**2-2**

@EnableAutoConfiguration

1、把启动类当成一个自动化且支持配置的bean

2、开启整个基于SpringBoot的工程的自动化配置

@EnableAutoConfiguration
public class App {
public static void main(String[] args) {
System.out.println("Hello World!");
}
}

什么是springboot的自动化配置？

如果要开启一个spring bean，可能要声明一个xml或者相当于注解形式的一个变量，让它被spring自动扫描到，把它变成一个bean

只要有@EnableAutoConfiguration，springboot项目会把我们对所有的比如说对数据库和redis的依赖以及对spring容器本身的aop的依赖管理统统以一个自动化的配置加载到我们对应的项目的工程当中

我们一旦开启了对应的@EnableAutoConfiguration之后，需要用一行代码来启动对应的springboot项目：

SpringApplication.run(App.class, args);

@EnableAutoConfiguration
public class App {
public static void main(String[] args) {
System.out.println("Hello World!");
SpringApplication.run(App.class, args);
}
}

这样springboot项目的后台应用程序就能运行起来了

**2-3**

springboot集成mybatis，能够完成项目对数据库的读写操作

在springboot项目的resources目录下的application.properties为springboot的默认配置，在文件内就可以对springboot的配置进行修改

如：

server.port=8090

修改tomcat的端口号

即springboot通过对应在固定的classpath的resources目录下寻找application.properties配置文件，通过key和value对加载内部的一些配置，就可以完成通过配置文件改变springboot中spring bean生成的默认配置策略

pom.xml依赖配置：

配置完依赖后，通过在application.properties配置文件中导入mybatis所需要的一些配置，用来启动一个带mybatis数据库访问的springboot工程

配置如下：

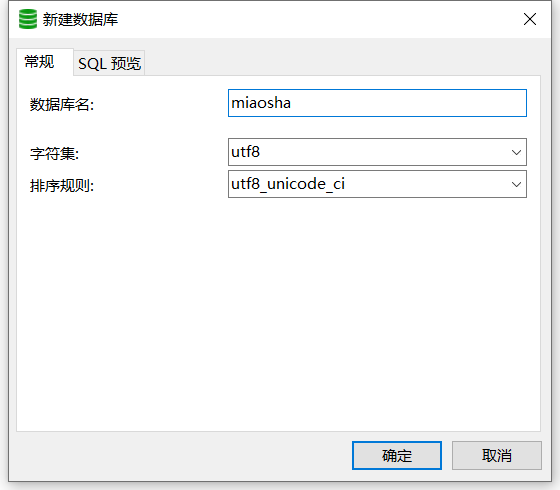
mybatis.mapper-locations=classpath:mapping/\*.xml

