# 基础

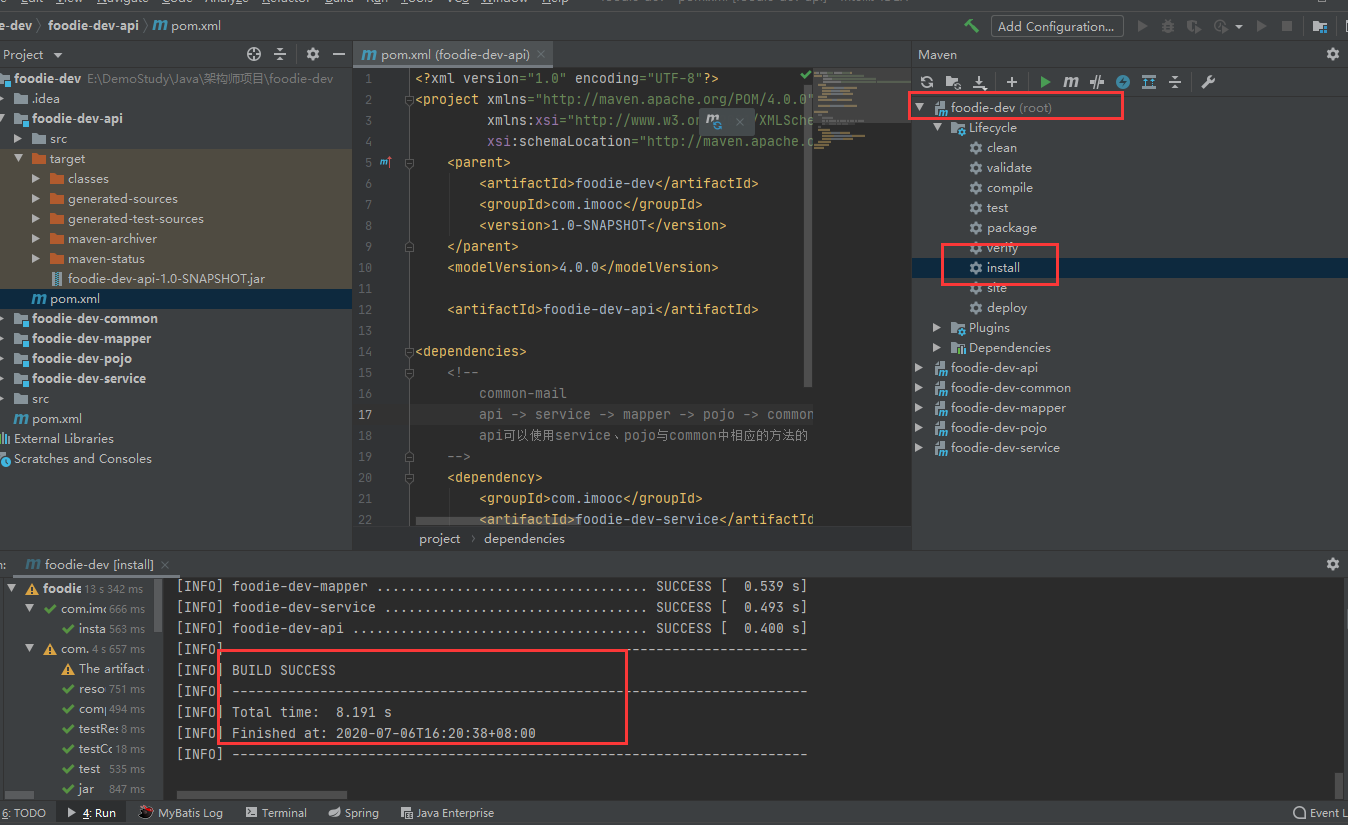
## 技术选型

* SpringMvc是框架 SpringBoot是工具
* SrpingBoot配置都设置在yml中
* SpringBoot集成了多样化中间件[\*-stater]
* SpringBoot从外置tomcat变为内置tomcat
* YAML是一种层级格式，它和.properties很容易互相转换，它的优点是去掉了大量重复的前缀，并且更加易读。

