了过滤器";
    }
}
```

4. 测试





1.3.2 FilterRegistrationBean 实现

1. 创建 FilterConfig

```
package com.yingxue.lesson.config;

import com.yingxue.lesson.filter.MyFilter;
import org.springframework.boot.web.servlet.FilterRegistrationBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

/**
 * @ClassName: FilterConfig
 * TODO:类文件简单描述
 * @Author: 小霍
 * @UpdateUser: 小霍
 * @Version: 0.0.1
 */

@Configuration
public class FilterConfig {
    @Bean
     public MyFilter myFilter(){
         return new MyFilter();
     }
     @Bean
     public FilterRegistrationBean getFilterRegistrationBean(MyFilter myFilter){
         FilterRegistrationBean filterRegistrationBean=new FilterRegistrationBean();
         /**
          * 设置过滤器
          */
         filterRegistrationBean.setFilter(MyFilter());
         /**
          * 拦截路径
          */
         filterRegistrationBean.addUrlPatterns("/api/*");
         /**
          * 设置名称
          */
         filterRegistrationBean.setName("myFilter");
         /**
          * 设置访问优先级 值越小越高
          */
         filterRegistrationBean.setOrder(1);
         return filterRegistrationBean;
     }
}
```

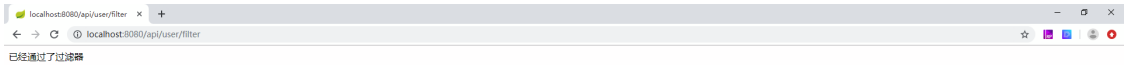
2. 修改 MyFilter.java

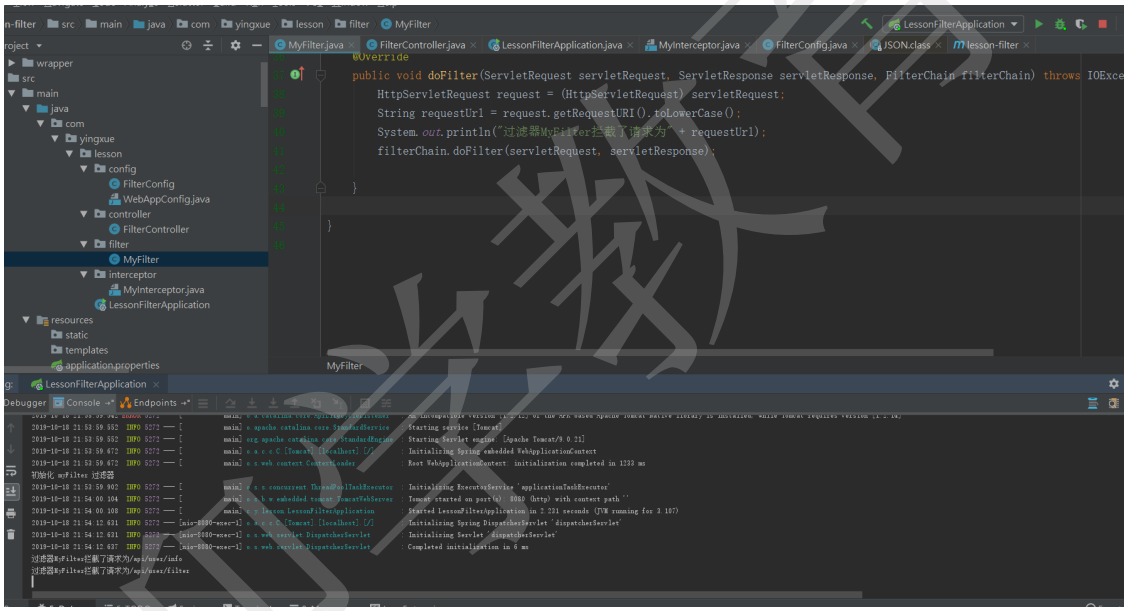
```
//@WebFilter(urlPatterns ={"/api/*"},filterName = "myFilter")
```

3. 修改启动类

```
//@ServletComponentScan
```

4. 测试





1.4 过滤校验用户是否登录实战

1. 修改 application.properties 加入开发接口通配地址

```
#凡是请求地址层级带有 open 都放行
open.url=/**/open/**
```

2. 修改 MyFilter

```
//@WebFilter(urlPatterns ={"/*"},filterName = "myFilter")
public class MyFilter implements Filter {
    @Value("${open.url}")
    private String openUrl;
    @Override
    public void init(FilterConfig filterConfig) {
        System.out.println("初始化 myFilter 过滤器");
    }

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse,
FilterChain filterChain) throws IOException, ServletException {
        HttpServletRequest request = (HttpServletRequest) servletRequest;
        String requestUrl = request.getRequestURI();
        System.out.println("过滤器MyFilter拦截了请求为" + requestUrl);
```

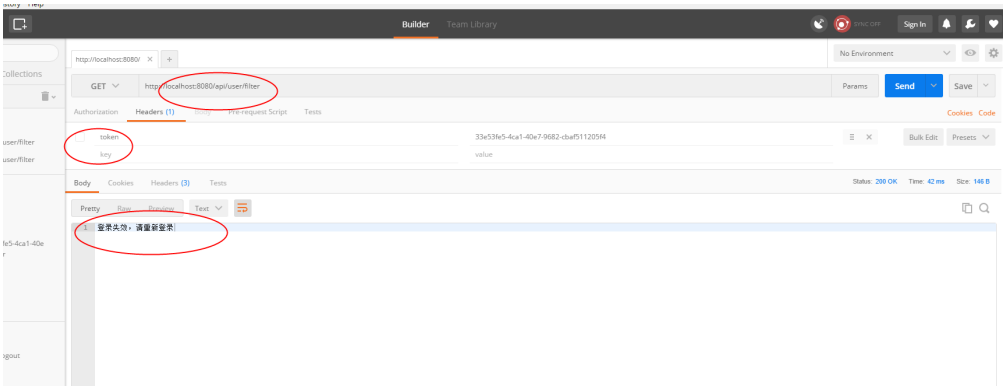
```
        //首先校验是否是开放 api
        //是直接放行，否再校验token
        PathMatcher matcher = new AntPathMatcher();
        if(matcher.match(openUrl,requestUrl)){
            filterChain.doFilter(servletRequest,servletResponse);
        }else {
            String token=request.getHeader("token");
            if(StringUtils.isEmpty(token)){
                servletRequest.getRequestDispatcher("/api/open/unLogin").forward(servletRequest,
                        servletResponse);
            }else {
                filterChain.doFilter(servletRequest,servletResponse);
            }
        }