**2-4**

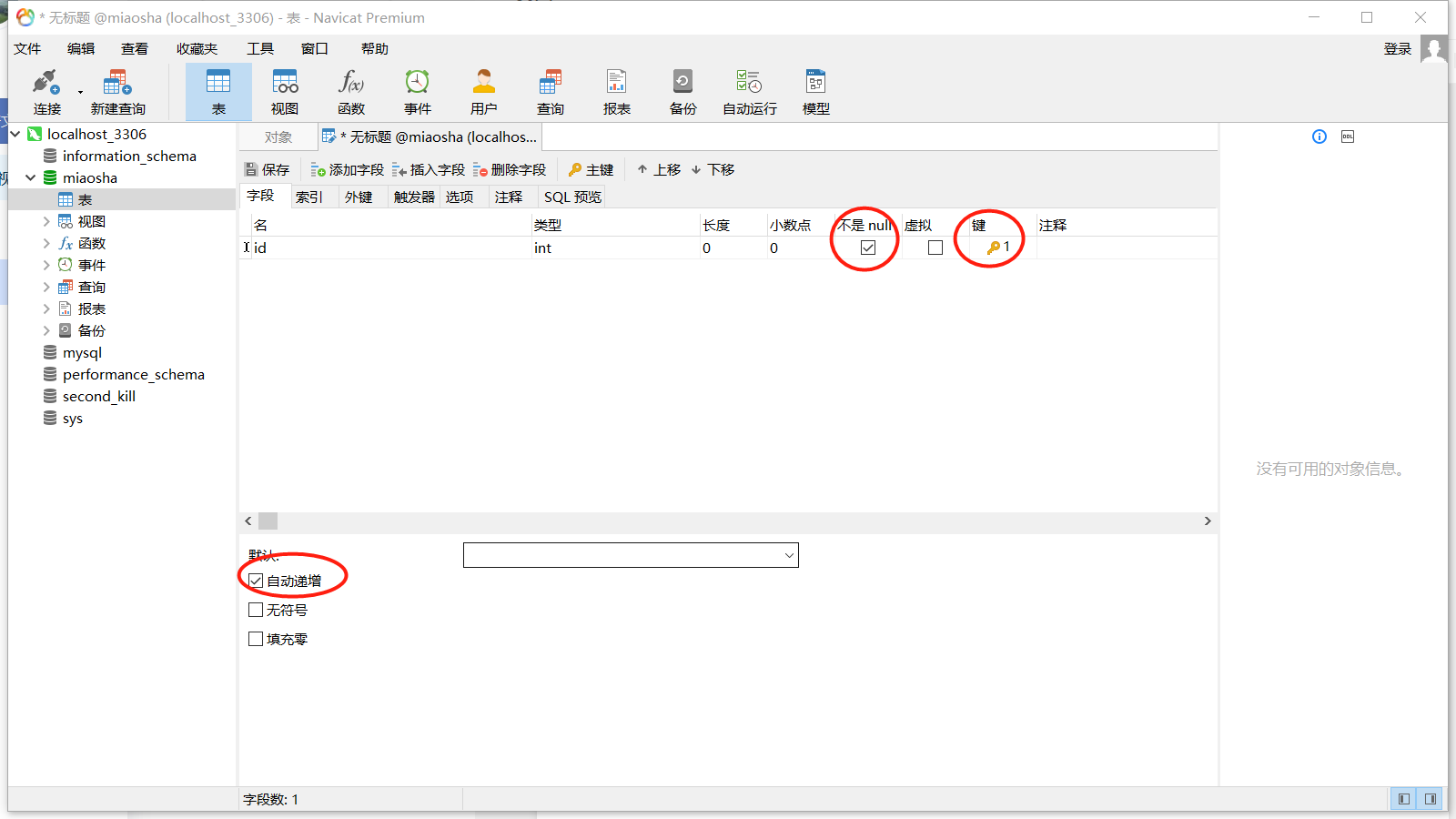
随后使用mybatis的自动生成工具用来生成对应数据库文件的映射

pom.xml中要引入mybatis自动生成文件的插件

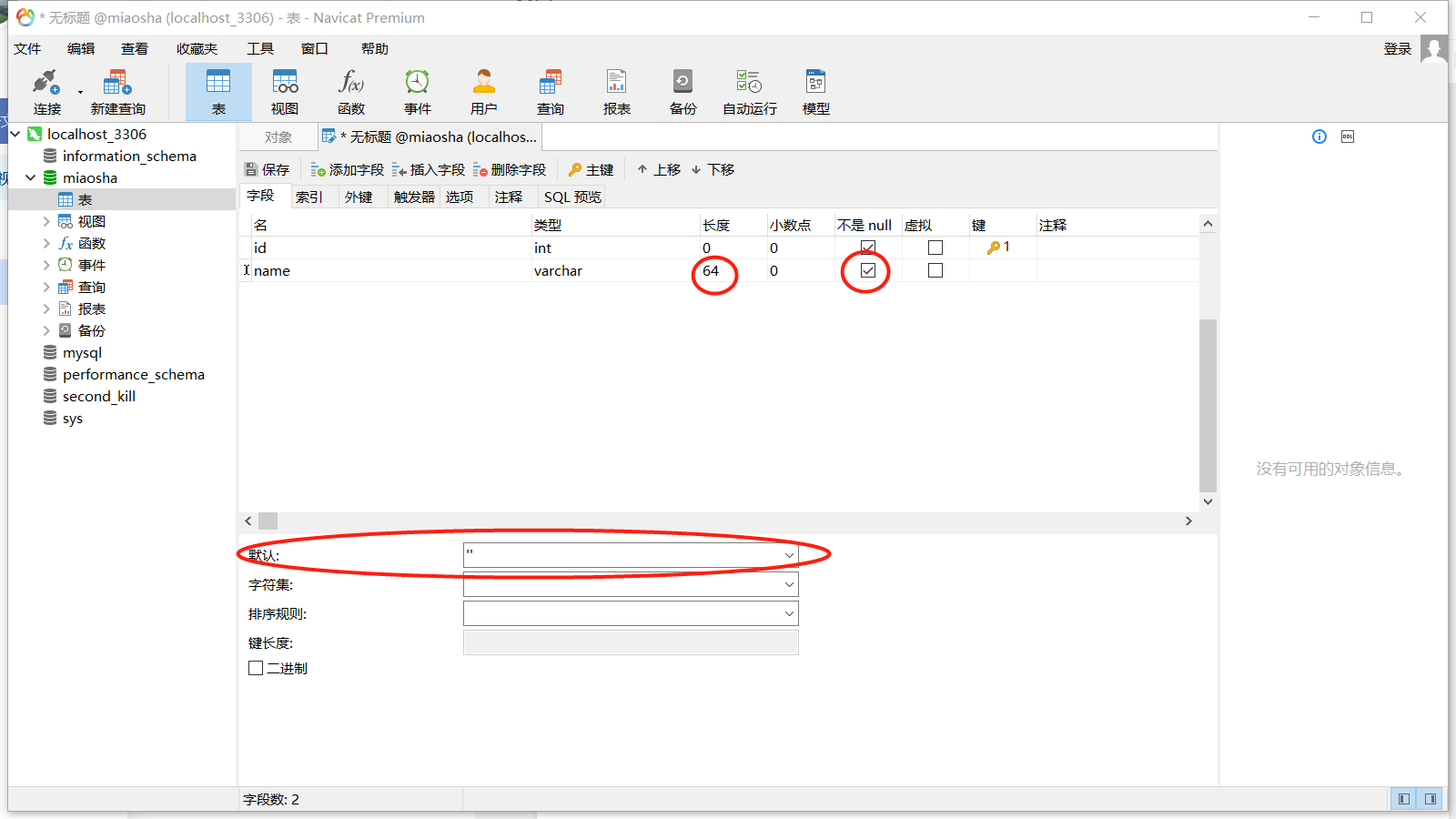
数据库的构建



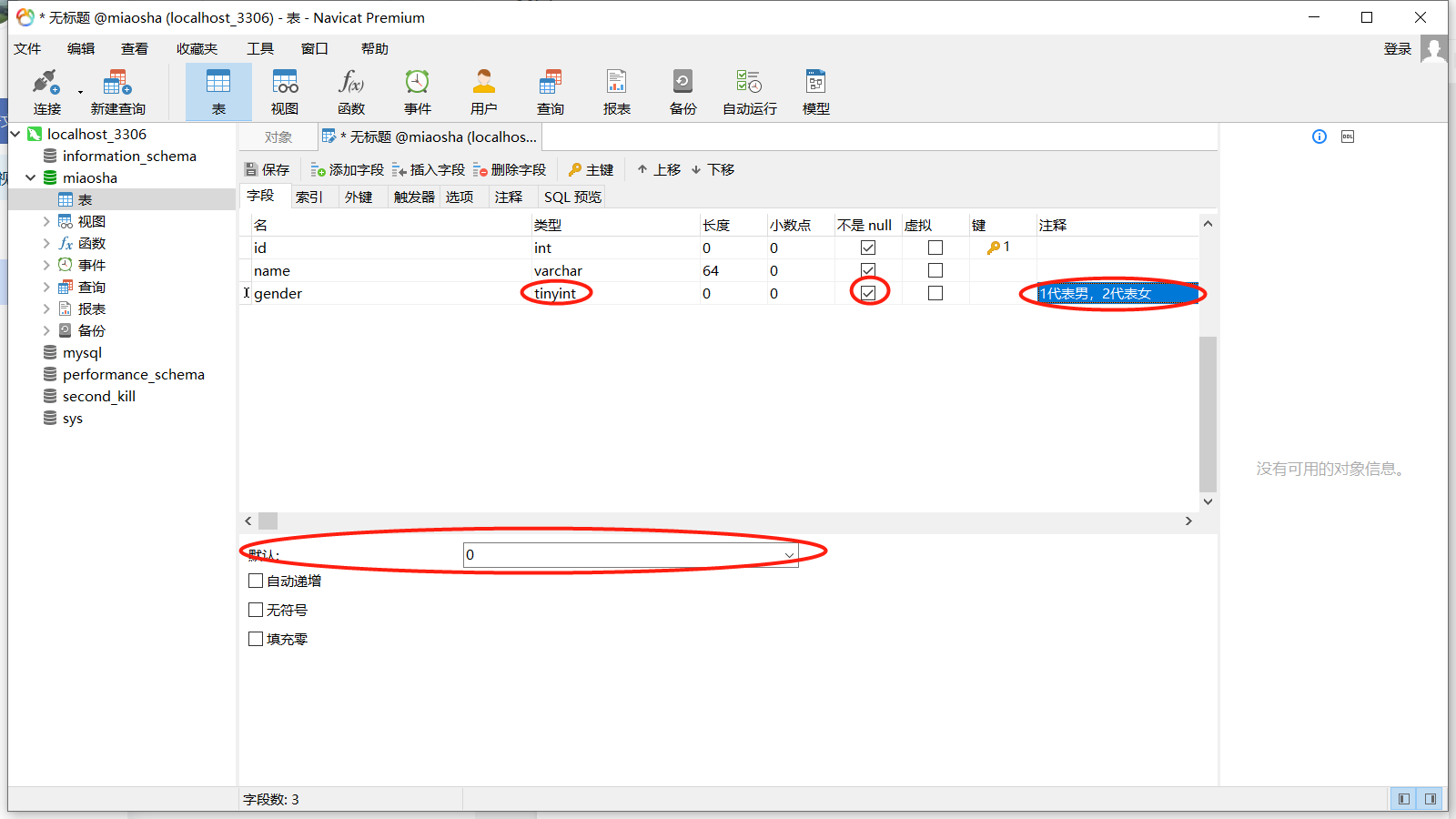
1、创建用户表



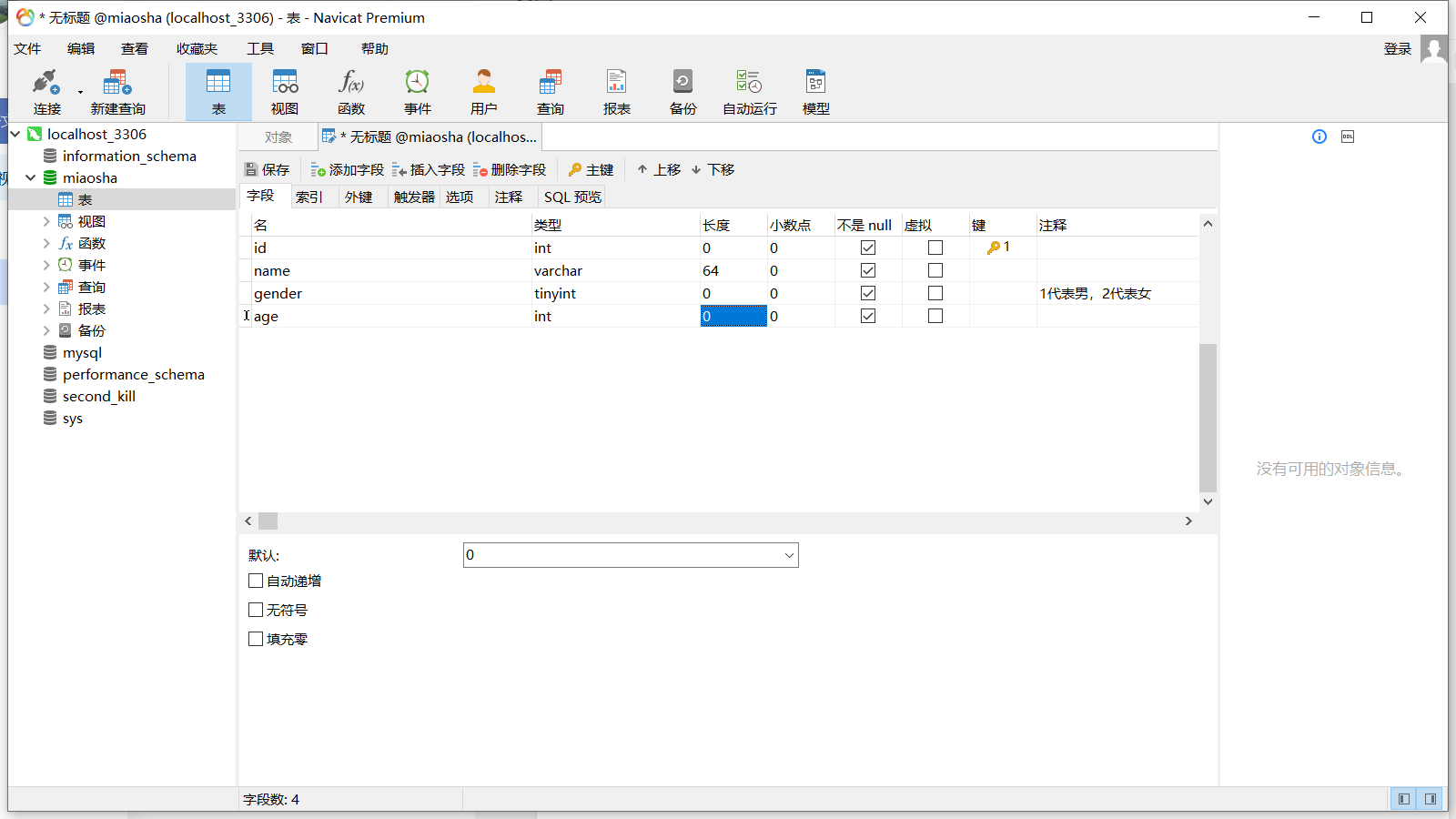
id为递增主键



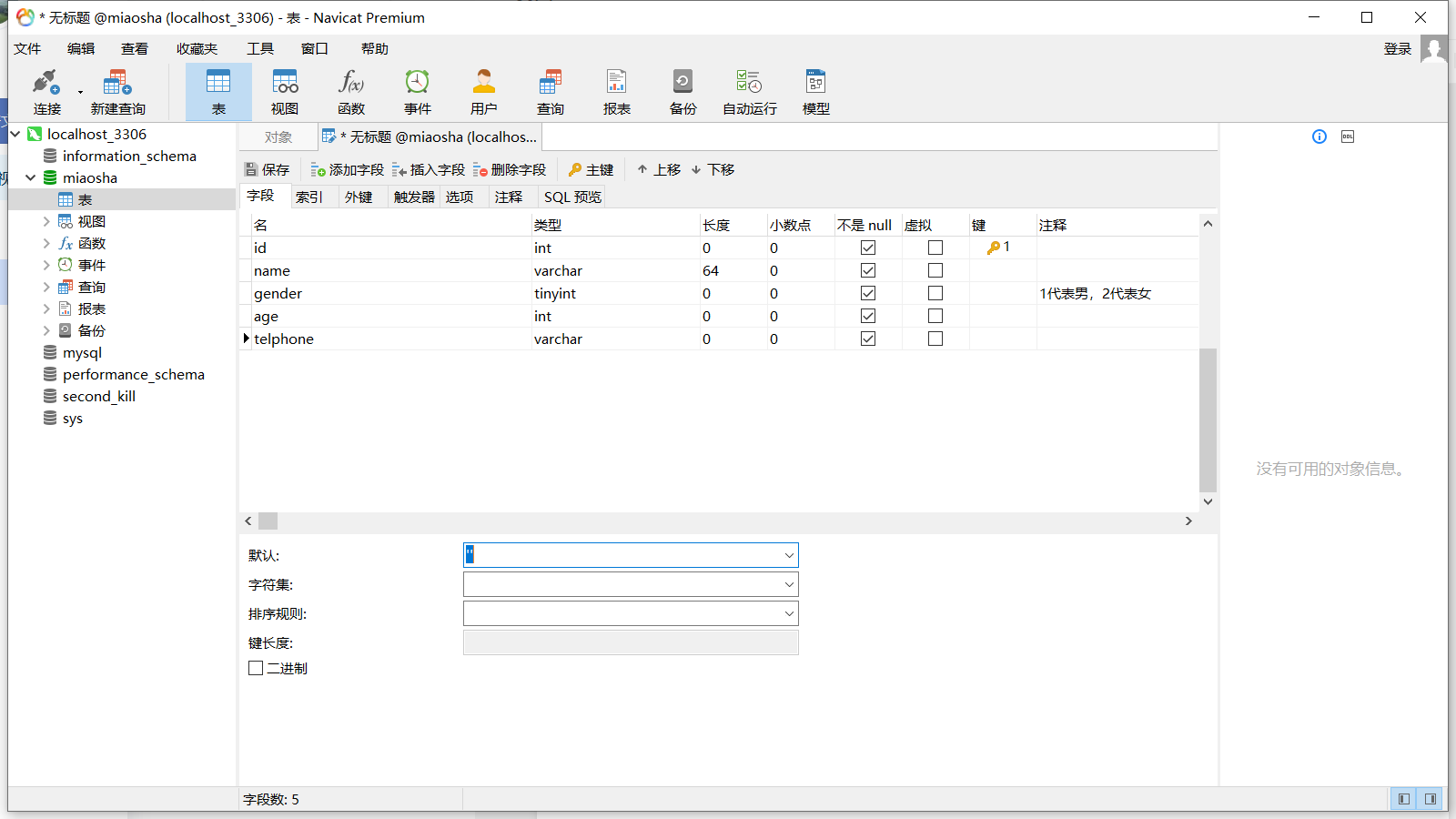
name为空字符串



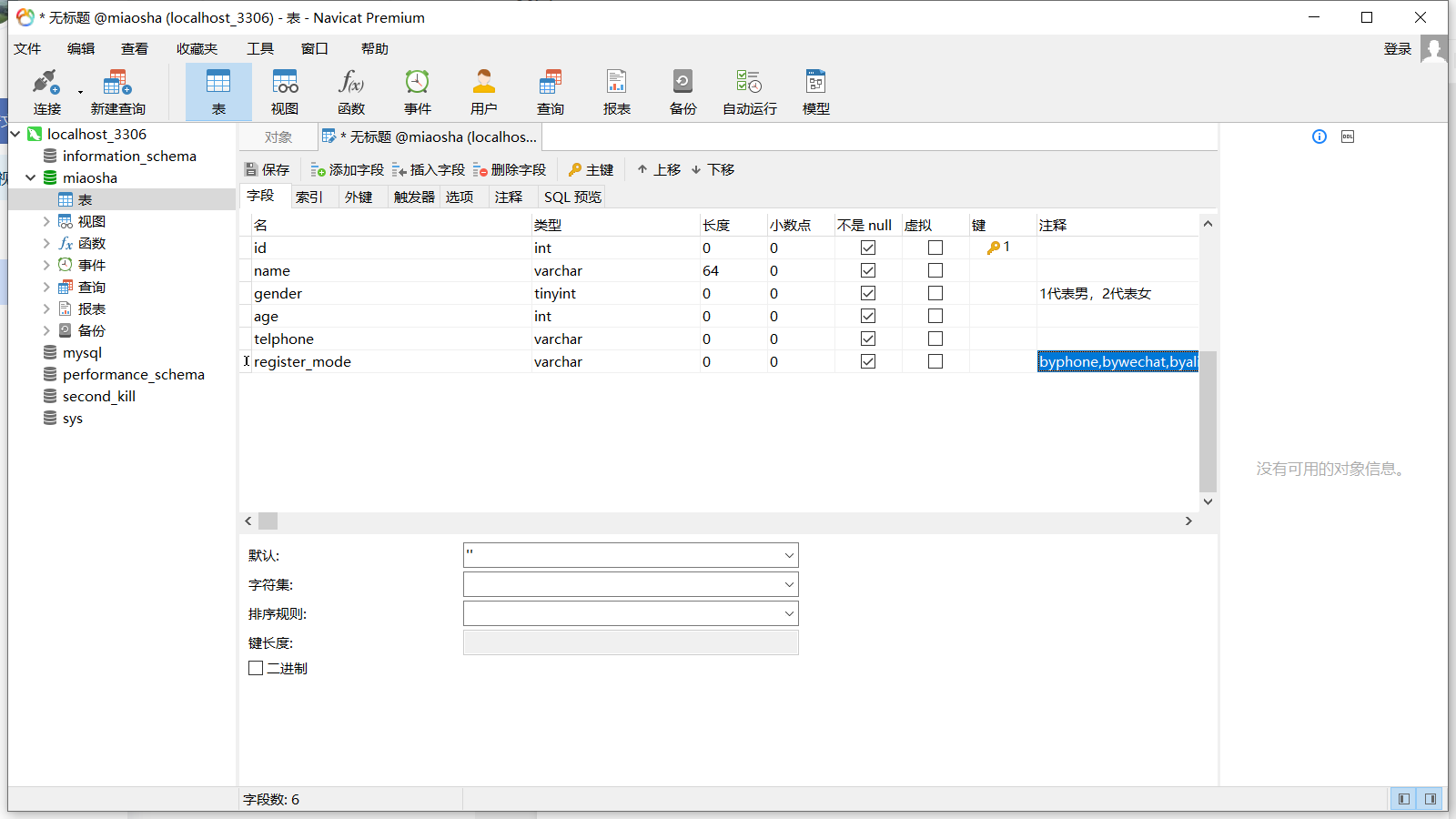
性别，tinyint只占一位



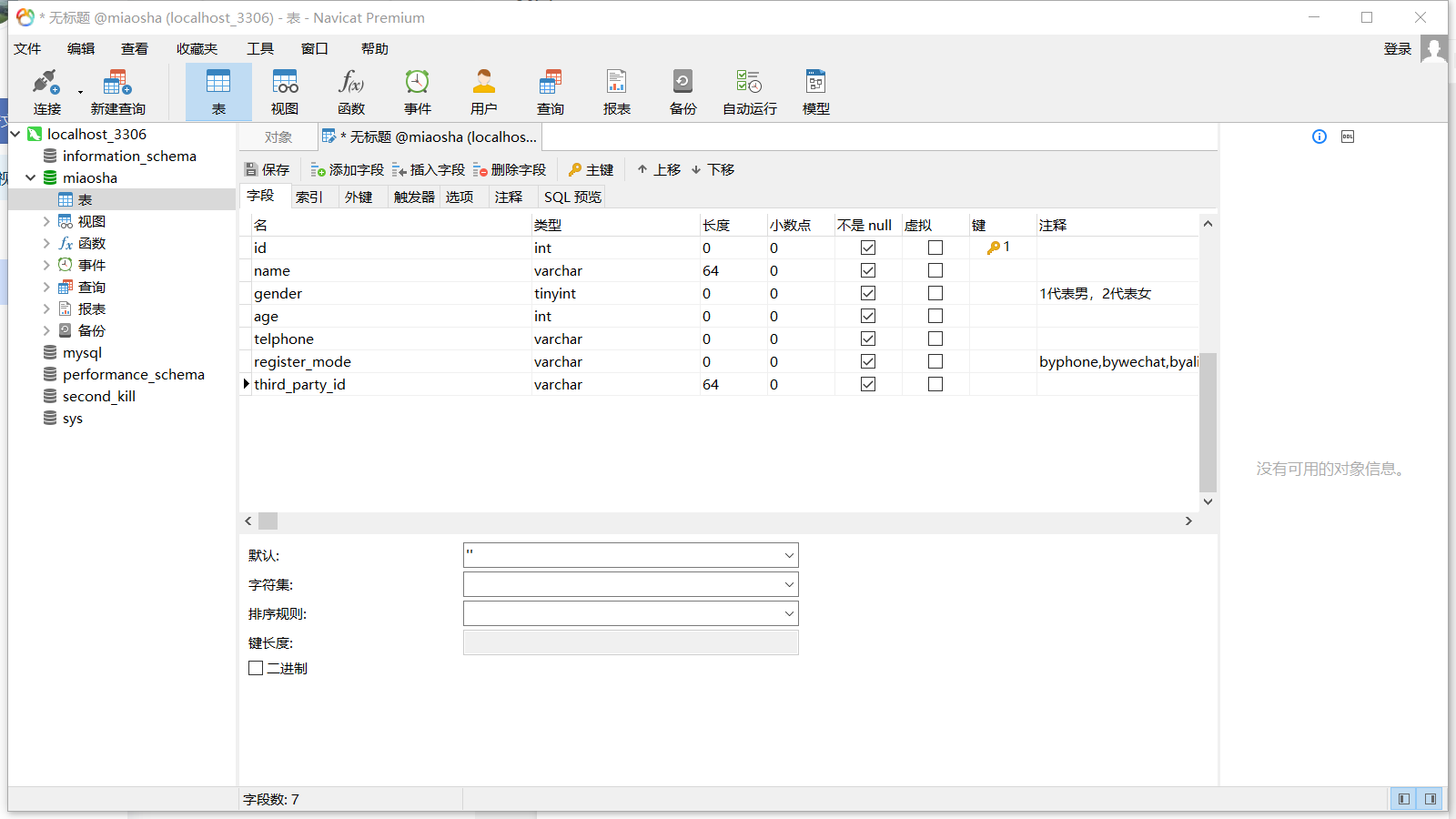
年龄



手机号：通过手机号给用户做短信的发送，即otp验证码的发送，以及用telphone做登录账号



登录方式



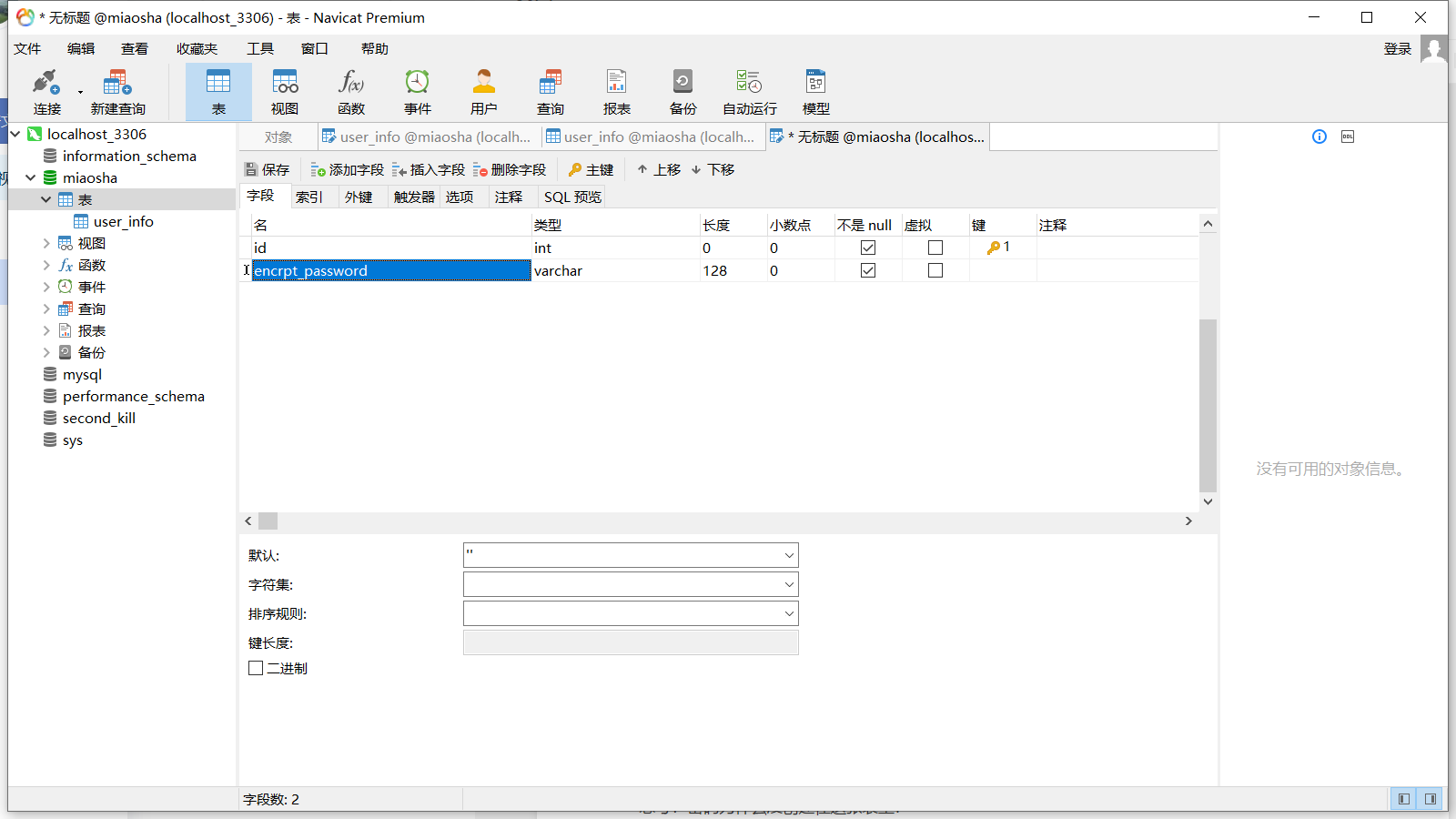
记录第三方登录的id

保存表为user\_info

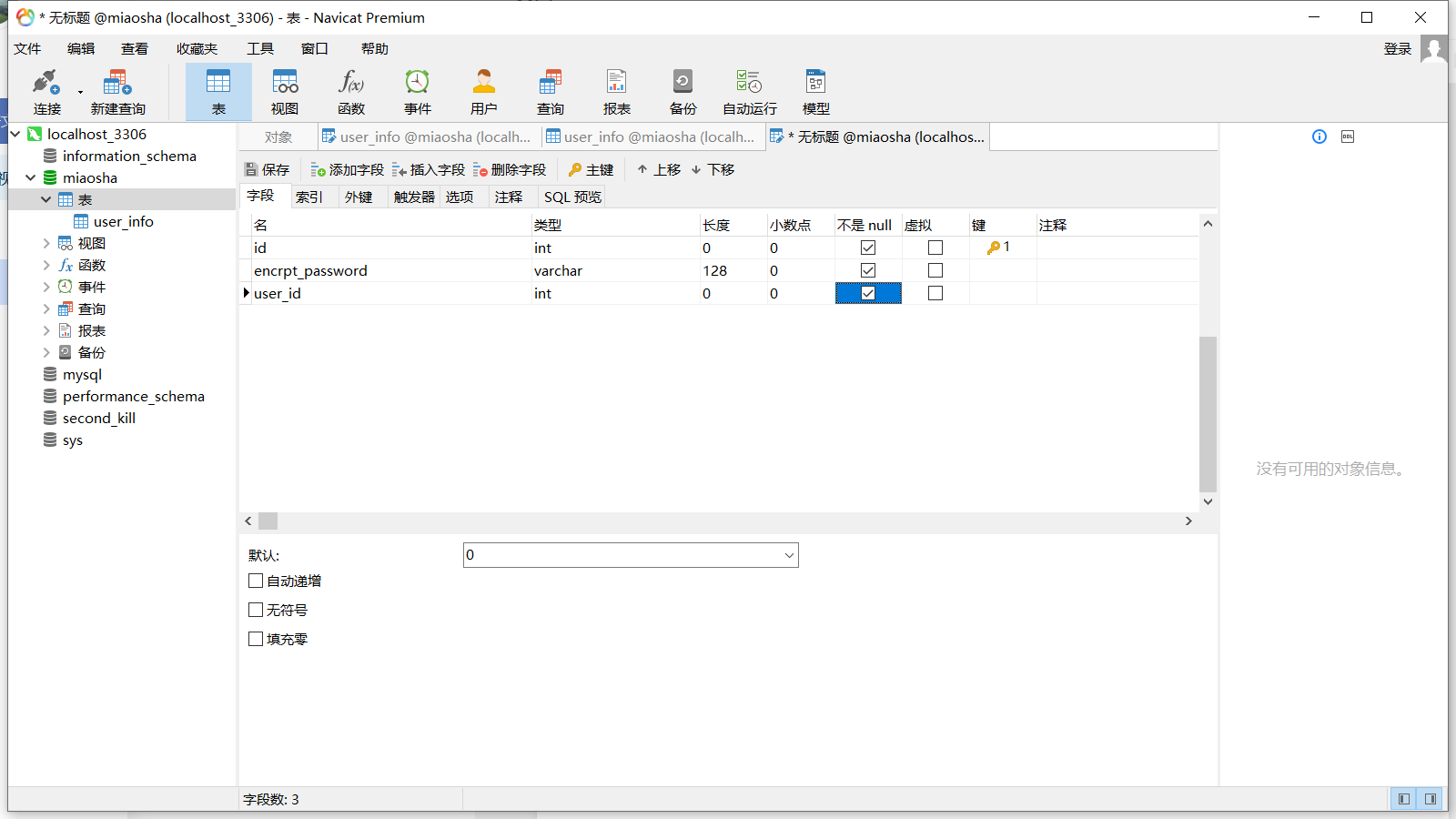
思考：密码为什么没创建在这张表里？

原因：密码是加密字符串，一般来说不跟主表创建在一起，二者要分开存取

创建密码表：



密码采用encrpt方式加密，以密文方式存储在数据库中，绝不能以明文方式存储在数据库中



user\_id作为外键关联到之前的用户表上

使用mybatis generator生成对应的数据库的mapper接口和dataObject中的java文件

加入注解@MapperScan("com.miaoshaproject.dao")，让springboot去扫描mybatis的配置文件，@SpringBootApplication和@EnableAutoConfiguration功能基本一样，都是使App这个类被spring托管，并且可以指定它是个主启动类

扫描这两个路径

@SpringBootApplication(scanBasePackages = {"com.miaoshaproject" })
@MapperScan("com.miaoshaproject.dao")

@SpringBootApplication(scanBasePackages = {"com.miaoshaproject" })
@RestController
@MapperScan("com.miaoshaproject.dao")
public class App {
public static void main(String[] args) {
System.out.println("Hello World!");
SpringApplication.run(App.class, args);
}
}

**3-1**

springmvc项目所需的模块

自上而下：

首先：controller层

然后：service层以及dao层

controller对应用户

以UserController为例：

@Controller("user")
@RequestMapping("/user")
public class UserController{
@RequestMapping("/get")
public void getUser(@RequestParam(name="id") Integer id){
//调用service服务获取对应id的用户对象并返回给前端
}
}

UserController需要制定一个Controller标记用来被Spring扫描到，这个controller的名字为user

@RequestMapping("/user")意思为在url上需要以/user的方式访问到

service内不仅需要xxx.service接口，还需要具体的实现类xxx.serviceImpl

public interface UserService{
//通过用户ID获取用户对象的方法
void getUserById(Integer id);
}

UserServiceImpl去实现UserService接口

@Service
//加这个注解来指明：对应的UserServiceImpl就是spring的一个service
public class UserServiceImpl implements UserService {
@Autowired
//导入UserDOMapper
private UserDOMapper userDOMaooer;
@Override
public void getUserById(Integer id){
//调用userDOMapper根据主键获取对应的用户：dataObject包中的UserDo
UserDo userDo = userDOMapper.selectByPrimaryKey(id);
}
}

但是在企业级的应用中，下面的userDo是不能透传给前端的

对于SpringMVC来说，model分为三层，第一层是我们定义的最简单的dataObject，它与数据库完全一一映射，即数据库有什么字段，dataObject中就有什么字段，并不含逻辑。

但是在service中不能简单的把对应数据库的映射透传返回给想要这个service的服务，即在service层必须要有一个model的概念，model才是真正意义上定义SpringMVC中业务逻辑交互的模型概念

UserDo中包含了所有信息和setter getter方法

package com.miaoshaproject.dataObject;
public class UserDO {
private Integer id;
private String name;
private Byte gender;
private Integer age;
private String telephone;
private String registerMode;
public Integer getId() {
return id;
}
public void setId(Integer id) {
this.id = id;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name == null ? null : name.trim();
}
public Byte getGender() {
return gender;
}
public void setGender(Byte gender) {
this.gender = gender;
}
public Integer getAge() {
return age;
}
public void setAge(Integer age) {
this.age = age;
}
public String getTelephone() {
return telephone;
}
public void setTelephone(String telephone) {
this.telephone = telephone == null ? null : telephone.trim();
}
public String getRegisterMode() {
return registerMode;
}
public void setRegisterMode(String registerMode) {
this.registerMode = registerMode == null ? null : registerMode.trim();
}
public String getThirdPartyId() {
return thirdPartyId;
}
public void setThirdPartyId(String thirdPartyId) {
this.thirdPartyId = thirdPartyId == null ? null : thirdPartyId.trim();
}
}

UserModel

/\*\*
\* \\@NotNull、@NotEmpty、@NotBlank的区别
\* 大致区别如下：@NotEmpty用在集合类上面 @NotBlank 用在String上面 @NotNull 用在基本类型上 链接: https://www.jianshu.com/p/c9e71b241daa
\*/
public class UserModel {
private Integer id;
@NotBlank(message = "用户名不能为空")
private String name;
@NotNull(message = "性别不能不填写")
private Byte gender;
@NotNull(message = "年龄不能不填写")
@Min(value = 0, message = "年龄必须大于0")
@Max(value = 150, message = "年龄必须小于150")
private Integer age;
@NotBlank(message = "手机号不能为空")
private String telephone;
private String registerMode;