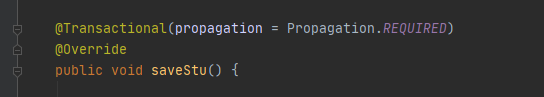
## 数据设计相关

* 数据库外键(最好不要)
  + 对性能有影响
  + 导致 热跟新 无法使用
  + 降低耦合度 影响
  + 难以 数据库分库分表

## 项目(聚合工程)创建完毕之后 因为有依赖关系，所以需要【安装】，安装之后依赖关系才会建立



## 事务的传播



事务传播 - Propagation

* REQUIRED: 使用当前的事务，如果当前没有事务，则自己新建一个事务，子方法是必须运行在一个事务中的；
  + 如果当前存在事务，则加入这个事务，成为一个整体。 举例：领导没饭吃，我有钱，我会自己买了自己吃；领导有的吃，会分给你一起吃。
* SUPPORTS: 如果当前有事务，则使用事务；如果当前没有事务，则不使用事务。
  + 举例：领导没饭吃，我也没饭吃；领导有饭吃，我也有饭吃。
* MANDATORY: 该传播属性强制必须存在一个事务，如果不存在，则抛出异常
  + 举例：领导必须管饭，不管饭没饭吃，我就不乐意了，就不干了（抛出异常）
* REQUIRES\_NEW: 如果当前有事务，则挂起该事务，并且自己创建一个新的事务给自己使用；如果当前没有事务，则同 REQUIRED
  + 举例：领导有饭吃，我偏不要，我自己买了自己吃
* NOT\_SUPPORTED: 如果当前有事务，则把事务挂起，自己不适用事务去运行数据库操作
  + 举例：领导有饭吃，分一点给你，我太忙了，放一边，我不吃
* NEVER: 如果当前有事务存在，则抛出异常
  + 举例：领导有饭给你吃，我不想吃，我热爱工作，我抛出异常
* NESTED: 如果当前有事务，则开启子事务（嵌套事务），嵌套事务是独立提交或者回滚；