    }

}
```

3. 新增 未登录接口、首页接口

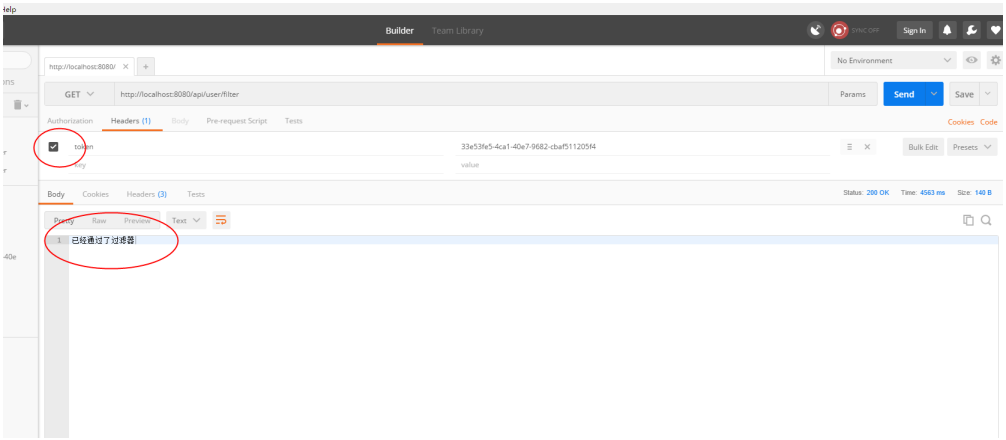
```
@GetMapping("/open/home/info")
    public Map<String,String> getHome(){
        Map<String,String> map=new HashMap<>();
        map.put("游客","欢迎访问首页");
        return map;
    }

    @GetMapping("/open/unLogin")
    public String getUnauthorized(){
        return "登录失效，请重新登录";
    }
```

4. 测试
    1. 首先访问 开放接口

    2. 访问需权鉴接口
        1. 不带 token



        2. 带上 token



2. Spring Boot2.x 拦截器基础入门&实战项目场景实现

2.1拦截器

拦截器是什么?

简单的来说，就是一道阀门，在某个方法被访问之前，进行拦截，然后在之前或之后加入某些操作，拦截器是AOP 的一种实现策略。

拦截器主要做什么？

对正在运行的流程进行干预。

拦截器的代码实现。

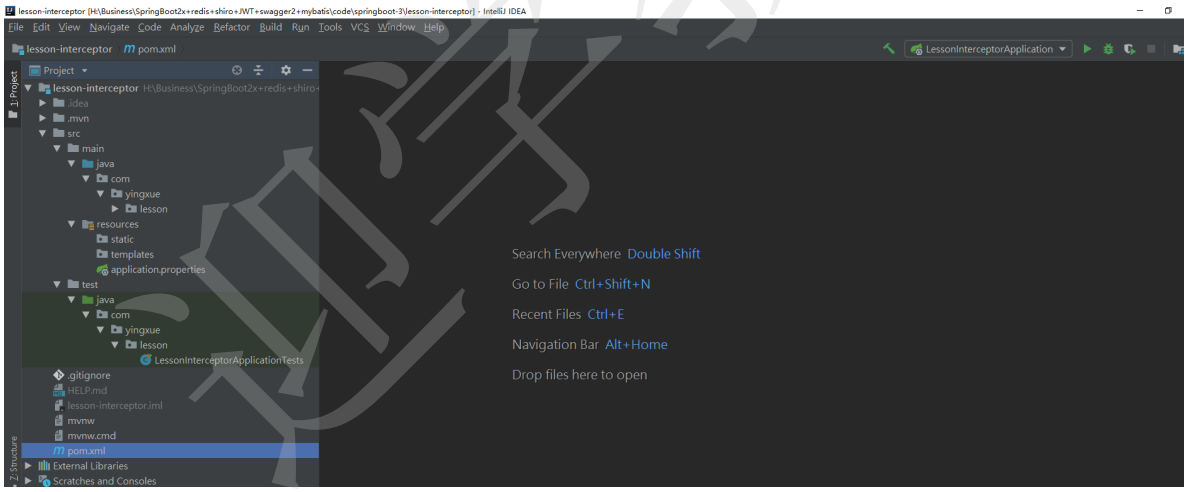
拦截器也主要有三个方法：

其中preHandle是在请求之前就进行调用，如果该请求需要被拦截，则返回false，否则true;

postHandle是在请求之后进行调用，无返回值;

afterCompletion是在请求结束的时候进行调用，无返回值。

2.2 创建项目 lesson-interceptor



2.3 Interceptor 快速入门

那么在 springBoot 中如何使用拦截器呢？

步骤：创建一个类实现 HandlerInterceptor 接口，再创建一个配置类实现 WebMvcConfigurer接口 ，重写 addInterceptors 方法。

1. 创建我们自己的拦截器类并实现 HandlerInterceptor 接口。

```
package com.yingxue.lesson.interceptor;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.util.AntPathMatcher;
import org.springframework.util.PathMatcher;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```
/**
 * @ClassName: MyInterceptor
 * TODO:类文件简单描述
 * @Author: 小霍
 * @UpdateUser: 小霍
 * @Version: 0.0.1
 */
public class MyInterceptor implements HandlerInterceptor {
    @Value("${open.url}")
    private String openUrl;
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object
handler) throws Exception {
        System.out.println("MyInterceptor....在请求处理之前进行调用（Controller方法调用之前）");
        String requestUrl=request.getRequestURI();
        System.out.println("过滤器MyFilter拦截了请求为"+requestUrl);
        return true;

    }


    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object
handler, ModelAndView modelAndView) throws Exception {
        System.out.println("MyInterceptor...请求处理之后进行调用，但是在视图被渲染之前（Controller方法调用之
后）");
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object
handler, Exception ex) throws Exception {
        System.out.println("MyInterceptor....在整个请求结束之后被调用，也就是在DispatcherServlet 渲染了对
应的视图之后执行（主要是用于进行资源清理工作）");
    }
}
```

2. 修改 application.properties 加入开发接口通配地址

```
#凡是请求地址层级带有 open 都放行
open.url=/**/open/**
```

3. 创建一个 Java 实现 WebMvcConfigurer ，并重写 addInterceptors 方法。

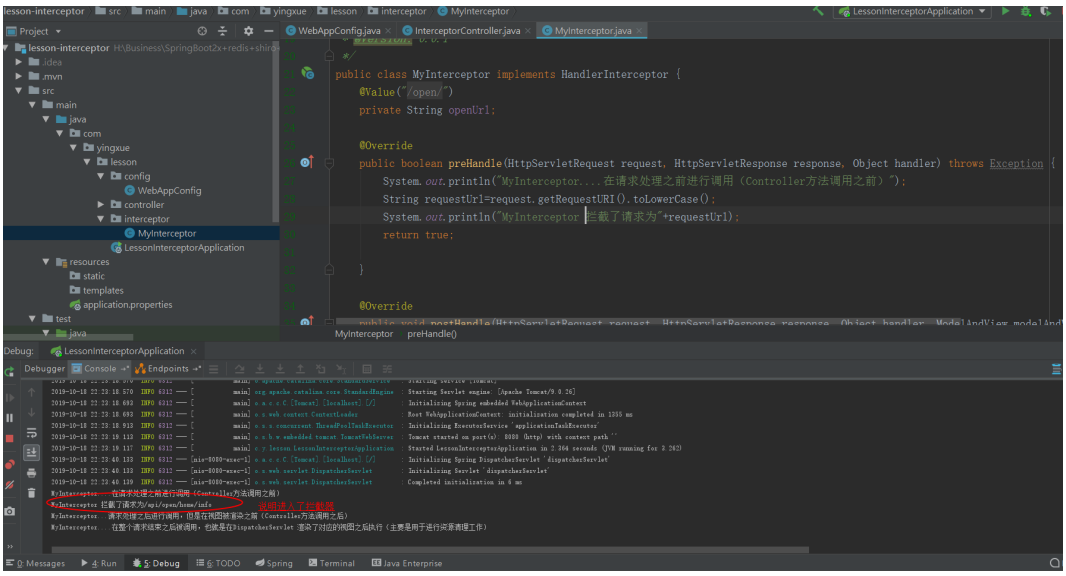
```
package com.yingxue.lesson.config;

import com.yingxue.lesson.interceptor.MyInterceptor;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