private String thirdPartyId;
/\*\*
\* encryptPassword 由于数据模型层的关系(加密)设置在两张不同的表里面,
\* 但是对 java \*领域模型\* 的对象的概念来说,encryptPassword 就是属于 UserModel 的.
\* 因此 UserModel 才是真正意义上, 处理业务逻辑的核心的模型, 而dataObject仅仅只是一个对数据库的映射
\* 所以我们重新定义一下 UserService 的返回值, 是一个 UserModel的对象
\*/
@NotBlank(message = "密码不能为空")
private String encryptPassword;
public String getencryptPassword() {
return encryptPassword;
}
public void setencryptPassword(String encryptPassword) {
this.encryptPassword = encryptPassword;
}
public Integer getId() {
return id;
}
public void setId(Integer id) {
this.id = id;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public Byte getGender() {
return gender;
}
public void setGender(Byte gender) {
this.gender = gender;
}
public Integer getAge() {
return age;
}
public void setAge(Integer age) {
this.age = age;
}
public String getTelephone() {
return telephone;
}
public void setTelephone(String telephone) {
this.telephone = telephone;
}
public String getRegisterMode() {
return registerMode;
}
public void setRegisterMode(String registerMode) {
this.registerMode = registerMode;
}
public String getThirdPartyId() {
return thirdPartyId;
}
public void setThirdPartyId(String thirdPartyId) {
this.thirdPartyId = thirdPartyId;
}
}

一个UserModel对象，不仅包含UserDo中的内容，还有UserPasswordDO中的内容，因为encrptPassword是属于用户对象的，但因为数据模型层的关系，把用户信息和密码设计在两张不同的表里面。但是对应用模型来说，它就是属于UserModel的。因此UserModel的model层面才是我们处理业务逻辑的核心模型，dataObject仅仅是对数据库的映射

所以UserService中getUserById(Integer id)方法的返回值是个UserModel对象

public interface UserService {
//通过用户ID获取用户对象地方的方法
UserModel getUserById(Integer id);
}

也就是说通过UserService可以获取到用户领域模型的对象

@Service
public class UserServiceImpl implements UserService {
@Autowired
private UserDOMapper userDOMapper;
@Override
public UserModel getUserById(Integer id) {
//调用 UserDOMapper 获取 dataObject
UserDO userDO = userDOMapper.selectByPrimaryKey(id);
if (userDO == null) {
return null;
}
/\*\*
\* UserPasswordDO userPasswordDO = userPasswordDOMapper.\*\*selectByPrimaryKey\*\*(userDO.getId());
\* mybatis帮我们生成的方法里边, 仅仅只有 selectByPrimaryKey ,
\* 但是业务逻辑要求 通过对应用户id ,查询到加密的密码.
\* userPassword表中, 各字段分别是 (主键)id, encrypt\_password, user\_id
\* 所以, 改造一下 UserPasswordDOMapper.xml
\*/
//通过 用户id 获取对应的用户加密密码信息
UserPasswordDO userPasswordDO = userPasswordDOMapper.selectByUserId(userDO.getId());
return convertFromDataObject(userDO, userPasswordDO);
}
private UserModel convertFromDataObject(UserDO userDO, UserPasswordDO userPasswordDO) {
//处理一下空的问题
if (userDO == null) {
return null;
}
UserModel userModel = new UserModel();
BeanUtils.copyProperties(userDO, userModel);
//处理一下空的问题
if (userPasswordDO != null) {
//这里不能使用 copyProperties了, 因为对应内部还有 重名字段, 所以用 简单的set
userModel.setencryptPassword(userPasswordDO.getencryptPassword());
}
return userModel;
}
}

下面的这个方法可以把UserDo对象和UserPasswordDO组装成一个UserModel对象

private UserModel convertFromDataObject(UserDO userDO, UserPasswordDO userPasswordDO) {
//处理一下空的问题
if (userDO == null) {
return null;
}
UserModel userModel = new UserModel();
BeanUtils.copyProperties(userDO, userModel);
//处理一下空的问题
if (userPasswordDO != null) {
//这里不能使用 copyProperties了, 因为对应内部还有 重名字段, 所以用 简单的set
userModel.setencryptPassword(userPasswordDO.getencryptPassword());
}
return userModel;
}

这个函数：source是userDO，target是userModel

把userDO的属性copy到userModel内

BeanUtils.copyProperties(userDO, userModel);

下面就不能再使用copy了，因为userPasswordDO中和userModel中都有id字段，重复了

因此使用简单的get set方法，把userPasswordDO中的encrptPassword字段set进userModel中

userModel.setencryptPassword(userPasswordDO.getencryptPassword());

再加上判空的条件，保护一下

在UserPasswordDOMapper.xml中添加一个selectByUserId，使得对象可以通过UserId检索到

<!--
改造UserPasswordDOMapper.xml部分
然后去 UserPasswordDOMapper.java 添加方法
-->
<select id="selectByUserId" parameterType="java.lang.Integer" resultMap="BaseResultMap" >
select
<include refid="Base\_Column\_List" />
from user\_password
where user\_id = #{user\_id,jdbcType=INTEGER}
</select>
<delete id="deleteByPrimaryKey" parameterType="java.lang.Integer">

再去UserPasswordDOMapper.java这个接口中新建这个方法

/\*\*
\* 添加的根据 user\_id 获取 encrypt\_password的 mybatis方法
\* @param id
\* @return
\*/
UserPasswordDO selectByUserId(Integer id);