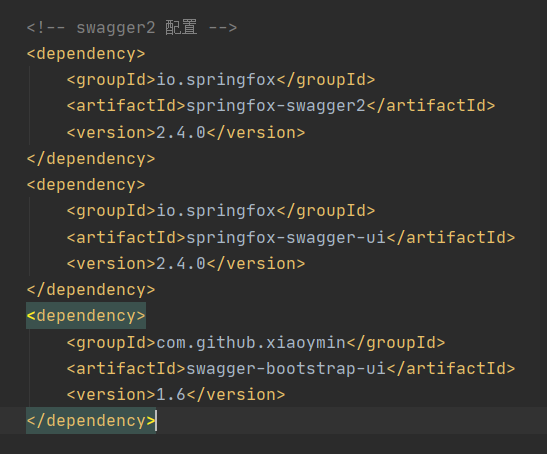
如果当前没有事务，则同 REQUIRED。

但是如果主事务提交，则会携带子事务一起提交。

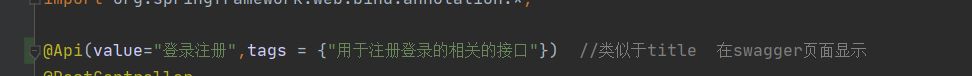
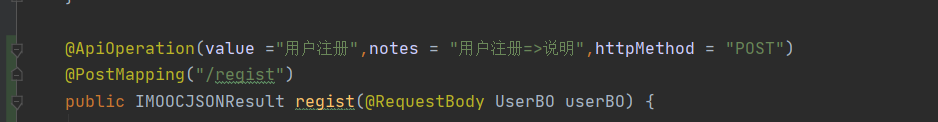
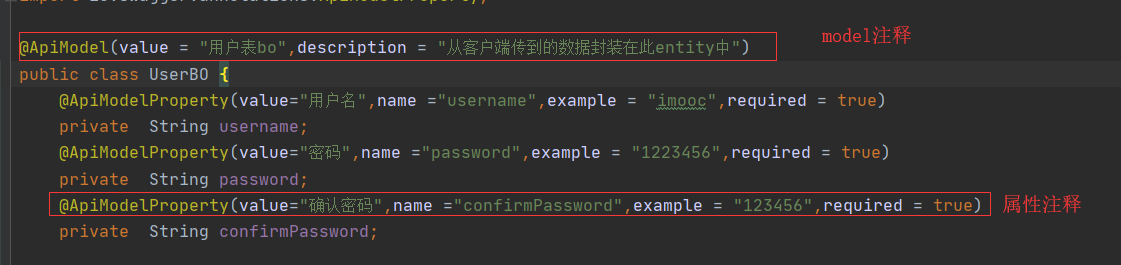
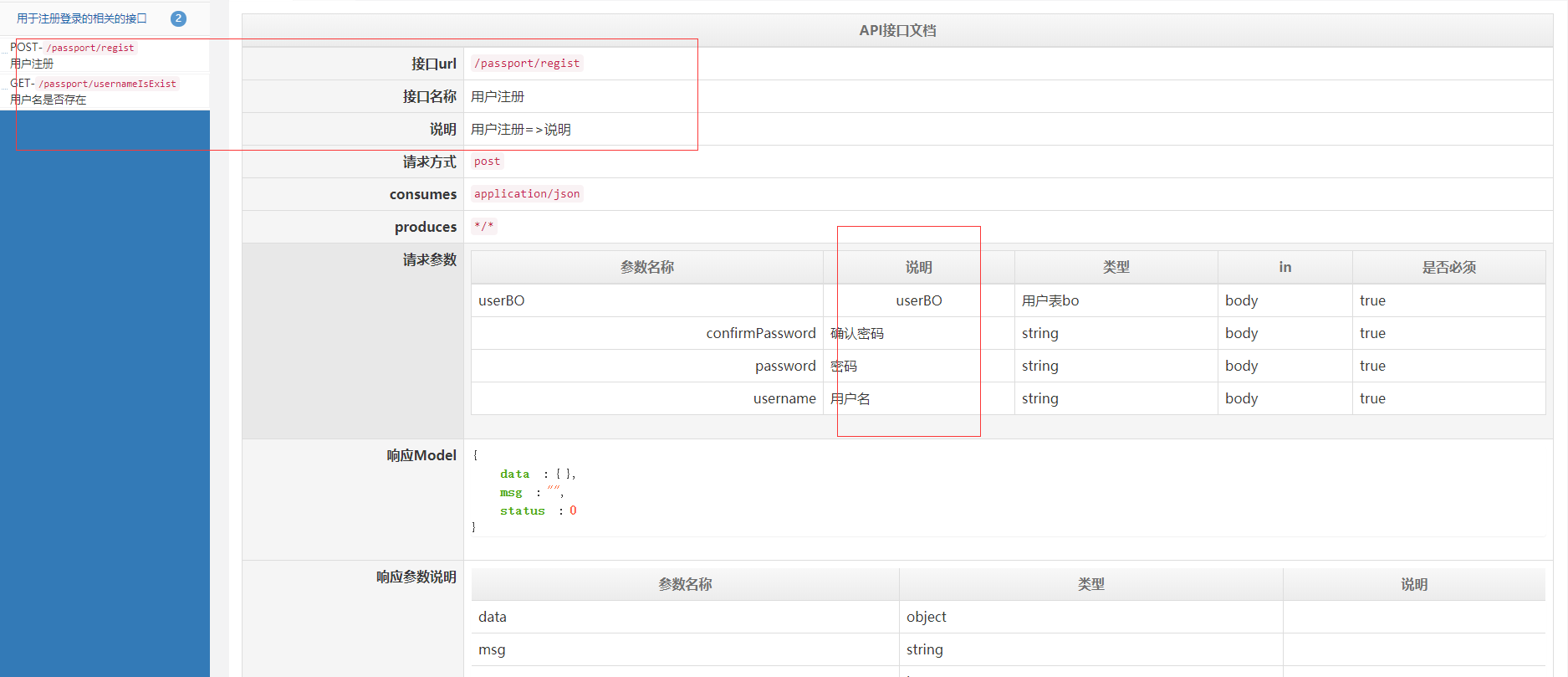
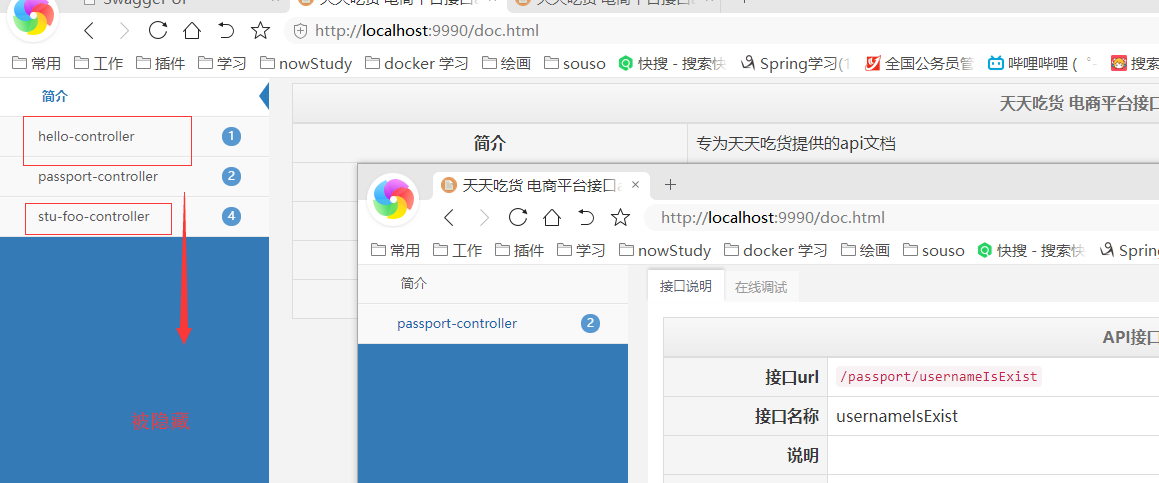
如果主事务回滚，则子事务会一起回滚。相反，子事务异常，则父事务可以回滚或不回滚。