/**
 * @ClassName: WebAppConfig
 * TODO:类文件简单描述
 * @Author: 小霍
 * @UpdateUser: 小霍
 * @Version: 0.0.1
 */
@Configuration
public class WebAppConfig implements WebMvcConfigurer {
    @Value("${open.url}")
    private String openUrl;
    @Bean
    public MyInterceptor getMyInterceptor(){
        return new MyInterceptor();
    }
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
```

```
 registry.addInterceptor(getMyInterceptor()).addPathPatterns("/api/**").excludePathPatterns(openUrl
);
    }
}
```

4. 创建开放首页接口

```
package com.yingxue.lesson.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.HashMap;
import java.util.Map;

/**
 * @ClassName: InterceptorController
 * TODO:类文件简单描述
 * @Author: 小霍
 * @UpdateUser: 小霍
 * @Version: 0.0.1
 */
@RestController
@RequestMapping("/api")
public class InterceptorController {
       @GetMapping("/home/open/info")
    public String home(){
        return "欢迎来到首页";
    }

    @GetMapping("/user/interceptor")
    public String interceptor(){
        return "我被拦截了并通过了拦截器";
    }
}
```

5. 测试
    1. 带有open http://localhost:8080/api/open/home/info 不拦截



2.4 拦截校验用户是否登录实战

1. 加入需要权鉴接口、未登录接口

```
    @GetMapping("/user/filter")
    public String testFilter(){
        return "已经通过了拦截器";
    }
    @GetMapping("/open/unLogin")
    public String getUnauthorized(){
        return "登录失效，请重新登录";
    }
```

2. 修改拦截器校验逻辑

```
public class MyInterceptor implements HandlerInterceptor {
    @Value("${open.url}")
    private String openUrl;

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object
handler) throws Exception {
        System.out.println("MyInterceptor....在请求处理之前进行调用（Controller方法调用之前）");
        System.out.println("MyInterceptor 拦截了请求为"+requestUrl);
         String requestUrl=request.getRequestURI();
        System.out.println(requestUrl+"被 MyInterceptor 拦截了");
        //判断是否携带凭证就可以了
        String token = request.getHeader("token");
        if(StringUtils.isEmpty(token)){
            request.getRequestDispatcher("/api/open/unLogin").forward(request,response);
            return false;
        }
        return true;

        }

    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object
handler, ModelAndView modelAndView) throws Exception {
        System.out.println("MyInterceptor...请求处理之后进行调用，但是在视图被渲染之前（Controller方法调用之
后）");
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object
handler, Exception ex) throws Exception {
        System.out.println("MyInterceptor....在整个请求结束之后被调用，也就是在DispatcherServlet 渲染了对
应的视图之后执行（主要是用于进行资源清理工作）");
    }
```

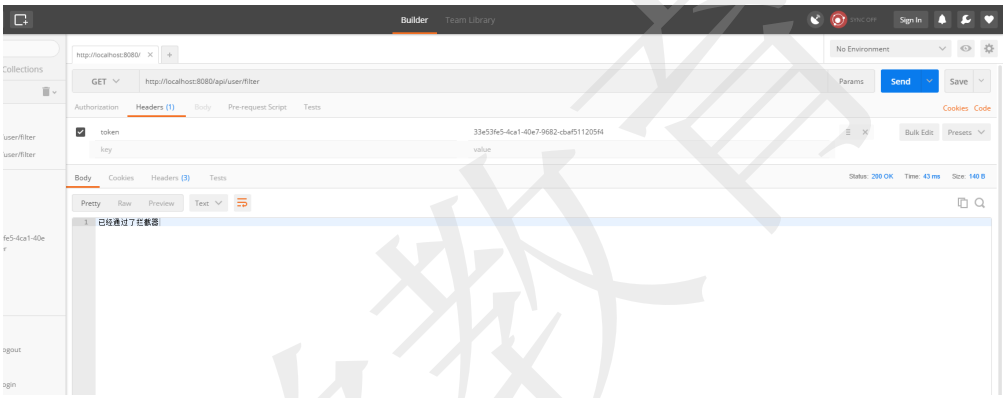
```
}
```

3. 测试

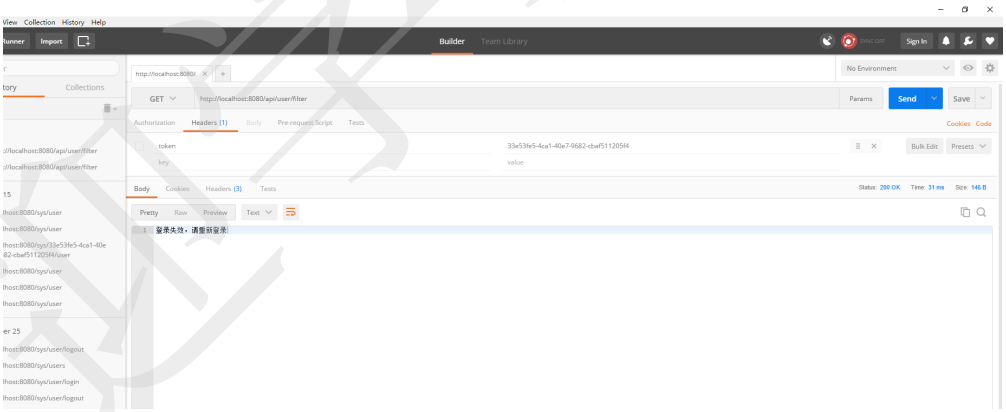
    1. 首先访问 开放接口

    2. 访问需权鉴接口

        1. 带 token

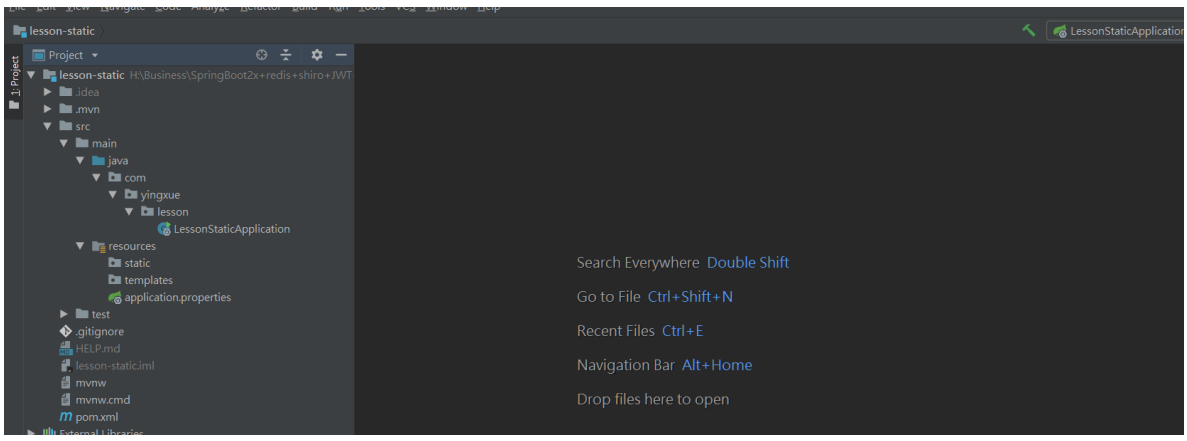


        2. 不带 token



3. Spring Boot 静态资源访问

3.1 创建项目



3.2 源码分析

1. 我们打开 ResourceProperties 资源配置类