再在UserServiceImpl.java中实现这个方法

@Override
public UserModel getUserById(Integer id) {
//调用 userDOMapper 获取 userDO
UserDO userDO = userDOMapper.selectByPrimaryKey(id);
if (userDO == null) {
return null;
}
/\*\*
\* UserPasswordDO userPasswordDO = userPasswordDOMapper.\*\*selectByPrimaryKey\*\*(userDO.getId());
\* mybatis帮我们生成的方法里边, 仅仅只有 selectByPrimaryKey,
\* 但是业务逻辑要求 通过对应用户id ,查询到加密的密码.
\* userPassword表中, 各字段分别是 (主键)id, encrypt\_password, user\_id
\* 所以, 改造一下 UserPasswordDOMapper.xml
\*/
//通过 用户id 获取对应的用户加密密码信息
UserPasswordDO userPasswordDO = userPasswordDOMapper.selectByUserId(userDO.getId());
//通过convertFromDataObject方法，把userDO和userPasswordDO转成userModel
return convertFromDataObject(userDO, userPasswordDO);
//返回给controller层
//service层的返回值返回给controller层
}

return convertFromDataObject(userDO, userPasswordDO);
//返回给controller层
//service层的返回值返回给controller层

UserController中构建一个方法

@Controller("user")
@RequestMapping("/user")
public class UserController{
@RequestMapping("/get")
public UserModel getUser(@RequestParam(name="id") Integer id){
//调用service服务获取对应id的用户对象并返回给前端
UserModel userModel = userService.getUserById(id);
//返回给前端页面
return userModel;
}
}

启动后端服务之后，url输入为localhost:8090/user/get?id=1,出现信息

！！！但是，此时前端页面中包含用户的密码，不安全，设计不专业

原因是：在于下面的代码中：

//调用service服务获取对应id的用户对象并返回给前端
UserModel userModel = userService.getUserById(id);
//返回给前端页面
return userMode

这样是把一个领域模型，也就是用于自己业务逻辑处理的领域模型返回给了前端用户，而前端只需要拿到用于展示的业务处理的字段即可，而不能是领域模型本身

因此需要在controller层再加一层模型对象存放在viewobject包中

新建下面的类：

public class UserVO {
private Integer id;
private String name;
private Integer gender;
private Integer age;
private String telephone;
//下面再加上setter和getter方法
}

把getUser()方法的返回值改成UserVO，再写一个convertFromModel(UserModel userModel)方法，把userModel的值导进UserVO中，并返回UserVO

@Controller("user")
@RequestMapping("/user")
public class UserController{
@RequestMapping("/get")
public UserVO getUser(@RequestParam(name="id") Integer id){
//调用service服务获取对应id的用户对象并返回给前端
UserModel userModel = userService.getUserById(id);
//返回给前端页面
//将核心领域模型用户对象转换为可供UI使用的viewModel
return convertFromModel(userModel);
}
private UserVO convertFromModel(UserModel userModel) {
if (userModel == null) {
return null;
}
UserVO userVO = new UserVO();
BeanUtils.copyProperties(userModel, userVO);
return userVO;
}
}

总结：本节做了完整的读操作的展示，通过controller层到对应的service层，再到对应的dao层，并且每一层贯穿了始终。也就是说dataObject层（以UserDO为例）负责数据存储到service层的传输，并且在用户的service服务里面，组装了对应的需要的领域模型做下一步的处理，controller层（以UserController为例）做了到用户viewobject（以UserVO为例）之间的透传，保证了UI只使用它需要展示的字段即可

**3-2**

上面的缺陷在于，不能出错，一旦出错的话页面会报错，但是用户不知道哪里出 现了问题，出现的是错误码，前端需要的是有意义的错误信息

只要服务器能受理请求（页面能打开，即使是whitelabel error page也是能受理的），status按理说都是200

业务逻辑中有任何错误，应该通过业务逻辑的错误标识方式来做

分为两步处理：

1、归一化@RespnseBody 的返回参数，归一化成统一的status+data格式

新建一个包response，用来处理http返回，在这个包里新建一个类，CommonReturnType类

package com.miaoshaproject.response;
public class CommonReturnType {
// 表明对应请求的 返回结果 "success" 或 "fail"
private String status;
// 若status = "success", 则data内, 返回前端需要的json数据
// 若status = "fail", 则data内 返回 通用的错误码格式.
private Object data;
//定义一个通用的创建方法
public static CommonReturnType create(Object result) {
// 重载 (Object) 调用 (Object, String)
//如果 CommonReturnType.create() 不带任何status的话, 则直接添加success的status,并返回
return CommonReturnType.create(result, "success");
}
public static CommonReturnType create(Object result, String status) {
CommonReturnType type = new CommonReturnType();
type.setStatus(status);
type.setData(result);
return type;
}
public String getStatus() {
return status;
}
public void setStatus(String status) {
this.status = status;
}
public Object getData() {
return data;
}
public void setData(Object data) {
this.data = data;
}
}

修改UserController的getUser方法

@RequestMapping("/get")
@ResponseBody
/\*\*
\* 问题:
\* public \*UserModel\* getUser(@RequestParam(name = "id") Integer id) {
\* 此处我们将领域模型返回给了前端用户, 非常不科学, 非常不专业
\* 前端仅仅只需要, 他拿到作业务展示处理的字段即可, 而不是整个领域模型本身
\* 处理方法:
\* UserVO 代替 UserModel, 隐去不必要字段
\*/
/\*\*
\* 问题: 对于正确返回的json, 格式凌乱, 对前端UI处理不方便
\* 处理方法: CommonReturnType 统一返回格式
\*/
public CommonReturnType getUser(@RequestParam(name = "id") Integer id) throws BusinessException {
// 调用service服务获取对应id 的用户对象并返回给前端
UserModel userModel = userService.getUserById(id);
if (userModel == null) {
throw new BusinessException(EmBusinessError.USER\_NOT\_EXIST);
}
// 将核心领域模型用户对象, 转化为 可供UI使用的viewobject
UserVO userVO = convertFromModel(userModel);
// 返回通用对象
return CommonReturnType.create(userVO);
}

对于前端代码，其实很简单，只要http返回值是正常的200状态，只需要去解析对应的status和对应的data数据

**3-3返回错误信息**

当status返回false的时候，只需要把data定义成一个固定的错误码格式，比如errcode是多少，这个errcode所对应的errmsg的描述是什么，这样前端研发人员就能判断当status=false时，对应取errcode和errmsg的值就行了

1、定义一个通用的错误形式

在error包中

public interface CommonError {
public int getErrCode();
public String getErrMsg();
public CommonError setErrMsg(String errMsg);
}

接下来定义一个枚举类型，把error的错误信息取出来

package com.miaoshaproject.error;
public enum EmBusinessError implements CommonError{
//通用错误类型10000
// PARAMETER\_VALIDATION\_ERROR, 参数不合法统一使用, 此通用错误码, 区别在于 errMsg里的信息
PARAMETER\_VALIDATION\_ERROR(10001,"参数不合法"),
UNKNOW\_ERROR(10002,"未知错误"),
//20000开头为用户信息相关错误定义
USER\_NOT\_EXIST(20001,"用户不存在"),
//这个信息 USER\_LOGIN\_FAIL 账户密码不能提示得太明显, 如果明确告诉它 手机号 或者 密码 不存在的话, 会被人异常攻击
USER\_LOGIN\_FAIL(20002,"用户手机号或者密码不正确"),
USER\_NOT\_LOGIN(20003,"用户未登录"),
//30000开头为交易信息错误定义
STOCK\_NOT\_ENOUGH(30001,"库存不足")
;
//USER\_NOT\_EXIST 需要一个构造函数, 否则会报错
private EmBusinessError(int errCode, String errMsg) {
this.errCode = errCode;
this.errMsg = errMsg;
}
//枚举类, 本质是一个普通java类, 所以可以有成员变量
private int errCode;
private String errMsg;
@Override
public int getErrCode() {
return this.errCode;
}
@Override
public String getErrMsg() {
return this.errMsg;
}
@Override
public CommonError setErrMsg(String errMsg) {
this.errMsg = errMsg;
return this;
}
}

在做分布式开发的时候，全局需要一个统一的状态码的流转，比如在企业级项目中，A负责用户模块，B负责商品模块，二者信息如何互通？尤其是错误信息，就需要标识全局错误码，当系统越多，对应的错误码的管理就要在一个通用的文件里去做，以上就是模仿这种做法

//USER\_NOT\_EXIST 需要一个构造函数, 否则会报错
private EmBusinessError(int errCode, String errMsg) {
this.errCode = errCode;
this.errMsg = errMsg;
}

有了这个构造函数之后，当对应的"USER\_NOT\_EXIST"被定义出来之后，可以直接通过对应的构造方法构造出来一个实现了CommonError接口的子类EmBussinessError，这个子类是enum枚举类型

@Override
public CommonError setErrMsg(String errMsg) {
this.errMsg = errMsg;
return this;
}

这个方法是用来做什么的？

举个例子：

我们定义一个通用的数据类型：

PARAMETER\_VALIDATION\_ERROR(10001,"参数不合法"),

这个东西可以解决什么问题呢？

我们有很多程序存在入参校验，比如必填校验，是否是邮箱号等等，有了这个通用错误码之后，我们可以不用去定义无数的比如“某个用户名的入参没有传”要定义一个错误码，“邮箱没传”再定义一个错误码。这里只要使用这个通用错误码10001，但是对应的errMsg的描述是要详尽的，即10001这个错误码对应的场景不同时，errMsg是不一样的，所以需要有个serErrMsg去改动对应的errMsg信息，通过定制化的方式去改变它。

有了这个枚举类型之后，如果我们以后要往某个对应的错误信息里填值，只要在这边无限延伸定义想要的错误信息的enum值即可：

比如下面这样：

//通用错误类型10000
// PARAMETER\_VALIDATION\_ERROR, 参数不合法统一使用, 此通用错误码, 区别在于 errMsg里的信息PARAMETER\_VALIDATION\_ERROR(10001,"参数不合法"),
UNKNOW\_ERROR(10002,"未知错误"),
//20000开头为用户信息相关错误定义
USER\_NOT\_EXIST(20001,"用户不存在"),
//这个信息 USER\_LOGIN\_FAIL 账户密码不能提示得太明显, 如果明确告诉它 手机号 或者 密码 不存在的话, 会被人异常攻击
USER\_LOGIN\_FAIL(20002,"用户手机号或者密码不正确"),
USER\_NOT\_LOGIN(20003,"用户未登录"),
//30000开头为交易信息错误定义
STOCK\_NOT\_ENOUGH(30001,"库存不足")
;

如何使用？

定义的流程控制的机制就是通过Exception机制，也就是说程序出现了任何跑不下去的异常，统一抛一个Exception，这个Exception会被最后的controller层的springboot的handler捕获并做处理

定义一个BusinessException类继承Exception并实现CommonError接口

package com.miaoshaproject.error;
// 包装器 业务异常类 实现
public class BusinessException extends Exception implements CommonError{
private CommonError commonError;
// 直接接受 EmBusinessError的传参, 构造业务异常
public BusinessException(CommonError commonError) {
//调用 父类 Exception自身初始化机制
super();
this.commonError = commonError;
}
// 接收 自定义errMsg的方式, 构造业务异常
public BusinessException(CommonError commonError, String errMsg) {
super();
this.commonError = commonError;
// 二次改写 errMsg
this.commonError.setErrMsg(errMsg);
}
@Override
public int getErrCode() {
return this.commonError.getErrCode();
}
@Override
public String getErrMsg() {
return this.commonError.getErrMsg();
}
@Override
public CommonError setErrMsg(String errMsg) {
this.commonError.setErrMsg(errMsg);
return this;
}
}

首先，这个类内部需要强关联一个对应的CommonError

private CommonError commonError;

其实在这边，这个commonError就是刚才的Enum类

再创建构造函数方便使用：

// 直接接收 EmBusinessError的传参, 构造业务异常
public BusinessException(CommonError commonError) {
//调用 父类 Exception自身初始化机制
super();
this.commonError = commonError;
}
// 接收 自定义errMsg的方式, 构造业务异常
public BusinessException(CommonError commonError, String errMsg) {
super();
this.commonError = commonError;
// 二次改写 errMsg
this.commonError.setErrMsg(errMsg);
}

在UserController类中尝试一下：

public CommonReturnType getUser(@RequestParam(name = "id") Integer id) throws BusinessException {
// 调用service服务获取对应id 的用户对象并返回给前端
UserModel userModel = userService.getUserById(id);
if (userModel == null) {
throw new BusinessException(EmBusinessError.USER\_NOT\_EXIST);
}
// 将核心领域模型用户对象, 转化为 可供UI使用的viewobject
UserVO userVO = convertFromModel(userModel);
// 返回通用对象
return CommonReturnType.create(userVO);
}

如果获取的类不存在：

if (userModel == null) {
throw new BusinessException(EmBusinessError.USER\_NOT\_EXIST);
}

**3-4 异常处理**

通过springboot自带的springmvc的handleexception去解决通用异常处理的方式

new BusinessException(EmBusinessError.USER\_NOT\_EXIST);

上面被抛出的BusinessException被抛到了Tomcat的容器层，tomcat处理这个异常就是返回一个status=500的错误页

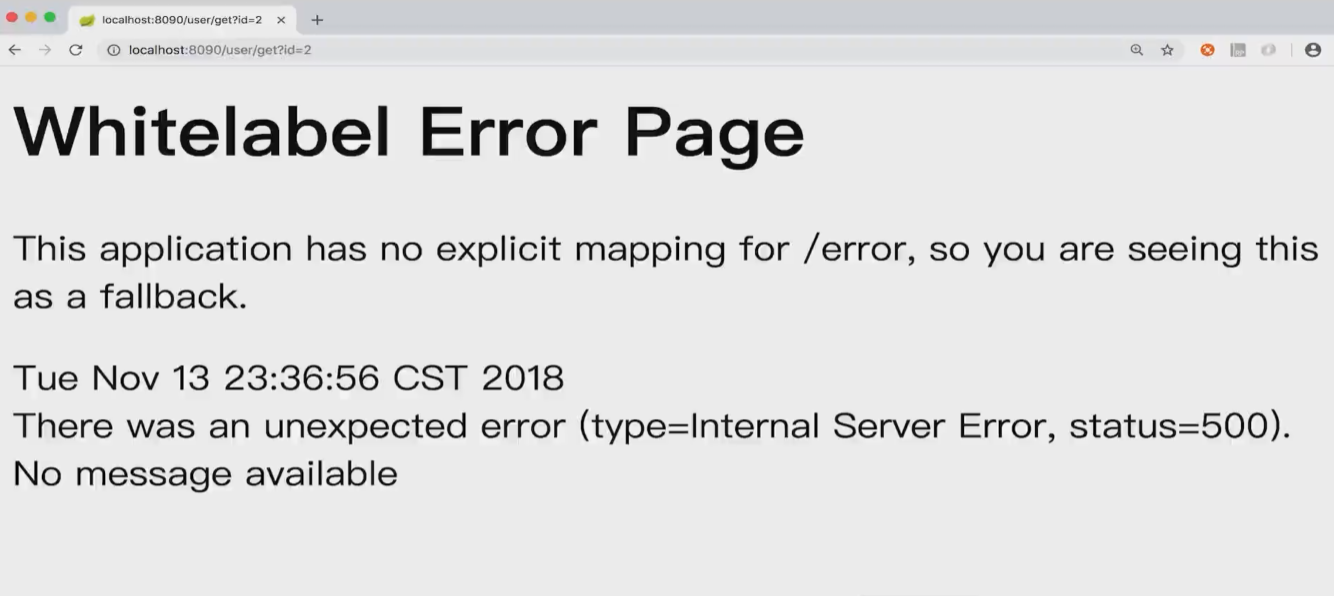
所以就需要一个机制来拦截掉tomcat的异常处理

springboot提供了一个通用化解决方式：

定义exceptionhandler解决未被controller层吸收的exception。

思想：

对于web系统来说，controller层的异常，算是业务处理的最后一道关口，controller层的异常如果被处理掉了，就能够在返回web前端之前能有一个很好的钩子？如果没有被处理掉的话，对用户体验来说就会出现这个界面：