* + 举例：领导决策不对，老板怪罪，领导带着小弟一同受罪。小弟出了差错，领导可以推卸责任。

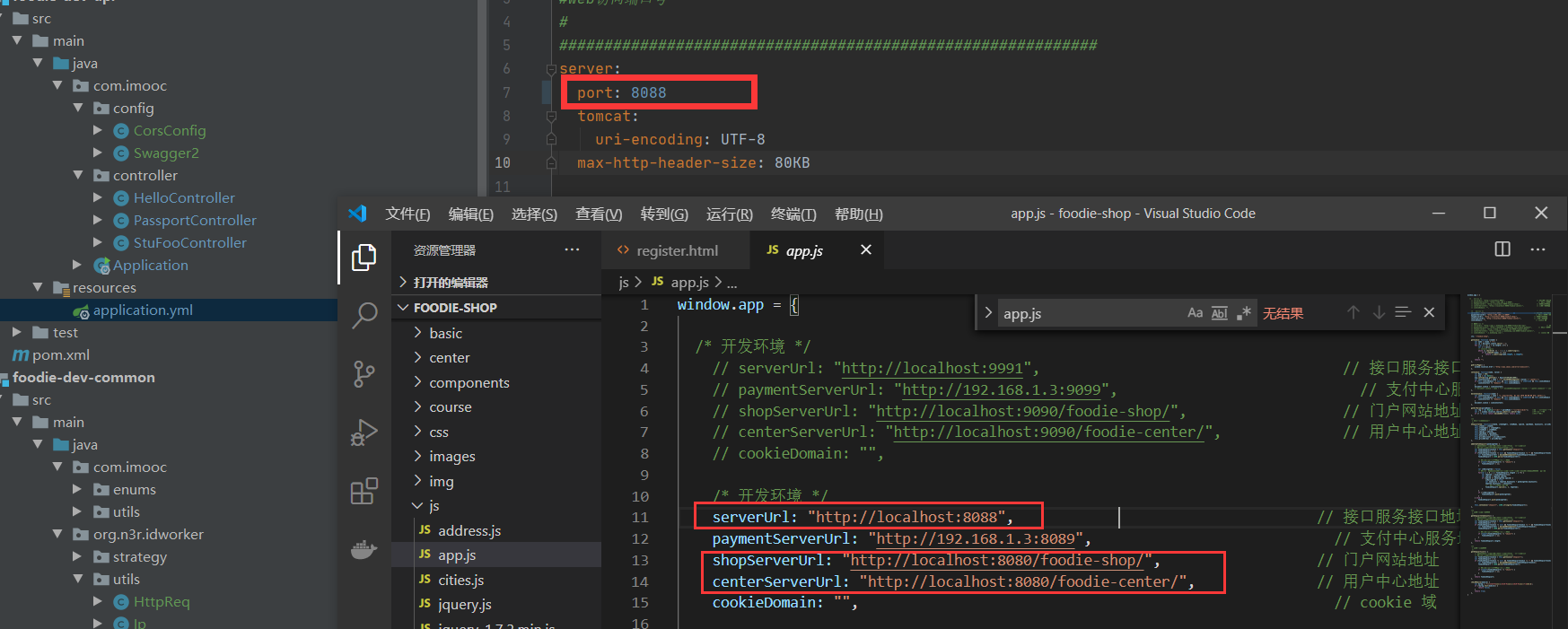
## api接口

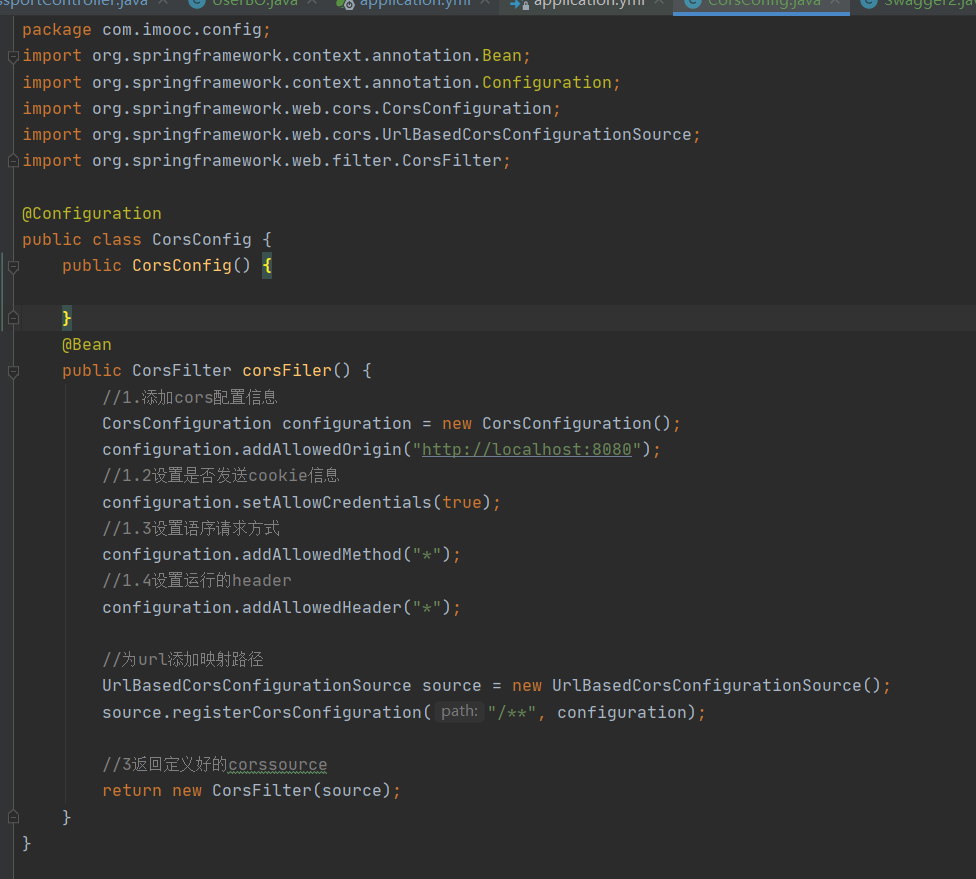
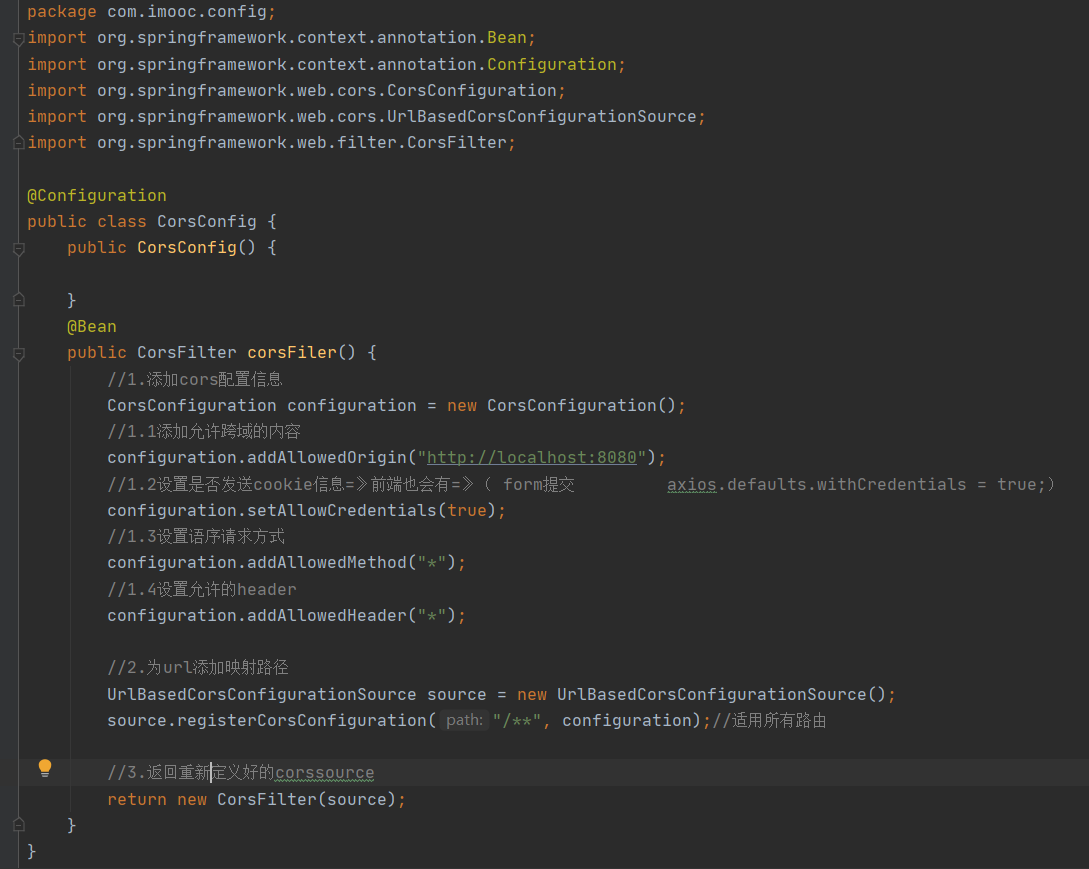
* 需要现在pom.xml中引入以下文件
  + 

|  |
| --- |
| <!-- swagger2 配置 -->  <dependency>  <groupId>io.springfox</groupId>  <artifactId>springfox-swagger2</artifactId>  <version>2.4.0</胜利ersion>  </dependency>  <dependency>  <groupId>io.springfox</groupId>  <artifactId>springfox-swagger-ui</artifactId>  <version>2.4.0</胜利ersion>  </dependency>  <dependency>  <groupId>com.github.xiaoymin</groupId>  <artifactId>swagger-bootstrap-ui</artifactId>  <version>1.6</胜利ersion>  </dependency> |

* 创建swagger相关内容
  + 
* 添加各种注释
  + 对controller的注释
    - 
  + 对active的注释
    - 
  + 对model的注释
    - 
    - 内容显示
      * 
* 取消页面显示
  + 
  + 

