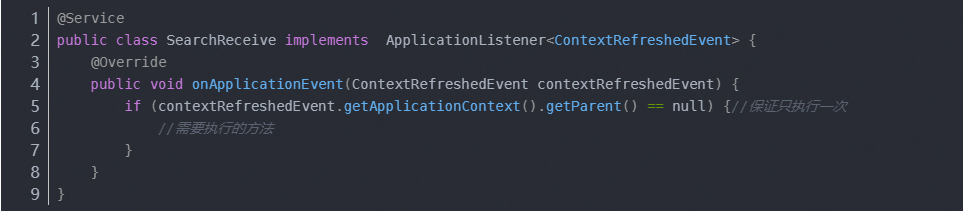
SpringBoot项目中实现项目启动时就在执行需要执行的代码？

1、实现ApplicationListener接口，并实现 onApplicationEvent(ContextRefreshedEvent contextRefreshedEvent)方法

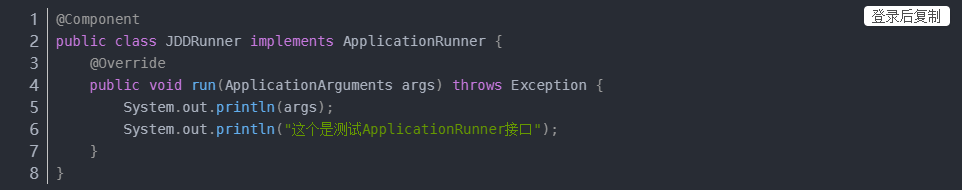


至于为什么先做判断，因为Spring存在两个容器，一个是root application context ,另一个就是我们自己的 projectName-servlet context（作为root application context的子容器）。这种情况下，就会造成onApplicationEvent方法被执行两次。为了避免上面提到的问题，我们可以只在root application context初始化完成后调用逻辑代码，其他的容器的初始化完成，则不做任何处理

2、CommandLineRunner和ApplicationRunner

SpringBoot给我们提供了两个接口来帮助我们实现这种需求。这两个接口分别为CommandLineRunner和ApplicationRunner。他们的执行时机为容器启动完成的时候。

这两个接口中有一个run方法，我们只需要实现这个方法即可。这两个接口的不同之处在于：ApplicationRunner中run方法的参数为ApplicationArguments，而CommandLineRunner接口中run方法的参数为String数组。



# 1、spring boot 基本使用

## 1.1 框架内使用

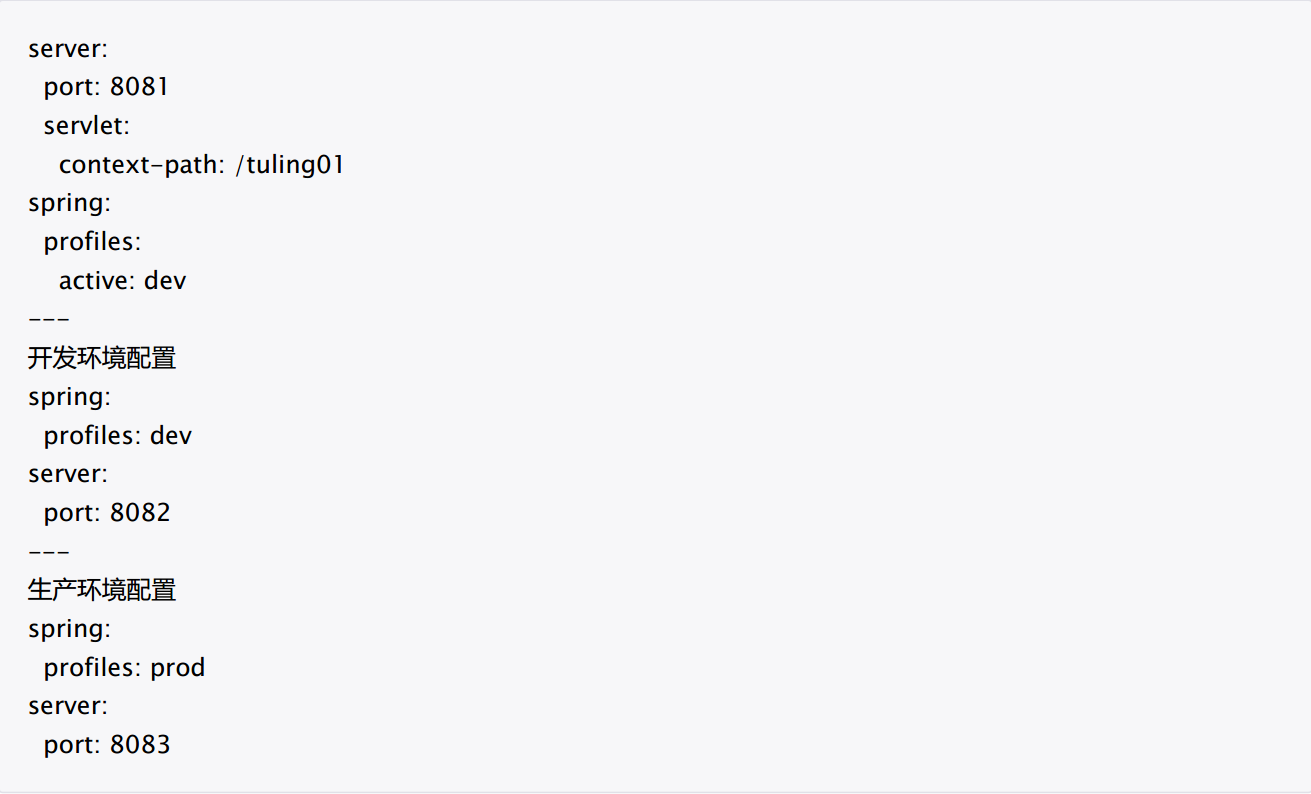
### 1.1 多profile切换

我们在开发应用时， 通常一个项目 会被部署到不同的环境中， 比如： 开发、 测试、 生产等。 其中每个环境的数据库地址、 服务器端口等等配置都会不同， 对于多环境的配置， 大部分构建工具或是框架解决的基本思路是一致的， 通过配置多份不同环境的配置文件， 再通过打包命令指定需要打包的内容之后进行区分打包