@ExceptionHandler(Exception.class)

springboot提供了上面的这个注解，内部参数是当它收到一个什么类型的Exception之后才会进入它的处理环节

在此处定义了Exception的根类Exception.class

因为刚才web页面收到的请求为status=500，但是对于BusinessException这种异常来说，属于业务逻辑处理上的问题或者是正常的业务逻辑错误，而不是服务端不能处理的错误

因此定义下面注解：

@ResponseStatus(HttpStatus.OK)

即便controller抛出了exception，就捕获它，并且返回HttpStatus.OK，然后定义一个handlerException的方法，首先要接收这两个参数：HttpServletRequest request, Exception ex

@ResponseBody

@ResponseBody的作用其实是将java对象转为json格式的数据。

@ResponseBody注解的作用是将controller的方法返回的对象通过适当的转换器转换为指定的格式之后，写入到response对象的body区，通常用来返回JSON数据或者是XML数据。

@ExceptionHandler(Exception.class)
@ResponseStatus(HttpStatus.OK)
@ResponseBody
public Object handleException(HttpServletRequest request, Exception ex) {
CommonReturnType commonReturnType = new CommonReturnType();
commonReturnType.setStatus("fail");
commonReturnType.setData(ex);
return commonReturnType;
}

这样返回的其实是BussinessException异常类的反序列化的json方式

接下来这么做：

因为异常处理类是所有controller都想使用的公用逻辑，所以要改成面向对象的方式，独立出一个BaseController用来存放这个逻辑，别的类继承BaseController即可，代码如下：

package com.miaoshaproject.controller;
public class BaseController {
public static final String CONTENT\_TYPE\_FORMED = "application/x-www-form-urlencoded";
/\*\*
\* SpringBoot 提供了
\* 定义exceptionhandler解决未被controller层吸收的exception
\* 根据OOP方式, 把公共的类从UserController, 提取到基类 BaseController 里, 然后让 UserController 去继承 BaseController
\*/
@ExceptionHandler(Exception.class)
@ResponseStatus(HttpStatus.OK)
@ResponseBody
public Object handleException(HttpServletRequest request, Exception ex) {
Map<String, Object> responseData = new HashMap<>(); //提出公共代码
//CommonReturnType commonReturnType = new CommonReturnType(); //可简化
//commonReturnType.setStatus("fail"); //可简化
//commonReturnType.setData(ex);
/\*\*
\* 由于 返回的json的data中的ex的错误信息, 是 stackTrace + BusinessException,
\* 从 UserController.getUser() 也可知抛出的是 BusinessException
\* 故强转 BusinessException， 取出 errCode 和 errMsg，并塞回 commonReturnType.setData()
\*/
if (ex instanceof BusinessException) {
BusinessException businessException = (BusinessException) ex;
responseData.put("errCode", businessException.getErrCode());
responseData.put("errMsg", businessException.getErrMsg());
//commonReturnType.setData(responseData);//可简化
} else {
responseData.put("errCode", EmBusinessError.UNKNOW\_ERROR.getErrCode());
responseData.put("errMsg", EmBusinessError.UNKNOW\_ERROR.getErrMsg());
}
return CommonReturnType.create(responseData,"fail");
}
}

总结：

首先定义了一个CommonReturnType能够用status和data返回所有的json序列化方式的固定对象供前端解析使用，摒弃掉使用Http的status状态码+内嵌tomcat自带error页的方式。

然后定义一个BussinessException的方式统一管理我们所要使用的错误码

然后再BaseController里定义通用的exceptionhandler来解决未被controller层吸收的exception，使用errCode和errMsg的统一定义方式吃掉内部不可预知的异常，

**3-6用户模型管理--otp验证码获取**

用户信息管理流程：

1、otp短信获取

2、opt注册用户

3、用户手机登录

otp实际上就是个手机短信验证码，把用户手机号作为注册信息

使用httpsession绑定用户手机号和otpcode

如何获取httpsession，只要获取到HttpServletRquest对象即可得到httpsession，通过bean的方式注入进来：

@Autowired
private HttpServletRequest httpServletRequest;

以bean的形式注入，说明是个httpServletRequest对象是个单例模式，单例怎么支持一个request支持多个并发访问呢？

通过Spring bean包装的HttpServletRequest本质是个proxy，内部拥有threadlocal方式的map，去让用户在每个线程中处理自己所对应的request，并且有threadlocal清除的机制，因此可以放心使用：httpServletRequest就是对应当前用户的对应的http请求

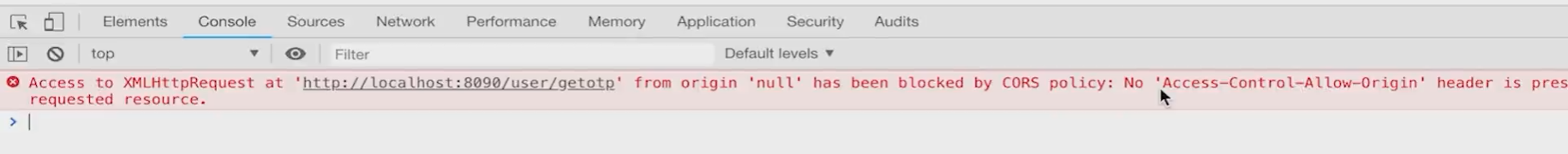
通过下面方法把session put到对应的telphone和otpCode

//使用httpsession绑定opt与telephone
request.getSession().setAttribute(telephone, otpCode);

通过这个方法就可以让用户telphone和otpCode做关联

**3-8**

如果前端出现这个错误，说明是跨域请求的错误



Ajax启动的这个html的域对应的是本地文件，但是请求的服务器是localhost这么一个域名，虽然请求能正确的到达服务端，服务端也能正确返回，但是Ajax的请求回调认定他们俩的域不同，所以说他们俩的回调是不安全的，会报错，且不能走到Ajax请求的accessblock里面

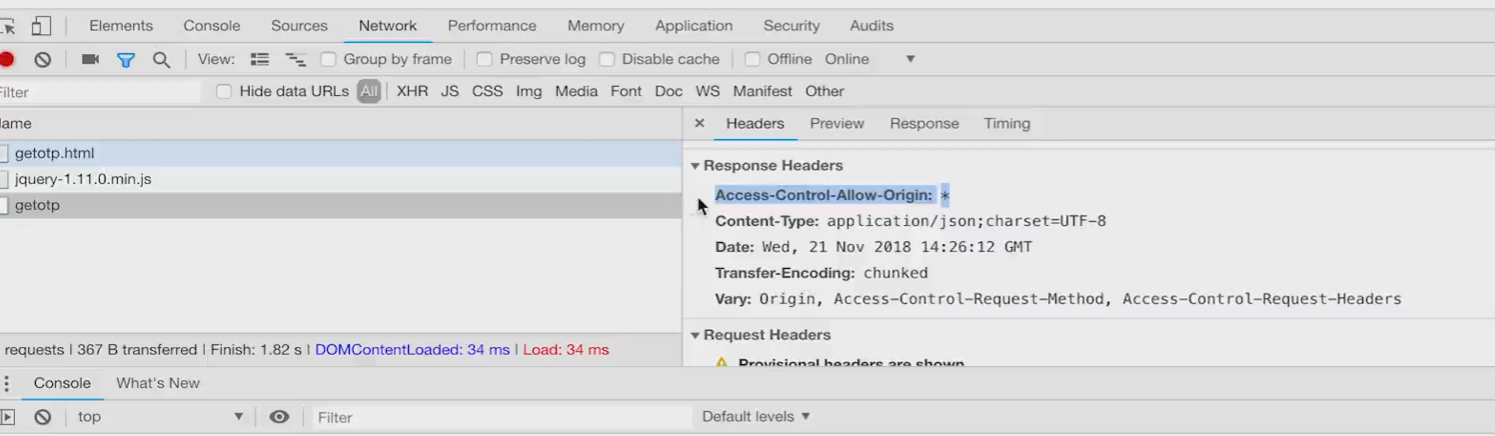
处理：

在springboot中，处理Ajax跨域请求的方式，只要让response时刻能够返回Access-Controller-Allow-Origin这个header为所有的域，也就是星号\*即可，springboot提供了一个简单的注解来解决这个问题@CrossOrigin

@CrossOrigin
public class UserController extends BaseController {
}

这个注解就可以让我们完成所有的springboot对应返回web请求当中加上跨域allow-header对应的标签

效果：



**3-10用户模型管理-用户注册功能实现**

在UserServiceImpl中实现一个用户注册接口

@Override
public void register(UserModel userModel) throws BusinessException {
if (userModel == null) {
throw new BusinessException(EmBusinessError.PARAMETER\_VALIDATION\_ERROR);
}
//org.apache.commons
//if (StringUtils.isEmpty(userModel.getName())
// || userModel.getGender() == null
// || userModel.getAge() == null
// || StringUtils.isEmpty(userModel.getTelephone())) {
// throw new BusinessException(EmBusinessError.PARAMETER\_VALIDATION\_ERROR);
//}
/\*\*
\* 校验过于繁琐, pom引入hibernate的 validator, 并新建 validator.ValidationResult + validator.ValidatorImpl
\*/
ValidationResult result = validator.validate(userModel);
if (result.isHasErrors()) {
throw new BusinessException(EmBusinessError.PARAMETER\_VALIDATION\_ERROR, result.getErrMsg());
}
// 实现model->dataobject方法
UserDO userDO = convertFromModel(userModel);
// 为什么使用 insertSelective 而不是 insert 方法?
// insertSelective, 如果字段为null则不更新这个字段, 讲了一段设计数据库的经验, 没大听懂.
try {
userDOMapper.insertSelective(userDO);
} catch (DuplicateKeyException ex) {
throw new BusinessException(EmBusinessError.PARAMETER\_VALIDATION\_ERROR, "手机号已重复注册 ");
}
//获取userid, 以便存入userPasswordDO
userModel.setId(userDO.getId());
UserPasswordDO userPasswordDO = convertPasswordFromModel(userModel);
userPasswordDOMapper.insertSelective(userPasswordDO);
}

定义convertFromModel方法把UserModel中的属性导入到UserDO中

// 实现model->dataobject方法

private UserDO convertFromModel(UserModel userModel) {
if (userModel == null) {
return null;
}
UserDO userDO = new UserDO();
BeanUtils.copyProperties(userModel, userDO);
return userDO;
}

使用这个方法

// 实现model->dataobject方法
UserDO userDO = convertFromModel(userModel);
// 为什么使用 insertSelective 而不是 insert 方法?
// insertSelective, 如果字段为null则不更新这个字段, 讲了一段设计数据库的经验, 没大听懂.
userDOMapper.insertSelective(userDO);

为什么使用insertSelective()方法？

原因：它会首先判断对应的字段在dataobject中是否为null，如果不为null就执行insert操作，如果为null就跳过（未定义，不覆盖），就不insert这个字段，不insert这个字段的意义是：完全依赖于数据库，数据库提供什么样的默认值，这里就提供什么样的默认值。在update操作时尤其有用，字段为null的话就不更新这个字段，

如果使用insert方法的话，如果字段为null，就会用null覆盖掉数据库中的默认值

好处：在数据库设计的时候，尽量避免使用null字段，设置空字符串作为默认值，

比如当要求用户必须有手机号的时候，选择not null，并且指定其为唯一索引，如果选择可以为null的话，唯一索引就会失效，

定义convertPasswordFromModel方法把UserModel中的属性导入到UserPasswordDO中

private UserPasswordDO convertPasswordFromModel(UserModel userModel) {
if (userModel == null) {
return null;
}
UserPasswordDO userPasswordDO = new UserPasswordDO();
userPasswordDO.setencryptPassword(userModel.getencryptPassword());
userPasswordDO.setUserId(userModel.getId());
return userPasswordDO;
}

给用户注册方法加一个@Transactional注解，表明是一个事务

在UserController中的register方法中加入注册流程：

// 用户注册接口
@RequestMapping(
value = "/register",
method = {RequestMethod.POST},
consumes = {CONTENT\_TYPE\_FORMED}
)
@ResponseBody
public CommonReturnType register(@RequestParam(name = "telephone") String telephone,
@RequestParam(name = "otpCode") String otpCode,
@RequestParam(name = "name") String name, @RequestParam(name = "age") Integer age,
@RequestParam(name = "gender") Integer gender,
@RequestParam(name = "password") String password,
HttpServletRequest request)
throws BusinessException, UnsupportedEncodingException, NoSuchAlgorithmException {
// 1验证 手机号 和 对应的optcode 相符
/\*\*
\* inSessionOtpCode 实际使用中为 null , 为什么呢?
\* 因为＠CrossOrigin　对ajax 无法对session共享
\* 看 ＠CrossOrigin 源码 --> boolean DEFAULT\_ALLOW\_CREDENTIALS = false;
\* DEFAULT\_ALLOW\_CREDENTIALS = ture, 需要配合前端设置xhrFields授信后使得跨域session共享
\*/
String inSessionOtpCode = (String) request.getSession().getAttribute(telephone);
if (!com.alibaba.druid.util.StringUtils.equals(otpCode, inSessionOtpCode)) {
throw new BusinessException(EmBusinessError.PARAMETER\_VALIDATION\_ERROR, "短信验证码不正确");
}
// 2用户注册流程
UserModel userModel = new UserModel();
userModel.setName(name);
// int → string → byte
userModel.setGender(new Byte(String.valueOf(gender)));
userModel.setAge(age);
userModel.setTelephone(telephone);
userModel.setRegisterMode("byphone");
// 16位MD5 MD5Encoder.encode(password.getBytes()) --> EncodeByMd5()
userModel.setencryptPassword(this.EncodeByMd5(password));
userService.register(userModel);
// 将登陆凭证加入到用户登陆成功的session内, 注册等同于登录, 个人添加
request.getSession().setAttribute("IS\_LOGIN", true);
request.getSession().setAttribute("LOGIN\_USER", userModel);
return CommonReturnType.create(null);
}

在用户注册流程中，密码使用md5做加密：

// 16位MD5 MD5Encoder.encode(password.getBytes()) --> EncodeByMd5()
userModel.setencryptPassword(this.EncodeByMd5(password));

上面的com.alibaba.druid.util.StringUtils.equals方法实际上已经重写过，是下面这样的：

有一个判空条件

第一个equals(String a, String b)方法为我们重写的equals方法

内部的a.equals(b)为为String类里的equals方法

public static boolean equals(String a, String b) {
if (a == null) {
return b == null;
} else {
return a.equals(b);
}
}

**3-11**

跨域请求：

上面的@CrossOrign对于Ajax来说无法做到跨域共享，只有一个@CrossOrign无法做到session共享

解决方案：

指定@CrossOrign的范围

即allowCredentials = "true" allowedHeaders = "\*"

DEFAULT\_ALLOW\_CREDENTIALS = true需要配合前端设置xhrFields授信后使得跨域session共享，xhrFields的意思是说允许跨域授信的请求，以使session变为跨域可授信

xhrFields:{withCredentials:true}

只有通过上面的设置才能使后端对应的allowCredentials = "true" allowedHeaders = "\*" 遥相呼应

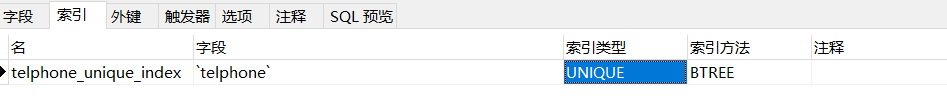
@CrossOrigin(
allowCredentials = "true",
allowedHeaders = "\*"
)
public class UserController extends BaseController {
}