## 关于跨域问题=》使用CorsFilter解决



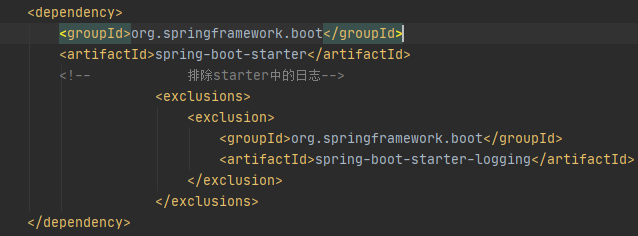
* 由于前端端口(8080)和后台端口(8088)不是同一个端口，在前端进行请求访问后台的时候，会出现跨域问题，所以需要在后台添加以下模块（当前在tomcat下运行）
  + 

|  |
| --- |
| package com.imooc.config;  import org.springframework.context.annotation.Bean;  import org.springframework.context.annotation.Configuration;  import org.springframework.web.cors.CorsConfiguration;  import org.springframework.web.cors.UrlBasedCorsConfigurationSource;  import org.springframework.web.filter.CorsFilter;  @Configuration  public class CorsConfig {  public CorsConfig() {  }  @Bean  public CorsFilter corsFiler() {  //1.添加cors配置信息  CorsConfiguration configuration = new CorsConfiguration();  //1.1添加允许跨域的内容  configuration.addAllowedOrigin("http://localhost:8080");  //1.2设置是否发送cookie信息=》前端也会有=》（ form提交 axios.defaults.withCredentials = true;）  configuration.setAllowCredentials(true);  //1.3设置语序请求方式  configuration.addAllowedMethod("\*");  //1.4设置允许的header  configuration.addAllowedHeader("\*");  //2.为url添加映射路径  UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();  source.registerCorsConfiguration("/\*\*", configuration);//适用所有路由  //3.返回重新定义好的corssource  return new CorsFilter(source);  }  } |