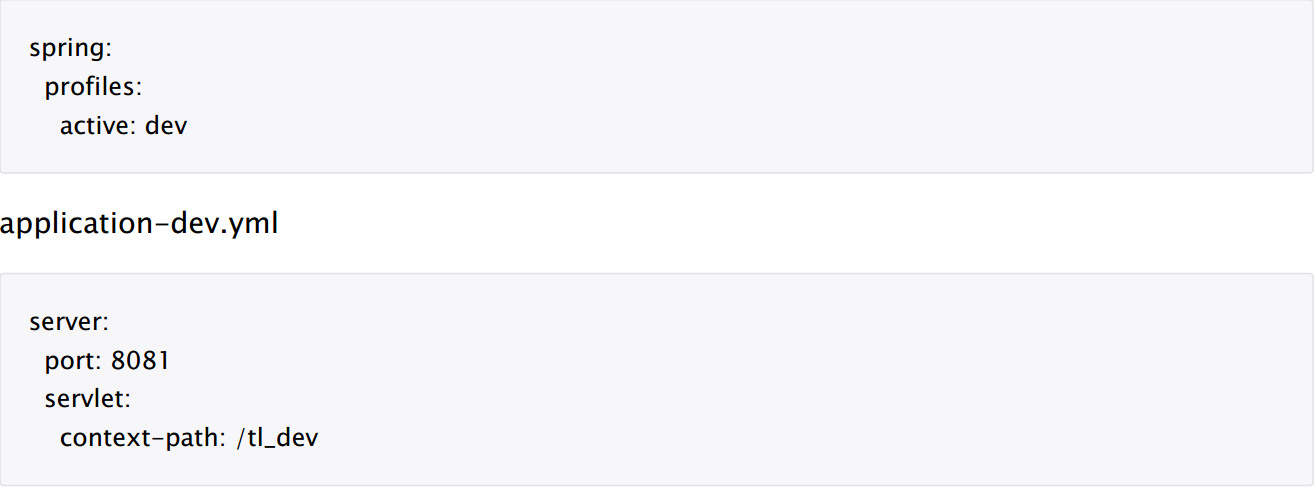
多profile切换有三种方式:

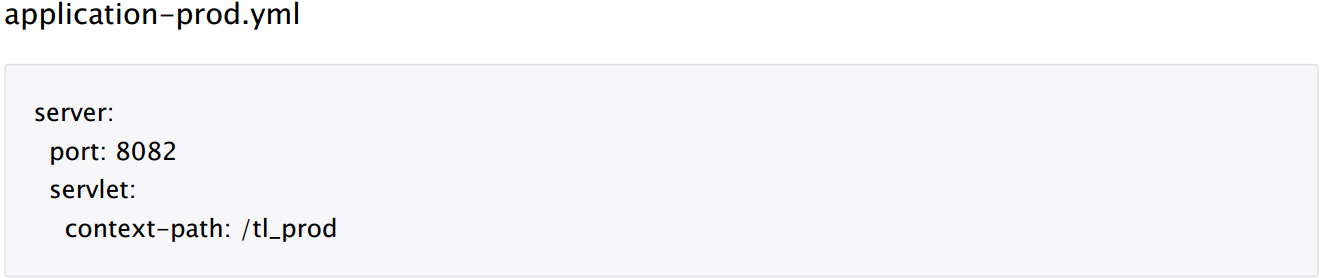
1、yml多模块文档快

在一个配置文件中配置多个



2、多配置文件切换





3、启动参数指定配置文件

命令行参数启动(打成Jar包时候) java -jar tuling-vip-springboot-02-0.0.1 -SNAPSHOT.jar --  
spring.profiles.active=prod

4、查看设置的jvm参数

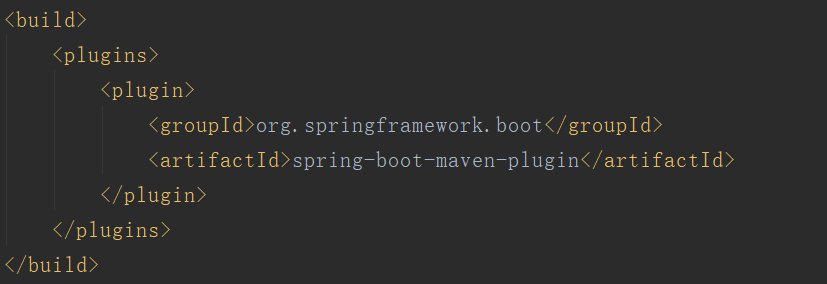
首先执行jps 查看jar启动进程号，然后jinfo +进程号查看配置信息

### 1.2 spring boot 打包

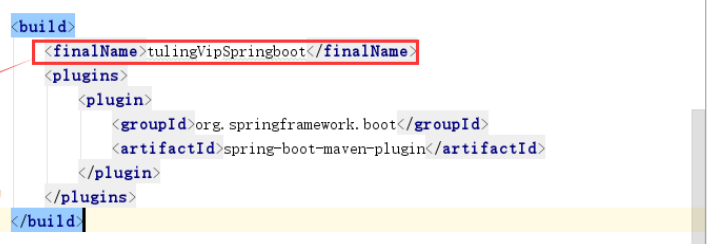
1、jar包

使用maven插件将项目打成jar包，会有两个jar包，第一个为可执行的jar包，但是不能被引用，解压后中只有class文件，第二个jar包可以被引用但不能被执行，将其解压后为正常的Java文件