MD5加密的问题：

JDK自带的MD5方法（MD5Encoder.encode()）只支持16位的md5，所以要自己写个加密方法

public String EncodeByMd5(String str) throws NoSuchAlgorithmException, UnsupportedEncodingException {
// 确定一个计算方法
MessageDigest md5 = MessageDigest.getInstance("MD5");
BASE64Encoder base64Encoder = new BASE64Encoder();
// 加密字符串
String newStr = base64Encoder.encode(md5.digest(str.getBytes("utf-8")));
return newStr;
}

为了防止重复注册，把telphone设置为唯一索引



**3-12 用户登录功能**

需要一个用户登录方法

@RequestMapping(value = "/login", method = {RequestMethod.POST}, consumes = {CONTENT\_TYPE\_FORMED})
@ResponseBody
// 用户登录接口
public CommonReturnType login(@RequestParam(name = "telephone") String telephone,
@RequestParam(name = "password") String password,
HttpServletRequest request)
throws BusinessException, UnsupportedEncodingException, NoSuchAlgorithmException {
}

首先：入参校验，判断手机号和密码是否为空

// 入参检验
if (org.apache.commons.lang3.StringUtils.isEmpty(telephone) || StringUtils.isEmpty(password)) {
throw new BusinessException(EmBusinessError.PARAMETER\_VALIDATION\_ERROR, "手机号或者密码为空");
}

然后：校验用户登录是否合法：

1、在UserService接口中定义validataLogin(String telephone, String encryptPassword)这个抽象方法：

/\*\*
\* @param telephone 用户注册手机
\* @param encryptPassword 用户加密后的密码
\* @throws BusinessException
\*/
UserModel validataLogin(String telephone, String encryptPassword) throws BusinessException;

2、在UserServiceImpl中实现这个validataLogin(String telephone, String encryptPassword)方法：

@Override
public UserModel validataLogin(String telephone, String encryptPassword) throws BusinessException {
//1、通过用户的手机 获取用户信息
UserDO userDO = userDOMapper.selectByTelephone(telephone);
if (userDO == null) {
throw new BusinessException(EmBusinessError.USER\_LOGIN\_FAIL);
}
UserPasswordDO userPasswordDO = userPasswordDOMapper.selectByUserId(userDO.getId());
UserModel userModel = convertFromDataObject(userDO, userPasswordDO);
//2、比对用户信息内加密的密码是否和传输进来的密码相匹配
if (!StringUtils.equals(encryptPassword, userModel.getencryptPassword())) {
throw new BusinessException(EmBusinessError.USER\_LOGIN\_FAIL);
}
return userModel;
}

如果要实现上面的第一个要求

// 在UserDOMapper.xml中写一个通过手机号码获取用户信息的方法，即selectByTelephone

<!-- 通过 手机 获取用户信息-->
<select id="selectByTelephone" parameterType="java.lang.String" resultMap="BaseResultMap">
<!--
WARNING - @mbg.generated
This element is automatically generated by MyBatis Generator, do not modify.
This element was generated on Mon Jun 07 05:44:13 CST 2021.
-->
select
<include refid="Base\_Column\_List"/>
from user\_info
where telephone = #{telephone ,jdbcType=VARCHAR}
</select>

然后在UserDOMapper.java中写一个映射

UserDO selectByTelephone(String telephone);

在UserController中的login方法中加入下面模块 ：

这里使用单点session登录，将登陆凭证加入到用户登陆成功的session内，"IS\_LOGIN"是登录标识，如果用户的会话标识里有这个登录标识说明登录成功了

// 用户登录服务, 校验 用户登录是否合法
UserModel userModel = userService.validataLogin(telephone, this.EncodeByMd5(password));
// 将登陆凭证加入到用户登陆成功的session内
// "IS\_LOGIN"是登录标识，如果session里有这个登录标识说明登录成功了
request.getSession().setAttribute("IS\_LOGIN", true);
// 如果用户登录成功，把userModel返回给controller层
// 如果登陆成功，就把userModel放入到用户session里
request.getSession().setAttribute("LOGIN\_USER", userModel);
// 然后返回给前端一个正确的信息
return CommonReturnType.create(null);

**3-13 优化校验规则**

引入依赖

做一个validator的封装

新建一个valdator的包目录，在里面建立一个ValidationResult的类

package com.miaoshaproject.validator;
public class ValidationResult {
// 默认值 防止空指针异常
//校验结果是否有错
private boolean hasErrors = false;
//存放错误信息的map
private Map<String, String> errMsgMap = new HashMap<>();
public boolean isHasErrors() {
return hasErrors;
}
public void setHasErrors(boolean hasErrors) {
this.hasErrors = hasErrors;
}
public Map<String, String> getErrMsgMap() {
return errMsgMap;
}
public void setErrMsgMap(Map<String, String> errMsgMap) {
this.errMsgMap = errMsgMap;
}
//使用 通用的 通过格式化字符串信息 获取 错误结果的 msg 方法
public String getErrMsg() {
return StringUtils.join(errMsgMap.values().toArray(), ",");
}
}

ValidationResult类的功能：

在应用程序和validator校验逻辑之间使用ValidationResult做对接

创建一个ValidatorImpl类并实现InitializingBean接口，实现接口中的 afterPropertiesSet()方法

@Component
public class ValidatorImpl implements InitializingBean {
@Override
public void afterPropertiesSet() throws Exception {
//用于 包装 javax.validation.Validator 接口
//将hibernate validator通过工厂的初始化方式 使其 实例化
this.validator = Validation.buildDefaultValidatorFactory().getValidator();
}
}

用@Component注解，表明ValidatorImpl是spring的一个bean，在进行类扫描的时候会扫描到它，

使用afterPropertiesSet()作用：当spring的bean初始化完成之后，会回调ValidatorImpl的afterPropertiesSet()方法，在这个方法里做这些事情：

在ValidatorImpl类中定义：

private Validator validator;

这个validator是通过包装 javax.validation.Validator 接口而实现的validator工具

然后对方法进行如下处理：

@Override
public void afterPropertiesSet() throws Exception {
//用于 包装 javax.validation.Validator 接口
//将hibernate validator通过工厂的初始化方式 使其 实例化
this.validator = Validation.buildDefaultValidatorFactory().getValidator();
}

上面是通过通过工厂的初始化方式实例化了一个校验器，然后实现校验方法并返回校验结果：

//实现校验方法 并 返回校验结果
public ValidationResult validate(Object bean) {
final ValidationResult result = new ValidationResult();
// 若对应的bean 里边的一些参数规则, 有违背了 ????? constraintViolationSet里边就有这个值
Set<ConstraintViolation<Object>> constraintViolationSet = validator.validate(bean);
if (constraintViolationSet.size() > 0) {
//有错误
result.setHasErrors(true);
constraintViolationSet.forEach(constraintViolation -> {
String errMsg = constraintViolation.getMessage();
String propertyName = constraintViolation.getPropertyPath().toString();
result.getErrMsgMap().put(propertyName, errMsg);
});
}
return result;
}

如何使用？

UserServiceImpl中加了很多对UserModel的校验，于是对UserModel中的属性做如下处理：

加注解：@NotNull、@NotEmpty、@NotBlank

@NotNull：不能为null，即不能不填，否则报错

例如：@NotNull(message = "性别不能不填写")，为null的话会报message的错误

@NotBlank：不能为null和空字符串，否则报错

例如：@NotBlank(message = "用户名不能为空")

@NotEmpty：

@Min(value = 0, message = "年龄必须大于0")

@Max(value = 150, message = "年龄必须小于150")

这俩@Min 和 @Max注解表明了所标注属性的最小值和最大值

/\*\*
\* \\@NotNull、@NotEmpty、@NotBlank的区别
\* 大致区别如下：@NotEmpty用在集合类上面 @NotBlank 用在String上面 @NotNull 用在基本类型上 链接: https://www.jianshu.com/p/c9e71b241daa
\*/
public class UserModel {
private Integer id;
@NotBlank(message = "用户名不能为空")
private String name;
@NotNull(message = "性别不能不填写")
private Byte gender;
@NotNull(message = "年龄不能不填写")
@Min(value = 0, message = "年龄必须大于0")
@Max(value = 150, message = "年龄必须小于150")
private Integer age;
@NotBlank(message = "手机号不能为空")
private String telephone;
private String registerMode;
private String thirdPartyId;
/\*\*
\* encryptPassword 由于数据模型层的关系(加密)设置在两张不同的表里面,
\* 但是对 java \*领域模型\* 的对象的概念来说,encryptPassword 就是属于 UserModel 的.
\* 因此 UserModel 才是真正意义上, 处理业务逻辑的核心的模型, 而dataObject仅仅只是一个对数据库的映射
\* 所以我们重新定义一下 UserService 的返回值, 是一个 UserModel的对象
\*/
@NotBlank(message = "密码不能为空")
private String encryptPassword;
public String getencryptPassword() {
return encryptPassword;
}
public void setencryptPassword(String encryptPassword) {
this.encryptPassword = encryptPassword;
}
public Integer getId() {
return id;
}
public void setId(Integer id) {
this.id = id;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public Byte getGender() {
return gender;
}
public void setGender(Byte gender) {
this.gender = gender;
}
public Integer getAge() {
return age;
}
public void setAge(Integer age) {
this.age = age;
}
public String getTelephone() {
return telephone;
}
public void setTelephone(String telephone) {
this.telephone = telephone;
}
public String getRegisterMode() {
return registerMode;
}
public void setRegisterMode(String registerMode) {
this.registerMode = registerMode;
}
public String getThirdPartyId() {
return thirdPartyId;
}
public void setThirdPartyId(String thirdPartyId) {
this.thirdPartyId = thirdPartyId;
}
}

然后在UserServiceImpl中引入validator

@Autowired
private ValidatorImpl validator;

然后下面代码即可注释掉：

/\*if (StringUtils.isEmpty(userModel.getName())
|| userModel.getGender() == null
|| userModel.getAge() == null
|| StringUtils.isEmpty(userModel.getTelephone())) {
throw new BusinessException(EmBusinessError.PARAMETER\_VALIDATION\_ERROR);
}\*/

以下面的代码来替换：

/\*\*
\* 校验过于繁琐, pom引入hibernate的 validator, 并新建 validator.ValidationResult + validator.ValidatorImpl
\* 以
\*/
ValidationResult result = validator.validate(userModel);
if (result.isHasErrors()) {
throw new BusinessException(EmBusinessError.PARAMETER\_VALIDATION\_ERROR, result.getErrMsg());
}

**4-1 商品模型--商品创建01**

问题：先设计库还是先设计模型？

答：先设计领域模型

领域模型可能是有级联嵌套的一种方式，比如商品模型与销量挂钩，而销量是从订单的交易数量来获得，那么对应的商品模型就要有一个销量的字段，但是映射到数据库里面的话，这个销量字段是否要设计在商品的基础表上面，还是通过交易表去count出来呢？要具体行为具体分析

所以，在做任何一个业务功能之前，不要去动数据库，首先应该想好商品模型应该如何设计

1、新建一个商品的model：ItemModel

public class ItemModel {
private Integer id;
//商品名
@NotBlank(message = "商品名称不能为空")
private String title;
@NotNull(message = "商品价格不能为空")
@Min(value = 0, message = "商品价格必须大于0")
private BigDecimal price;
//商品的库存
@NotNull(message = "库存必须填写")
private Integer stock;
/\*\*
\* mysql数据库中, 有个 stock字段
\* //商品库存虽是一对一, 但后面考虑到商品库存是跟交易流水相关的, 就是说每次对商品表的操作就是对库存表的操作,
\* 以后会用到分库分表的一些策略, 以及一些数据问题, 那库存字段最好跟user\_password设计思想一样, 拆到另一张表里面,
\* 以便于我们以后做一些性能优化以及水平的拆分
\*/
@NotBlank(message = "商品描述信息不能为空")
private String Description;
/\*\*
\* 考虑一下sales表, 应该跟着哪一张表去走. 销量对应到商品模型之中, 它是属于当交易行为发生之后所产生的一个计数的累加,
\* 那么对应于这个sales, 是交易模型过来之后累加, 还是独立出另外一张表, 每次去求和呢? 暂且用于展示
\* 销量暂时放在item表里边, 当交易行为之后, 我们通过异步的方式, 给对应的item的销量值 +1, 而不会影响 下单主链路, 因此暂时把销量 放在这个位置.
\*
\*/
//商品的销量 销量不是创建的时候传进来的, 而是我们通过其他方式统计进来的，不在入参校验范围之内
private Integer sales;
//描述商品的图片
@NotBlank(message = "商品图片信息不能为空")
private String imgUrl;
//使用聚合模型, 如果 promoModel不为空, 则表示其拥有 还未结束的秒杀活动
private PromoModel promoModel;
public PromoModel getPromoModel() {
return promoModel;
}
public void setPromoModel(PromoModel promoModel) {
this.promoModel = promoModel;
}
public Integer getId() {
return id;
}
public void setId(Integer id) {
this.id = id;
}
public String getTitle() {
return title;
}
public void setTitle(String title) {
this.title = title;
}
public BigDecimal getPrice() {
return price;
}
public void setPrice(BigDecimal price) {
this.price = price;
}
public Integer getStock() {
return stock;
}
public void setStock(Integer stock) {
this.stock = stock;
}
public String getDescription() {
return Description;
}
public void setDescription(String description) {
Description = description;
}
public Integer getSales() {
return sales;
}
public void setSales(Integer sales) {
this.sales = sales;
}
public String getImgUrl() {
return imgUrl;
}
public void setImgUrl(String imgUrl) {
this.imgUrl = imgUrl;
}
}

然后开始设计数据库

对于库存stock字段，商品库存虽然与商品是一对一的关系，但是后面考虑到商品库存与交易流水相关，也就是说每次对商品表的操作实际上就是对库存表的操作，以后可能会用到一些分库分表的策略和数据问题，对应的库存字段最好和user\_password设计思想一样，拆到另一张表里，以便于我们做性能优化和水平拆分

这就是领域模型ItemModel对应的库存与商品虽然确实是一对一的关系，但是对应到数据库级别为了数据库的性能设计来考虑，就要把库存放置到另外一张表中

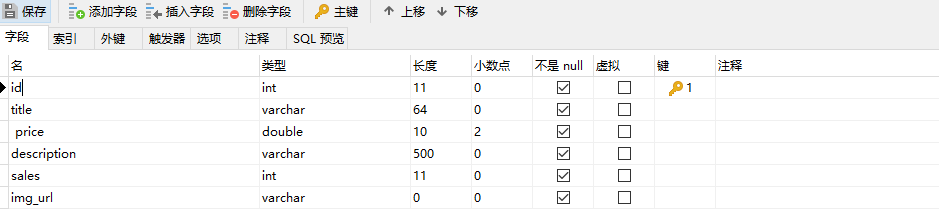
那么对于销量sales来说，仍然有和库存stock相同的问题，那么sales应该跟着哪一张表去走呢？

销量对应到商品模型之中，它属于当交易行为发生之后所产生的的计数的累加。

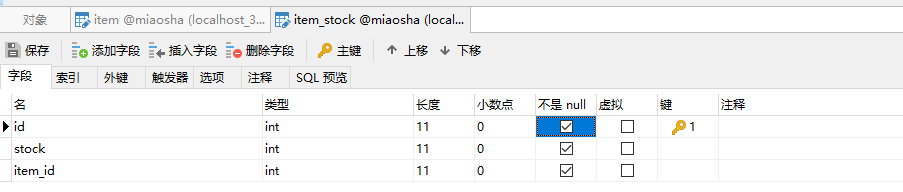
对应这个sales，考虑是在交易模型过来之后做累加，还是独立出另外一张表每次去求和呢？