```
@ConfigurationProperties(
    prefix = "spring.resources",
    ignoreUnknownFields = false
)
public class ResourceProperties {
    private static final String[] CLASSPATH_RESOURCE_LOCATIONS = new String[]{"classpath:/META-INF/resources/", "classpath:/resources/", "classpath:/static/", "classpath:/public/"};
```

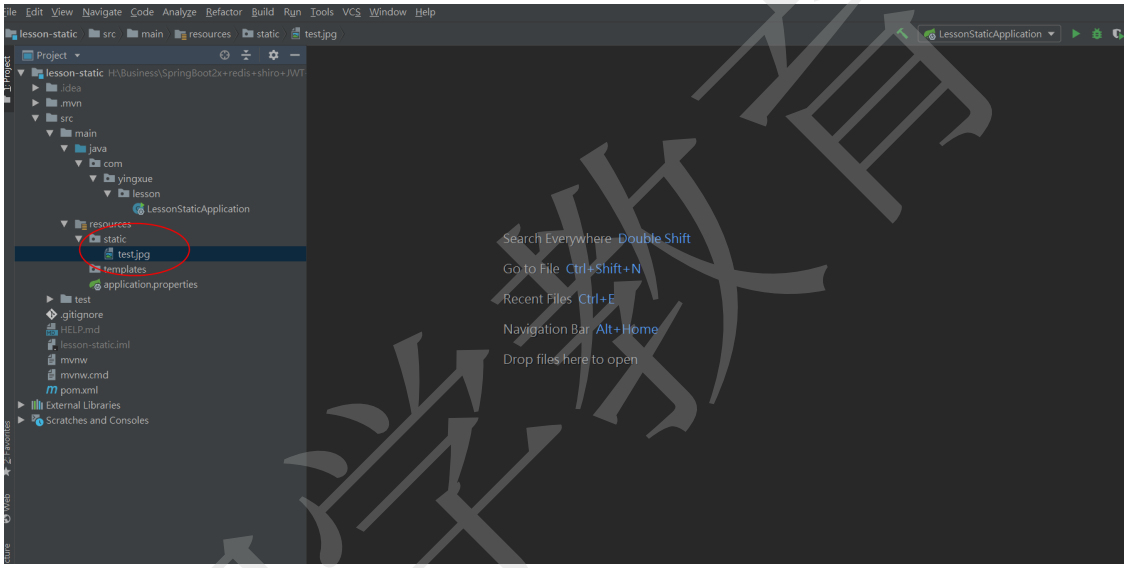
默认的静态资源路径为：

- classpath:/META-INF/resources/
- classpath:/resources/
- classpath:/static/
- classpath:/public

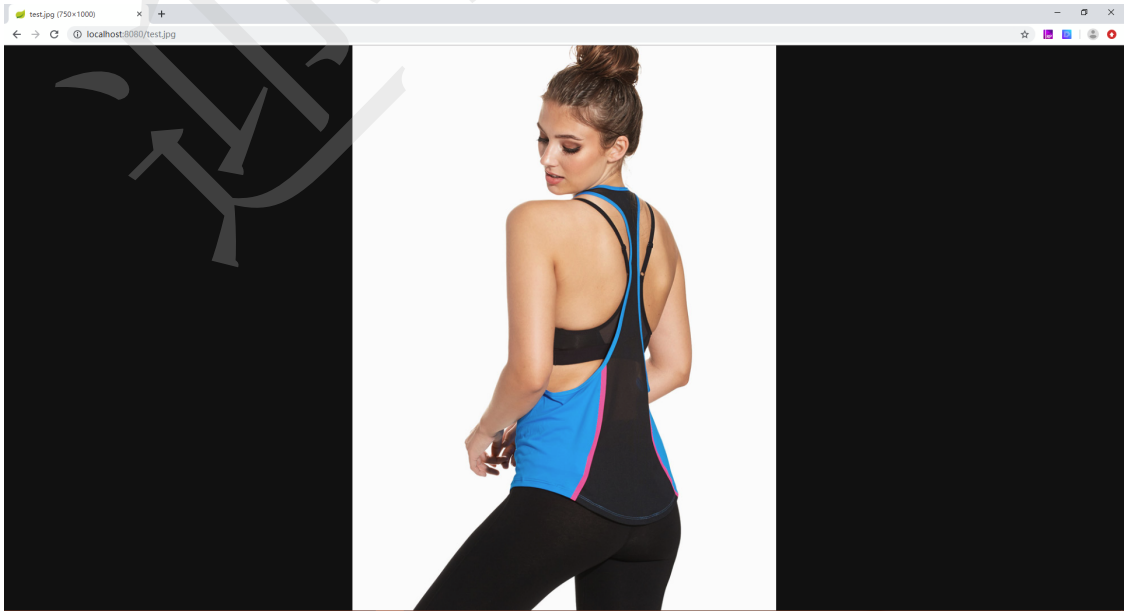
  只要静态资源放在这些目录中任何一个，SpringMVC都会帮我们处理。 我们习惯会把静态资源放在classpath:/static/ 目录下。

3.3 测试

1. 把 test.png 图片放入 static 文件夹



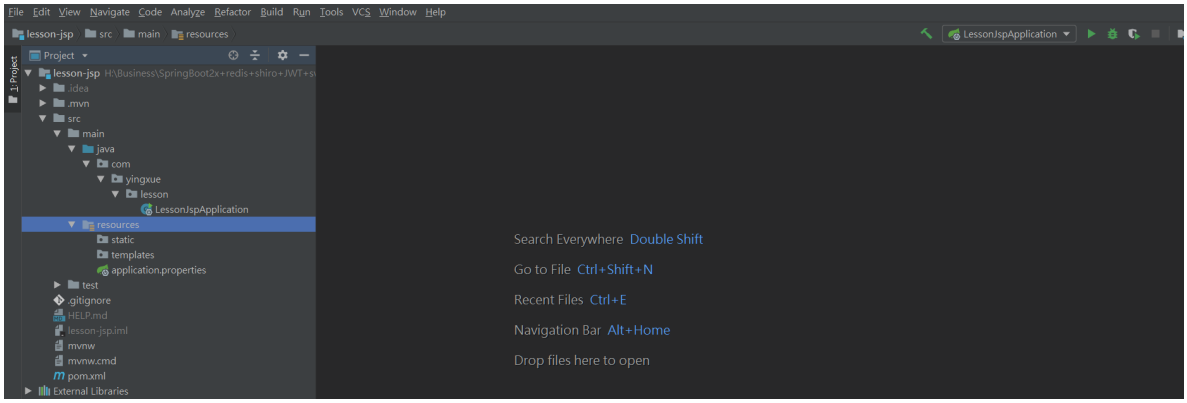
2. 重启项目



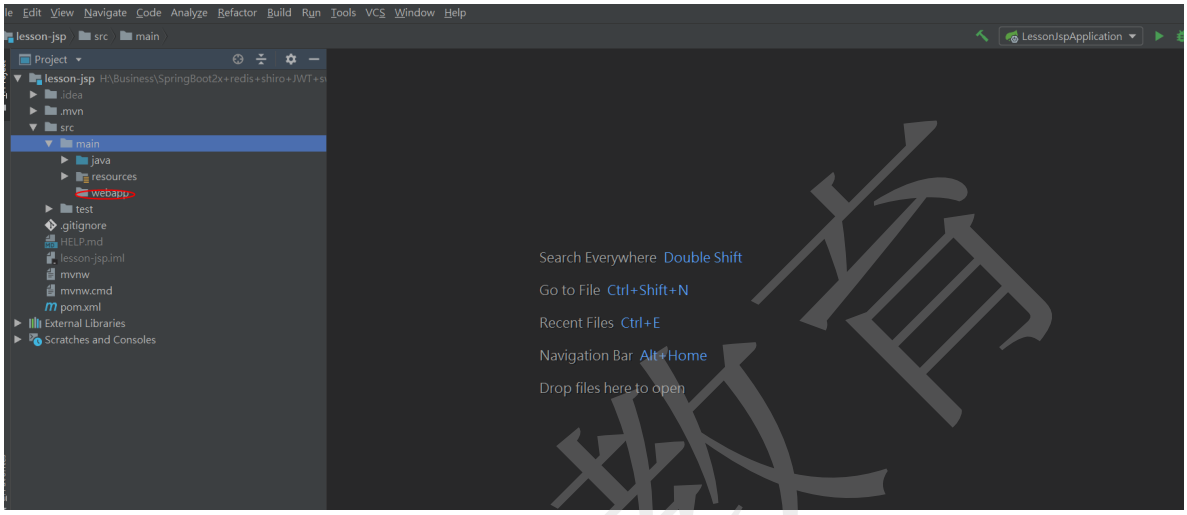
4. Spring Boot 整合 Jsp

提起 Java 不得不说的一个开发场景就是 Web 开发，说到 Web 开发绕不开的一个技术就是 JSP，因为目前市面上仍有很多的公司在使用 SSM+JSP,然后又想升级 Spring Boot。这节课程主要讲如何在 SpringBoot 项目使用 JSP

4.1 创建项目



4.2 创建 webapp

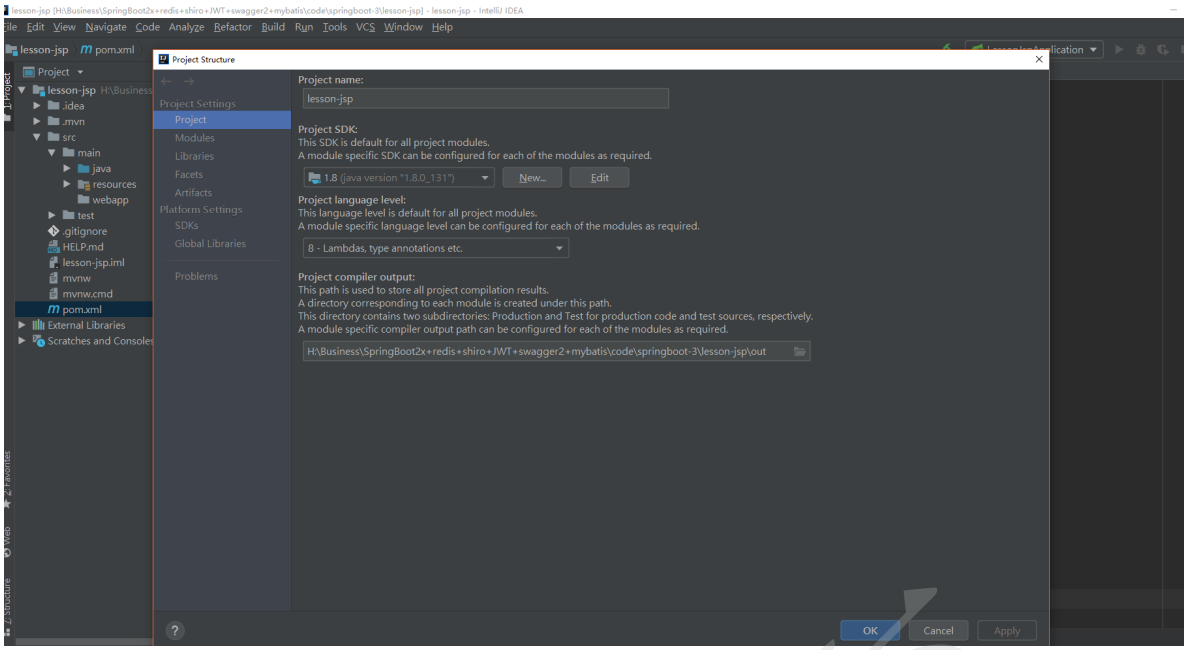


4.1.3 引入 JSP 相关依赖

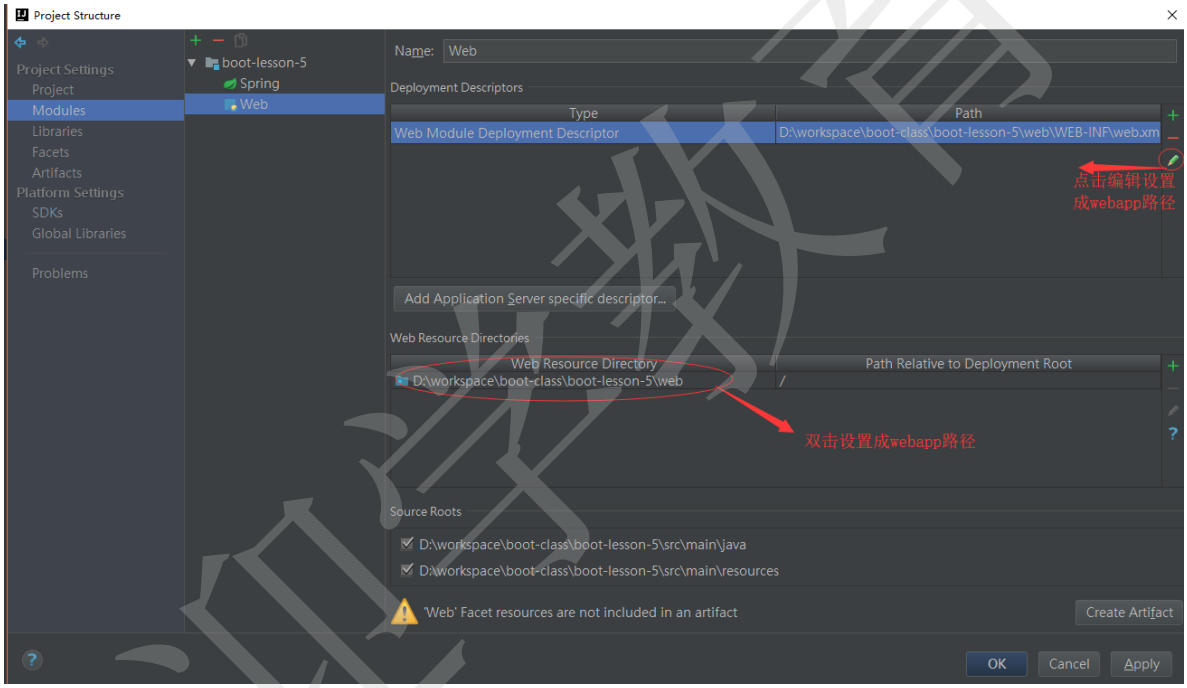
```
<!--JSP标准标签库-->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
</dependency>

