# 简介

## SpringBoot优势

简化配置：

SpringBoot是对Spring的进一步封装，基于注解开发，舍弃笨重的XML，确实需要配置的使用yml或properties进行简要的配置即可

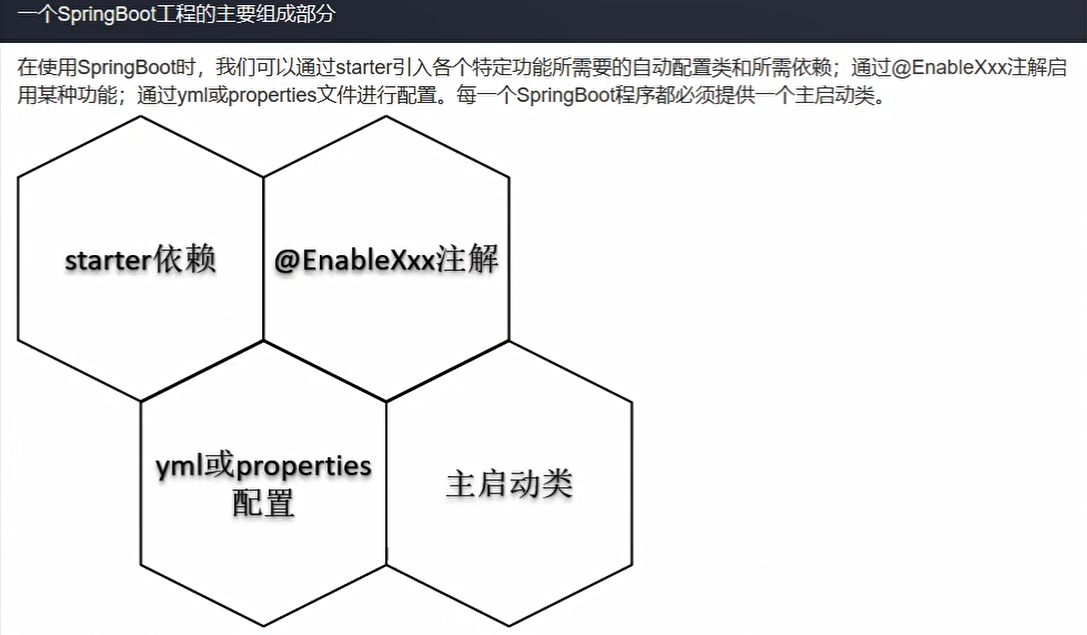
产品独立运行：

每一个工程都可以打成一个jar包，其中内置了Tomcat或其他Servlet容器，可以独立运行，这是和微服务理念最为契合的一点

强大的场景启动器：

每一个特定场景下的需求都封装成了一个starter，只要导入这个starter就有了这个场景所需要的一切。其中包括针对这个场景的自动化配置、依赖信息。

## SpringBoot工程组成



## 使用版本

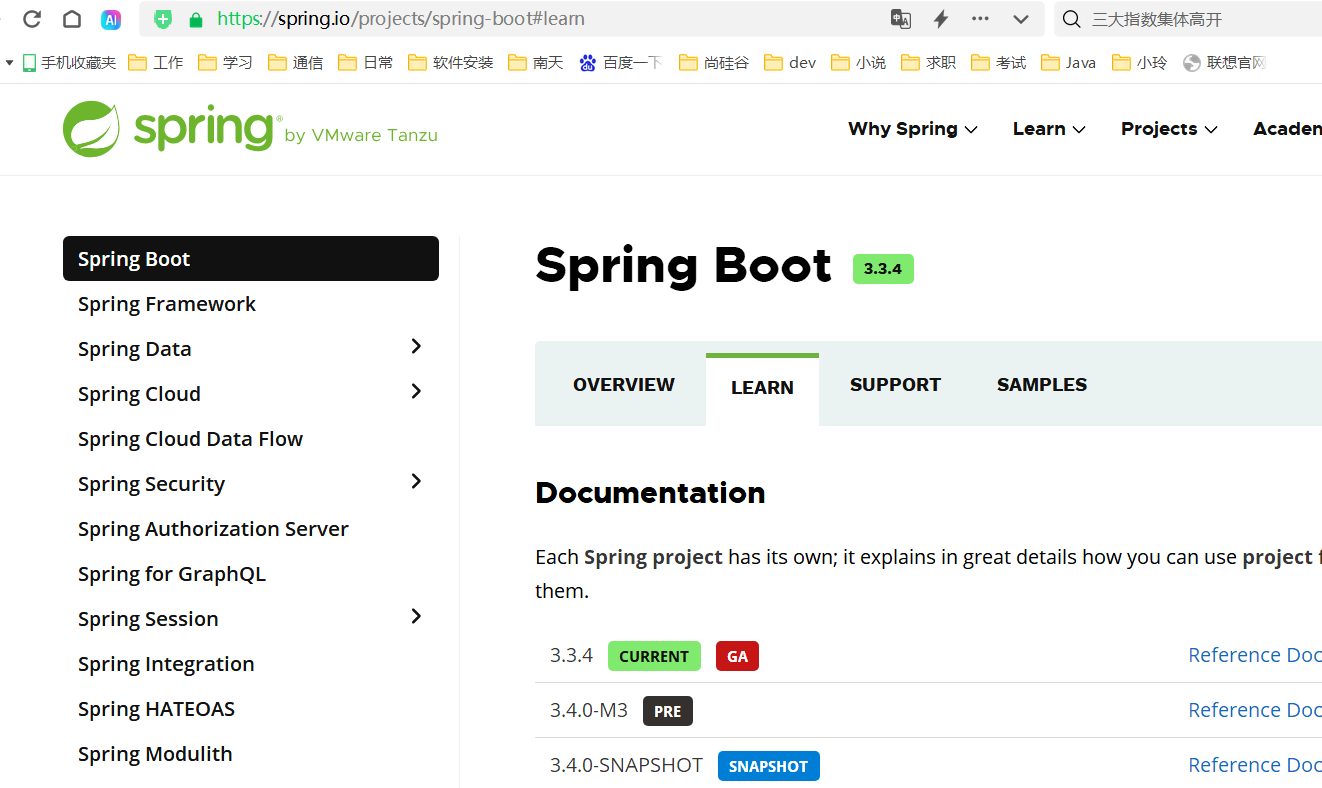
使用2.1.6版本

要求 spring5 jdk1.8以上

地址：

https://spring.io/projects/spring-boot#learn

GA是稳定版，snapshot是快照，不稳定版本



# 环境搭建

## 创建项目的三种方式

创建SpringBoot工程有两种方式

方式一：先创建Maven工程，然后引入SpringBoot环境

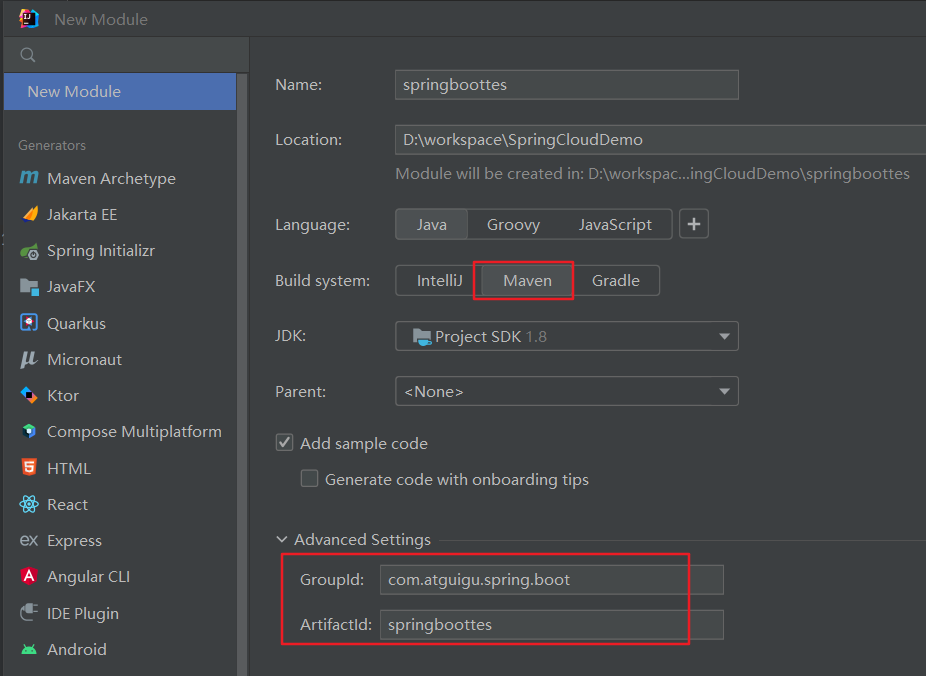
方式二：使用SpringBoot插件创建，主启动类、测试类、pom.xml都会自动创建好，要求必须联网

方式三：下载官网提供的包，解压即可

建议使用方式一，需要什么依赖手动添加即可

### 方式一

1.创建一个Maven工程



2.引入SpringBoot依赖，注意是打jar包

<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
  
 <groupId>com.atguigu.spring.boot</groupId>  
 <artifactId>springboot</artifactId>  
 <version>1.0-SNAPSHOT</version>  
 <packaging>jar</packaging>  
  
  
 <!-- 继承SpringBoot官方指定的父工程 -->  
 <parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>2.1.6.RELEASE</version>  
 </parent>  
  
 <dependencies>  
 <!-- 加入Web开发所需要的场景启动器 -->  
 <dependency>  
 <!-- 指定groupId和artifactId即可，版本已在父工程中定义 -->  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
 </dependencies>  
  
 <!-- Maven构建过程相关配置 -->  
 <build>  
 <!-- 构建过程中所需要用到的插件 -->  
 <plugins>  
 <!-- 这个插件将SpringBoot应用打包成一个可执行的jar包 -->  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 </plugin>  
 </plugins>  
 </build>  
  
</project>

3.创建主启动类

需要加上@SpringBootApplication注解

类名随便起

要有main方法，main方法里面的内容固定

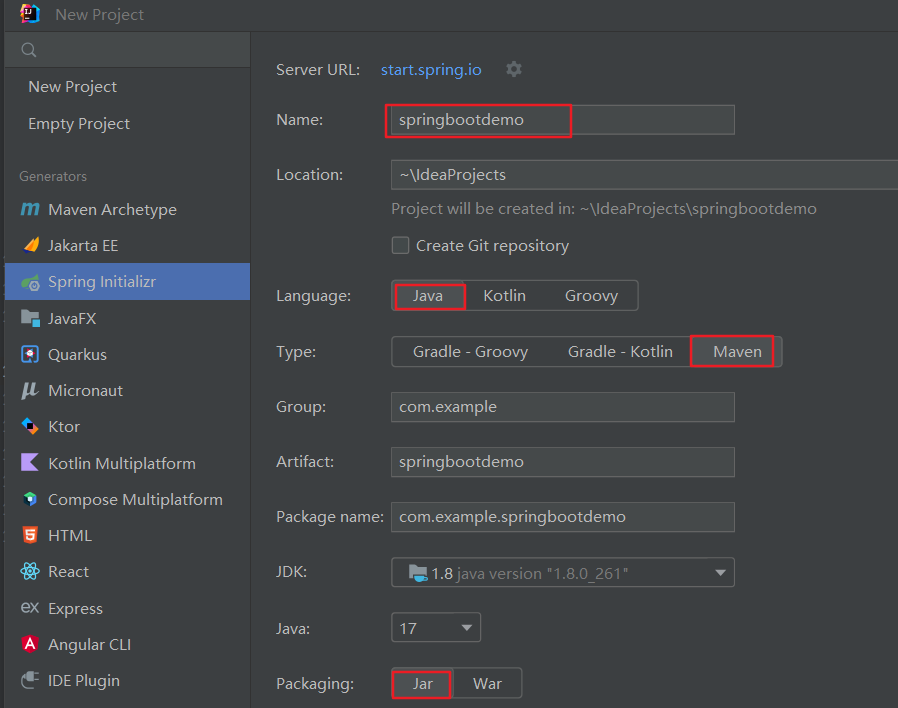
//将当前类标记为一个SpringBoot应用

@SpringBootApplication  
public class HelloWordMain {  
 public static void main(String[] args) {  
 SpringApplication.*run*(HelloWordMain.class,args);  
 System.*out*.println("Hello world!");  
 }  
}

### 方式二

新建项目，使用spring initializr

file->new project



2.选择SpringBoot版本和添加web目录



等待工程创建好，依赖下载好后，不需要修改即可启动

如果想要修改配置，改pom.xml即可

### 方式三

<http://start.spring.io/>



下载后解压，直接用IDEA打开即可

配置好Maven和jdk，下载依赖后即可启动

默认访问8080端口，可在配置文件修改

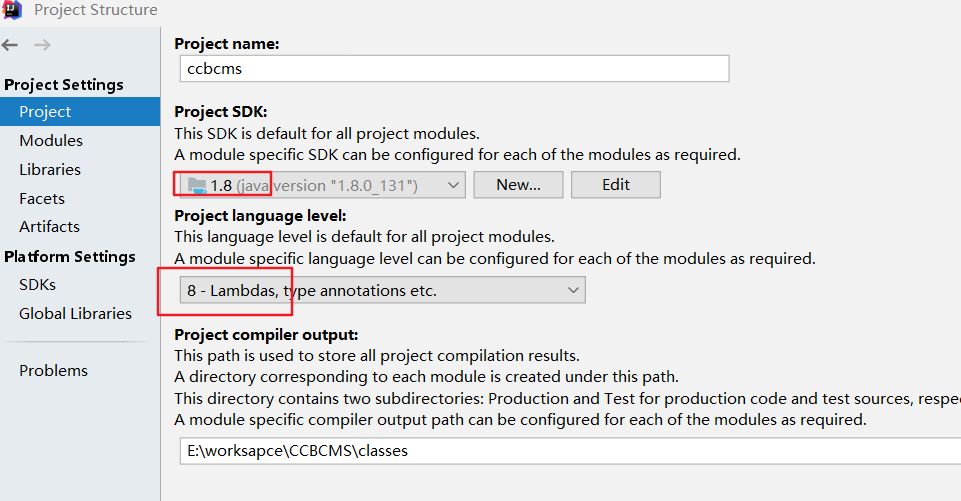
## 错误汇总

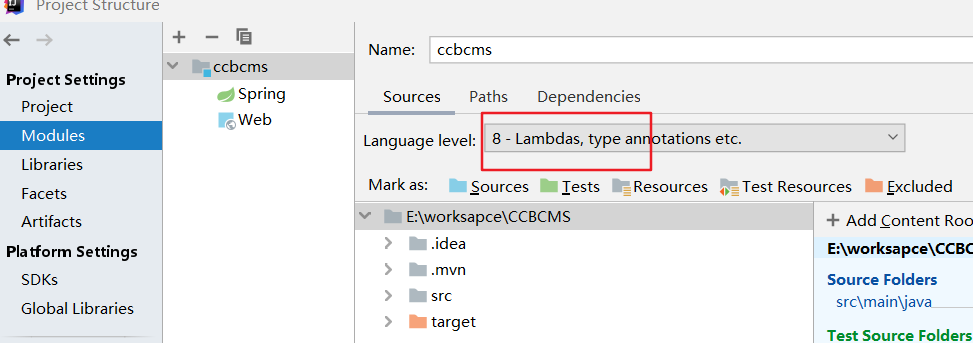
1.启动报错：Error:(3,32) java: 无法访问org.springframework.boot.SpringApplication