在本项目中，销量暂时用于展示，所以把销量放在商品详情表中，在用户发生交易行为之后，通过异步的方式给对应的item的销量值+1，而不会影响下单主链路

设计商品表：item



设计库存表item\_stock：



2、使用mybatis-generator生成所需的mapperDAO文件以及dataobject对象

此时要去pom.xml文件中修改为false，禁止自动覆盖

<!--允许自动覆盖文件\_测试使用\_一般是约定好第一次生成, 以后手动添加-->
<overwrite>false</overwrite>

**4-2 商品模型--商品创建01**

新建ItemService层，维护下面的几个接口

public interface ItemService {
//创建商品
ItemModel createItemModel(ItemModel itemModel) throws BusinessException;
//商品列表浏览
List<ItemModel> listItems();
//商品详情浏览
ItemModel getItemById(Integer id);
}

新建ItemServiceImpl，实现ItemService接口，首先实现创建Item对象并能成功入库

@Override
//创建一个Item必须在一个事务当中
@Transactional
public ItemModel createItemModel(ItemModel itemModel) throws BusinessException {
//校验入参
ValidationResult result = validator.validate(itemModel);
if (result.isHasErrors()) {
throw new BusinessException(EmBusinessError.PARAMETER\_VALIDATION\_ERROR, result.getErrMsg());
}
//转化 itemmodel -> dataobject
ItemDO itemDO = this.convertItemDOFromItemModel(itemModel);
//写入数据库
itemDOMapper.insertSelective(itemDO);//itemDO写入到数据库
itemModel.setId(itemDO.getId());//itemDO的id就是itemModel的id
ItemStockDO itemStockDO = this.convertItemStockDOFromItemModel(itemModel);//把对应的itemModel转成itemStockDO
itemStockDOMapper.insertSelective(itemStockDO);//讲itemStockDO写入到数据库
//返回创建完成的对象
return this.getItemById(itemModel.getId());
}

需要导入如下：

@Autowired
private ValidatorImpl validator;//校验
@Autowired
private ItemDOMapper itemDOMapper;//把ItemDO对象写入库所需
@Autowired
private ItemStockDOMapper itemStockDOMapper;//把ItemStockDO对象写入库所需

还需要如下方法：

1、把ItemModel对象转换成ItemDO对象

private ItemDO convertItemDOFromItemModel(ItemModel itemModel) {
if (itemModel == null) {
return null;
}
ItemDO itemDO = new ItemDO();
BeanUtils.copyProperties(itemModel, itemDO);//itemModel和itemDo中的price类型不同，BeanUtils.copyProperties()方法不会拷贝类型不同的对象
itemDO.setPrice(itemModel.getPrice().doubleValue());//double传到前端可能有精度损失，所以itemModel里的price是BigDecimal类型的，在这里通过doubleValue转化成double类型
return itemDO;
}

2、把ItemModel转成ItemStockDO对象

private ItemStockDO convertItemStockDOFromItemModel(ItemModel itemModel) {
if (itemModel == null) {
return null;
}
ItemStockDO itemStockDO = new ItemStockDO();
itemStockDO.setItemId(itemModel.getId());//把itemModel的id赋值给itemStockDO的itemid
itemStockDO.setStock(itemModel.getStock());//把itemModel的stock赋值给itemStockDO的stock
return itemStockDO;
}

实现通过id获取ItemModel对象的方法

@Override
public ItemModel getItemById(Integer id) {
ItemDO itemDO = itemDOMapper.selectByPrimaryKey(id);
if (itemDO == null) {
return null;
}
//操作获取库存数量
//通过ItemId获取到ItemStockDO对象
ItemStockDO itemStockDO = itemStockDOMapper.selectByItemId(id);
//将 dataobject -> model
ItemModel itemModel = this.convertModelFromDataObject(itemDO, itemStockDO);
return itemModel;
}

实现上面代码需要封装如下方法：

通过ItemDO对象和ItemStockDO对象获得ItemModel对象

private ItemModel convertModelFromDataObject(ItemDO itemDO, ItemStockDO itemStockDO) {
ItemModel itemModel = new ItemModel();
BeanUtils.copyProperties(itemDO, itemModel);
itemModel.setPrice(new BigDecimal(itemDO.getPrice()));
itemModel.setStock(itemStockDO.getStock());
return itemModel;
}

搞定controller层，声明ItemController（在企业级应用里, 我们所要达到的目的很简单, 就是尽可能让 Controller层简单, 让Service层复杂。把我们服务的逻辑, 尽可能聚合在Service内部, 用于实现它内部的 流转 处理）

@Controller("item")
@RequestMapping("/item")
@CrossOrigin(
allowCredentials = "true",
allowedHeaders = "\*"
)
public class ItemController extends BaseController {
/\*\*
\* 在企业级应用里, 我们所要达到的目的很简单, 就是尽可能让 Controller层简单, 让Service层复杂
\* 得把我们的逻辑, 尽可能聚合在Service内部, 用于实现它内部的 流转 处理
\*/
@Autowired
private ItemService itemService;
@RequestMapping("/create")
@ResponseBody
//创建商品
public CommonReturnType createItem(
@RequestParam(name = "title") String title,
@RequestParam(name = "price") BigDecimal price,
@RequestParam(name = "stock") Integer stock,
@RequestParam(name = "description") String description,
@RequestParam(name = "imgUrl") String imgUrl
) throws BusinessException {
//封装service请求用来创建商品
ItemModel itemModel = new ItemModel();
itemModel.setTitle(title);
itemModel.setPrice(price);
itemModel.setStock(stock);
itemModel.setDescription(description);
itemModel.setImgUrl(imgUrl);
ItemModel itemModelForReturn = itemService.createItemModel(itemModel);
ItemVO itemVO = convertItemVOFromModel(itemModelForReturn);
return CommonReturnType.create(itemVO);
}
@RequestMapping(value = "/get", method = {RequestMethod.GET})
@ResponseBody
//商品详情页浏览
public CommonReturnType getItem(@RequestParam(name = "id") Integer id) {
ItemModel itemModel = itemService.getItemById(id);
ItemVO itemVO = this.convertItemVOFromModel(itemModel);
return CommonReturnType.create(itemVO);
}
@RequestMapping(value = "/list", method = {RequestMethod.GET})
@ResponseBody
//商品列表页面浏览
public CommonReturnType listItem() {
List<ItemModel> itemModelList = itemService.listItems();
//使用 stream api 将list内的itemModel转化为ItemVO
List<ItemVO> itemVOList = itemModelList.stream().map(itemModel -> {
ItemVO itemVO = this.convertItemVOFromModel(itemModel);
return itemVO;
}).collect(Collectors.toList());
return CommonReturnType.create(itemVOList);
}
//把ItemModel转换成ItemVO对象
private ItemVO convertItemVOFromModel(ItemModel itemModel) {
/\*\*
\* 企业级应用里面, 许多时候 VO层 的定义, 和 M层 的定义, 是完全不一样的.
\* 而且许多都会用到聚合操作. --> 比如说, M层 到 DO层, 已经用到了一些 比如库存字段是通过DO层聚合出来的.
\* 对于 VO层, 我们为了一些前端交互逻辑上的方便, 很多时候把 VO 定义得 M层更大. --> 比如说, 为了前端的性能考虑, 我们会聚合上活动价格信息, 等等.
\* 因此, 对应的一些用户请求, 在VO层, 会变得非常非常复杂, 分层是必须的.
\*/
if (itemModel == null) {
return null;
}
ItemVO itemVO = new ItemVO();
BeanUtils.copyProperties(itemModel, itemVO);
//添加 秒杀 逻辑
if (itemModel.getPromoModel() != null) {
//有 正在进行 或 即将进行的 秒杀活动
//ItemVO 则会 出现 static问题, 因为是根据类直接调用
itemVO.setPromoStatus(itemModel.getPromoModel().getStatus());
itemVO.setPromoId(itemModel.getPromoModel().getId());
// 注意是 DateTimeFormat 而不是 DateTimeFormatter
// 由于 DateTime 序列化后, 有很多多余东西,故改为String 并用 .toString(DateTimeFormat.forPattern("yyyy-MM-dd HH:mm:ss")) 格式化
itemVO.setStartDate(itemModel.getPromoModel().getStartDate().toString(DateTimeFormat.forPattern("yyyy-MM-dd HH:mm:ss")));
itemVO.setPromoPrice(itemModel.getPromoModel().getPromoItemPrice());
} else {
itemVO.setPromoStatus(0);
}
return itemVO;
}
}

声明ItemVO负责给前端用户展示 ：

public class ItemVO {
private Integer id;
//商品名
private String title;
private BigDecimal price;
//商品的库存
private Integer stock;
private String Description;
//商品的销量 销量不是创建的时候传进来的, 而是我们通过其他方式统计进来的
private Integer sales;
//描述商品的图片
private String imgUrl;
//记录 商品 是否在秒杀活动中, 以及对应的状态: 0->没有秒杀 1->秒杀待开始 2->秒杀进行
private Integer promoStatus;
//秒杀活动价格
private BigDecimal promoPrice;
//秒杀活动ID
private Integer promoId;
//秒杀活动开始键
private String startDate;
public Integer getPromoStatus() {
return promoStatus;
}
public void setPromoStatus(Integer promoStatus) {
this.promoStatus = promoStatus;
}
public BigDecimal getPromoPrice() {
return promoPrice;
}
public void setPromoPrice(BigDecimal promoPrice) {
this.promoPrice = promoPrice;
}
public Integer getPromoId() {
return promoId;
}
public void setPromoId(Integer promoId) {
this.promoId = promoId;
}
public String getStartDate() {
return startDate;
}
public void setStartDate(String startDate) {
this.startDate = startDate;
}
public Integer getId() {
return id;
}
public void setId(Integer id) {
this.id = id;
}
public String getTitle() {
return title;
}
public void setTitle(String title) {
this.title = title;
}
public BigDecimal getPrice() {
return price;
}
public void setPrice(BigDecimal price) {
this.price = price;
}
public Integer getStock() {
return stock;
}
public void setStock(Integer stock) {
this.stock = stock;
}
public String getDescription() {
return Description;
}
public void setDescription(String description) {
Description = description;
}
public Integer getSales() {
return sales;
}
public void setSales(Integer sales) {
this.sales = sales;
}
public String getImgUrl() {
return imgUrl;
}
public void setImgUrl(String imgUrl) {
this.imgUrl = imgUrl;
}
}

为什么要大费周章的做ItemModel转换为ItemVO呢？

很多时候ViewObject的定义和ModelObject的定义是完全不一样的，很多都会用 到聚合操作，比如Model层dataObject层已经用到了一些，比如说库存stock字段是通过dataobject层聚合出来的，对于viewobject层，为了一些前端交互逻辑的方便，很多时候把viewobject层定义的比model层更大，比如为了前端的性能考虑，需要聚合一些活动价格信息等，因此对应的一些用户请求在viewobject层变得很复杂，所以分层是必须的

**4-3商品详情**

在ItemController里添加一个方法：

使用的method = {RequestMethod.GET}，用get请求，保证幂等性

@RequestMapping(value = "/get", method = {RequestMethod.GET})
@ResponseBody
//商品详情页浏览
public CommonReturnType getItem(@RequestParam(name = "id") Integer id) {
ItemModel itemModel = itemService.getItemById(id);
ItemVO itemVO = this.convertItemVOFromModel(itemModel);
return CommonReturnType.create(itemVO);
}

**4-3商品列表**

ItemServiceImpl里实现获取商品列表页面方法：

@Override
public List<ItemModel> listItems() {
List<ItemDO> itemDOList = itemDOMapper.listItem();
List<ItemModel> itemModelList = itemDOList.stream().map(itemDO -> {
ItemStockDO itemStockDO = itemStockDOMapper.selectByItemId(itemDO.getId());
ItemModel itemModel = this.convertModelFromDataObject(itemDO, itemStockDO);
return itemModel;
}).collect(Collectors.toList());
return itemModelList;
}

进入ItemController，编写一个商品列表页面浏览

@RequestMapping(value = "/list", method = {RequestMethod.GET})
@ResponseBody
//商品列表页面浏览
public CommonReturnType listItem() {
List<ItemModel> itemModelList = itemService.listItems();
//使用 stream api 将list内的itemModel转化为ItemVO
List<ItemVO> itemVOList = itemModelList.stream().map(itemModel -> {
ItemVO itemVO = this.convertItemVOFromModel(itemModel);
return itemVO;
}).collect(Collectors.toList());
return CommonReturnType.create(itemVOList);
}

**5-1交易模型管理--交易模型创建**

仍然是先考虑交易模型，而不是数据库的操作

流程与上面一样，在model层文件里新建OrderModel，用来解决用户下单的交易模型，可以理解为一个订单就是交易模型的一部分

public class OrderModel {
//2018102100012828
//orderModel对应的流水号, 有自己的生成规则, 故不自增
private String id;
//用户id，即哪个用户去下单的
private Integer userId;
//若非空, 则表示是以秒杀商品方式下单
private Integer promoId;
//购买的 商品id
private Integer itemId;
//若promo非空, 则表示秒杀商品价格
//因为商品的总金额orderPrice = 购买数量amount \* 商品价格，但是商品价格是在变化的
//购买的 商品 的 下单时的 单价, 以后原商品价格变化, 订单中的这个价格不会变
//购买商品的单价=购买时商品的价格
//即使以后商品价格发生了变化，已经购买的商品的总金额不会发生变化
private BigDecimal itemPrice;
//购买数量
private Integer amount;
//购买金额
private BigDecimal orderPrice;
public Integer getPromoId() {
return promoId;
}
public void setPromoId(Integer promoId) {
this.promoId = promoId;
}
public BigDecimal getOrderPrice() {
return orderPrice;
}
public void setOrderPrice(BigDecimal orderPrice) {
this.orderPrice = orderPrice;
}
public String getId() {
return id;
}
public void setId(String id) {
this.id = id;
}
public Integer getUserId() {
return userId;
}
public void setUserId(Integer userId) {
this.userId = userId;
}
public Integer getItemId() {
return itemId;
}
public void setItemId(Integer itemId) {
this.itemId = itemId;
}
public BigDecimal getItemPrice() {
return itemPrice;
}
public void setItemPrice(BigDecimal itemPrice) {
this.itemPrice = itemPrice;
}
public Integer getAmount() {
return amount;
}
public void setAmount(Integer amount) {
this.amount = amount;
}
}

建立好OrderModel之后，就可以建立对应的数据库了

本项目只考虑用户一个订单只购买一种商品，且只支付一次费用

建好数据表之后，使用mybatis-generator来生成对应的mapper映射文件

然后新建OrderService，用来处理订单交易的service

/\*\*
\* 下单方式:
\* 方式1: 通过前端url上传过来秒杀活动id, 然后下单接口内校验对应id是否属于对应商品, 且 活动已开始
\* - 扩展性: 同一商品, 可能在不同 入口 有不同的 秒杀活动, 必须依赖前端判断是哪个秒杀活动
\*
\* 方式2: 直接在下单接口内 判断对应的商品是否存在秒杀活动, 若存在进行中的秒杀, 则以秒杀价格下单
\* - 如果普通下单, 也要校验的话, 对普通下单, 或者说 对日常性能 是很伤的.
\*
\* 方案: 采用方案1, 前端传送过来则校验, 否则认为是普通下单
\*/
//参数顺序也很重要
public interface OrderService {
OrderModel createOrder(Integer userId, Integer itemId, Integer amount) throws BusinessException;
}