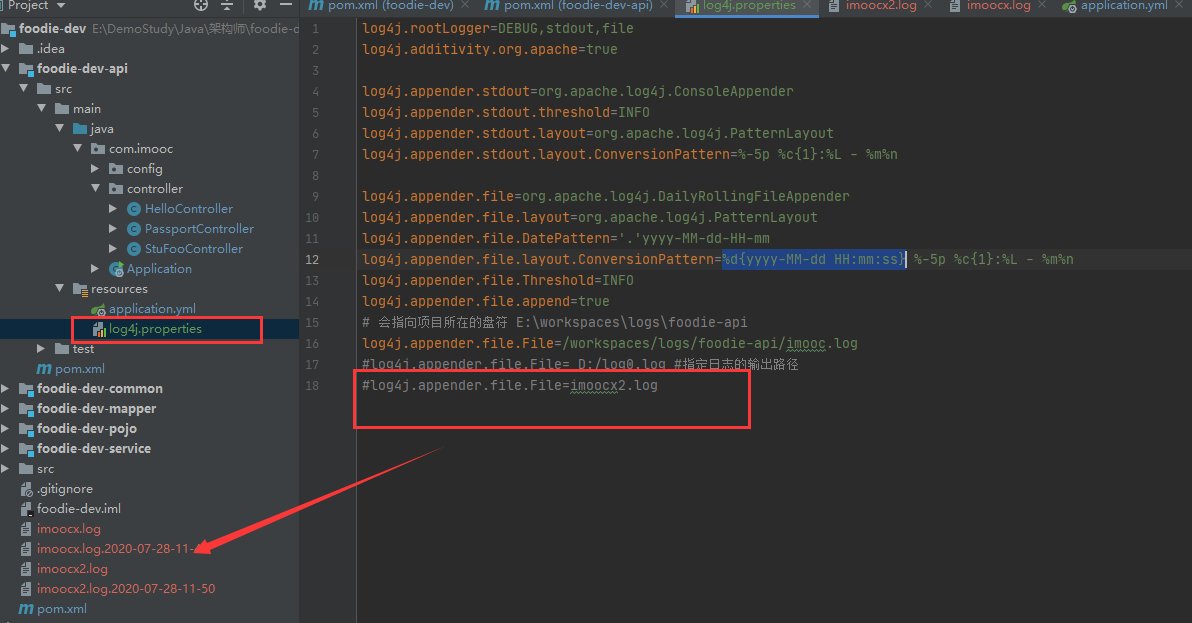
## log4j 记录日志

* 首先在pom.xml中添加以下配置
  + 

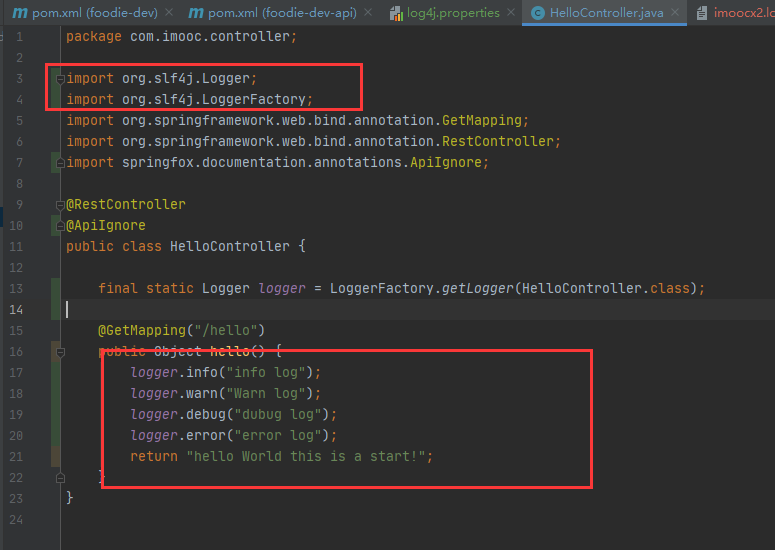
|  |
| --- |
| <!--引入日志依赖 抽象层 与 实现层-->  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-api</artifactId>  <version>1.7.21</version>  </dependency>  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-log4j12</artifactId>  <version>1.7.21</version>  </dependency> |

* 如果pom.xml中有引用 spring-boot-starter，则需要去掉本身日志
  + 

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter</artifactId>  <!-- 排除starter中的日志-->  <exclusions>  <exclusion>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-logging</artifactId>  </exclusion>  </exclusions>  </dependency> |

* 然后添加相关文件 log4j.properties
  + 

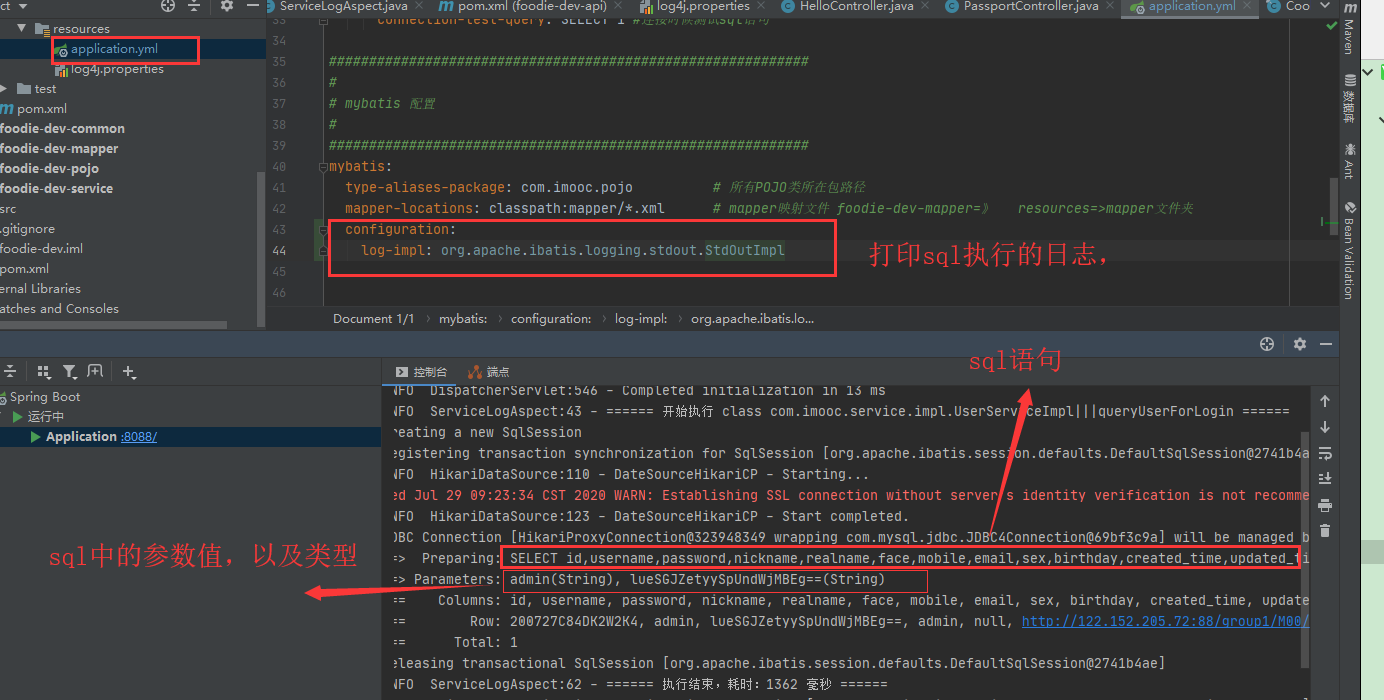
|  |
| --- |
| log4j.rootLogger=DEBUG,stdout,file  log4j.additivity.org.apache=true  log4j.appender.stdout=org.apache.log4j.ConsoleAppender  log4j.appender.stdout.threshold=INFO  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  log4j.appender.stdout.layout.ConversionPattern=%-5p %c{1}:%L - %m%n  log4j.appender.file=org.apache.log4j.DailyRollingFileAppender  log4j.appender.file.layout=org.apache.log4j.PatternLayout  log4j.appender.file.DatePattern='.'yyyy-MM-dd-HH-mm  log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n  log4j.appender.file.Threshold=INFO  log4j.appender.file.append=true  # 会指向项目所在的盘符 E:\workspaces\logs\foodie-api  log4j.appender.file.File=/workspaces/logs/foodie-api/imooc.log  #log4j.appender.file.File= D:/log0.log #指定日志的输出路径  #log4j.appender.file.File=imoocx2.log |