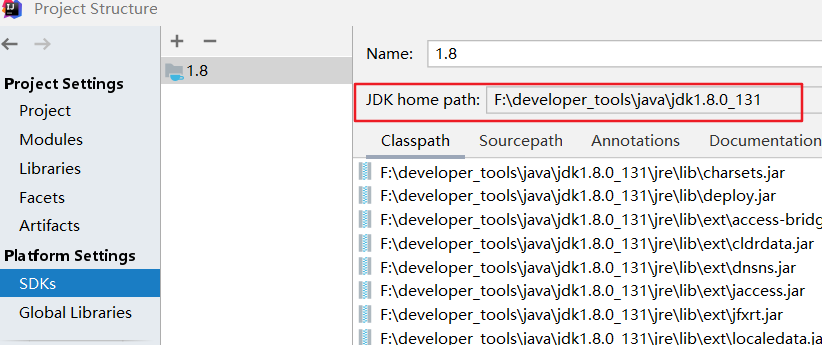
解决方案：springboot的版本过高，适当调低springboot版本

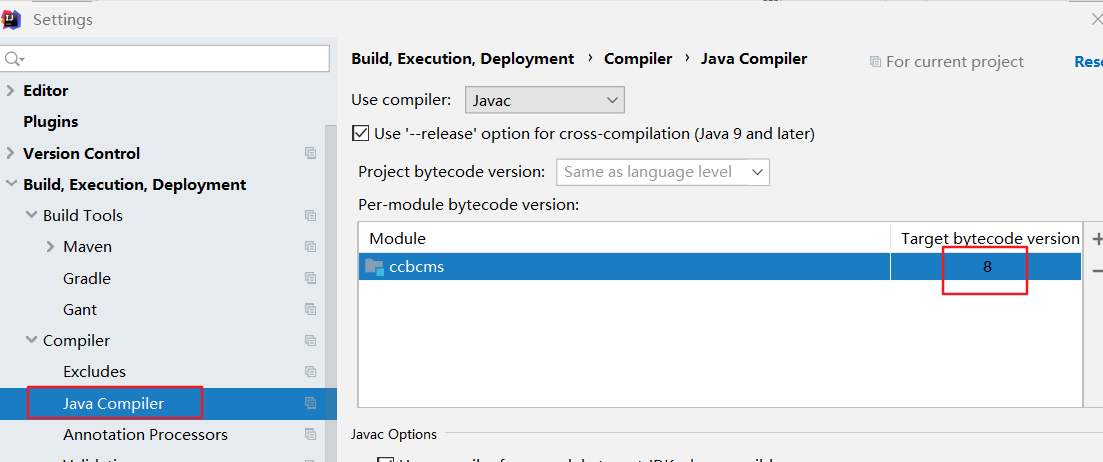
编译报错：Failed to execute goal org.apache.maven.plugins:maven-compiler-plugin:3.10.1:compile (default-compil

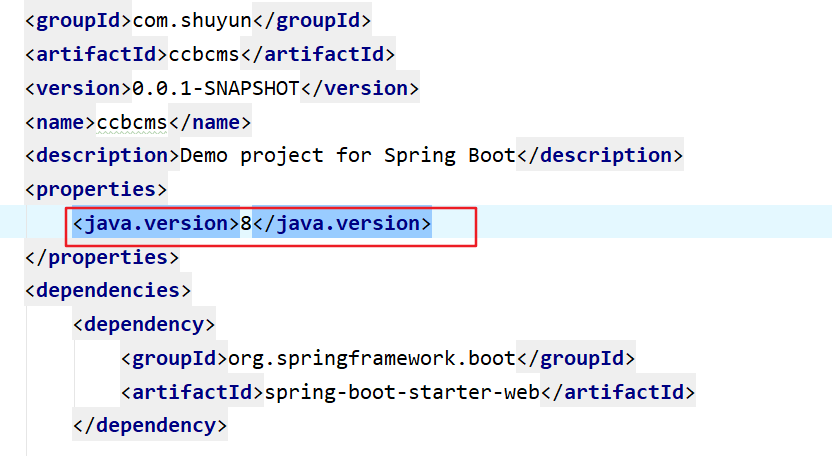
解决方案：











# Application.yml配置

注意：在冒号后腰加空格，否则配置不生效且启动不报错

多环境配置

不管采纳哪种形式，它们都能通过文件名的命名形式区分出不同的环境的配置，文件命名格式为：application-{profile}.properties 和 application-{profile}.yml。

这里使用yml方式进行配置

步骤一：在resource根目录下创建以下文件：

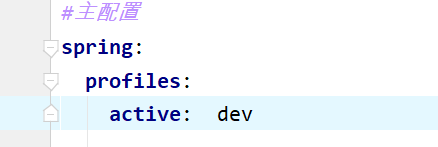
application.yml #项目主配置文件，所有公共配置；

application-dev.yml #开发环境配置文件；

application-test.yml #测试环境配置文件；

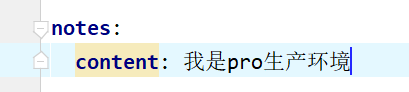
application-prod.yml #生产环境配置文件。

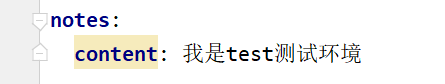
步骤二：在主配置文件application.yml中指定生效的配置文件



步骤三：分别在dev pro test三个配置文件中加一个属性配置：



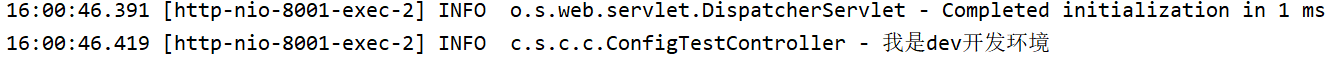




步骤四：测试生效的配置文件

@Value(**"${notes.content}"**)  
**private** String **content**;  
  
@RequestMapping(**"/environment"**)  
**public** String ConfigEnvironmentTest(){  
  
 ***logger***.info(**content**);  
 **return "success"**;  
}

结果：



# 日志

1.日志级别

trace：微量，少许的意思，级别最低；

debug：需要调试时候的关键信息打印；

info：普通的打印信息（默认⽇志级别）；

warn：警告，不影响使⽤，但需要注意的问题；

error：错误信息，级别较⾼的错误⽇志信息；

fatal：致命的，因为代码异常导致程序退出执⾏的事件。

即越往上接收到的消息就越少，如设置了 warn 就只能收到 warn、error、fatal 级别的⽇志

2.日志格式

%m 输出代码中指定的消息

%p 输出优先级，即DEBUG，INFO，WARN，ERROR，FATAL

%r 输出自应用启动到输出该log信息耗费的毫秒数

%c 输出所属的类目，通常就是所在类的全名

%t 输出产生该日志事件的线程名

%n 输出一个回车换行符，Windows平台为“\\r\\n”，Unix平台为“\\n”

%d 输出日志时间点的日期或时间,默认格式为ISO8601,也可以在其后指定格式，比 如%d{yyy MMM dd HH:mm:ss,SSS}，输出类似：2018年6月15日22:10:28，921

%l 输出日志事件的发生位置，包括类目名、发生的线程，以及在代码中的行数。举 例：estlog4.main(TestLog4.java: 10)

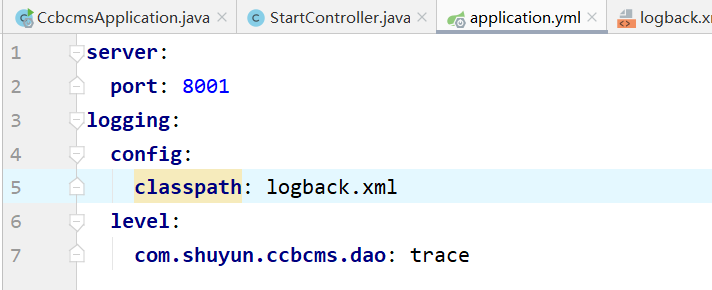
3.日志配置文件位置

Src/main/resource/

slf4j简单日志门面

步骤一：aoolication.yml中指定日志配置文件logback.xml

日志级别：level用于指定mapper 的输出级别 trace会把sql打印出来



步骤二：在logback.xml中配置日志的相关信息

注意logback.xml和application.yml是同在一个目录下，即resource根目录下

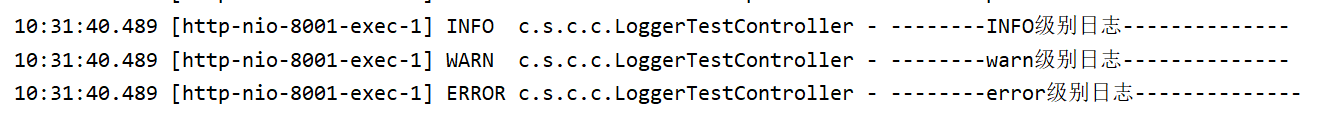
*<?***xml version="1.0" encoding="UTF-8"***?>  
<!---->*<**configuration**>  
 *<!--首先定义一个格式，命名为 “LOG\_PATTERN”，该格式中 %d表示日期，%t表示线程名，%-5level 表示级别从左显示5个字符宽度，  
 %logger{36} 表示 logger 名字最长36个字符，%msg 表示日志消息，%n 是换行符。-->* <**property name="LOG\_PATTERN" value="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"** />  
 *<!--定义名为 “FILE\_PATH” 文件路径，日志都会存储在该路径下。%i 表示第 i 个文件，当日志文件达到指定大小时，会将日志生成到新的文件里，这里的 i 就是文件索引，日志文件允许的大小可以设置。  
 这里需要注意的是，不管是 windows 系统还是 Linux 系统，日志存储的路径必须要是绝对路径。-->* <**property name="FILE\_PATH" value="E:/logs/ccbcms/ccbcms.%d{yyyy-MM-dd}.%i.log"** />  
  
 *<!--定义控制台输出-->* <**appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender"**>  
 <**encoder**>  
 *<!--按照上面定义的格式输出日志-->* <**pattern**>${LOG\_PATTERN}</**pattern**>  
 </**encoder**>  
 </**appender**>  
  
 *<!--使用 定义一个名为 “FILE” 的文件配置，主要是配置日志文件保存的时间、  
 单个日志文件存储的大小、以及文件保存的路径和日志的输出格式-->* <**appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender"**>  
 <**rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy"**>  
 *<!-- 按照上面配置的FILE\_PATH路径来保存日志 -->* <**fileNamePattern**>${FILE\_PATH}</**fileNamePattern**>  
 *<!-- 日志保存15天 -->* <**maxHistory**>15</**maxHistory**>  
 <**timeBasedFileNamingAndTriggeringPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP"**>  
 *<!-- 单个日志文件的最大，超过则新建日志文件存储 -->* <**maxFileSize**>10MB</**maxFileSize**>  
 </**timeBasedFileNamingAndTriggeringPolicy**>  
 </**rollingPolicy**>  
  
 <**encoder**>  
 *<!-- 打印日志的格式 -->* <**pattern**>${LOG\_PATTERN}</**pattern**>  
 </**encoder**>  
 </**appender**>  
  
 *<!--指定日志级别-->* <**logger name="com.shuyun.ccbcms" level="INFO"** />  
 *<!--控制台输出和日志文件-->* <**root level="INFO"**>  
 <**appender-ref ref="CONSOLE"** />  
 <**appender-ref ref="FILE"** />  
 </**root**>  
  
</**configuration**>

日志使用

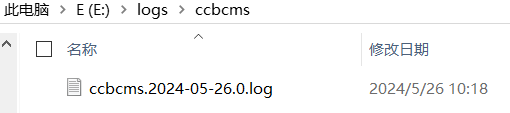
在代码中，我们一般使用 Logger 对象来打印出一些 log 信息，可以指定打印出的日志级别，也支持占位符，很方便

@RestController  
@RequestMapping(**"test"**)  
**public class** LoggerTestController {  
  
 **private final static** Logger ***logger*** = LoggerFactory.*getLogger*(LoggerTestController.**class**);  
  
 @RequestMapping(**"logger"**)  
 **public** String LoggerTest(){  
 ***logger***.info(**"--------INFO级别日志--------------"**);  
 ***logger***.warn(**"--------warn级别日志--------------"**);  
 ***logger***.error(**"--------error级别日志--------------"**);  
 ***logger***.debug(**"--------debug级别日志--------------"**);  
  
 **return "success"**;  
 }  
  
}

因为info级别比debug级别高，所以debug级别日志没打印



同时在本地目录也生成了日志文件

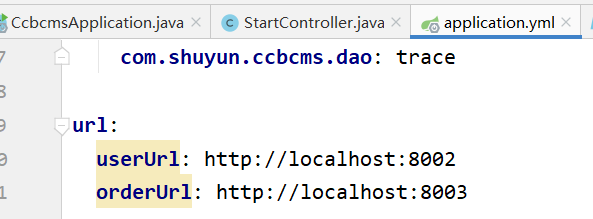


# 配置

配置文件方式

在application.yml中配置，使用@Value注解获取

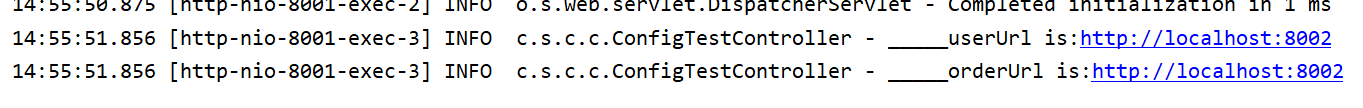
配置：



控制器：

@RestController  
@RequestMapping(**"/test"**)  
**public class** ConfigTestController {  
  
 **private final static** Logger ***logger*** = LoggerFactory.*getLogger*(ConfigTestController.**class**);  
  
 @Value(**"${url.userUrl}"**)  
 **private** String **userUrl**;  
  
 @Value(**"${url.orderUrl}"**)  
 **private** String **orderUrl**;  
  
 @RequestMapping(**"/config"**)  
 **public** String ConfigerTest(){  
  
 ***logger***.info(**"\_\_\_\_\_userUrl is:"**+**userUrl**);  
 ***logger***.info(**"\_\_\_\_\_orderUrl is:"**+**userUrl**);  
 **return "success"**;  
 }  
}

结果：



配置类方式

配置类：

使用 @ConfigurationProperties 注解并且使用 prefix 来指定一个前缀，然后该类中的属性名就是配置中去掉前缀后的名字，一一对应即可。即：前缀名 + 属性名就是配置文件中定义的 key。同时，该类上面需要加上 @Component 注解，把该类作为组件放到Spring容器中，让 Spring 去管理，我们使用的时候直接注入即可



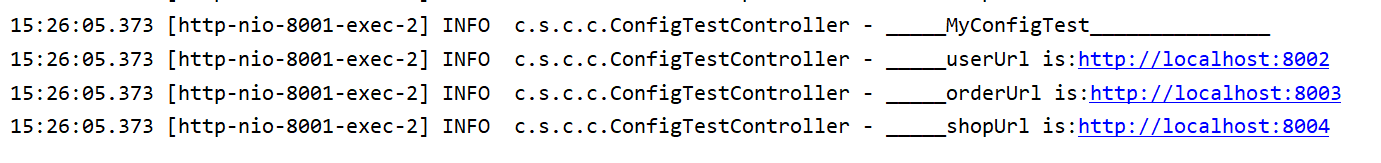
需要注意的是，使用 @ConfigurationProperties 注解需要导入它的依赖：