再新建OrderServiceImpl来实现OrderService接口，实现createOrder()方法

OrderServiceImpl加@Service注解，createOrder()方法加@Transactional注解，保证创建订单在同一个事务之中

/\*\*
\* 电商减库存逻辑一般有两种: 落单减库存 or 支付减库存
\* 支付减库存, 下单时有就可以支付, 下单到支付的时间差内可能库存已空, 有超卖的风险. 对用户来说支付后没库存就要走退款流程.这对用户来说就很奇怪, 明明我支付成功了, 为什么要告诉我没货.
\* 这是商家为了保证用户交易率, 比如说有些用户会利用落单减库存去恶意的拍, 等着订单自动超时关单后再去做对应的释放操作,
\* ok这时商家为了让用户尽快购买, 使用支付减库存, 商家说我可以承担超卖的风险, 因为我有备货, 我在系统上导了100件商品,其实我的库存有120件, 超卖20件对我来说是可以承受的, 但是如果是更多的超卖, 会想尽一切办法去调货, 如果说再不行的话, 会把对应的钱 扣给对应的用户
\* 所以, 目前, 使用 落单减库存.
\*/
@Service
public class OrderServiceImpl implements OrderService {
@Autowired
private ItemService itemService;
@Autowired
private UserService userService;
@Autowired
private OrderDOMapper orderDOMapper;
@Autowired
private SequenceDOMapper sequenceDOMapper;
@Override
@Transactional
public OrderModel createOrder(Integer userId, Integer itemId, Integer amount) throws BusinessException {
// 1、校验下单状态 ①下单的商品 是否存在 ②用户是否合法 ③购买数量是否正确
// 此时可以看到 service 独立 的好处了, 也就是所有的 service 互相关联, 互相调用, 所有的逻辑都在service内部去做
ItemModel itemModel = itemService.getItemById(itemId);
if (itemModel == null) {
throw new BusinessException(EmBusinessError.PARAMETER\_VALIDATION\_ERROR, "商品信息不存在");
}
UserModel userModel = userService.getUserById(userId);
if (userModel == null) {
throw new BusinessException(EmBusinessError.PARAMETER\_VALIDATION\_ERROR, "用户信息不存在");
}
// 用户不能 买 <=0 件商品, 也不能买超过 99件商品
if (amount <= 0 || amount > 99) {
throw new BusinessException(EmBusinessError.PARAMETER\_VALIDATION\_ERROR, "数量信息不正确");
}
// 2、落单减库存
// ItemServiceImpl中实现了ItemService中的decreaseStock库存扣减方法，返回值是boolean类型，代表成功与否
boolean result = itemService.decreaseStock(itemId,amount);
if (!result) {
throw new BusinessException(EmBusinessError.STOCK\_NOT\_ENOUGH);
}
// 3、订单入库
OrderModel orderModel = new OrderModel();
//orderModel对应的流水号, 有自己的生成规则, 故不自增
//orderModel.setId(); <-- 少了这一步
orderModel.setUserId(userModel.getId());
orderModel.setItemId(itemId);
orderModel.setPromoId(promoId);
orderModel.setAmount(amount);
if (promoId != null) {
orderModel.setItemPrice(itemModel.getPromoModel().getPromoItemPrice());
} else {
orderModel.setItemPrice(itemModel.getPrice());
}
orderModel.setOrderPrice(orderModel.getItemPrice().multiply(new BigDecimal(amount)));
/\*\*
\* 生成交易流水号, 即订单号 --> 此种写法的问题 ①\\@Transactional 会在插入失败后回滚, 且下一个订单会使用失败的sequence
\* 及时插入失败, 也不应该使用重复的 sequence, 这是为了保证全局唯一性的策略
\* 解决方案 generateOrderNo() 开启新的事务, 因为是子事务, 所以会提前提交, 就算回滚了也不会拿到相同sequence
\*/
// 生成交易流水号, 即订单号
orderModel.setId(generateOrderNo());
OrderDO orderDO = convertFromOrderModel(orderModel);
//因为 OrderDO 对应的 ItemPrice 和 OrderPrice 是Double, 而 OderModel是BigDecimal, 所以要人肉处理一下 bigdecimal-->double
orderDO.setItemPrice(orderModel.getOrderPrice().doubleValue());
orderDO.setOrderPrice(orderModel.getOrderPrice().doubleValue());
orderDOMapper.insertSelective(orderDO);
// 加上商品的销量, 前端显示用刷新重新获取解决
itemService.increaseSales(itemId, amount);
// 4、返回前端
return orderModel;
}
@Transactional(propagation = Propagation.REQUIRES\_NEW)
public String generateOrderNo() {
//订单号有16位
StringBuilder stringBuilder = new StringBuilder();
//前8位为时间信息, 年月日
// java8 里边应该使用 LocalDateTime, Date打印不美观, 且SimpleDateFormat时, 线程不安全
LocalDateTime now = LocalDateTime.now();
String nowDate = now.format(DateTimeFormatter.ISO\_DATE).replace("-", "");
stringBuilder.append(nowDate);
//中间6位为自增序列
//获取当前sequence
int sequence = 0;
SequenceDO sequenceDO = sequenceDOMapper.getSequenceByName("order\_info");
sequence = sequenceDO.getCurrentValue();
//这次拿走了, 下一次就应该, +步长
sequenceDO.setCurrentValue(sequenceDO.getCurrentValue() + sequenceDO.getStep());
sequenceDOMapper.updateByPrimaryKeySelective(sequenceDO);
//拼接 凑足6位
//拼接 凑足6位 --> 此种写法的问题 ①超过了6位最大值, 异常
String sequenceStr = String.valueOf(sequence);
for (int i = 0; i < 6 - sequenceStr.length(); i++) {
stringBuilder.append(0);
}
stringBuilder.append(sequenceStr);
//最后2位, 为 \*\*分库分表位\*\* , 暂时写死
/\*\*
\* 例如, ↓ 拓展
\* Integer userId = 1000122;
\* userId % 100
\*/
stringBuilder.append("00");
return stringBuilder.toString();
}
private OrderDO convertFromOrderModel(OrderModel orderModel) {
if (orderModel == null) {
return null;
}
OrderDO orderDO = new OrderDO();
BeanUtils.copyProperties(orderModel, orderDO);
return orderDO;
}
}

对于减库存来说，一般分为两种：

1、落单减库存 ：在调用订单的createOrder下单之前，先把库存数量amount锁定给这个订单使用，如果锁库存失败的话，说明商品库存不足，认定下单失败。如果下单成功的话，锁库存的操作必然是成功的

2、支付减库存 ：落单的时候去看一下，如果库存还是有的，而不去锁库存，返回一个可以继续落单的消息，等下单支付成功之后。才去把库存扣减下来。但是，由于在落单的时候并没有锁库存，因此在做对应的支付减库存的时候无法保证支付成功之后对应的库存数量还有，可能会导致超卖。

对于减库存来说，要对库存表item\_stock进行修改，所以要加行锁，在ItemStockDOMapper.xml中实现如下SQL语句：

<update id="decreaseStock" >
update item\_stock
set stock = stock - #{amount}
where item\_id = #{item\_id} and stock >= #{amount}
</update>

再 在ItemStockDOMapper.java中加入如下代码：

int decreaseStock(@Param("item\_id")Integer itemId, @Param("amount")Integer amount);

再在ItemServiceImpl中实现decreaseStock方法：

@Override
@Transactional
public boolean decreaseStock(Integer itemId, Integer amount) throws BusinessException {
//加锁到 item\_stock, 此时能够看到 把stock抽出来的好处, 由于item常用于展示, 所以对 item\_stock表的操作, 不会影响到 item表的 展示
/\*\*
\* 新手程序员, 容易 写成2次sql, 一次取 stock, 再根据 amount< stock, 再update. 以下这么写可以在插入的同时, 检查插入条件
\* <update id="decreaseStock" > <-- src\main\resources\mapping\ItemStockDOMapper.xml
\* update item\_stock
\* set stock = stock - #{amount}
\* where item\_id = #{item\_id} and stock >= #{amount}
\* </update>
\*/
//这个affectedRow是影响的条目数，如果SQL语句中where后面的条件均满足，则受影响的条目数为1，如果条件不满足，则受影响的条目数为0
int affectedRow = itemStockDOMapper.decreaseStock(itemId,amount);
if (affectedRow > 0) {//where后面条件满足，说明更新库存成功
return true;
} else {//where后面条件不满足，更新失败
return false;
}
}

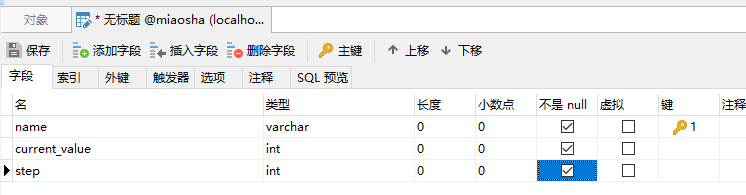
对于订单入库：先写如下方法：把OrderModel转换成OrderDO

private OrderDO convertFromOrderModel(OrderModel orderModel) {
if (orderModel == null) {
return null;
}
OrderDO orderDO = new OrderDO();
BeanUtils.copyProperties(orderModel, orderDO);
return orderDO;
}

OrderModel的交易流水号id有专门的生成规则，所以写一个按照一定规则生成订单号的方法：

@Transactional(propagation = Propagation.REQUIRES\_NEW)
public String generateOrderNo() {
//订单号有16位
StringBuilder stringBuilder = new StringBuilder();
//前8位为时间信息, 年月日
// java8 里边应该使用 LocalDateTime, Date打印不美观, 且SimpleDateFormat时, 线程不安全
LocalDateTime now = LocalDateTime.now();
String nowDate = now.format(DateTimeFormatter.ISO\_DATE).replace("-", "");
stringBuilder.append(nowDate);
//中间6位为自增序列
//获取当前sequence
int sequence = 0;
SequenceDO sequenceDO = sequenceDOMapper.getSequenceByName("order\_info");
sequence = sequenceDO.getCurrentValue();
//这次拿走了, 下一次就应该, +步长
sequenceDO.setCurrentValue(sequenceDO.getCurrentValue() + sequenceDO.getStep());
sequenceDOMapper.updateByPrimaryKeySelective(sequenceDO);
//拼接 凑足6位
//拼接 凑足6位 --> 此种写法的问题 ①超过了6位最大值, 异常
String sequenceStr = String.valueOf(sequence);
for (int i = 0; i < 6 - sequenceStr.length(); i++) {
stringBuilder.append(0);
}
stringBuilder.append(sequenceStr);
//最后2位, 为 \*\*分库分表位\*\* , 暂时写死
/\*\*
\* 例如, ↓ 拓展
\* Integer userId = 1000122;
\* userId % 100
\*/
stringBuilder.append("00");
return stringBuilder.toString();
}

创建一张sequence\_info表



这张表的意义是：先初始化一些sequence，sequence的初始值为0，每次从sequence\_info表中获取数据的时候，就给它加对应的步长step

然后使用mybatis-generator来生成sequence\_info所对应的mapper映射文件

对SequenceDOMapper.xml进行一些修改：这里我们不需要selectByPrimaryKey方法，而是需要getSequenceByName方法：

这里是要上锁的，因为每次获取sequence就要对sequence进行更新，需要使用select……for update方式上锁

<!-- for update是一种行级锁，又叫排它锁。 -->
<select id="getSequenceByName" parameterType="java.lang.String" resultMap="BaseResultMap">
select
<include refid="Base\_Column\_List" />
from sequence\_info
where name = #{name,jdbcType=VARCHAR} for update
</select>

然后再SequenceDOMapper.java中加入对应的抽象方法：

SequenceDO getSequenceByName(String name);

然后再OrderServiceImpl中引入

@Autowired
private SequenceDOMapper sequenceDOMapper;

然后再写获取当前sequence的方法：

//中间6位为自增序列
//获取当前sequence
int sequence = 0;
SequenceDO sequenceDO = sequenceDOMapper.getSequenceByName("order\_info");
sequence = sequenceDO.getCurrentValue();
//拿到sequence之后，生成新的sequence
//这次拿走了, 下一次就应该, +步长
sequenceDO.setCurrentValue(sequenceDO.getCurrentValue() + sequenceDO.getStep());
//更新sequence
sequenceDOMapper.updateByPrimaryKeySelective(sequenceDO);
//拼接 凑足6位 --> 此种写法的问题 ①超过了6位最大值, 异常
String sequenceStr = String.valueOf(sequence);
for (int i = 0; i < 6 - sequenceStr.length(); i++) {
stringBuilder.append(0);
}
stringBuilder.append(sequenceStr);

接下来就是拼接了。

总结：

1、首先根据今天的日期获取前八位

2、从mysql数据库里拿出之前预设好的sequence信息，通过原sequence信息和步长step拼接，凑足6位，不足的补0

3、再拼接上分库分表位，得到最终的订单号，返回。

问题：

createOrder方法已经有@Transactional注解了，我们写的生成订单号的方法generateOrderNo()在createOrder方法内，且也有@Transactional注解，那么如果创建订单失败的话，generateOrderNo()也要回滚，就会重复使用之前的序列号，后面的订单可能就会使用前面失败的订单的订单号，这样无法保证订单的全局唯一性

解决：

在@Transcational注解内，通过设置Propagation来解决

@Transactional(propagation = Propagation.REQUIRES\_NEW)
public String generateOrderNo() {
}

意思是，只要执行完这个genrerateOrderNo这个代码块，外部的事务成功与否，这个代码块对应的事务都会提交，对应的sequence都已经被更新保存了

接下来新建一个OrderController

@Controller("order")
@RequestMapping("/order")
@CrossOrigin(
allowCredentials = "true",
allowedHeaders = "\*"
)
public class OrderController extends BaseController {
@Autowired
private OrderService orderService;
@Autowired
private HttpServletRequest httpServletRequest;
@RequestMapping(value = "/createOrder", method = {RequestMethod.POST}, consumes = {CONTENT\_TYPE\_FORMED})
@ResponseBody
//封装下单请求
public CommonReturnType createOrder(@RequestParam(name = "itemId") Integer itemId,
@RequestParam(name = "promoId", required = false) Integer promoId, // 如果不传则认定为平销价格
@RequestParam(name = "amount") Integer amount) throws BusinessException {
//获取用户的登录信息
Boolean isLogin = (Boolean)this.httpServletRequest.getSession().getAttribute("IS\_LOGIN");
if (isLogin == null || !isLogin.booleanValue()) {
throw new BusinessException(EmBusinessError.USER\_NOT\_LOGIN, "用户未登录, 不能下单");
}
UserModel userModel = (UserModel) this.httpServletRequest.getSession().getAttribute("LOGIN\_USER");
OrderModel orderModel = orderService.createOrder(userModel.getId(), itemId, promoId, amount);
return CommonReturnType.create(null);
}
}

一旦用户下单成功，对应的商品销量就要增加

所以在ItemService接口中加一个抽象方法：

//商品销量增加
void increaseSales(Integer itemId, Integer amount) throws BusinessException;

商品的销量增加意味着库存扣减成功且下单成功

在本项目中，只要落单成功，商品销量就会增加

在ItemDOMapper.xml中加一条SQL语句：

<update id="increaseSales">
update item
set sales = sales + #{amount}
where id = #{id,jdbcType=INTEGER}
</update>

再在ItemDOMapper.java文件中加一个

int increaseSales(@Param("id") Integer id, @Param("amount") Integer amount);

所以再在ItemServiceImpl中实现这个ItemService接口中的increaseSales方法：

@Override
@Transactional
public void increaseSales(Integer itemId, Integer amount) throws BusinessException {
itemDOMapper.increaseSales(itemId, amount);
}