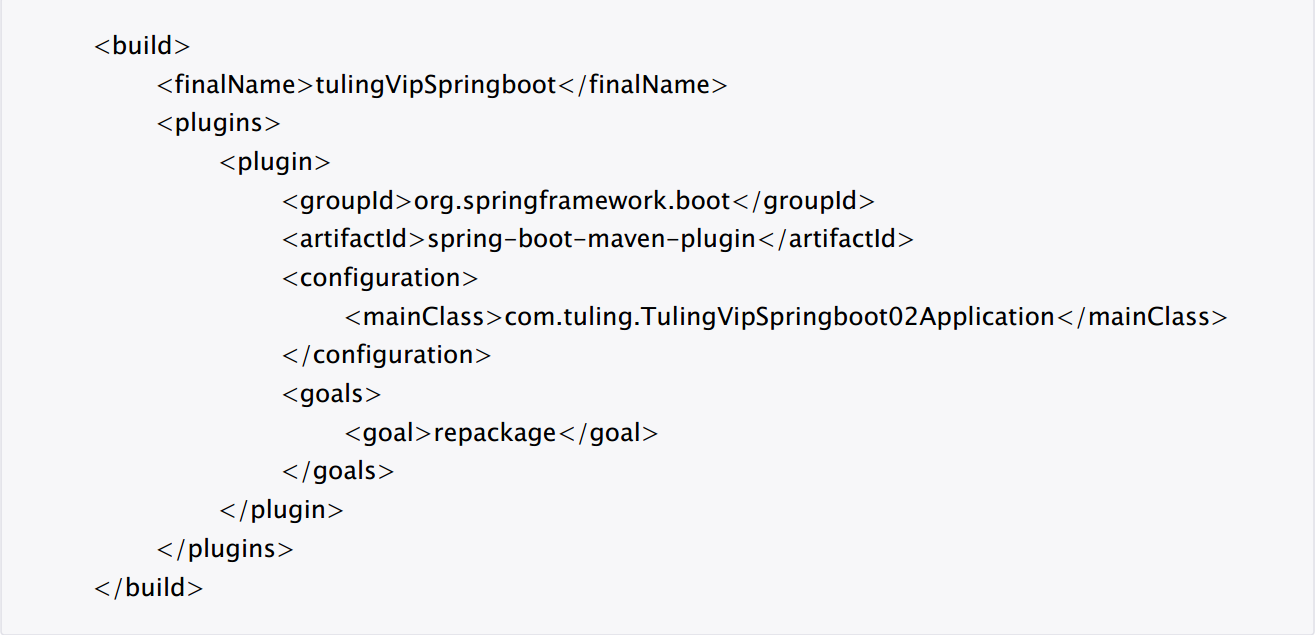




打包并指定jar包名称

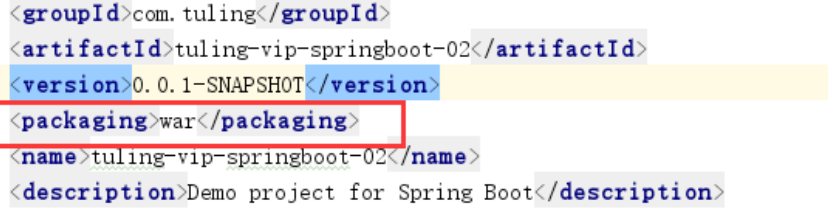


若工程中有多个main方法，需要制定主启动类



2、打war包

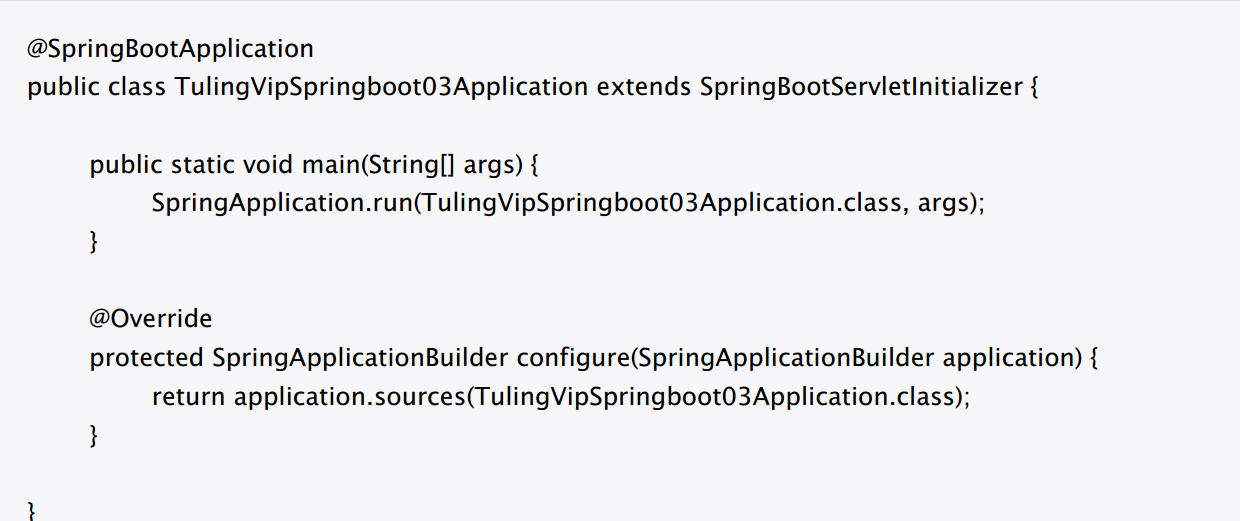
第一步:指定springboot pom中的打包方式 由jar改为war



第二步:在spring-boot-starter-web模块打包必依赖与 tomcat



第三步:主启动类上 实现SpringBootServletInitializer 从写confiure方法(原理第三节课节讲)