*<!--@ConfigurationProperties注解依赖-->*<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-configuration-processor</**artifactId**>  
 <**optional**>true</**optional**>  
</**dependency**>

控制器：

@Resource  
**private** MyConfigTest **config**;  
  
@RequestMapping(**"/myconfig"**)  
**public** String getConfigTest(){  
  
 ***logger***.info(**"\_\_\_\_\_MyConfigTest\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"**);  
 ***logger***.info(**"\_\_\_\_\_userUrl is:"**+**config**.getUserUrl());  
 ***logger***.info(**"\_\_\_\_\_orderUrl is:"**+**config**.getOrderUrl());  
 ***logger***.info(**"\_\_\_\_\_shopUrl is:"**+**config**.getShopUrl());  
 **return "success"**;  
}

结果：



# SpringBoot中MVC的支持

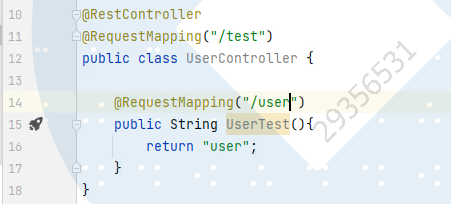
## @RestController

@RestController注解是SpringBoot新增的注解，相当于@Controller和@ResponseBody两个注解。

@Controller：标注一个控制层组件，交给Spring 管理

@ResponseBody：将返回数据结构转换为Json格式

注意：如果是前后端分离，使用了thymeleaf模板引擎，使用了@RestController注解后就不是返回页面了，而是返回Json字符串。这时候要改成使用@Controlle注解



## @RequestMapping

@RequestMapping注解是处理请求和控制器方法之间的映射关系。

作用在类上，表示以该地址作为父路径

作用在方法上，则进一步指定处理方法的映射关系

Value属性：值就是请求地址，斜杠表示上下文地址。如果多个控制器方法对应同一个请求，会报错

Method属性：该属性是一个RequestMethod类型数组，可以匹配多种请求方式

## @PathVariable

@PathVariable注解主要是用来获取url参数，SpringBoot支持restful风格的url

原始方式：/deleteUser?id=1

Rest方式：/deleteUser/1

restful风格：当请求路径中将某些数据通过路径的方式传到服务器中，就可以在相应@RequestMapping注解的value属性中通过占位符{xxx}的方式表示传输的数据，再通过@PathVariable注解，将占位符所表示的数据赋值给控制器方法的形参

控制器代码：

@RestController  
@RequestMapping("/test")  
public class UserController {  
  
 private static final Logger *log* = LoggerFactory.*getLogger*(UserController.class);  
  
 @RequestMapping("/user/{id}/{username}")  
 public String UserTest(@PathVariable("id") String uid,@PathVariable("username") String uname){  
 *log*.info("id="+uid+",username="+uname);  
 return "user";  
 }  
}

请求地址：

http://localhost:8001/test/user/333/shuyun

结果：



## @RequestParam

@RequestParam注解也是获取请求参数，而@PathVariable注解也是用于获取请求参数，两者的区别在于：

@PathVariable注解用于获取restful风格的url的参数：/deleteUser/1

@RequestParam注解用于获取原始方式url的参数：/deleteUser?id=1

控制器方法代码：

@RequestMapping("/user")  
public String UserTest(@RequestParam(value="id",defaultValue ="1001") String uid,@RequestParam(value="name",defaultValue ="shuyun") String uname){  
 *log*.info("id="+uid+",username="+uname);  
 return "success";  
}

请求url:

<http://localhost:8001/test/user?id=1111&name=shuyun333>

结果：



## 使用POJO获取请求参数

可以在控制器方法的形参中使用一个实体类来接收请求参数，要求请求参数的参数名和JavaBean的属性名保持一致，否则会赋值为null

控制器方法代码：

这里控制器类的注解使用的是：@RestController，所以会将User转换成Json格式

@RequestMapping("/userbean")  
public User getUserTest(User user){  
  
 return user;  
}

Postman请求参数与结果：因为password没对应上，所以为Null



## @RequestBody

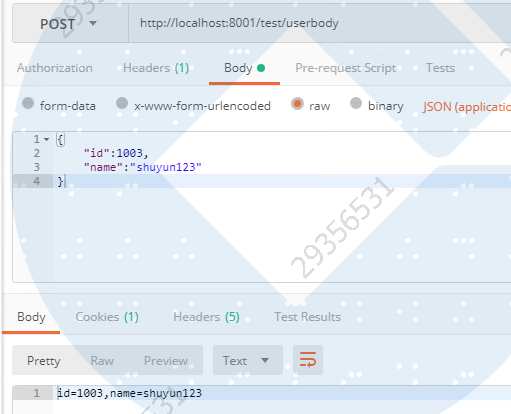
@RequestBody注解用于接收前端请求的请求体，例如：前端使用json传来两个参数id和name，此时可以在后端用User实体类来接收。

或者前端使用表单提交数据时，可以使用@RequestBody注解获取表单提交的数据

请求控制器方法：

@RequestMapping("/userbody")  
public String userbodyTest(@RequestBody User user){  
  
 return "id="+user.getId()+",name="+user.getName();  
}

Postman请求参数和结果：



# Thymeleaf模板引擎

## 相关配置

1.依赖导入

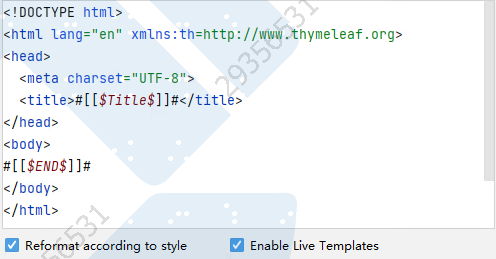
*<!--thymeleaf模板引擎-->*<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-thymeleaf</artifactId>  
</dependency>

2.在html页面的标签中引入thymeleaf模板

可以直接在html模板中加上

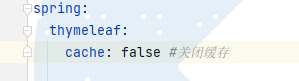
File-setting-edit-file and code templates

在HTML FILE中添加： <html lang="en" xmlns:th=http://www.thymeleaf.org>



3.在application.yml中关闭缓存

Thymeleaf默认是开启缓存的，会导致页面没法及时看到更新后的效果，需要手动关闭



## 访问静态页面

一般在做网站的时候，都会做一个404页面和500页面，为了在出错时给用户一个友好的展示，而不至于一堆的异常信息抛出来。

SpringBoot中会自动识别模板目录(templates/)下的404.html和500.html文件。可以在templates/目录下新建一个error文件夹，专门放置错误的html页面，然后分别打印信息。以500.html页面为例：

控制器方法代码：

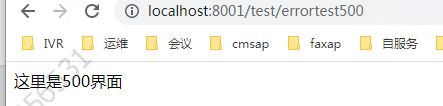
注意，这里的控制器注解不能使用@RestController了，因为要返回视图名称而不是返回Json格式的字符串了

@Controller  
@RequestMapping("test")  
public class ThymeleafController {  
  
 @RequestMapping("errortest500")  
 public String thymeleafErrorTest(){  
 int i = 1 / 0;  
 return "index";  
 }  
  
}

500.html页面：

<!DOCTYPE html>  
<html lang="en" xmlns:th=http://www.thymeleaf.org>  
<head>  
 <meta charset="UTF-8">  
 <title>Title</title>  
</head>  
<body>  
 这里是500界面  
</body>  
</html>

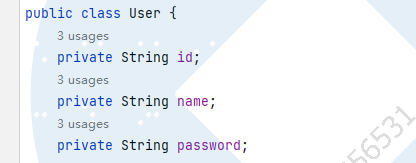
请求url及结果：



## 对象访问

1.创建User实体类

3个属性和空参构造和全属性的构造



2.控制器方法向request域共享数据

注意这里是返回视图

@RequestMapping("/userinfo")  
public String getUserInfo( Model model){  
 User user = new User("10001","shuyun","123456");  
 model.addAttribute("user",user);  
 return "user";  
}

3.前端页面渲染数据

<body>  
<table border="1" cellpadding="6" style="text-align:center" id="usertable">  
 <tr>  
 <th colspan="5">用户信息</th>  
 </tr>  
 <tr>  
 <th>id</th>  
 <th>name</th>  
 <th>password</th>  
 </tr>  
  
 <tr th:object="${user}">  
 <td th:text="${user.id}"></td>  
 <td th:value="\*{name}"></td>  
 <td th:text="${user.getPassword()}"></td>  
 </tr>  
</table>  
</body>

注意：user.html页面要放在templates目录下

说明：在thymeleaf模板中可以通过三种方式来获取对象属性

首先获取对象：

th:object="${user}"

然后获取属性：

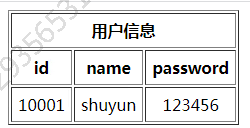
这里text和value都可以

方式一： th:text="${user.id}"

方式二：th:value="\*{name}"

方式三：th:text="${user.getPassword()}"

结果展示：



## List集合处理

控制器方法：

@RequestMapping("/userlist")  
public String getUserList( Model model){  
  
 User user1 = new User("1001", "shuyun1", "123");  
 User user2 = new User("1002", "shuyun2", "124");  
 User user3 = new User("1003", "shuyun3", "125");  
 List<User> list = new ArrayList<>();  
 list.add(user1);  
 list.add(user2);  
 list.add(user3);  
 model.addAttribute("list",list);  
 return "userlist";  
}

Userlist.html页面

<body>  
<table border="1" cellpadding="6" style="text-align:center" id="usertable">  
 <tr>  
 <th colspan="5">用户信息</th>  
 </tr>  
 <tr>  
 <th>id</th>  
 <th>name</th>  
 <th>password</th>  
 </tr>  
  
 *<!-- 遍历list，从list中取出user对象-->* <tr th:each="user:${list}">  
 <td th:text="${user.id}"></td>  
 <td th:text ="${user.name}"></td>  
 <td th:text="${user.getPassword()}"></td>  
 </tr>  
</table>  
</body>

**注意：**

**1.这里不能使用\*{属性名}来获取实体类的属性**

**2.不要用value，要用text**

# SpringBoot中全局异常处理

前端或者其他服务请求本服务的接口时，该接口需要返回对应的json数据，一般该服务只需要返回请求者需要的参数即可，但是在实际项目中，我们需要封装更多的信息，比如状态码code、相关信息msg等等等。

一方面是在项目中可以有个统一的返回结构，整个项目组都适用，另一方面是方便结合全局异常处理信息，因为异常处理信息中一般我们需要把状态码和异常内容反馈给调用方。

## 定义返回的统一json结构

1.导入fastjson依赖

*<!--fastjson依赖-->*<dependency>  
 <groupId>com.alibaba</groupId>  
 <artifactId>fastjson</artifactId>  
 <version>1.2.35</version>  
</dependency>

2.统一返回数据格式

public class JsonResult<T> {  
 private T data;  
 private String code;  
 private String msg;  
  
 */\*\*  
 \* 无数据返回  
 \*/* public JsonResult(){  
 this.code="0";  
 this.msg="操作成功";  
 }  
  
 */\*\*  
 \* 无数据返回，可以自定义状态码和提示信息  
 \* @param code  
 \* @param msg  
 \*/* public JsonResult(String code,String msg){  
 this.code=code;  
 this.msg=msg;  
 }  
  
  
 */\*\*  
 \* 有数据返回，默认状态码和提示信息  
 \* @param data  
 \*/* public JsonResult(T data){  
 this.data=data;  
 this.code="0";  
 this.msg="操作成功";  
 }  
  
 */\*\*  
 \* 有数据返回，默认状态码为0，人为指定提示信息  
 \* @param data  
 \* @param msg  
 \*/* public JsonResult(T data,String msg){  
 this.data=data;  
 this.code="0";  
 this.msg=msg;  
 }  
  
 public T getData() {  
 return data;  
 }  
  
 public void setData(T data) {  
 this.data = data;  
 }  
  
 public String getCode() {  
 return code;  
 }  
  
 public void setCode(String code) {  
 this.code = code;  
 }  
  
 public String getMsg() {  
 return msg;  
 }  
  
 public void setMsg(String msg) {  
 this.msg = msg;  
 }  
}

## 处理异常

新建一个GlobalExceptionHandler全局异常处理类，然后加上@ControllerAdvice注解，即可拦截项目中抛出的异常

@ControllerAdvice注解包含了@Component注解，说明在SpringBoot启动时也会把该类作为组件交给Spring来管理

此外，@ControllerAdvice注解还有个basePackages属性，该属性是用来拦截哪个包中的异常信息，一般不指定这个属性，我们拦截项目工程中的所有异常

使用方式为：创建一个方法，在该方法上通过@ExceptionHandler注解来指定具体的异常，然后在该方法中处理该异常信息，最后将结果通过统一json格式返回给调用者

### 举例1：空指针

空指针出现的场景

场景1：微服务中经常会调用其他服务获取数据，这个数据主要是json格式，但是在解析json的过程中有可能会出现空的情况，所以我们在获取某个jsonObject的相关信息时，应该要先做非空判断

场景2：从数据库查询数据时，不管是查询一条记录封装在某个对象中，还是查询多条记录封装在一个list中，我们接下来都要处理数据，那么就有可能出现空指针，因为谁也不保证从数据库中查询回来的东西一定不为空，所以在使用数据前要做非空判断

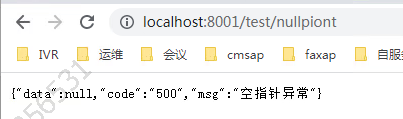
1.定义异常处理方法及返回json

@ControllerAdvice  
@ResponseBody  
public class GlobalExceptionHandler {  
  
 private static final Logger *log* = LoggerFactory.*getLogger*(GlobalExceptionHandler.class);  
  
 @ExceptionHandler(NullPointerException.class)  
 @ResponseStatus(value= HttpStatus.*INTERNAL\_SERVER\_ERROR*)  
 public JsonResult handleNullPionterException(NullPointerException ex){  
 *log*.error("空指针异常,{}",ex.getMessage());  
 return new JsonResult("500","空指针异常");  
 }  
  
  
}

2.控制器方法产生空指针

@RestController  
@RequestMapping("/test")  
public class ExceptionController {  
  
 @RequestMapping("/nullpiont")  
 public String nullPionterTest(){  
 User user =null;  
 user.getName();  
 return "success";  
 }  
}

3.请求和结果



### 举例2：参数缺失异常

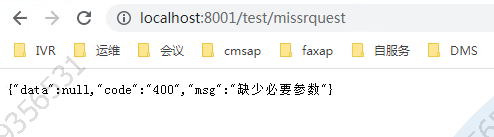
1.定义异常处理方法及返回json

@ControllerAdvice  
@ResponseBody  
public class GlobalExceptionHandler {  
 private static final Logger *log* = LoggerFactory.*getLogger*(GlobalExceptionHandler.class);  
 */\*\*  
 \* 空指针异常  
 \* @param ex  
 \* @return  
 \*/* @ExceptionHandler(NullPointerException.class)  
 @ResponseStatus(value= HttpStatus.*INTERNAL\_SERVER\_ERROR*)  
 public JsonResult handleNullPionterException(NullPointerException ex){  
 *log*.error("空指针异常,{}",ex.getMessage());  
 return new JsonResult("500","空指针异常");  
 }  
  