* 测试
  + 

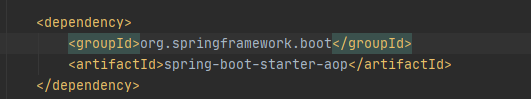
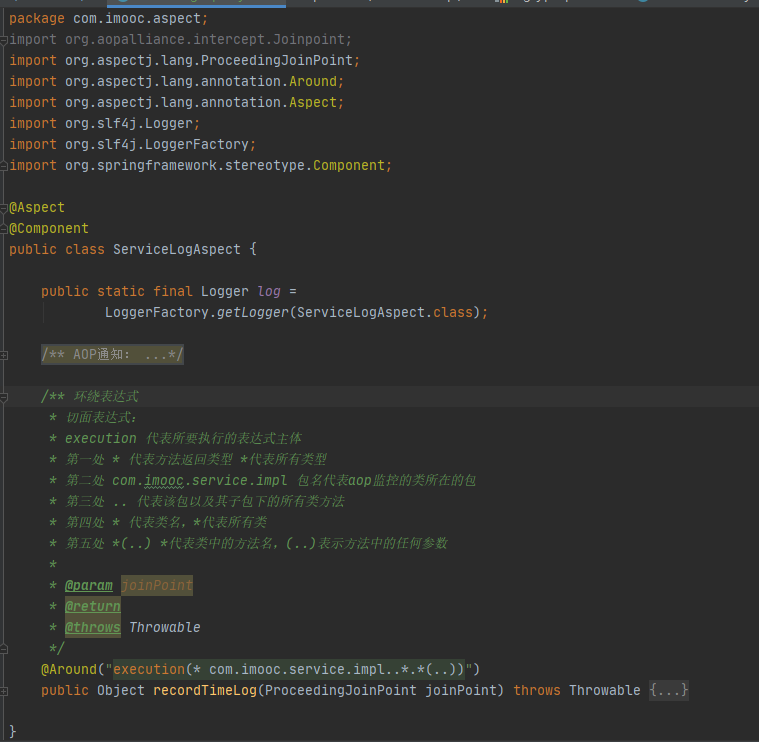
### 记录sql执行记录

configuration:

log-impl: org.apache.ibatis.logging.stdout.StdOutImpl



## Spring Aop

* Spring aop有五种类型
  + 前置通知：在方法调用之前执行 before
  + 后置通知：在方法正常调用之后执行 afterReturn
  + 环绕通知：在方法调用之前和之后，都分别可以执行的通知 around
  + 异常通知：如果在方法调用过程中发生异常，则通知 afterThrow
  + 最终通知：在方法调用之后执行 after
* 添加相应依赖
  + 
* 添加相对应类
  + 
* 代码部分

|  |
| --- |
| package com.imooc.aspect;  import org.aopalliance.intercept.Joinpoint;  import org.aspectj.lang.ProceedingJoinPoint;  import org.aspectj.lang.annotation.Around;  import org.aspectj.lang.annotation.Aspect;  import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  import org.springframework.stereotype.Component;  @Aspect  @Component  public class ServiceLogAspect {  public static final Logger log =  LoggerFactory.getLogger(ServiceLogAspect.class);  /\*\* 环绕表达式  \* 切面表达式：  \* execution 代表所要执行的表达式主体  \* 第一处 \* 代表方法返回类型 \*代表所有类型  \* 第二处 com.imooc.service.impl 包名代表aop监控的类所在的包  \* 第三处 .. 代表该包以及其子包下的所有类方法  \* 第四处 \* 代表类名，\*代表所有类  \* 第五处 \*(..) \*代表类中的方法名，(..)表示方法中的任何参数  \*  \* @param joinPoint  \* @return  \* @throws Throwable  \*/  @Around("execution(\* com.imooc.service.impl..\*.\*(..))")  public Object recordTimeLog(ProceedingJoinPoint joinPoint) throws Throwable {  //某一个service 中的某一个方法  log.info("====== 开始执行 {}|||{} ======",  joinPoint.getTarget().getClass(),  joinPoint.getSignature().getName());  // 记录开始时间  long begin = System.currentTimeMillis();  // 执行目标 service  Object result = joinPoint.proceed();  // 记录结束时间  long end = System.currentTimeMillis();  long takeTime = end - begin;  if (takeTime > 3000) {  log.error("====== 执行结束，耗时：{} 毫秒 ======", takeTime);  } else if (takeTime > 2000) {  log.warn("====== 执行结束，耗时：{} 毫秒 ======", takeTime);  } else {  log.info("====== 执行结束，耗时：{} 毫秒 ======", takeTime);  }  return result;  }  } |

# 看到