第四步:打成war包 放在tomcat上运行

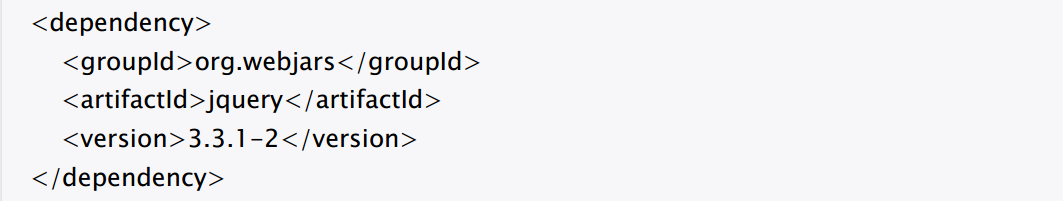
## 1.2 spring boot web开发

### 2.1 webjar

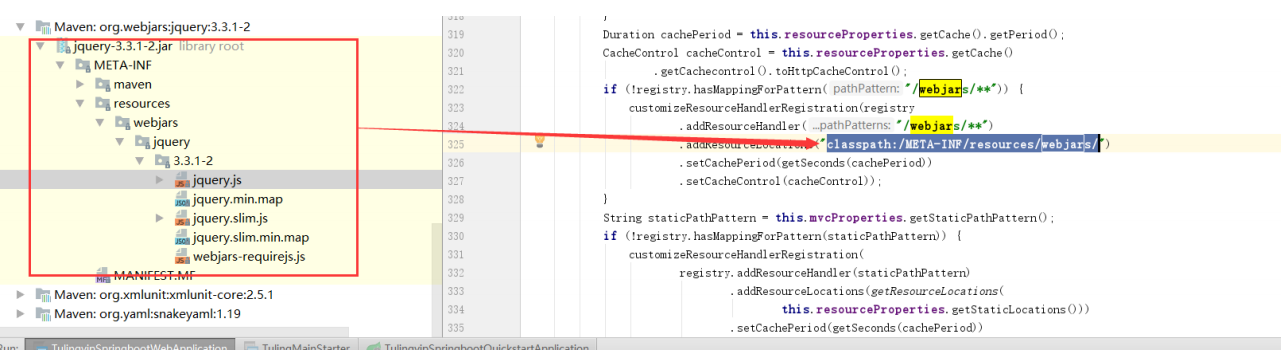
https://www.webjars.org/

什么是webJar： 以jar包的形式来引入前端资源,比如jquery 或者是BootStrap

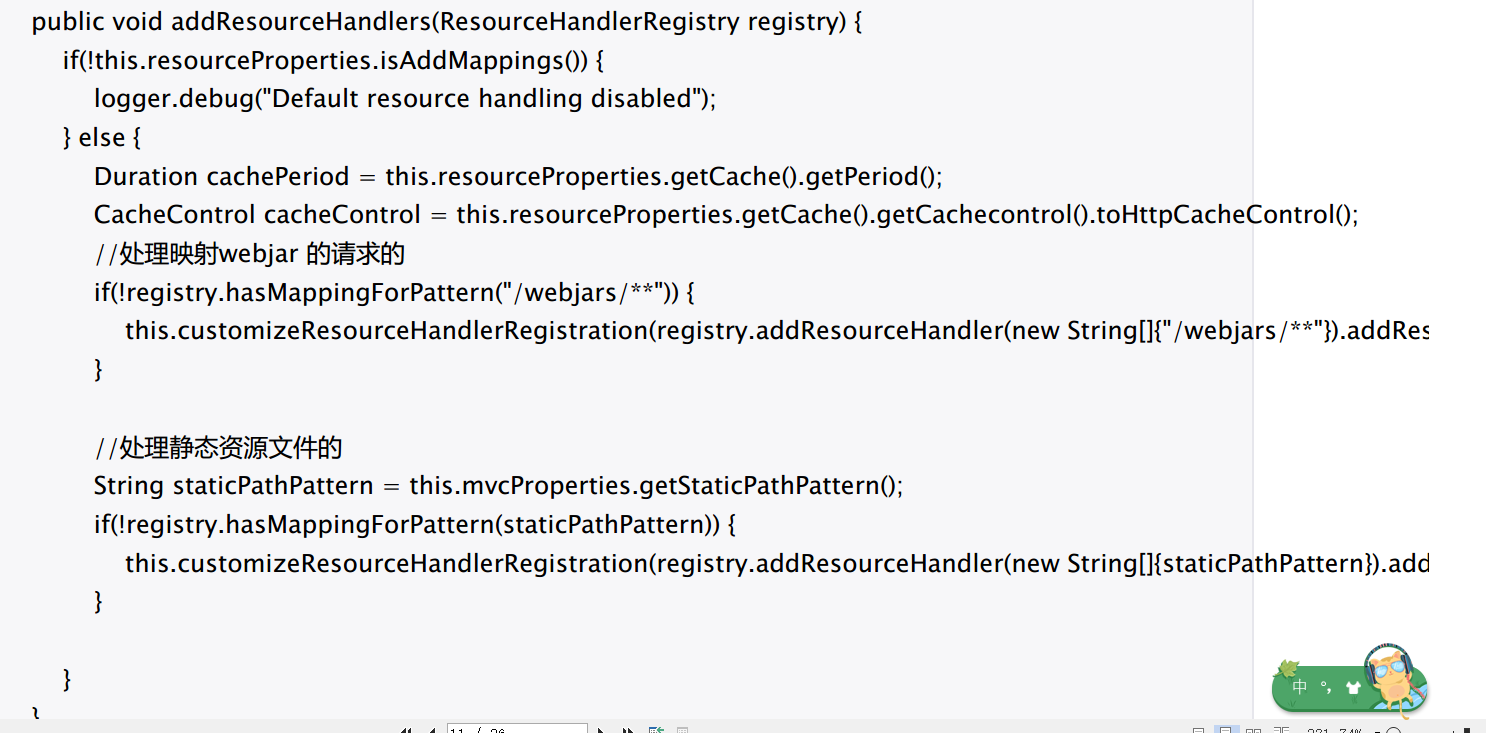
引入jar包



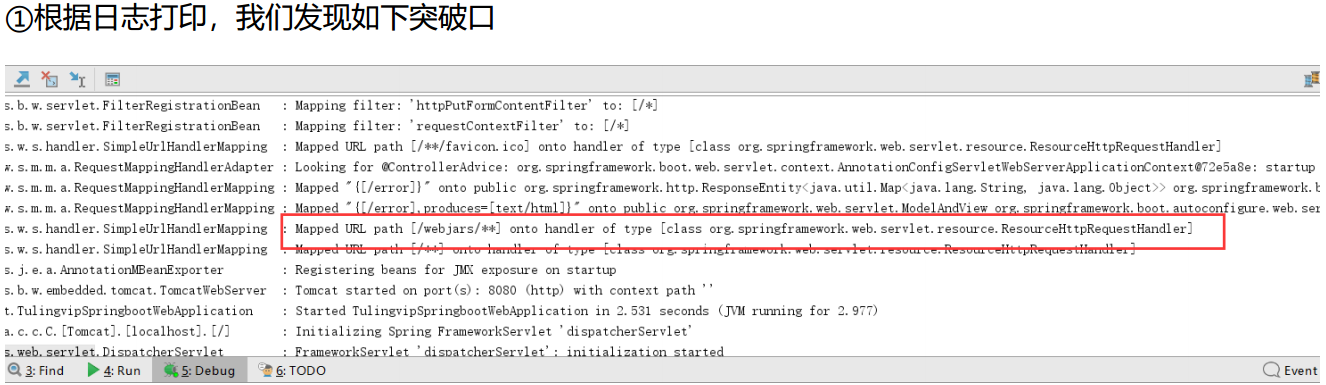
映射规则 /webjars/\*\* 都会被映射到classpath:/META-INF/resources/webjars/ 目录下去处理