 @ExceptionHandler(MissingServletRequestParameterException.class)  
 public JsonResult handleHttpMessageNotReadableException(){  
  
 return new JsonResult("400","缺少必要参数");  
 }  
}

2.控制器方法

@RequestMapping("/missrquest")  
public String missRequestParameterTest(@RequestParam("id") String id,@RequestParam("name") String name){  
 *log*.info("id="+id+",name="+name);  
 return "success";  
}

3.请求url和结果



## 自定义异常拦截

在实际项目中，除了拦截一些系统异常外，在某些业务上，我们需要自定义一些业务异常。比如在微服务中，服务之间的相互调用很常见，要处理一个服务的调用时，可能会调用失败或超时等，此时我们需要一个自定义异常，当调用失败时抛出该异常，给GlobalExceptionHandler去捕获

### 业务异常枚举类

由于在业务中有很多异常，针对不同业务可能给出的提示信息不同，所以为了方便项目异常信息管理，我们一般会定义一个异常信息枚举类

public enum BusinessExceptionEnum {  
   
 *PARAMETER\_EXCEPTION*("102","参数异常"),  
 *SERVICE\_TIME\_OUT*("103","服务调用超时"),  
 *UNEXPECTED\_EXCEPTION*("500","系统发生异常，请联系管理员");  
   
 private String code;  
 private String msg;  
   
 private BusinessExceptionEnum(String code,String msg){  
 this.code=code;  
 this.msg=msg;  
 }  
  
 public String getCode() {  
 return code;  
 }  
  
 public void setCode(String code) {  
 this.code = code;  
 }  
  
 public String getMsg() {  
 return msg;  
 }  
  
 public void setMsg(String msg) {  
 this.msg = msg;  
 }  
}

### 自定义异常类

自定义一个异常类，继承Runtimeexception

public class BusinessErrorException extends RuntimeException {  
  
 private static final long *serialVersionUID* = -1685891363433592448L;  
  
 private String code;  
 private String msg;  
  
 public BusinessErrorException(){}  
  
 public BusinessErrorException(BusinessExceptionEnum bEnum) {  
 this.code = bEnum.getCode();  
 this.msg = bEnum.getMsg();  
 }  
  
 public String getCode() {  
 return code;  
 }  
  
 public void setCode(String code) {  
 this.code = code;  
 }  
  
 public String getMsg() {  
 return msg;  
 }  
  
 public void setMsg(String msg) {  
 this.msg = msg;  
 }  
}

### 拦截自定义异常

在GlobalExceptionHandler中增加自定义异常拦截方法

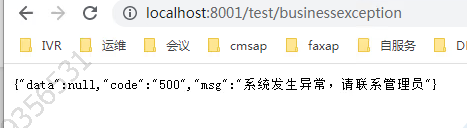
@ControllerAdvice  
@ResponseBody  
public class GlobalExceptionHandler {  
  
 private static final Logger *log* = LoggerFactory.*getLogger*(GlobalExceptionHandler.class);  
  
 */\*\*  
 \* 空指针异常  
 \* @param ex  
 \* @return  
 \*/* @ExceptionHandler(NullPointerException.class)  
 @ResponseStatus(value= HttpStatus.*INTERNAL\_SERVER\_ERROR*)  
 public JsonResult handleNullPionterException(NullPointerException ex){  
 *log*.error("空指针异常,{}",ex.getMessage());  
 return new JsonResult("500","空指针异常");  
 }  
  
  
 */\*\*  
 \* 参数缺失  
 \* @return  
 \*/* @ExceptionHandler(MissingServletRequestParameterException.class)  
 public JsonResult handleHttpMessageNotReadableException(){  
  
 return new JsonResult("400","缺少必要参数");  
 }  
  
 */\*\*  
 \* 拦截自定义业务异常  
 \* @param ex  
 \* @return  
 \*/* @ExceptionHandler(BusinessErrorException.class)  
 @ResponseStatus(value= HttpStatus.*INTERNAL\_SERVER\_ERROR*)  
 public JsonResult handleBusinessErrorException(BusinessErrorException ex){  
  
 return new JsonResult( ex.getCode(),ex.getMsg());  
 }  
  
}

### 控制器方法

在控制器方法中模拟抛出业务异常

@RequestMapping("/businessexception")  
public String businessExceptionTest(){  
 try {  
 int i =1 / 0;  
 }catch ( Exception ex){  
 throw new BusinessErrorException(BusinessExceptionEnum.*UNEXPECTED\_EXCEPTION*);  
 }  
 return "success";  
}

请求url及结果：



# SpringBoot中AOP处理

## AOP依赖

*<!--aop依赖-->*<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-aop</**artifactId**>  
</**dependency**>

## Aop常用注解

@Pointcut：定义一个切面，即关注的某件事入口，被增强的方法

@Before：前置通知，即在进入切面之前做的事

@After：最终通知，即在完成切面方法之后做的事，类似finally

@AfterReturning：后置通知，对切面方法返回值做增强处理

@AfterThrowing：在切面方法出现异常时，进行处理

## 切入点表达式

方式一：execution()

（1）切入点表达式作用：知道对哪个类里面的哪个方法进行增强

（2）语法结构： execution( [ 权限修饰符 ] [ 返回类型 ] [ 类全路径 ] [ 方法名称 ]( [ 参数列表 ] ) )

举例 1：对 com.atguigu.dao.BookDao 类里面的 add 进行增强。

\*号表示四种权限修饰符都可以。add(..)：两个点表示方法中的参数

execution(\* com.atguigu.dao.BookDao.add(..))

举例 2：对 com.atguigu.dao.BookDao 类里面的所有的方法进行增强。第二个\*表示类中的所有方法

execution(\* com.atguigu.dao.BookDao.\* (..))

举例 3：对 com.atguigu.dao 包里面所有类，类里面所有方法进行增强

execution(\* com.atguigu.dao.\*.\* (..))

方式二：annotation()

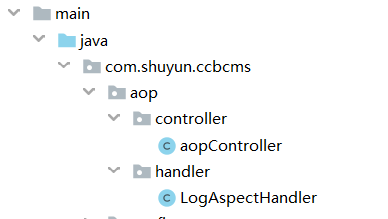
annotation()方式是针对某个注解来定义切面，比如@GetMapping

## 注解使用

对aop.controller包下所有类的所有方法进行增强

请求url：http://localhost:8001/test/aop/shuyun

目录结构：



1. 增强代理类

*/\*\*  
 \** ***@author*** *shuyun  
 \** ***@date*** *2024-06-10 9:49  
 \*/*@Aspect *//代理类对象*@Component *//交给spring管理***public class** LogAspectHandler {  
 **private final static** Logger ***log*** = LoggerFactory.*getLogger*(LogAspectHandler.**class**);  
  
 *//对aop.controller包下所有类的所有方法进行增强* @Pointcut(**"execution(\* com.shuyun.ccbcms.aop.controller.\*.\*(..))"**)  
 **public void** pointCut(){}  
  
 *//pointCut() 要增强的方法名，通过方法名来完成切入点表达式的注入* @Before(**"pointCut()"**)  
 **public void** doBefore(JoinPoint joinPoint){  
 ***log***.info(**"====开始进入doBefore方法"**);  
  
 *//获取签名* Signature signature = joinPoint.getSignature();  
 *//获取切入的包名* String declaringTypeName = signature.getDeclaringTypeName();  
 *//获取即将执行的方法名* String funcName = signature.getName();  
 ***log***.info(**"即将执行的方法为：{}，属于{}包"**,funcName,declaringTypeName);  
  
 *//获取url和ip* ServletRequestAttributes attributes = (ServletRequestAttributes) RequestContextHolder.*getRequestAttributes*();  
 HttpServletRequest request = attributes.getRequest();  
 String url = request.getRequestURL().toString();  
 String ip = request.getRemoteAddr();  
 ***log***.info(**"用户请求的url为：{}，ip地址为：{}"**,url,ip);  
 }  
  
 @After(**"pointCut()"**)  
 **public void** doAfter(JoinPoint joinPoint){  
 ***log***.info(**"====进入doAfter方法"**);  
 Signature signature = joinPoint.getSignature();  
 String methodName = signature.getName();  
 ***log***.info(**"方法{}已经执行完了"**,methodName);  
 }  
  
 *//returning属性的值必须和doAfterReturning方法的形参的名字一样，否则检测不到，此处为result* @AfterReturning(pointcut = **"pointCut()"**,returning = **"result"**)  
 **public void** doAfterReturning(JoinPoint joinPoint,Object result){  
 Signature signature = joinPoint.getSignature();  
 String methodName = signature.getName();  
 ***log***.info(**"方法{}已经执行，返回结果为：{}"**,methodName,result);  
 String enhandResult = (String) result + **", wellcome to aop"**;  
 ***log***.info(**"对返回参数进行增强:"**+enhandResult);  
  
 }

*//throwing属性的值必须和doAfterThrowing方法的形参的名字一样，否则检测不到，此处为ex* @AfterThrowing(pointcut = **"pointCut()"**,throwing = **"ex"**)  
 **public void** doAfterThrowing(JoinPoint joinPoint,Throwable ex){  
 Signature signature = joinPoint.getSignature();  
 String methodName = signature.getName();  
 ***log***.info(**"执行方法{}出现异常，异常为：{}"**,methodName,ex);  
 }  
}

1. 控制器方法

*/\*\*  
 \** ***@author*** *shuyun  
 \** ***@date*** *2024-06-10 9:58  
 \*/*@RequestMapping(**"/test"**)  
@RestController  
**public class** aopController {  
  
 @GetMapping(**"/aop/{name}"**)  
 **public** String aopAnnotationTest(@PathVariable String name){  
  
 **int** i = 1/0;  
 **return "Hello "**+name;  
 }  
}

注意，如果异常被cath掉了，不会进入界面异常方法了

# 集成MyBatis

## 依赖引入

### oracle

*<!--mybatis依赖-->*<dependency>  
 <groupId>org.mybatis.spring.boot</groupId>  
 <artifactId>mybatis-spring-boot-starter</artifactId>  
 <version>1.3.2</version>  
</dependency>  
  
*<!-- Oracle驱动 -->*<dependency>  
 <groupId>com.oracle</groupId>  
 <artifactId>ojdbc8</artifactId>  
 <version>12.2.0.1</version>  
</dependency>  
  
*<!-- 阿里数据库连接池 -->*<dependency>  
 <groupId>com.alibaba</groupId>  
 <artifactId>druid-spring-boot-starter</artifactId>  
 <version>${druid.version}</version>  
</dependency>

### mysql

## SpringBoot配置

要在启动类加上注解：

### oracle

spring:  
 thymeleaf:  
 cache: false *#关闭缓存  
  
 #数据源配置* datasource:  
 driver-class-name: oracle.jdbc.OracleDriver  
 type: com.alibaba.druid.pool.DruidDataSource  
 url: jdbc:oracle:thin:@//128.196.124.157:1521/ZXCFGCF  
 username: zxdb\_cc  
 password: db10$ZTE  
  
 druid:  
 *# 初始连接数* initialSize: 5  
 *# 最小连接池数量* minIdle: 10  
 *# 最大连接池数量* maxActive: 20  
 *# 配置获取连接等待超时的时间* maxWait: 60000  
 *# 配置连接超时时间* connectTimeout: 30000  
 *# 配置网络超时时间* socketTimeout: 60000  
 *# 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒* timeBetweenEvictionRunsMillis: 60000  
 *# 配置一个连接在池中最小生存的时间，单位是毫秒* minEvictableIdleTimeMillis: 300000  
 *# 配置一个连接在池中最大生存的时间，单位是毫秒* maxEvictableIdleTimeMillis: 900000  
 *# 配置检测连接是否有效* validationQuery: SELECT 1 FROM DUAL  
 testWhileIdle: true  
 testOnBorrow: false  
 testOnReturn: false  
 webStatFilter:  
 enabled: true  
 statViewServlet:  
 enabled: false  
 *# 设置白名单，不填则允许所有访问* allow:  
 url-pattern: /druid/\*  
 *# 控制台管理用户名和密码* login-username:  
 login-password:  
 filter:  
 stat:  
 enabled: true  
 *# 慢SQL记录* log-slow-sql: true  
 slow-sql-millis: 1000  
 merge-sql: true  
 wall:  
 config:  
 multi-statement-allow: true  
  
*#mybatis配置*mybatis:  
 *#mapper扫描，找到所有的mapper.xml文件* mapper-locations: classpath:mapper/\*.xml  
 *#实体类所在位置，指定别名* type-aliases-package: com.shuyun.ccbcms.vo  
 configuration:  
 *#控制台日志打印* log-impl: org.apache.ibatis.logging.stdout.StdOutImpl  
 *#驼峰映射开关* map-underscore-to-camel-case: true

### mysql

## 基于xml的整合

使用原始的xml方式，需要新建Mapper.xml文件。在application.yml配置文件中，已经定义了xml文件的路径：classpath:mapper/\*.xml，所以需要在resource目录下新建一个mapper文件夹，然后创建mapper.xml文件

classpath:mapper/\*.xml和classpath\*:mapper/\*\*/\*.xml的区别

classpath：表示类路径，即打包后.class文件的路径，即WEB-INFO/class/目录

classpath：只会在class/目录下查找

classpath\*：会在class/目录和jar文件下的class/目录下查找

1.实体类

com.shuyun.ccbcms.vo包下，新建实体类：operskill

public class operskill {  
  
 private String vcid;  
 private String operatoid;  
 private String skillgrpid;  
 private String id;

2.mapper接口

com.shuyun.ccbcms.mapper包下

新建接口：operskillMapper

接口方法：selectOperSkillByVcid

public interface OperskillMapper {  
  
 */\*\*  
 \* 查询所有操作员技能信息  
 \* @return  
 \*/* List<Operskill> selectOperSkillByVcid(Map<String,String> map);  
}

3.mapper.xml文件

在resource目录下新建文件夹mapper

在mapper文件夹下新建operskillMapper.xml文件（注意和接口名一致）

*<?*xml version="1.0" encoding="UTF-8" *?>*<!DOCTYPE mapper  
 PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd"*>  
<!--命名空间：绑定一个mapper接口，即实现该接口-->*<mapper namespace="com.shuyun.ccbcms.mapper.OperskillMapper">  
 <resultMap id="baseResultMap" type="operskill">  
 <id column="id" property="id"/>  
 <result column="vcid" property="vc"/>  
 <result column="operatoid" property="operatoid"/>  
 <result column="skillgrpid" property="skillgrpid"/>  
 </resultMap>  
  
 *<!--这里的id要和接口方法名一致-->* <select id="selectOperSkillByVcid" resultMap="baseResultMap" parameterType="map">  
 select \* from zxdb\_kf.cc\_operskill where vcid=#{vcid}  
 </select>  
  