<!--内置tocat对Jsp支持的依赖，用于编译Jsp-->
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-jasper</artifactId>
</dependency>
```

4.2 web 配置

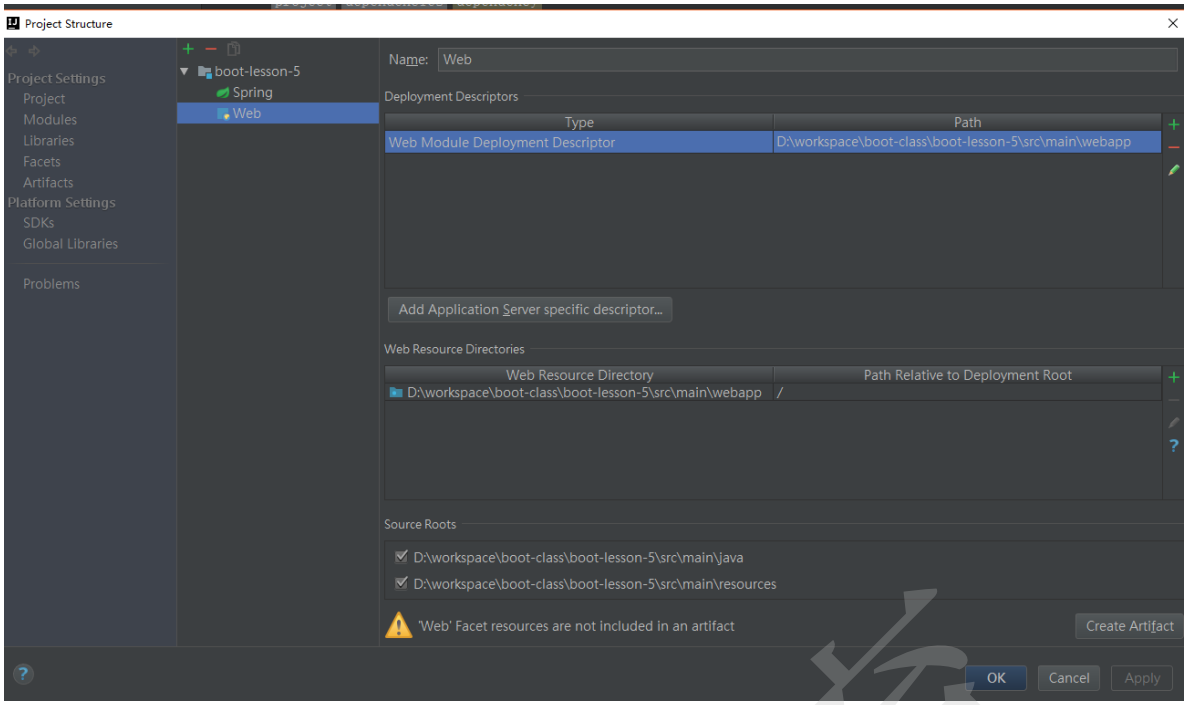
4.2.1 编辑 project Structure



4.2.2 配置根路径



4.2.3 apply OK



4.2.4 Spring Mvc 视图解析器配置

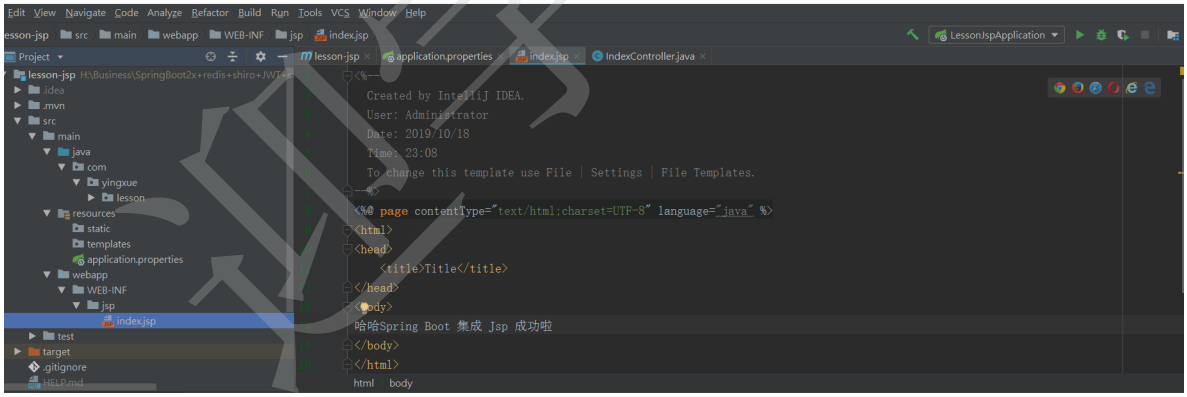
修改 application .properties加入如下代码