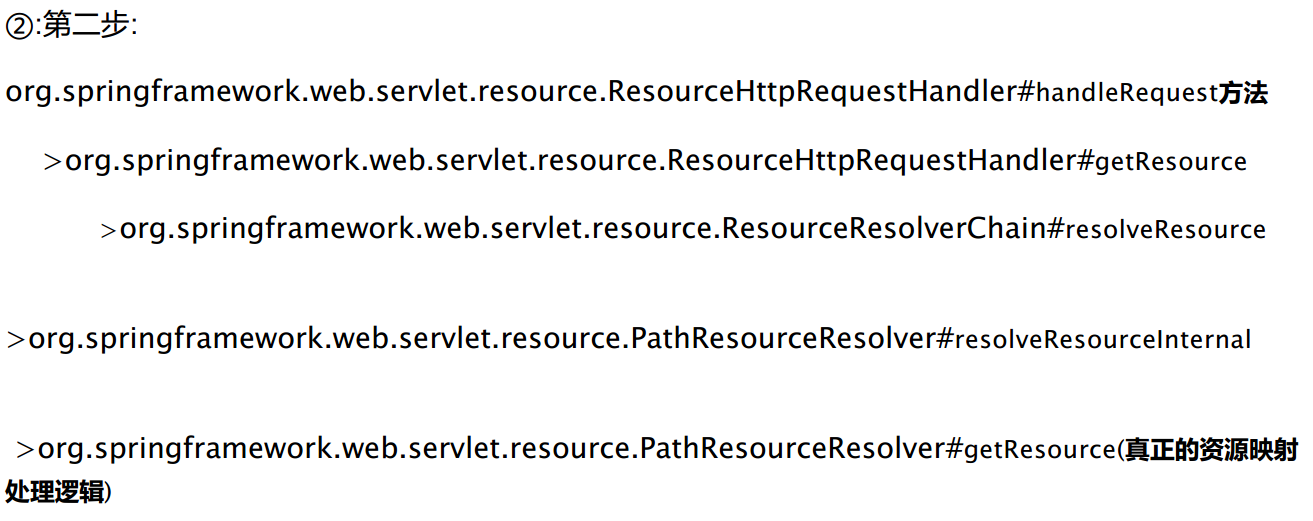


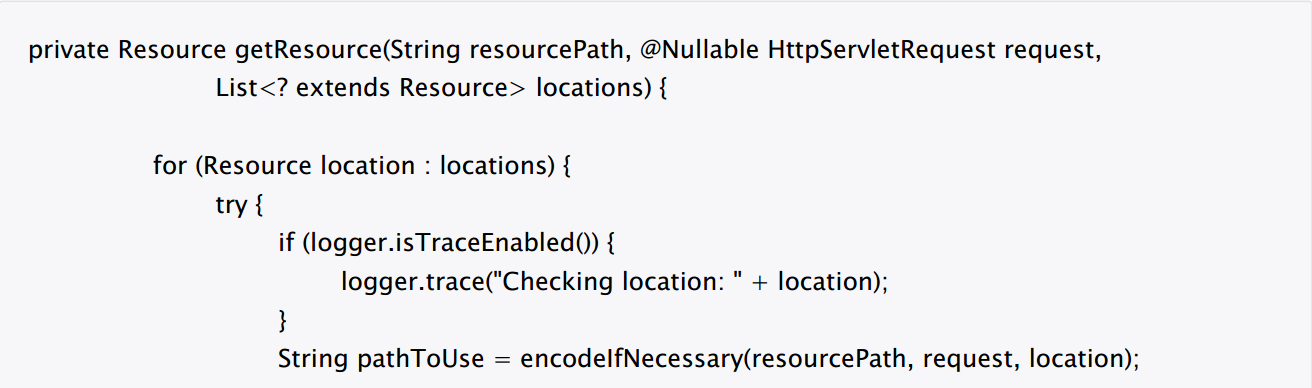
前端资源映射规则 核心源代码:

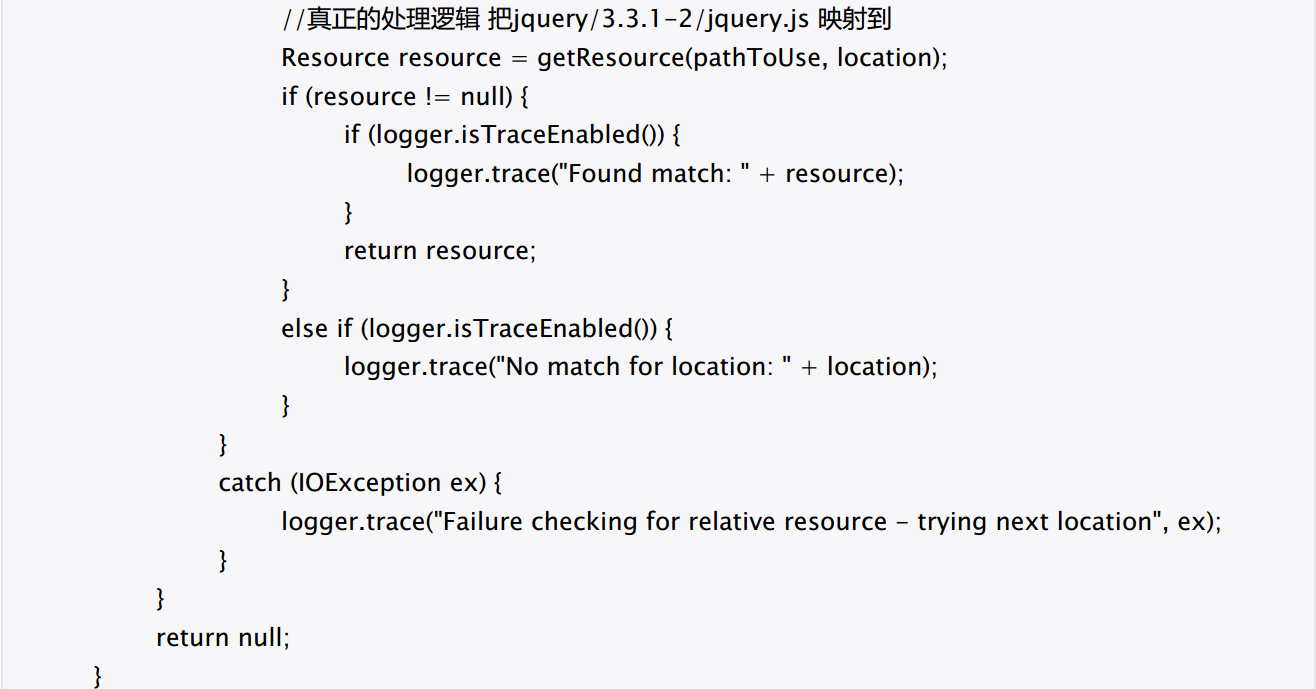


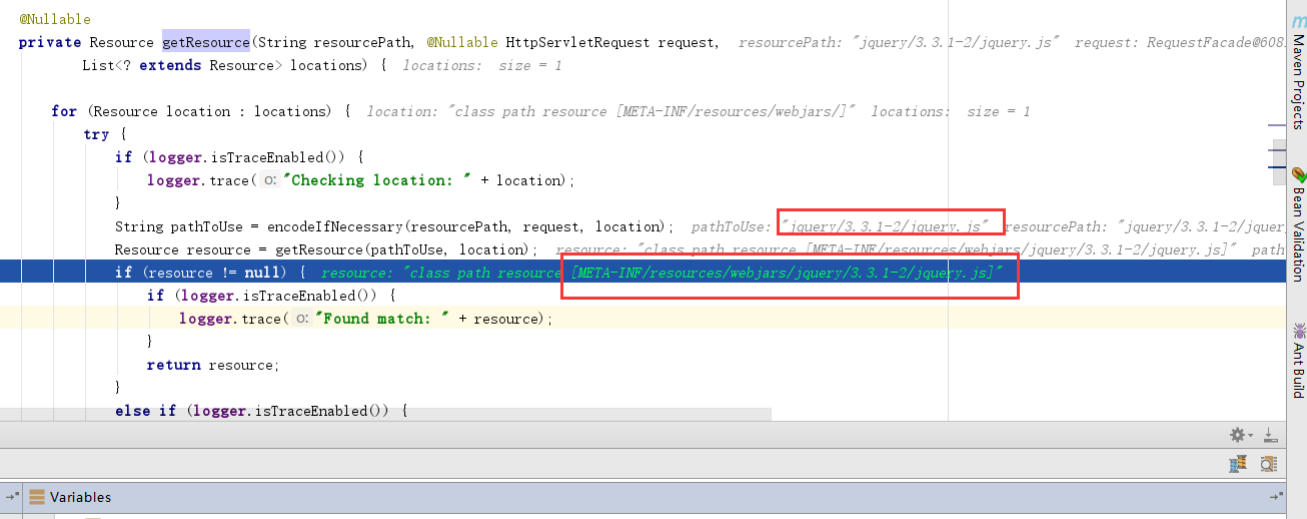
http://localhost:8080/webjars/jquery/3.3.1 -2/jquery.js 请求如何拦截处理请求的



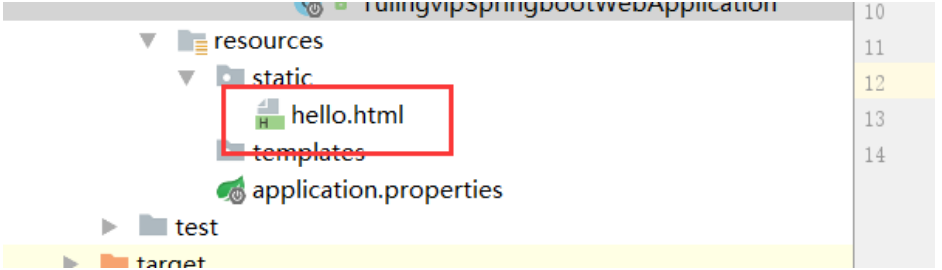




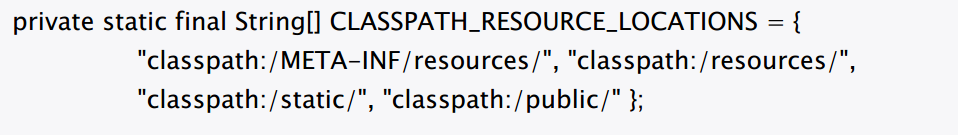


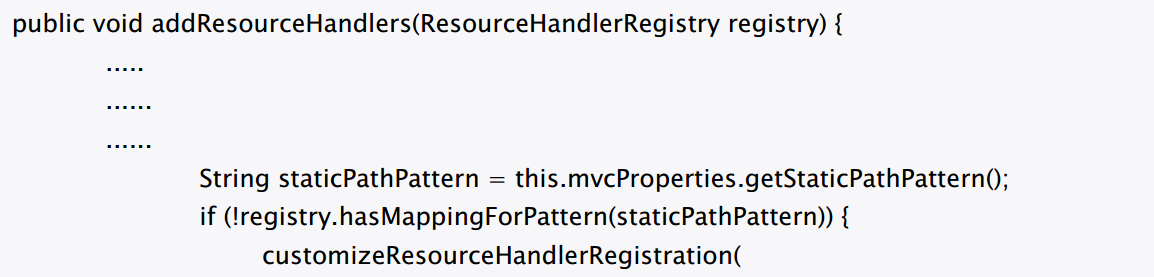


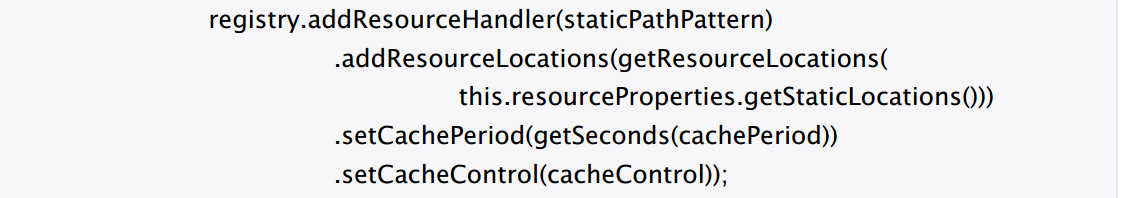
访问静态html页面 我们直接把静态页面放在static的目录下， 直接可以在路径直接访问