</mapper>

4.service层代码

新建service包

接口：

com.shuyun.ccbcms.service.inf包下，新建接口：OperSkillService

接口方法：selectOperSkillByVcid

public interface OperSkillService {  
  
 */\*\*  
 \* 查询所有操作员技能信息  
 \* @return  
 \*/* List<Operskill> selectOperSkillByVcid(Map<String,String> map);  
  
}

实现类：

com.shuyun.ccbcms.service.impl包下新建实现类：OperSkillServiceImpl

重新方法：selectOperSkillByVcid

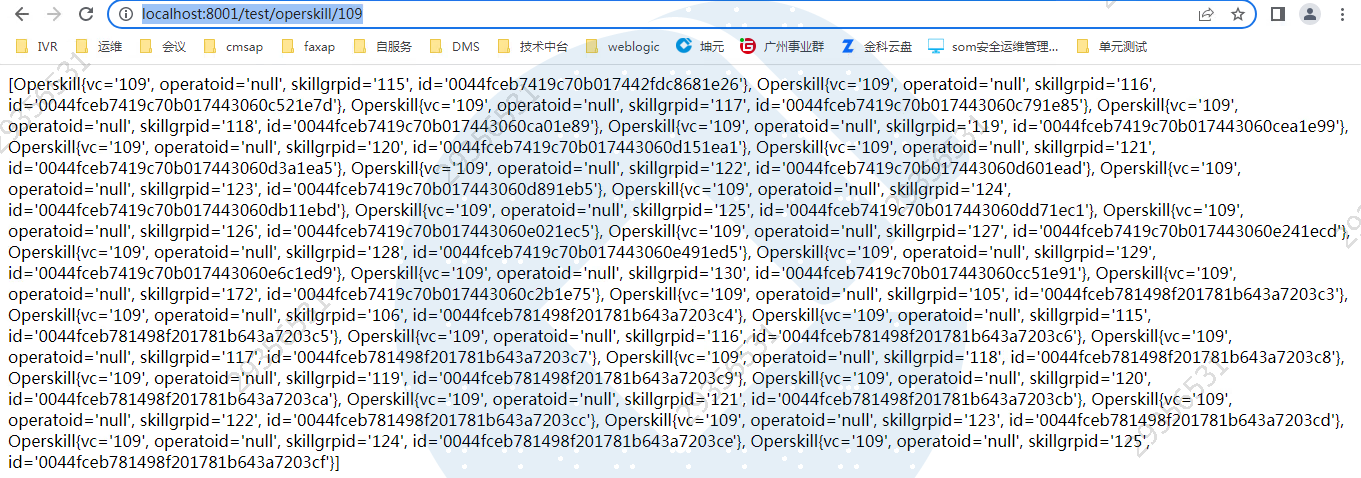
@Service("operSkillService")  
public class OperSkillServiceImpl implements OperSkillService {  
  
 @Resource  
 private OperskillMapper operskillMapper;  
  
 @Override  
 public List<Operskill> selectOperSkillByVcid(Map<String,String> map) {  
 return operskillMapper.selectOperSkillByVcid(map);  
 }  
}

5.控制器

@RequestMapping("/test")  
@RestController  
public class OperSkillController {  
  
 @Resource  
 private OperSkillService operSkillService;  
  
 @RequestMapping("/operskill/{vcid}")  
 public String operSkillList(@PathVariable("vcid") String vc){  
 Map<String,String> map = new HashMap<>();  
 map.put("vcid",vc);  
 List<Operskill> operskills = operSkillService.selectOperSkillByVcid(map);  
 return operskills.stream().limit(30).collect(Collectors.*toList*()).toString();  
 }  
}

6.请求和结果

<http://localhost:8001/test/operskill/109>



## 基于注解的整合

MyBatis提供了四个注解：@Select，@Insert，@Update，@Delete

在注解后面跟上对应的sql语句即可

当sql语句存在多个参数时，可以使用@Param注解指定参数名字，该名字要与sql中#{}的参数名一致

@Param注解用于给方法参数起一个名字。以下是总结的使用原则：

在方法只接受一个参数的情况下，可以不使用@Param。

在方法接受多个参数的情况下，建议一定要使用@Param注解给参数命名。

如果参数是 JavaBean ， 则不能使用@Param。

不使用@Param注解时，参数只能有一个，并且是Javabean

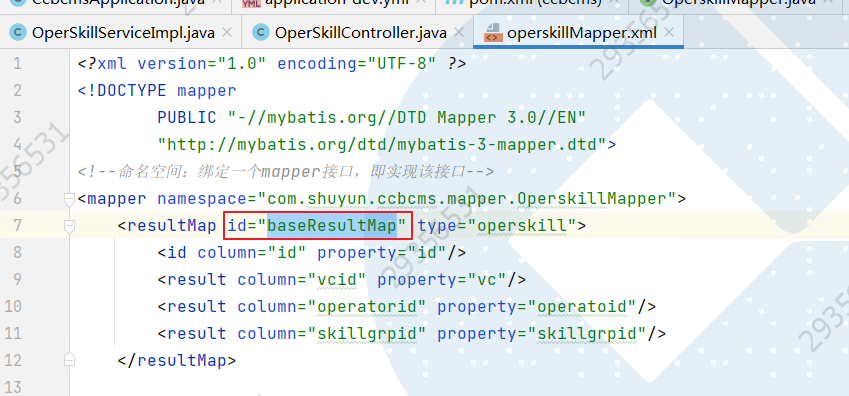
注意：官方不推荐使用注解代替xml配置文件，因为注解对于复杂的sql语句，处理起来很麻烦，比如：数据库字段与JavaBean属性名不一致时，查询不到数据。**使用注解和配置文件协同开发，才是MyBatis的最佳实践。**

在Mapper接口中新增查询方法，加上@Select注解

*/\*\*  
 \* 工号和Vc查询技能  
 \* @param vc  
 \* @param operid  
 \* @return  
 \*/*@Select("select \* from zxdb\_kf.cc\_operskill where vcid=#{vcid} and operatorid=#{oid}")  
*//字段映射，方式一*@Results({  
 @Result(property = "operatoid",column="operatorid"),  
 @Result(property = "vcid",column = "vcid")  
})  
*//字段映射，方式一二  
//@ResultMap("baseResultMap")*List<Operskill> selectOperSkillByVcidAndOper(@Param("vcid") String vc, @Param("oid")String operid);

数据库字段和JavaBean属性不匹配时，可以用@Result注解或者@ResultMap注解

@ResultMap注解引用的是Mapper.xml中的映射配置



# 事务配置管理

事务管理是SpringBoot框架中最为常用的功能之一。我们在实际应用开发时，基本上是在service层处理业务逻辑的时候加上事务

## 依赖引入

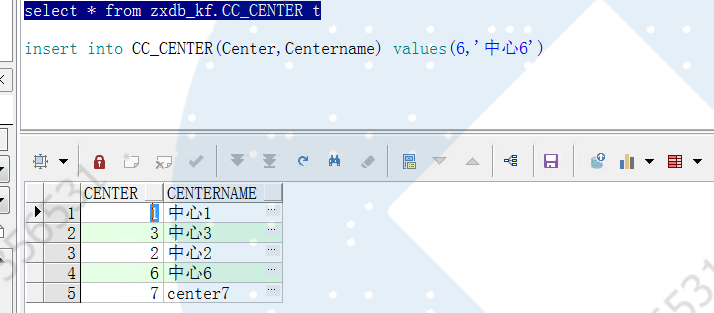
在SpringBoot中使用事务只需要导入mybatis依赖即可。因为之前集成mybatis已经引入依赖，这里不再引入

*<!--mybatis依赖-->*<**dependency**>  
 <**groupId**>org.mybatis.spring.boot</**groupId**>  
 <**artifactId**>mybatis-spring-boot-starter</**artifactId**>  
 <**version**>1.3.2</**version**>  
</**dependency**>

## 事务简单使用

导入mybatis依赖以后，SpringBoot会自动注入DataSourceTransactionManager，我们不需要任何其他配置就可以使用@Transactional注解进行事务的使用

1.数据库center表



2.JavaBean

**package** com.shuyun.ccbcms.vo;  
  
**import** lombok.NoArgsConstructor;  
  
@NoArgsConstructor *//无参构造***public class** Center {  
 **private** String **center**;  
 **private** String **centerName**;  
  
 **public** Center(String center,String centerName){  
 **this**.**center**=center;  
 **this**.**centerName**=centerName;  
 }  
  
}

3.Mapper接口

**package** com.shuyun.ccbcms.mapper;  
  
**import** com.shuyun.ccbcms.vo.Center;  
**import** org.apache.ibatis.annotations.Insert;  
  
*/\*\*  
 \** ***@author*** *shuyun  
 \** ***@create*** *2024-06-13-16:54  
 \*/***public interface** CenterMapper {  
 */\*\*  
 \* 插入一条数据  
 \** ***@param center*** *\** ***@return*** *\*/* @Insert(**"insert into CC\_CENTER(center,centername) values(#{center},#{centerName})"**)  
 Integer insertCenter(Center center);  
}

4.service接口

**package** com.shuyun.ccbcms.service.inf;  
  
**import** com.shuyun.ccbcms.vo.Center;  
  
**public interface** CenterService {  
 */\*\*  
 \* 插入一条数据  
 \** ***@param center*** *\** ***@return*** *\*/* Integer insertCenter(Center center);  
}

5.实现类

在此实现类中模拟异常，导致数据回滚，数据库中没有插入数据

**package** com.shuyun.ccbcms.service.impl;  
  
**import** com.shuyun.ccbcms.mapper.CenterMapper;  
**import** com.shuyun.ccbcms.service.inf.CenterService;  
**import** com.shuyun.ccbcms.vo.Center;  
**import** org.springframework.stereotype.Service;  
**import** org.springframework.transaction.annotation.Transactional;  
  
**import** javax.annotation.Resource;  
  
@Service(**"centerService"**)  
**public class** CenterServiceImpl **implements** CenterService {  
  
 @Resource  
 **private** CenterMapper **centerMapper**;  
  
 @Override  
 @Transactional  
 **public** Integer insertCenter(Center center) {  
 Integer integer = **centerMapper**.insertCenter(center);

*//模拟异常***int** i = 1/0;

**return** integer;  
 }  
}

6.控制器

**package** com.shuyun.ccbcms.controller;  
  
**import** com.shuyun.ccbcms.service.inf.CenterService;  
**import** com.shuyun.ccbcms.vo.Center;  
**import** org.springframework.web.bind.annotation.PathVariable;  
**import** org.springframework.web.bind.annotation.RequestMapping;  
**import** org.springframework.web.bind.annotation.RestController;  
  
**import** javax.annotation.Resource;  
  
@RequestMapping(**"/test"**)  
@RestController  
**public class** CenterController {  
 @Resource  
 **private** CenterService **centerService**;  
  
 @RequestMapping(**"/transation/{centerid}/{centername}"**)  
 **public** String addCenterById(@PathVariable(**"centerid"**) String id,@PathVariable(**"centername"**) String name){  
 Center center = **new** Center(id,name);  
 Integer integer = **centerService**.insertCenter(center);  
 **return "suceess"**;  
 }  
  
}

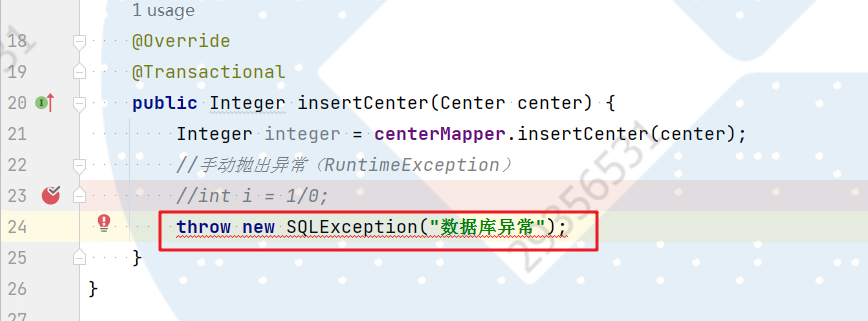
7.小结

在service层中增加@Transational注解，在调用完插入sql后模拟异常，导致数据回滚，数据库中没有插入数据

## 异常未被捕获

@Transational注解默认“捕获”的异常为RuntimeExcetion（运行时异常）和Error（程序错误）。如果程序抛出的是其他异常（如SQLException），而事务未捕获该异常，程序报错但是数据没有回滚，导致数据插入到库中。

---现在注解限制了，无法处理的异常，编译时会报错



## 异常被catch掉

异常处理有两种方式，一种是往上层抛出，由调用者解决；另一种是使用try-catch捕获异常。

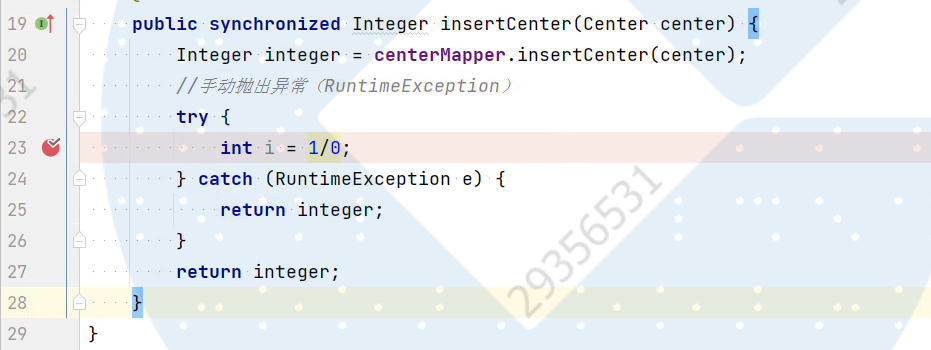
当使用try-catch处理异常时，因为异常已经被处理了，事务没有捕获到异常，事务无法回滚。

代码如下，数据没有回滚，能正常入库

@Override  
 @Transactional  
 **public** Integer insertCenter(Center center) {  
 Integer integer = **centerMapper**.insertCenter(center);  
 *//手动抛出异常（RuntimeException）* **try** {  
 **int** i = 1/0;  
 } **catch** (RuntimeException e) {  
 **return** integer;  
 }  
 **return** integer;  
 }  
}

## 事务范围大于锁范围

因为要考虑并发问题，使用同步方法时，事务的范围大于锁的范围，导致锁失效



上面代码中，使用synchronized同步方法，正常情况是先判断该用户是否存在，不存在插入，存在则更新，一个用户只有一条记录。

但是在进行压测时，因为并发量很高，在A线程执行完插入动作释放锁而事务还未结束时，B线程也进入该方法拿到了锁。此时数据库的状态和第一个线程进来的时候一样，即由于mysql Innodb引擎默认的隔离级别是可重复读（在同一个事务里，select的结果是事务开始时间点的状态）

线程二开始的时候，线程一还未提交完成，导致读取的数据还未更新。第二个线程也做了插入动作，导致插入了脏数据

根本原因是事务的范围比锁的范围大。解决方案有两个，一种是去掉事务（不推荐），另一种是在调用该方法的地方再加锁，保证锁的范围比事务的范围大即可

# 监听器的使用

web监听器是一种Servlet中特殊的类，它们能帮助开发者监听web中的特定事件，比如ServletContext,HttpSession,ServletRequest的创建和销毁；变量的创建、修改、销毁等。可以在某些动作前后增加处理，实现监控。