```
spring.mvc.view.prefix=/WEB-INF/jsp/
spring.mvc.view.suffix=.jsp
server.port=8080
```

4.3 测试

4.3.1 创建 index.jsp

webapp->WEB-INF->jsp->index.jsp 在webapp创建一个WEB-INF文件再WEB-INF下创建一个jsp文件夹再jsp下创建一个index.jsp 文件



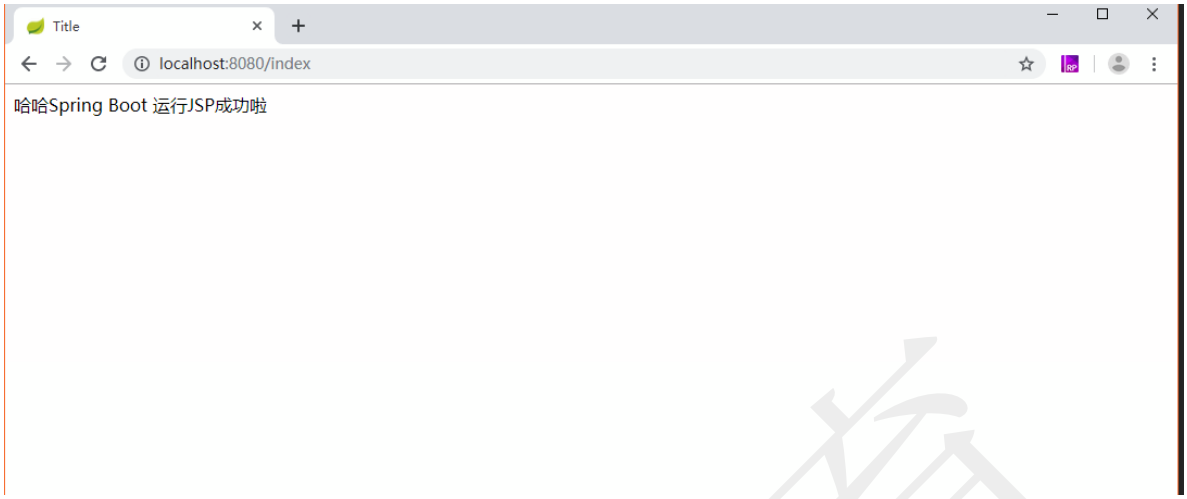
4.3.2 创建 IndexController

```
package com.yingxue.lesson.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

/**
 * @ClassName: IndexController
 * TODO:类文件简单描述
 * @Author: 小霍
 * @UpdateUser: 小霍
 * @Version: 0.0.1
 */
@Controller
public class IndexController {
    @GetMapping("/index")
    public String index(){
```