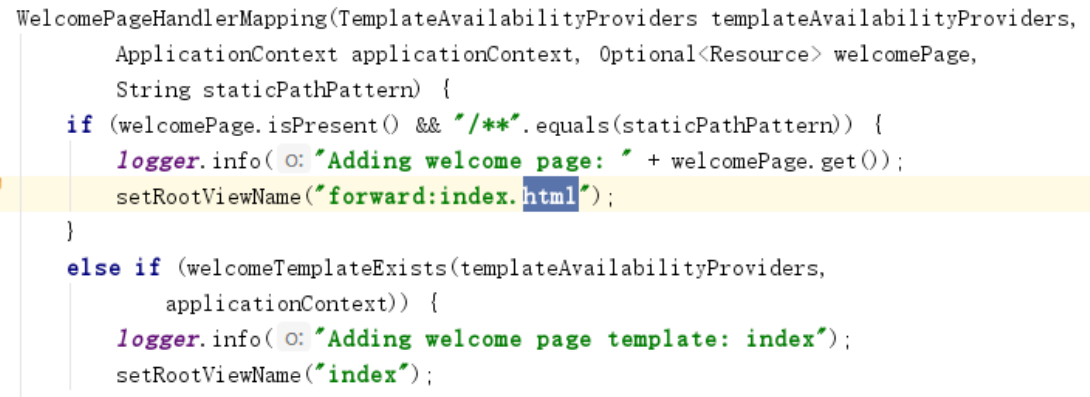
映射原理 /\*\*请求都会被映射到



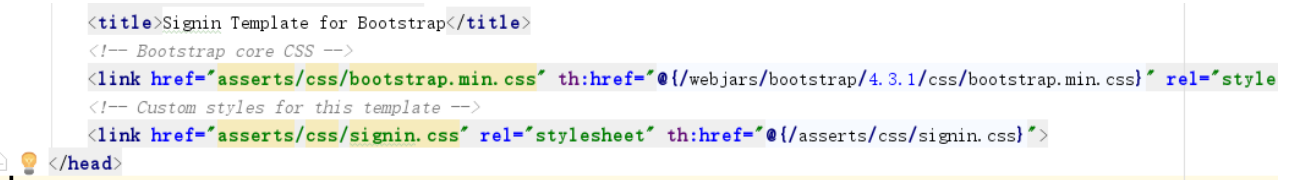




欢迎页； 静态资源文件夹下的所有index.html页面； 被"/\*\*"映射；



使用webjar的方式修前端页面修改引用路径





### 2.2 springboot是如何整合springmvc功能的（WebMvcAutoConfiguration）

自动装配的组件

①:ContentNegotiatingViewResolver 和 BeanNameViewResolver 视图解析器  
视图解析器的作用:根据方法的值找到对应的视图  
②:Support for serving static resources, including support for WebJars 支持静态资源和webJars  
③:Converter ,日期格式化器 Formatter  
④:消息装换器: HttpMessageConverters  
⑤:首页设置index.html  
⑥:图标支持 Favicon

#### 1、定义拦截器

我们需要自己写一个配置类 继承 WebMvcConfigurerAdapter 需要什么组件 就注册什么组件

第一步:创建一个拦截器

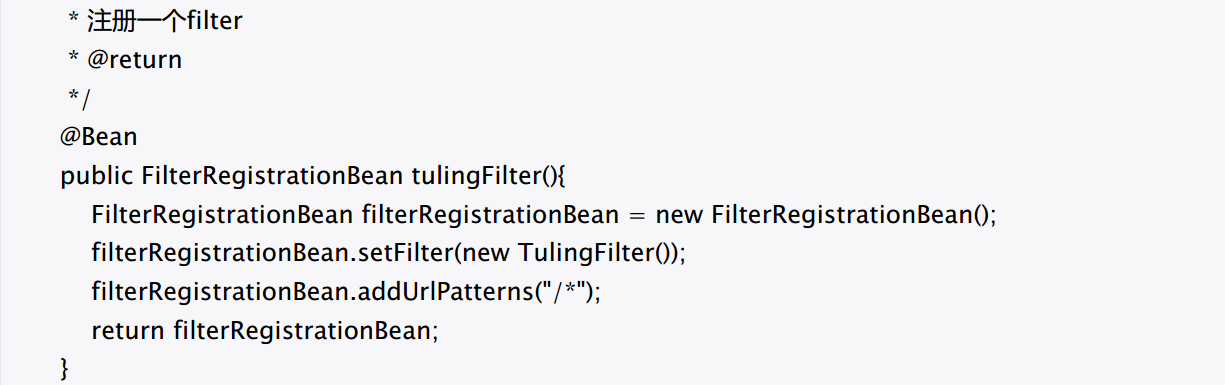


第二步:注册拦截器



#### 2、定义过滤器





#### 3、定义servlet





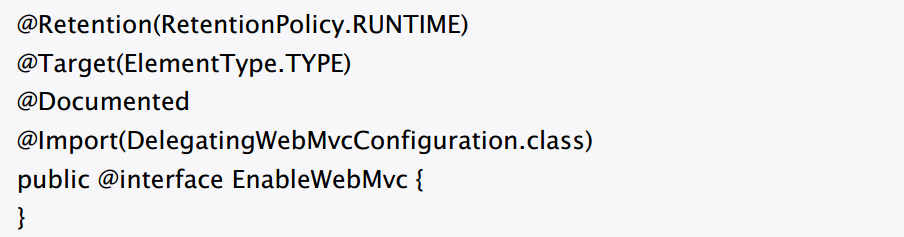
### 2.3 接管springboot mvc配置，使自动装配功能失效

官网：

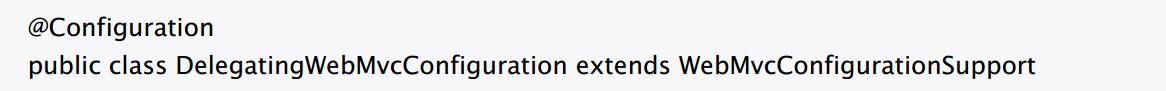
If you want to keep Spring Boot MVC features and you want to add additional MVC  
configuration (interceptors, formatters, view controllers, and other features), you can add your  
own @Configuration class of type WebMvcConfigurer but without @EnableWebMvc . If you wish to provide  
custom instances of RequestMappingHandlerMapping , RequestMappingHandlerAdapter ,  
or ExceptionHandlerExceptionResolver , you can declare a WebMvcRegistrationsAdapter instance to provide such  
components

即在配置文件中使用一个@EnableWebMvc来标识到配置类上,就会导致配置失效

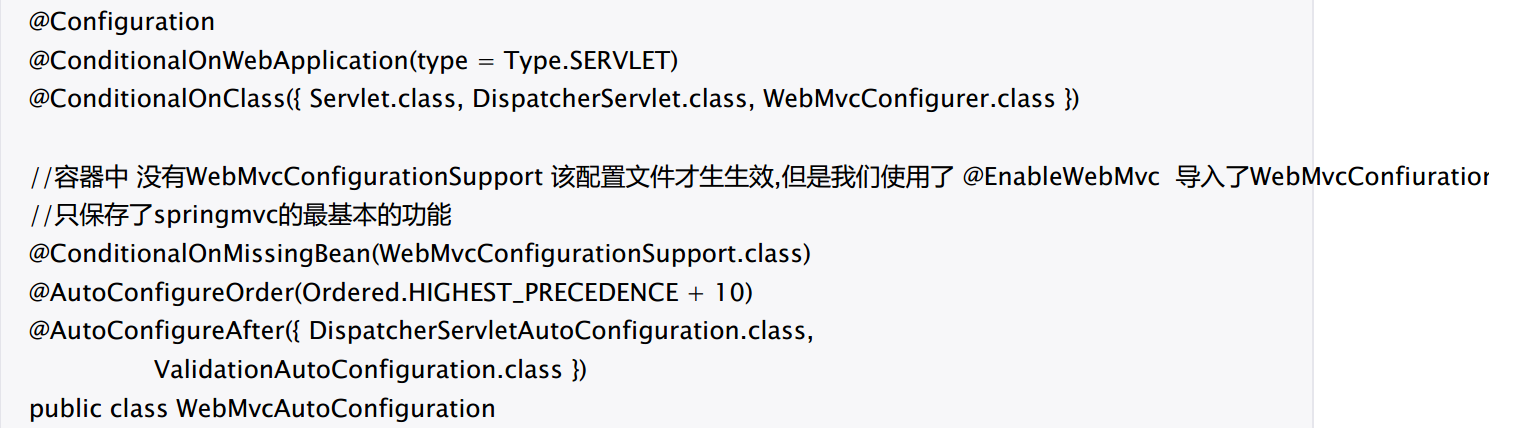
原理：原理: @EnableWebMvc 为容器中导入了DelegatingWebMvcConfiguration的组件