## 监听Servlet上下文对象

监听Servlet上下文对象可以用来初始化数据，用于缓存。比如用户在点击某个站点的首页时，一般会展示出首页的一些信息，而这些信息基本上是保持不变的，但是这些信息来自于数据库。如果用户的每次点击都要从数据库中获取数据，在用户量很大的时候，数据库的开销很大。

针对这种首页数据，大部分不常更新的话，我们完全可以缓存起来，每次用户点击的时候直接从缓存中获取，这样既可以提高首页的访问速度，又可以降低服务器的压力。如果做的更灵活一点，可以再加个定时器，定期来更新这个首页缓存。

步骤：

1.自定义一个监听器，实现 ApplicationListener 接口， ContextRefreshedEvent 作为参数传进去，重写 onApplicationEvent 方法

2.在该方法中，用 ContextRefreshedEvent 获取 application上下文信息，再通过 application 来获取 Service的Bean，获取到相应的数据，存到 application 域中

3.前端从 application 域来获取存入的数据，因为application域数据是缓存在内存中，可以减少数据库压力

代码如下：

1.service层代码复用之前的查询数据库操作



2.自定义监控器

在项目启动时，初始化数据到application域中

**package** com.shuyun.ccbcms.listener;  
**import** com.shuyun.ccbcms.service.inf.OperSkillService;  
**import** com.shuyun.ccbcms.vo.Operskill;  
**import** org.springframework.context.ApplicationContext;  
**import** org.springframework.context.ApplicationListener;  
**import** org.springframework.context.event.ContextRefreshedEvent;  
**import** org.springframework.stereotype.Component;  
**import** javax.servlet.ServletContext;  
**import** java.util.List;  
*/\*\*  
 \** ***@author*** *shuyun  
 \** ***@create*** *2024-06-18 14:56  
 \*/*@Component  
**public class** MyServletContextListener **implements** ApplicationListener<ContextRefreshedEvent> {  
 */\*\*  
 \* 应用启动初始化时进行  
 \** ***@param event*** *\*/* @Override  
 **public void** onApplicationEvent(ContextRefreshedEvent event) {  
 *//获取上下文信息* ApplicationContext context = event.getApplicationContext();  
 *//获取到对应的service* OperSkillService service = context.getBean(OperSkillService.**class**);  
 *//获取信息* List<Operskill> operskills = service.selectOperSkillByVcidAndOper(**"202"**, **"202777"**);  
 *//获取application域对象，将查到的信息放到application域中* ServletContext application = context.getBean(ServletContext.**class**);  
 application.setAttribute(**"operskillList"**,operskills);  
 }  
}

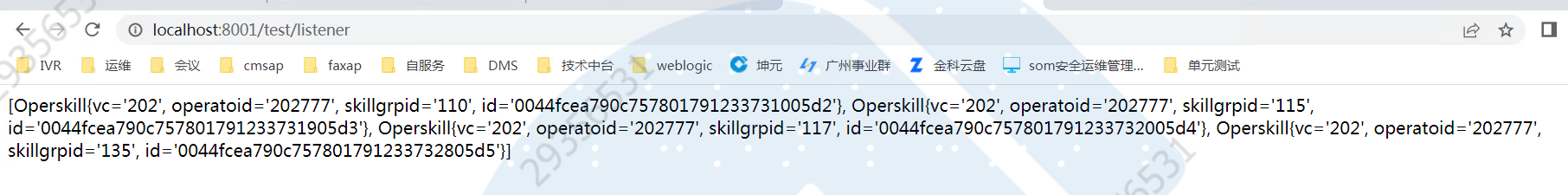
3.控制器

控制器中获取application域中的数据

**package** com.shuyun.ccbcms.controller;  
  
**import** com.shuyun.ccbcms.vo.Operskill;  
**import** org.springframework.web.bind.annotation.RequestMapping;  
**import** org.springframework.web.bind.annotation.RestController;  
  
**import** javax.servlet.ServletContext;  
**import** javax.servlet.http.HttpServletRequest;  
**import** java.util.List;  
  
*/\*\*  
 \** ***@author*** *shuyun  
 \** ***@create*** *2024-06-18 15:11  
 \*/*@RequestMapping(**"/test"**)  
@RestController  
**public class** MyListenerController {  
  
 @RequestMapping(**"/listener"**)  
 **public** String getOperskillList(HttpServletRequest request){  
 ServletContext application = request.getServletContext();  
 List<Operskill> resultList = (List<Operskill>) application.getAttribute(**"operskillList"**);  
 **return** resultList.toString();  
 }  
}

4.请求url及结果

http://localhost:8001/test/listener



## 监听HTTP会话Session对象

监听器还有一个比较常用的地方就是用来监听session对象，来获取在线用户数量。

实现步骤：

1.创建类 MyHttpSessionListener 实现 HttpSessionListener 接口

2.重写sessionCreated和sessionDestroyed方法，将业务数据放到session域中

3.控制器从session域获取到数据返回前端

1.自定义监控器代码：

**package** com.shuyun.ccbcms.listener;  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
**import** org.springframework.stereotype.Component;  
**import** javax.servlet.http.HttpSessionEvent;  
**import** javax.servlet.http.HttpSessionListener;  
  
*/\*\*  
 \* 使用HttpSessionListener统计在线用户数量  
 \** ***@author*** *shuyun  
 \** ***@create*** *2024-06-19 8:53  
 \*/*@Component  
**public class** MyHttpSessionListener **implements** HttpSessionListener {  
 **private static final** Logger ***log*** = LoggerFactory.*getLogger*(MyHttpSessionListener.**class**);  
 */\*\*  
 \* 记录在线的用户数量  
 \*/* **public** Integer **count** = 0;  
 @Override  
 **public synchronized void** sessionCreated(HttpSessionEvent event){  
 ***log***.info(**"新用户上线了"**);  
 **count**++;  
 event.getSession().getServletContext().setAttribute(**"count"**,**count**);  
 }  
 @Override  
 **public synchronized void** sessionDestroyed(HttpSessionEvent event) {  
 ***log***.info(**"用户下线了"**);  
 **count**--;  
 event.getSession().getServletContext().setAttribute(**"count"**,**count**);  
 }  
}

2.控制器获取数据代码

*/\*\*  
 \* 获取当前在线人数（该方法有bug）  
 \** ***@param request*** *\** ***@return*** *\*/*@RequestMapping(**"/online"**)  
**public** String getOnlineNumber(HttpServletRequest request){  
 Integer count = (Integer) request.getSession().getServletContext().getAttribute(**"count"**);  
 **return "当前在线人数为："**+count;  
}

3.请求url和结果

http://localhost:8001/test/online

打开谷歌浏览器，输入url可看到返回结果为1。再打开IE浏览器输入url，可看到返回结果为2

4.存在问题

关闭一个浏览器再打开，理论上在线人数为2，实际却是3

原因是关闭浏览器的时候session销毁的方法没有执行（可观察上线和下线的打印日志）。当重新打开浏览器时，服务器找不到用户原来的session，于是又创建了一个session。解决办法如下：

@RequestMapping(**"/totalOnline"**)  
**public** String getTotalOnline(HttpServletRequest req, HttpServletResponse resp){  
 Cookie cookie;  
 **try** {  
 *//将sessionid记录到浏览器中* cookie = **new** Cookie(**"JSESSIONID"**, URLEncoder.*encode*(req.getSession().getId(),**"utf-8"**));  
 cookie.setPath(**"/"**);  
 *//cookie有效时间设置为2天* cookie.setMaxAge(48\*60\*60);  
 resp.addCookie(cookie);  
 } **catch** (UnsupportedEncodingException e) {  
 **throw new** RuntimeException(e);  
 }  
 Integer count = (Integer) req.getSession().getServletContext().getAttribute(**"count"**);  
 **return "当前在线人数为："**+count;  
}

## 监听客户端请求Servlet Request对象

使用监听器获取用户的访问信息比较简单，实现接口即可，然后通过request对象获取请求信息

实现步骤：

1.创建类 MyServletRequestListener 实现 ServletRequestListener 接口，重写该接口的两个方法，将数据存入request域中

2.控制器获取request域中的数据

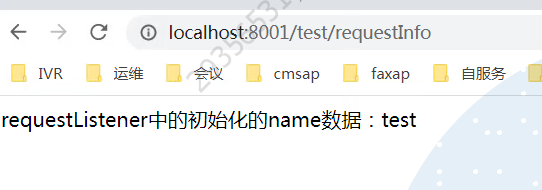
1.监视器代码：

**package** com.shuyun.ccbcms.listener;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
**import** org.springframework.stereotype.Component;  
  
**import** javax.servlet.ServletRequestEvent;  
**import** javax.servlet.ServletRequestListener;  
**import** javax.servlet.http.HttpServletRequest;  
  
*/\*\*  
 \** ***@author*** *shuyun  
 \** ***@create*** *2024-06-19 14:08  
 \*/*@Component  
**public class** MyServletRequestListener **implements** ServletRequestListener {  
 **private static final** Logger ***log*** = LoggerFactory.*getLogger*(MyHttpSessionListener.**class**);  
  
 @Override  
 **public void** requestDestroyed(ServletRequestEvent event) {  
 ***log***.info(**"request end"**);  
 HttpServletRequest request = (HttpServletRequest) event.getServletRequest();  
 ***log***.info(**"request域中保存的值为：{}"**,request.getAttribute(**"name"**));  
 }  
  
 @Override  
 **public void** requestInitialized(ServletRequestEvent event) {  
  
 HttpServletRequest request = (HttpServletRequest) event.getServletRequest();  
 ***log***.info(**"session id为：{}"**,request.getRequestedSessionId());  
 ***log***.info(**"request url为：{}"**,request.getRequestURL());  
 request.setAttribute(**"name"**,**"test"**);  
  
 }  
  
}

2.控制器方法

@RequestMapping(**"/requestInfo"**)  
**public** String getRequestInfo(HttpServletRequest request){  
 String attribute = (String) request.getAttribute(**"name"**);  
 **return "requestListener中的初始化的name数据："**+attribute;  
}

3．请求url和结果



## 自定义事件监听

在实际项目中，我们往往需要自定义一些事件和监听器来满足业务场景，比如在微服务中会有这样的场景：微服务A在处理完某个逻辑之后，需要通知微服务B去处理另一个逻辑；或者微服务A处理完某个逻辑之后，需要将数据同步到微服务B。这种场景非常普遍，这个时候我们可以自定义事件以及监听器来监听，一旦微服务A中的某个事件发生，就去通知微服务B处理对应的逻辑。

步骤：

1.自定义事件，继承…类

2.自定义监听器，实现…接口，监听自定义事件，

3.手动发布事件，这样监听器才能监听到

代码：

1.自定义监听事件MyEvent，继承ApplicationEvent

**package** com.shuyun.ccbcms.event;  
  
**import** com.shuyun.ccbcms.vo.User;  
**import** org.springframework.context.ApplicationEvent;  
  
*/\*\*  
 \** ***@author*** *shuyun  
 \** ***@create*** *2024-06-22 22:12  
 \*/***public class** MyEvent **extends** ApplicationEvent {  
  
 **private** User **user**;  
 **public** MyEvent(Object source,User user) {  
 **super**(source);  
 **this**.**user**=user;  
 }  
  
 **public** User getUser() {  
 **return user**;  
 }  
  
 **public void** setUser(User user) {  
 **this**.**user** = user;  
 }  
}

2.自定义监听器MyEventListener，实现ApplicationListener<MyEvent>，并将自定义的监听事件传进去

**package** com.shuyun.ccbcms.listener;  
  
**import** com.shuyun.ccbcms.event.MyEvent;  
**import** com.shuyun.ccbcms.vo.User;  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
**import** org.springframework.context.ApplicationListener;  
**import** org.springframework.stereotype.Component;  
  
*/\*\*  
 \** ***@author*** *shuyun  
 \** ***@create*** *2024-06-22 22:17  
 \*/*@Component  
**public class** MyEventListener **implements** ApplicationListener<MyEvent> {  
 **private static final** Logger ***log*** = LoggerFactory.*getLogger*(MyEventListener.**class**);  
 @Override  
 **public void** onApplicationEvent(MyEvent event) {  
 *//获取事件中的信息* User user = event.getUser();  
 *//处理时间，实际项目中可以通知别的微服务或者处理其他逻辑* ***log***.info(**"监听器收到的用户名：{}"**+**",密码：{}"**,user.getName(),user.getPassword());  
 }  
}

3.模拟业务处理逻辑，然后发布事件

需要注入ApplicationContext，通过其发布事件

**package** com.shuyun.ccbcms.service.impl;  
  
**import** com.shuyun.ccbcms.event.MyEvent;  
**import** com.shuyun.ccbcms.service.inf.UserService;  
**import** com.shuyun.ccbcms.vo.User;  
**import** org.springframework.context.ApplicationContext;  
**import** org.springframework.stereotype.Service;  
  
**import** javax.annotation.Resource;  
  
*/\*\*  
 \** ***@author*** *shuyun  
 \** ***@create*** *2024-06-22 22:25  
 \*/*@Service(**"userService"**)  
**public class** UserServiceImpl **implements** UserService {  
 @Resource  
 **private** ApplicationContext **context**;  
  
 @Override  
 **public** User publishUser(){  
 User user = **new** User(**"1001"**, **"shuyun"**, **"123456"**);  
 *//发布事件* MyEvent event = **new** MyEvent(**this**, user);  
 **context**.publishEvent(event);  
 **return** user;  
 }  
  
}

4.控制器模拟请求，进入业务处理逻辑，并处罚事件发布，然后监听器监听到事件后打印日志

@Resource  
**private** UserService **userService**;  
  
@RequestMapping(**"/MyEvent"**)  
**public** String listenMyEvent(){  
 User user = **userService**.publishUser();  
 **return "用户名："**+user.getName()+**",密码："**+user.getPassword();  
}

# 拦截器的使用

拦截器的原理很简单，是AOP的一种实现，专门拦截对动态资源的后台请求，即拦截对控制层的请求

使用场景比较多的是判断用户是否有权限请求后台，还可以结合websocket一起使用，来拦截websocket请求，然后做相应的处理。

拦截器不会拦截静态资源，SpringBoot默认的静态目录为：resource/static，该目录下的静态页面、js、css、图片等等，不会被拦截（具体还要分析）

**使用拦截器分两步：定义拦截器、配置拦截器**

## 自定义拦截器

1.自定义拦截器MyInterceptor，实现接口HandlerInterceptor（该接口是SpringBoot提供的拦截器的鼻祖）

2.实现HandlerInterceptor接口的三个方法

①preHandle：该方法的执行时机是，当某个url已经匹配到对应的Controller中的某个方法，且在该方法执行之前。所以preHandle方法可以决定是否将请求放行，这是通过该方法的返回值决定的。返回true则放行，返回false则被拦截

②postHandle：该方法的执行时机是，当某个url已经匹配到Controller中的某个方法，且在该方法执行完之后，但是在DispatcherServlet视图渲染之前。所以这个方法中有个ModelAndView参数，可以在视图渲染之前做一些修改动作。

③afterCompletion：该方法是在整个请求处理完成后（包括视图渲染）执行，这时做一些资源清理的工作。这个方法只有在preHandle方法被成功执行后且返回true时才会被执行。返回false时，该请求会被拦截，但是不会报错，只是没返回。