```
        return "index";
    }
}
```

4.3.3 启动项目

浏览器输入 localhost:8080/index



测试成功。

所以我们在以后遇到，老旧的项目升级成Spring Boot 项目时候，首先得配置好 webapp 这个跟路径、配置好 web、再配置 ORM 所需的一些配置，最后记得配置视图解析器。所需的配置配置好后就可以直接把代码拷入新的项目了。

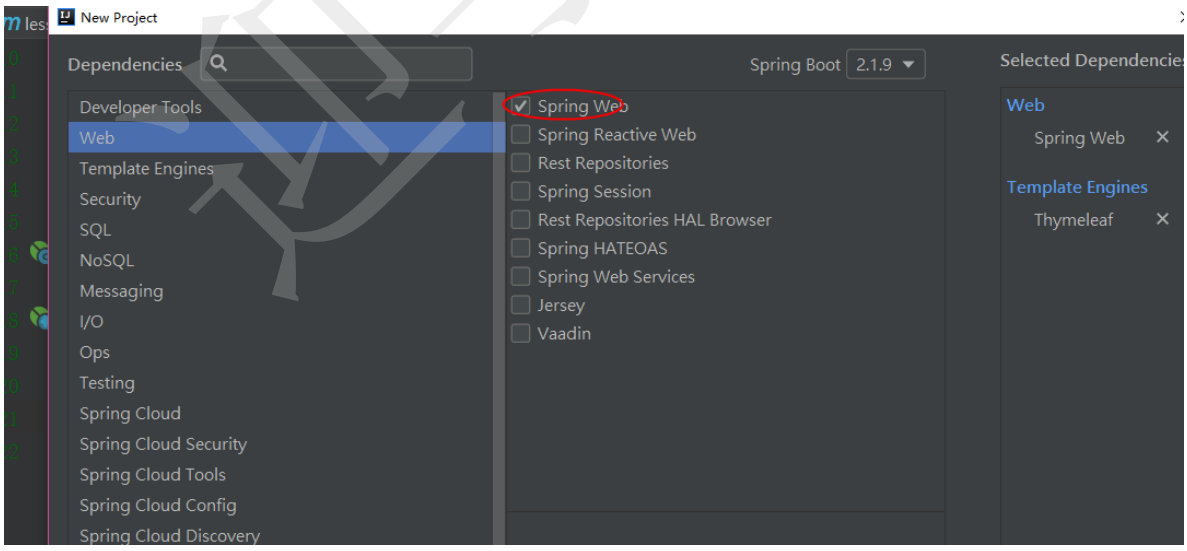
5. Spring Boot 整合 Thymeleaf

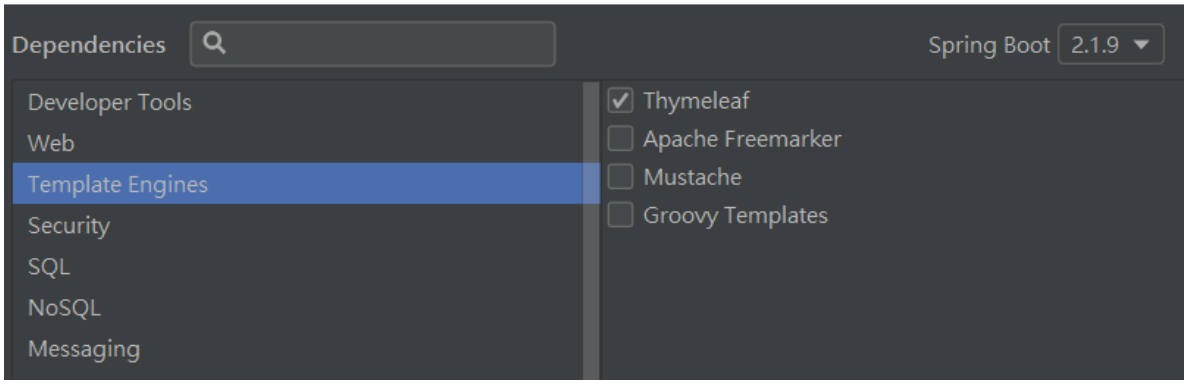
Thymeleaf是一款用于渲染XML/XHTML/HTML5内容的模板引擎。类似JSP，FreeMaker等， 它也可以轻易的与 Web 框架进行集成作为 Web 应用的模板引擎。与其它模板引擎相比， Thymeleaf 最大的特点是能够直接在浏览器中打开并正确显示模板页面，而不需要启动整个Web应用。

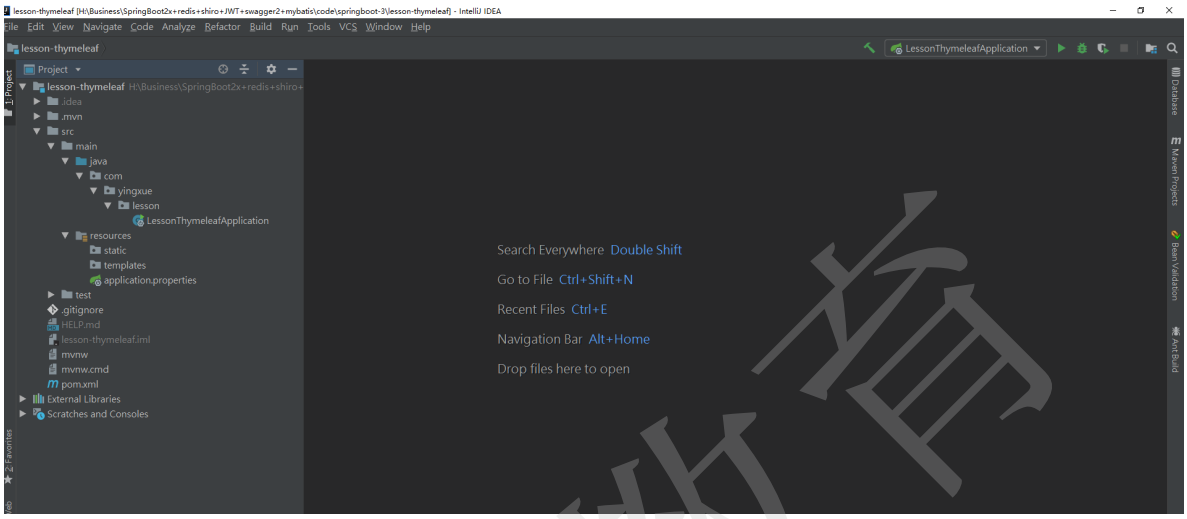
thymeLeaf支持Spring Expression Language语言作为方言，也就是SpEL，SpEL是可以用于Spring中的一种EL表达式。它与我们使用过的JSP不同，thymeleaf是使用html的标签来完成逻辑和数据的传入进行渲染。

可以说用 thymeleaf 完全替代 jsp 是可行的。

5.1 创建项目







5.2 Thymeleaf 配置

5.2.1 Spring Mvc 视图解析器配置

修改 application .properties加入如下代码

```
#thymeleaf
# 前缀 默认读取classpath:/templates/
#无需配置
#spring.thymeleaf.prefix=classpath:/templates/
# 后缀
spring.thymeleaf.suffix=.html
spring.thymeleaf.charset=UTF-8
spring.thymeleaf.servlet.content-type=text/html
```

5.3 测试

5.3.1 创建 hello.html

在templates下创建一个hello.html 必须加入xmlns:th="http://www.thymeleaf.org" Thymeleaf声明空间

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<p th:text="'hello:'+${username}"></p>
</body>
</html>
```