我们来分析一下DelegatingWebMvcConfiguration是一个什么东西？ ？ ？ ？ ？  
我们发现DelegatingWebMvcConfiguration是WebMvcConfiurationSupport（只保证了springmvc的基本功能） 类型的



我们来看下WebMvcAutoConfiguration上的注解



## 1.3 spring boot 整合mybatis

### 3.1 druid的配置

spring:

datasource:

druid:

username: root

password: 123456

jdbcUrl: jdbc:mysql://47.104.128.12:3306/tuling-vip

driverClassName: com.mysql.jdbc.Driver

initialSize: 5

minIdle: 5

maxActive: 20

maxWait: 60000

timeBetweenEvictionRunsMillis: 60000

minEvictableIdleTimeMillis: 300000

validationQuery: SELECT 1 FROM DUAL

testWhileIdle: true

testOnBorrow: false

testOnReturn: false

poolPreparedStatements: true

filters: stat,wall #用于监控

maxPoolPreparedStatementPerConnectionSize: 20

useGlobalDataSourceStat: true

connectionProperties: druid.stat.mergeSql=true;druid.stat.slowSqlMillis=500

仅在yml这种配置druid即可，也可以在Java代码中获取yml中的配置信息，然后手动创建相应的bean，如下将所需要的属性通过封装的形式自动获取yml中配置的值

@ConfigurationProperties(prefix = "spring.datasource.druid")

public class DruidDataSourceProperties {

private String username;

private String password;

public String getFilters() {

return filters;

}

public void setFilters(String filters) {

this.filters = filters;

}

public String filters;

public String getUsername() {

return username;

}

然后在druidconfig中配置如下实例

@Configuration

@EnableConfigurationProperties(value = DruidDataSourceProperties.class)

public class DruidDataSourceConfig {

@Autowired

private DruidDataSourceProperties druidDataSourceProperties;

@Bean

public DataSource dataSource() throws SQLException {

System.out.println(druidDataSourceProperties);

DruidDataSource druidDataSource = new DruidDataSource();

druidDataSource.setUsername(druidDataSourceProperties.getUsername());

druidDataSource.setPassword(druidDataSourceProperties.getPassword());

druidDataSource.setUrl(druidDataSourceProperties.getJdbcUrl());

druidDataSource.setDriverClassName(druidDataSourceProperties.getDriverClassName());

druidDataSource.setInitialSize(druidDataSourceProperties.getInitialSize());

druidDataSource.setMinIdle(druidDataSourceProperties.getMinIdle());

druidDataSource.setMaxActive(druidDataSourceProperties.getMaxActive());

druidDataSource.setMaxWait(druidDataSourceProperties.getMaxWait());

druidDataSource.setFilters(druidDataSourceProperties.getFilters());

druidDataSourceProperties.setPoolPreparedStatements(druidDataSourceProperties.isPoolPreparedStatements());

return druidDataSource;

}

/\*\*

\* 配置druid管理后台的servlet

\* @return

\*/

@Bean

public ServletRegistrationBean statViewSerlvet() {

ServletRegistrationBean bean = new ServletRegistrationBean(new StatViewServlet(),"/druid/\*");

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

initParameters.put("loginUsername","admin");

initParameters.put("loginPassword","123456");

bean.setInitParameters(initParameters);

return bean;

}

@Bean

public FilterRegistrationBean filterRegistrationBean() {

FilterRegistrationBean filterRegistrationBean = new FilterRegistrationBean(new WebStatFilter());

filterRegistrationBean.setUrlPatterns(Arrays.asList("/\*"));

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

initParams.put("exclusions","\*.js,\*.css,/druid/\*");

filterRegistrationBean.setInitParameters(initParams);

return filterRegistrationBean;

}

}

### 3.2 整合mybatis

首先配置mybatis对接口Java文件的加载，有两种方式

1、在Java接口文件上使用@Mapper注解

2、在启动类或者配置类上添加

@MapperScan({"com.tuling.tulingmall.mapper","com.tuling.tulingmall.portal.dao","com.tuling.tulingmall.dao"})，用来标明接口类所在位置，可配置多个

然后配置对映射xml文件的配置，可以配置多个，包括引入jar包的xml文件

mybatis:

mapper-locations:

- classpath:dao/\*.xml

- classpath\*:com/\*\*/mapper/\*.xml

## 1.4 spring boot 整合swagger

引入依赖

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger2</artifactId>

<version>2.6.1</version>

</dependency>

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger-ui</artifactId>

<version>2.6.1</version>

</dependency>

创建swagger的配置类

@Bean

public Docket createRestApi() {

return new Docket(DocumentationType.SWAGGER\_2)

.apiInfo(apiInfo())

.select()

//接口所在路径

.apis(RequestHandlerSelectors.basePackage("com.tuling.controller"))

.paths(PathSelectors.any())

.build();

}

private ApiInfo apiInfo() {

return new ApiInfoBuilder()

.title("springboot利用swagger构建api文档")

.description("tuling-swagger2整合测试用例")

//联系人姓名 URL 邮箱

.contact(new Contact("Kevin", "", "qq.com"))

//项目描述

.termsOfServiceUrl("http://www.baidu.com")

.version("1.0")

.build();

}

@Api：修饰整个类，描述Controller的作用

@ApiOperation：描述一个类的一个方法，或者说一个接口

@ApiParam：单个参数描述

@ApiModel：用对象来接收参数

@ApiProperty：用对象接收参数时，描述对象的一个字段

@ApiResponse：HTTP响应其中1个描述

@ApiResponses：HTTP响应整体描述

@ApiIgnore：使用该注解忽略这个API

@ApiError ：发生错误返回的信息

@ApiImplicitParam：一个请求参数

@ApiImplicitParams：多个请求参数