自定义拦截器的代码如下：

**package** com.shuyun.ccbcms.interceptor;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
**import** org.springframework.web.method.HandlerMethod;  
**import** org.springframework.web.servlet.HandlerInterceptor;  
**import** org.springframework.web.servlet.ModelAndView;  
  
**import** javax.servlet.http.HttpServletRequest;  
**import** javax.servlet.http.HttpServletResponse;  
**import** java.lang.reflect.Method;  
  
*/\*\*  
 \** ***@author*** *shuyun  
 \** ***@create*** *2024-06-24 15:34  
 \*/***public class** MyInterceptor **implements** HandlerInterceptor {  
  
 **private static final** Logger ***log*** = LoggerFactory.*getLogger*(MyInterceptor.**class**);  
  
 @Override  
 **public boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) **throws** Exception {  
 HandlerMethod handlerMethod = (HandlerMethod)handler;  
 Method method = handlerMethod.getMethod();  
 String methodName = method.getName();  
 ***log***.info(**"====拦截到了方法：{}，在该方法执行之前===="**,methodName);  
  
 *//返回true才会继续执行被拦截的方法，返回false则取消当前请求* **return false**;  
 }  
  
 @Override  
 **public void** postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) **throws** Exception {  
  
 ***log***.info(**"====执行完Controller方法之后，视图渲染之前====="**);  
  
 }  
  
 @Override  
 **public void** afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) **throws** Exception {  
 ***log***.info(**"=====视图已完成渲染========"**);  
 }  
}

## 拦截器配置

在SpringBoot2.0之前，我们都是直接继承WebMvcConfigurerAdapter类，然后重写addInterceptors方法来实现拦截器的配置。

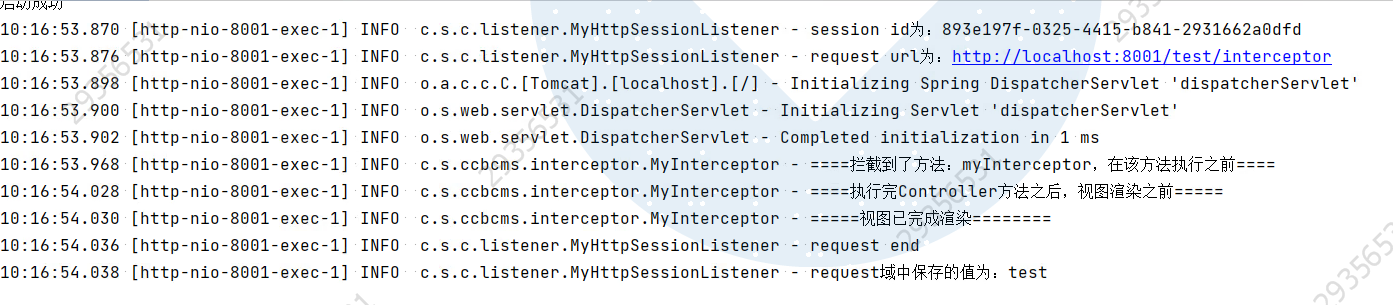
但是在SpringBoot2.0之后，该方法已经被废弃了（虽然也可继续使用），取而代之的是WebMvcConfigurationSupport方法

在该配置方法中重写addInterceptors方法，将自定义的拦截器添加进去。addPathPatterns方法是添加要拦截的请求，这里我们拦截所有请求。这样就配置好拦截器了

拦截器配置的代码如下：

**package** com.shuyun.ccbcms.config;  
  
**import** com.shuyun.ccbcms.interceptor.MyInterceptor;  
**import** org.springframework.context.annotation.Configuration;  
**import** org.springframework.web.servlet.config.annotation.InterceptorRegistry;  
**import** org.springframework.web.servlet.config.annotation.WebMvcConfigurationSupport;  
  
*/\*\*  
 \** ***@author*** *shuyun  
 \** ***@create*** *2024-06-24 16:39  
 \*/*@Configuration  
**public class** MyInterceptorConfig **extends** WebMvcConfigurationSupport {  
  
 @Override  
 **protected void** addInterceptors(InterceptorRegistry registry) {  
 registry.addInterceptor(**new** MyInterceptor()).addPathPatterns(**"/\*\*"**);  
 **super**.addInterceptors(registry);  
 }  
}

使用简单的控制器方法测试下即可，测试结果如下：



## 解决静态资源被拦截

前面的拦截器配置存在一个缺陷，那就是静态资源被拦截了。可以在/resource/static/目录下放置一些图片资源或者html文件，启动项目后发现无法访问

这是因为继承的WebMvcConfigurationSupport类会导致默认的静态资源被拦截，这就需要我们手动将静态资源放开。

做法是在拦截器的配置类 MyInterceptorConfig 中重写addInterceptors方法外，还需要重写一个方法：addresourceHandlers，将静态资源放开

**package** com.shuyun.ccbcms.config;  
  
**import** com.shuyun.ccbcms.interceptor.MyInterceptor;  
**import** org.springframework.context.annotation.Configuration;  
**import** org.springframework.web.servlet.config.annotation.InterceptorRegistry;  
**import** org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;  
**import** org.springframework.web.servlet.config.annotation.WebMvcConfigurationSupport;  
  
*/\*\*  
 \** ***@author*** *shuyun  
 \** ***@create*** *2024-06-24 16:39  
 \*/*@Configuration  
**public class** MyInterceptorConfig **extends** WebMvcConfigurationSupport {  
  
 @Override  
 **protected void** addInterceptors(InterceptorRegistry registry) {  
 registry.addInterceptor(**new** MyInterceptor()).addPathPatterns(**"/\*\*"**);  
 **super**.addInterceptors(registry);  
 }  
  
 @Override  
 **protected void** addResourceHandlers(ResourceHandlerRegistry registry) {  
 registry.addResourceHandler(**"/\*\*"**).addResourceLocations(**"classpath:/static/"**);  
 **super**.addResourceHandlers(registry);  
 }  
}

另外一种方法是：不继承WebMvcConfigurationSupport类，直接实现WebMvcConfigurer，重写addInterceptors方法，把自定义的拦截器添加进去即可

**推荐使用这种方式**

**package** com.shuyun.ccbcms.config;  
  
**import** com.shuyun.ccbcms.interceptor.MyInterceptor;  
**import** org.springframework.context.annotation.Configuration;  
**import** org.springframework.web.servlet.config.annotation.InterceptorRegistry;  
**import** org.springframework.web.servlet.config.annotation.WebMvcConfigurer;  
  
*/\*\*  
 \** ***@author*** *shuyun  
 \** ***@create*** *2024-06-24 16:39  
 \*/*@Configuration  
**public class** MyInterceptorConfig **implements** WebMvcConfigurer {  
  
 @Override  
 **public void** addInterceptors(InterceptorRegistry registry) {  
 registry.addInterceptor(**new** MyInterceptor()).addPathPatterns(**"/\*\*"**);  
 }  
}

两种方式的区别：

1.继承WebMvcConfigurationSupport类，用于前后端分离的项目，后台不需要访问静态资源

2.实现WebMvcConfigurer接口方式，可以用在非前后端的项目中，因为需要读取一些图片、css、js文件等等

## 拦截器使用实例

### 判断用户是否登录

一般用户登录功能 可以这么做，要么往session中写一个user，要么针对每个user生成一个token，第二种要更好点。

针对第二种方式，如果用户登录成功，每次请求的时候都会带上该用户的token；如果未登录，则没有该token。服务端通过检测该token参数的有无来判断用户有没有登录，从而实现拦截功能。

改造下preHandle方法：

@Override  
**public boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) **throws** Exception {  
 HandlerMethod handlerMethod = (HandlerMethod)handler;  
 Method method = handlerMethod.getMethod();  
 String methodName = method.getName();  
 ***log***.info(**"====拦截到了方法：{}，在该方法执行之前===="**,methodName);  
  
 *//判断用户有没有登录，一般登录之后的用户都会有一个对应的token* String token = request.getParameter(**"token"**);  
 **if** (StringUtils.*isNotBlank*(token)){  
 ***log***.info(**"用户未登录，没有权限执行。请登录后操作"**);  
 **return false**;  
 }  
 *//返回true才会继续执行被拦截的方法，返回false则取消当前请求* **return true**;  
}

被拦截：

http://localhost:8001/test/interceptor

可正常执行：

http://localhost:8001/test/interceptor?token=123

### 取消拦截操作

根据上文，如果我要拦截所有的/admin开头的url请求的话，需要在拦截器配置添加这个前缀，但是实际项目中，可能会出现这种场景：某个请求也是/admin开头的，但是不能拦截，比如：/admin/login等，这样的话又要去配置。

那么可不可以做成一个类似开关的东西，哪里不需要拦截，我就在哪里弄个开关上去，做成一个灵活的可插拔的效果

可以自定义一个注解，该注解专门用来取消拦截操作，如果某个Controller中的方法我们不需要拦截，即可在该方法上加上我们自定义的注解即可。

1.自定义注解

jdk 提供的4种元注解：  
Retention：指定所修饰的 Annotation 的生命周期：SOURCE\CLASS（默认行为\RUNTIME  
 只声明为RUNTIME生命周期的注解，才能通过反射获取。  
Target:用于指定被修饰的 Annotation 能用于修饰哪些程序元素  
\*\*\*\*\*\*\*出现的频率较低\*\*\*\*\*\*\*  
Documented:表示所修饰的注解在被javadoc解析时，保留下来。  
Inherited:被它修饰的 Annotation 将具继承性。

**package** com.shuyun.ccbcms.annotation;  
  
**import** java.lang.annotation.ElementType;  
**import** java.lang.annotation.Retention;  
**import** java.lang.annotation.RetentionPolicy;  
**import** java.lang.annotation.Target;  
  
*/\*\*  
 \* 该注解用于指定某个某个方法不被拦截  
 \** ***@author*** *shuyun  
 \** ***@create*** *2024-07-13 22:52  
 \*/*@Target(ElementType.***METHOD***)  
@Retention(RetentionPolicy.***RUNTIME***)  
**public** @**interface** UnInterception {  
}

2.修改拦截器拦截规则

通过方法获取注解，如果有UnInterception注解则不拦截

@Override  
 **public boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) **throws** Exception {  
 HandlerMethod handlerMethod = (HandlerMethod)handler;  
 Method method = handlerMethod.getMethod();  
 String methodName = method.getName();  
 ***log***.info(**"====拦截到了方法：{}，在该方法执行之前===="**,methodName);  
*//  
// //判断用户有没有登录，一般登录之后的用户都会有一个对应的token  
// String token = request.getParameter("token");  
// if (StringUtils.isBlank(token)){  
// log.info("用户未登录，没有权限执行。请登录后操作");  
// return false;  
// }  
 //获取该方法的注解，如果有UnInterception注解则不拦截* UnInterception annotation = method.getAnnotation(UnInterception.**class**);  
 **if** ( **null** != annotation){  
 **return true**;  
 }  
 *//返回true才会继续执行被拦截的方法，返回false则取消当前请求* **return true**;  
 }

# SpringBoot中集成Shiro

Shiro是一个强大、简单易用的Java安全框架，主要用来更便捷的认证、授权、加密、会话管理等。可以为任何应用提供安全保障

## Shiro三大核心组件

Shiro三大核心组件是：Subject、SecurityManager和Realm

1.Subject：认证主体。主要包含两个信息 Principals和Credentials。

Principals：身份，可以是用户名、邮件、手机号码等，用来标识一个登陆主体的身份信息

Credentials：凭证，常见有密码、数字证书等

Subject 认证主体简单说就是需要认证的东西，常见的是用户名和密码。比如用户在登陆时，需要去验证身份，就需要Subject认证主体

2.SecurityManager：安全管理员。这是Shiro架构的核心，它就像Shiro内部所有元件的保护伞一样。我们在项目中一般都会配置SecurityManager，开发人员大部分精力主要在Subject认证主体上面。我们在于Subject进行交互的时候，实际上是SecurityManager在背后做一些安全操作。

3.Realm：Realm是一个域，它是连接Shiro和具体应用的桥梁，当需要与安全数据交互的时候，比如用户账户、访问控制等，Shiro就会从一个或多个Realm中与查找。我们一般会自己定制Realm。

## Shiro身份和权限认证

一、身份认证：

1.应用程序在调用Subject.login(token)方法后，传入代表最终用户的身份和凭证的AuthenticationToken实例token

2.将Subject实例委托给应用程序的SecurityManager（Shiro的安全管理）来开始实际的认证工作，这里开始真正的认证工作了。

3/4/5：然后SecurityManager就会根据具体的realm去进行安全认证了。这里的realm可以自定义

二、权限认证：

权限认证，也就是访问控制，即在应用中控制谁能访问哪些资源。在权限认证中，最核心的三个要素是：权限、角色、用户

权限（permission）：即操作资源的权利，比如访问某个页面，以及对某个模块的数据的添加、修改、删除、查看的权利

角色（role）：指的是用户担任的角色，一个角色可以有多个权限

用户（user）：在Shiro中，代表访问系统的用户，即Subject认证主体。

一个用户可以有多个角色，而不同角色可以有不同权限，也可有相同权限。比如说现在有3个角色：1是普通角色，2也是普通角色，3是管理员。角色1只能查看信息，角色2只能添加信息，角色3两者都可以，而且还可以删除信息。

## SpringBoot集成Shiro

### 依赖引入

*<!--Shiro安全工具-->*<**dependency**>  
 <**groupId**>org.apache.shiro</**groupId**>  
 <**artifactId**>shiro-spring</**artifactId**>  
 <**version**>1.4.0</**version**>  
</**dependency**>

### 数据库表初始化

这里主要涉及三张表：用户表、角色表、权限表。

用户表，存储用户信息：

角色表，存储角色信息：

权限表，存储权限信息。这里的user:\*表示权限可以是user:create或者其他，\*表示一个占位符，我们可以自定义。

### 自定义Realm

自定义realm需要继承AuthorizingRealm类，因为该类封装了很多方法，它也进一步继承自Realm类的。继承AuthorizingRealm类后，需要重写两个方法：

AuthenticationInfo：用来验证当前登录的用户，获取认证信息

doGetAuthorizationInfo：用来为当前登录成功的用户授予权限和角色

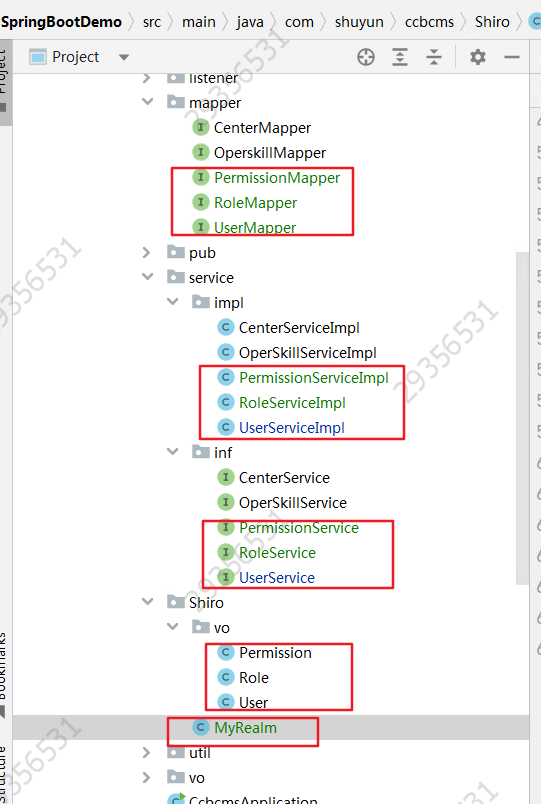
在身份验证的时候，根据用户输入的用户名，先从数据库中查出该用户名对应的用户，这时候并没有涉及到密码。也就是说到这一步的时候，即使用户输入的密码不对，也是可以查出来该用户的。然后将该用户的正确信息封装到authcInfo中返回给Shiro，接下来就是Shiro的事了。Shiro会根据authcInfo里的真实信息与用户前台输入的用户名和密码进行校验，这个时候也要校验密码了。如果校验通过就让用户登录，否则跳转到指定页面。