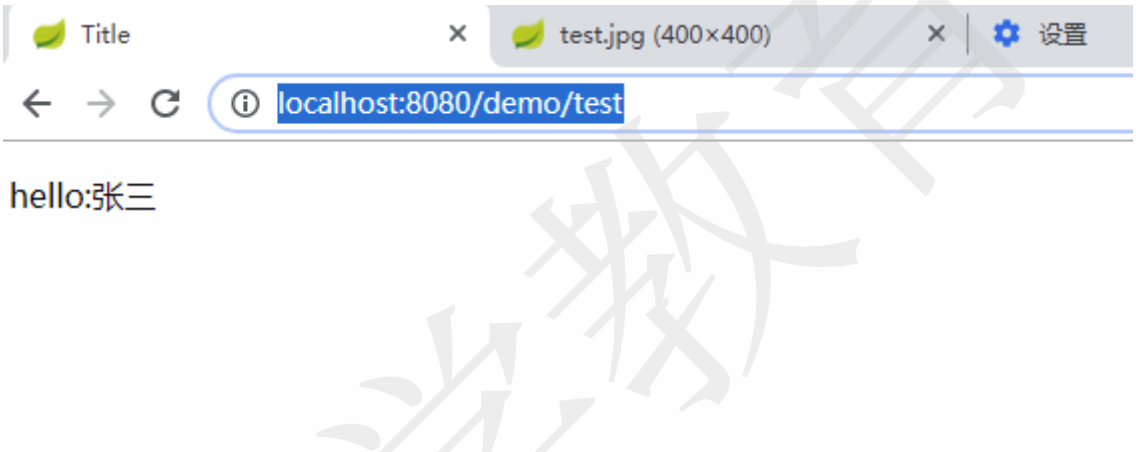
5.3.2 创建 HelloController

```
package com.yingxue.lesson.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
```

```java
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;

/**
 * @ClassName: HelloController
 * TODO:类文件简单描述
 * @Author: 小霍
 * @UpdateUser: 小霍
 * @Version: 0.0.1
 */
@Controller
public class HelloController {

    @GetMapping("/hello")
    public String hello(Model model){
        model.addAttribute("username","zhangsan");
        return "hello";
    }
}
```

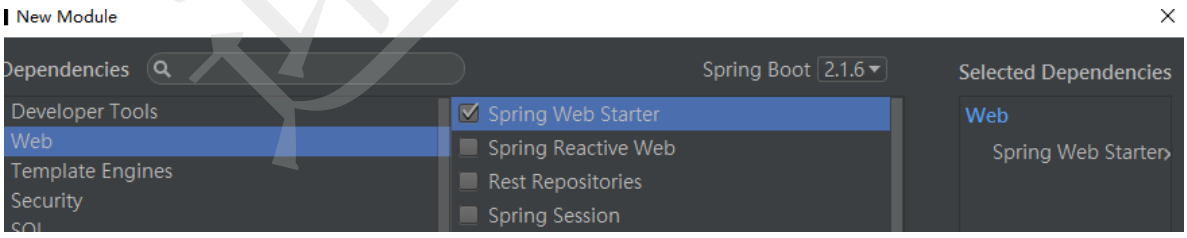
5.3.3 启动项目

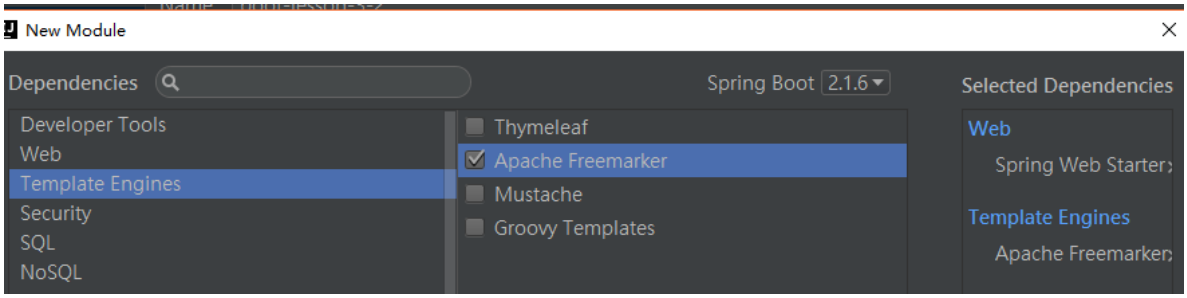
浏览器输入 http://localhost:8080/demo/test

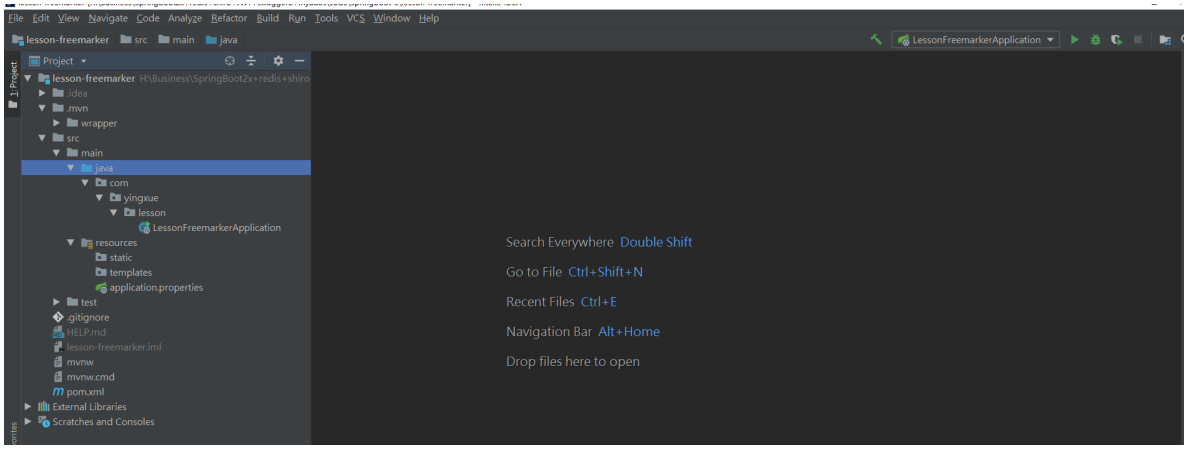


6. Spring Boot 整合 Freemarker

6.1 创建项目







6.2 Freemarker 配置

6.2.1 Spring Mvc 视图解析器配置

修改 application .properties加入如下代码

```
#freemarker
spring.freemarker.template-loader-path=classpath:/templates/
# 后缀
spring.freemarker.suffix=.html
spring.freemarker.charset=UTF-8
spring.freemarker.content-type=text/html
```

6.3 测试

6.3.1 创建 hello.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<h1>${username}</h1>欢迎使用Freemarker模板
</body>
</html>
```

6.3.2 创建 HelloController

```
package com.yingxue.lesson.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;

/**
 * @ClassName: HelloController
 * TODO:类文件简单描述
 * @Author: 小霍
 * @UpdateUser: 小霍
 * @Version: 0.0.1
 */
@Controller
public class HelloController {
    @GetMapping("/hello")
    public String hello(Model model){
        model.addAttribute("username","张三");
        return "hello";
    }
}
```

6.3.3 启动项目

浏览器输入： http://localhost:8080/hello