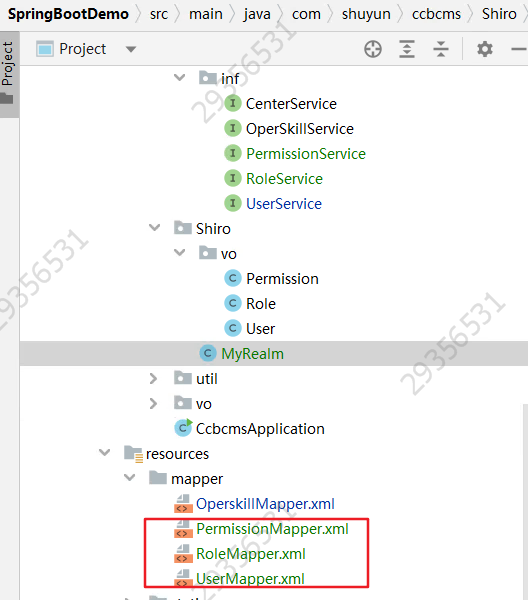
同理，权限验证的时候也是先根据用户名从数据库中获取与该用户名有关的角色和权限，然后封装到authorizationInfo中返回给Shiro。

1.Myrealm代码

**package** com.shuyun.ccbcms.Shiro;  
**import** com.shuyun.ccbcms.service.inf.PermissionService;  
**import** com.shuyun.ccbcms.service.inf.RoleService;  
**import** com.shuyun.ccbcms.service.inf.UserService;  
**import** com.shuyun.ccbcms.vo.User;  
**import** org.apache.shiro.SecurityUtils;  
**import** org.apache.shiro.authc.AuthenticationException;  
**import** org.apache.shiro.authc.AuthenticationInfo;  
**import** org.apache.shiro.authc.AuthenticationToken;  
**import** org.apache.shiro.authc.SimpleAuthenticationInfo;  
**import** org.apache.shiro.authz.AuthorizationInfo;  
**import** org.apache.shiro.authz.SimpleAuthorizationInfo;  
**import** org.apache.shiro.realm.AuthorizingRealm;  
**import** org.apache.shiro.subject.PrincipalCollection;  
  
**import** javax.annotation.Resource;  
**import** java.util.Objects;  
  
*/\*\*  
 \** ***@author*** *shuyun  
 \** ***@create*** *2024-07-05 23:42  
 \*/***public class** MyRealm **extends** AuthorizingRealm {  
  
 @Resource  
 **private** UserService **userService**;  
  
 @Resource  
 **private** PermissionService **permissionService**;  
 @Resource  
 **private** RoleService **roleService**;  
 @Override  
 **protected** AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principalCollection) {  
 *//为用户授予权限和角色  
 //获取用户名* String username = (String)principalCollection.getPrimaryPrincipal();  
 SimpleAuthorizationInfo authorizationInfo = **new** SimpleAuthorizationInfo();  
  
 *//给该用户设置角色，角色信息在TMP\_LOG\_ROLE表中取* authorizationInfo.setRoles(**roleService**.selectRoleByUsername(username));  
 *//给该用户设置权限信息，从TMP\_LOG\_PERMISSION* authorizationInfo.setStringPermissions(**permissionService**.selectPermissionsByUsername(username));  
 **return** authorizationInfo;  
 }  
  
 @Override  
 **protected** AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken authenticationToken) **throws** AuthenticationException {  
 *//验证当前用户，获取验证信息  
 //根据token获取用户名* String username = (String)authenticationToken.getPrincipal();  
 *//根据用户名从数据库中查询出该用户的信息* User user = **userService**.selectUserByUsername(username);  
 **if** (Objects.*nonNull*(user)){  
 *//把当前用户存到session中* SecurityUtils.*getSubject*().getSession().setAttribute(**"user"**,user);  
 *//传入用户名和密码进行身份验证，并返回验证信息* SimpleAuthenticationInfo authcInfo = **new** SimpleAuthenticationInfo(user.getName(), user.getPassword(), **"myRealm"**);  
 **return** authcInfo;  
 }  
 **return null**;  
 }  
}

2.用户、角色、权限的查询语句





### Shiro配置

Shiro配置主要配置三个东西：自定义realm、安全管理器SecurityManager和Shiro过滤器。

配置SecurityManager时需要将自定义的realm添加进来，这样的话Shiro才会走到自定义的realm中

配置Shiro过滤器时，我们需要自定义一个ShiroFactoryBean，然后将SecurityManager添加进来。主要的配置如下：

默认登录的url：身份验证失败时会访问该url

认证成功之后要跳转的url：权限认证失败会访问该url

需要拦截或者放行的url：这些都放在一个有序的map中

# 集成Redis

Redis是一种非关系型数据库（NoSQL），是以key-value的形式存储的，和传统的关系型数据库不一样，不一定遵循传统数据库的一些基本要求。比如说SQL标准，ACID属性，表结构等。这类数据库主要有以下特性：非关系型的、分布式的、开源的、水平可扩展的。

NoSQL使用场景：对数据高并发读写、对海量数据的高效存储和访问、对数据

的高可扩展性和高可用性等等。

Redis的key可以是：字符串、哈希、链表、集合和有序集合。value可以是String、list、set、zset。这些数据都支持push/pop、add/remove、取交集和并集以及更多丰富的操作，也支持不同方式的排序操作。

为了保证效率，数据都是缓存在内存中，它也可以周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件中。

# 集成ActiveMQ

## JMS和ActiveMQ简介

JMS即Java消息服务应用程序接口，是一个java平台中关于面向消息中间件的（MOM）的API，用于在两个应用程序之间或分布式系统中发送消息，进行异步通信。Java消息服务是一个与具体平台无关的API，绝大多数MOM提供商都对JMS提供支持。

JMS只是接口，不同的提供商或者开源组织对其有不同的实现，ActiveMQ就是其中之一，它支持JMS，是Apache推出的。JMS有几个对象模型：

连接工厂：ConnectionFactory

JMS连接：Connection

JMS会话：Session

JMS目的：Destination

JMS生产者：Producer

JMS消费者：Consumer

JMS消息两种类型：点对点和发布/订阅

可以看出JMS和JDBC有点类似，JDBC是可以用来访问许多不同关系数据库的API，而JMS则提供同样与厂商无关的访问方法，以访问消息收发服务

ActiveMQ是Apache的一个能力强劲的开源消息总线。ActiveMQ完全支持JMS1.1和J2EE1.4规范，尽管JMS规范出台已经是很久的事情了，但是JMS在当今的Java EE应用中仍然扮演着特殊的地位。ActiveMQ用在异步消息的处理上（所谓异步消息即消息发送者无需等待消息接受者的处理及返回，甚至无需关心消息是否发送成功）

异步消息主要有两种目的地形式：队列（queue）和主体（topic）。队列用于点对点形式的消息通信，主体用于发布/订阅式的消息通信

## ActiveMQ安装

官网下载地址：<http://activemq.apche.org/>

本次使用版本：apache-activemq-6.1.2

下载后解压即可使用

# 其他工具

## lombok插件

主要功能：免去JavaBean中写get()、set()、toString()、构造器、equals()、hashcode()等方法

通过注解实现：

@Data 除了构造方法以外的所有方法

@AllArgsConstructor 有参构造

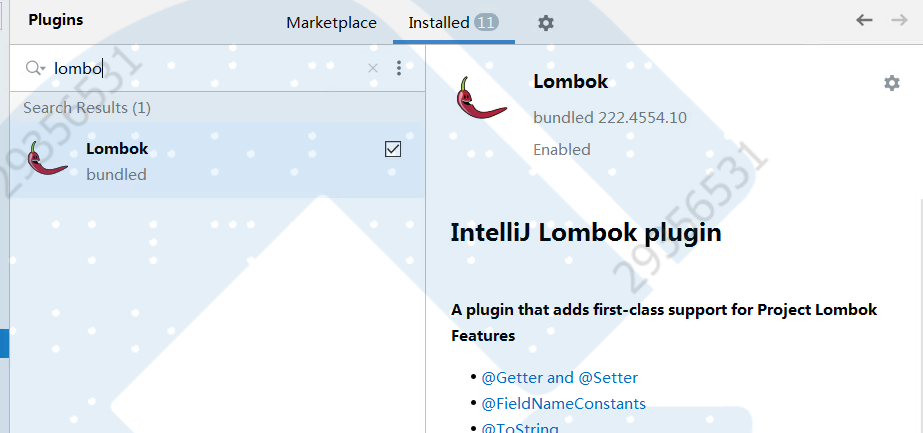
@NoArgsConstructor 无参构造

使用步骤：

1.引入依赖

*<!-- lombok辅助工具 -->*<**dependency**>  
 <**groupId**>org.projectlombok</**groupId**>  
 <**artifactId**>lombok</**artifactId**>  
 <**version**>1.18.30</**version**>  
</**dependency**>

2.IDEA中下载lombok插件



3.在javaBean中加上注解即可

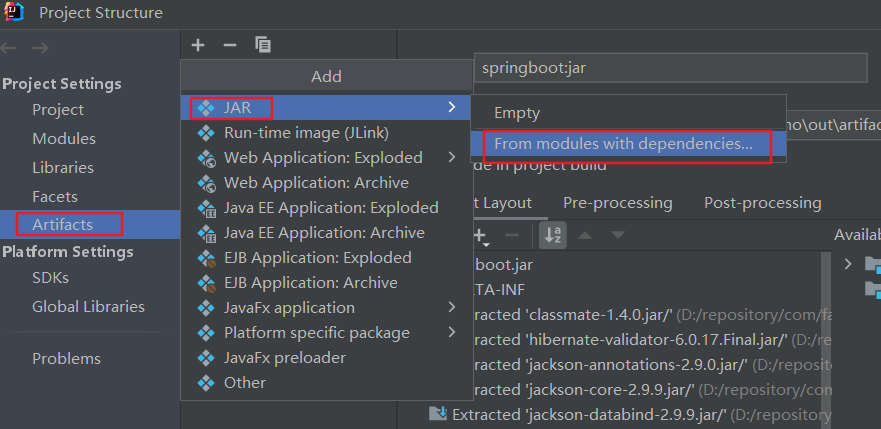
@Data *//get、set方法*@NoArgsConstructor *//无参构造***public class** Center {  
 **private** String **center**;  
 **private** String **centerName**;  
  
}

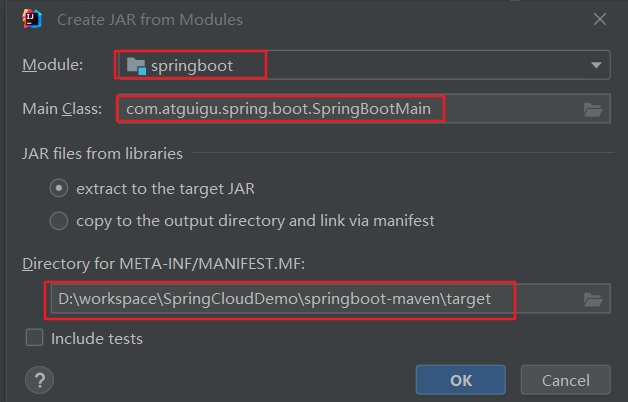
## 将工程打成可运行的jar包

### 依赖

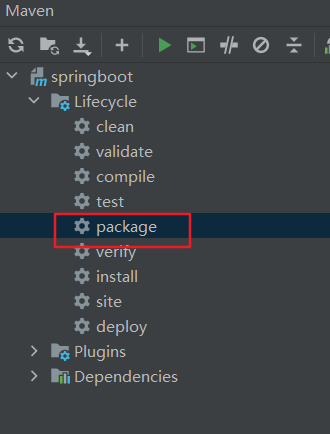
<!-- Maven构建过程相关配置 -->  
<build>  
 <!-- 构建过程中所需要用到的插件 -->  
 <plugins>  
 <!-- 这个插件将SpringBoot应用打包成一个可执行的jar包 -->  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 </plugin>  
 </plugins>  
</build>

### 指定目录





### 打包



会在指定目录下生成jar包

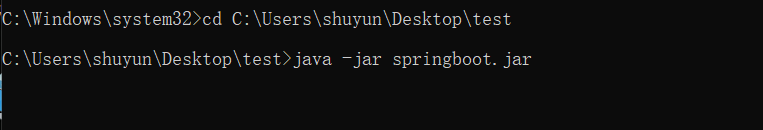


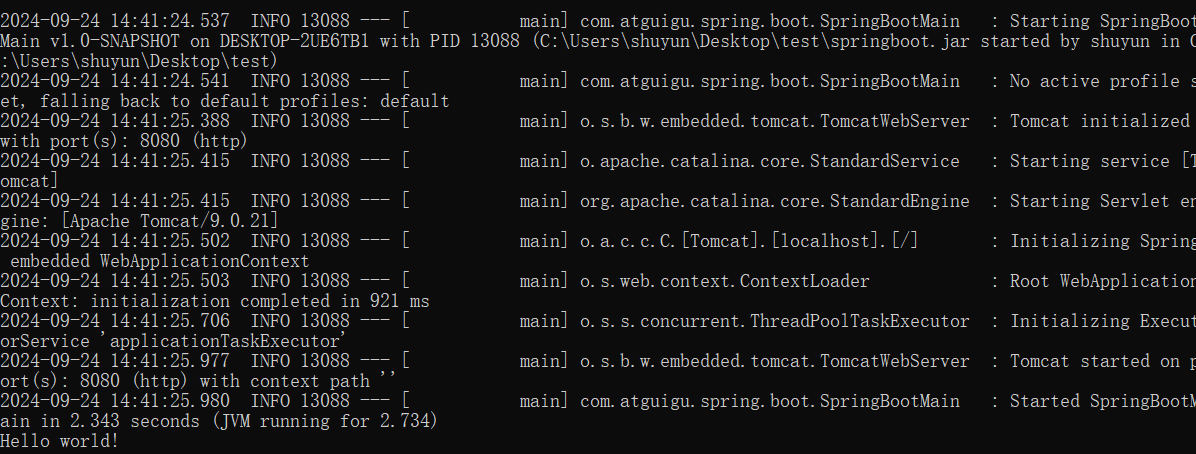
### 运行

把jar拷贝到一个目录下

cmd打开命令行窗口

java -jar jar包路径即可启动





# 最后