**企业做项目流程**

需求设计 ==> 设计 （概要设计，详细设计） ==> 技术选型 ==>

初始化/引入需要的技术 ==> 写Demo ==> 写代码（实现业务逻辑） ==> 测试 （单元测试） ==>

代码提交/代码评审 ==> 部署 ==> 发布

**需求分析**

登录 ， 注册

用户管理

用户校验

**技术选型**

**前端**

主要运用阿里 Ant Design 生态：

HTML + CSS + JavaScript 三件套

React 开发框架

Ant Design Pro 项目模板

Ant Design 端 组件库

Umi 开发框架

Umi Request 请求库

**后端**

Java 编程语言

Spring + SpringMVC + SpringBoot 框架

MyBatis + MyBatis Plus 数据访问框架

MySQL 数据库

jUnit 单元测试库

**部署**

单机

Nginx

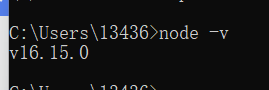
容器

**上**

**前端初始化项目**

**下载 node.js**

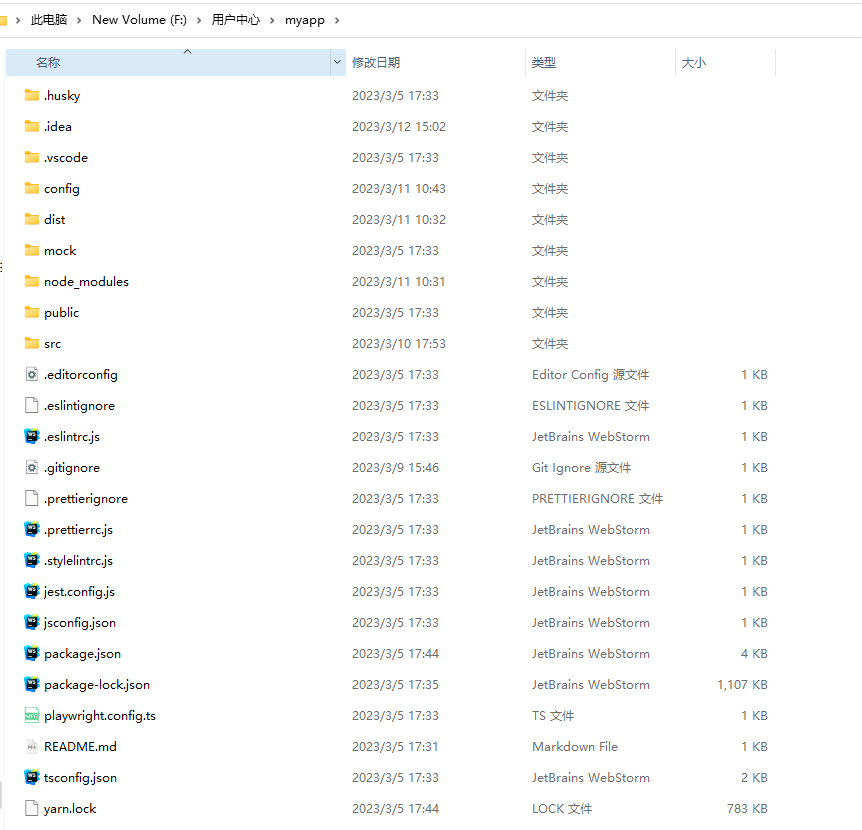
(3条消息) node.js安装及环境配置超详细教程【Windows系统安装包方式】\_nodejs安装及环境配置\_明金同学的博客-CSDN博客



**Ant Design Pro**



npm install --global yarn



**3.启动项目**



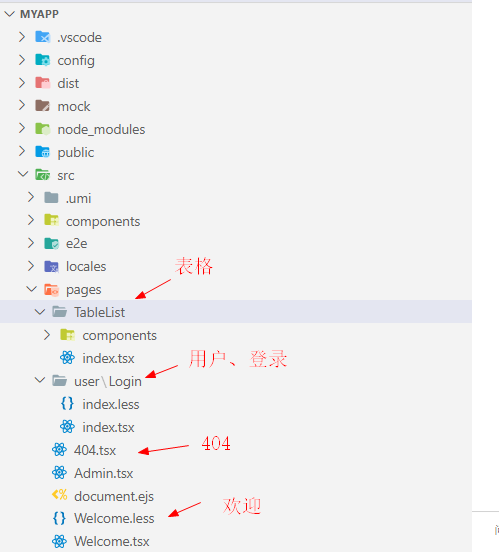
初始:

用户名: admin

密码： ant.design

**4. 安装 Umi UI**

yarn add @umijs/preset-ui -D



**5.项目瘦身**

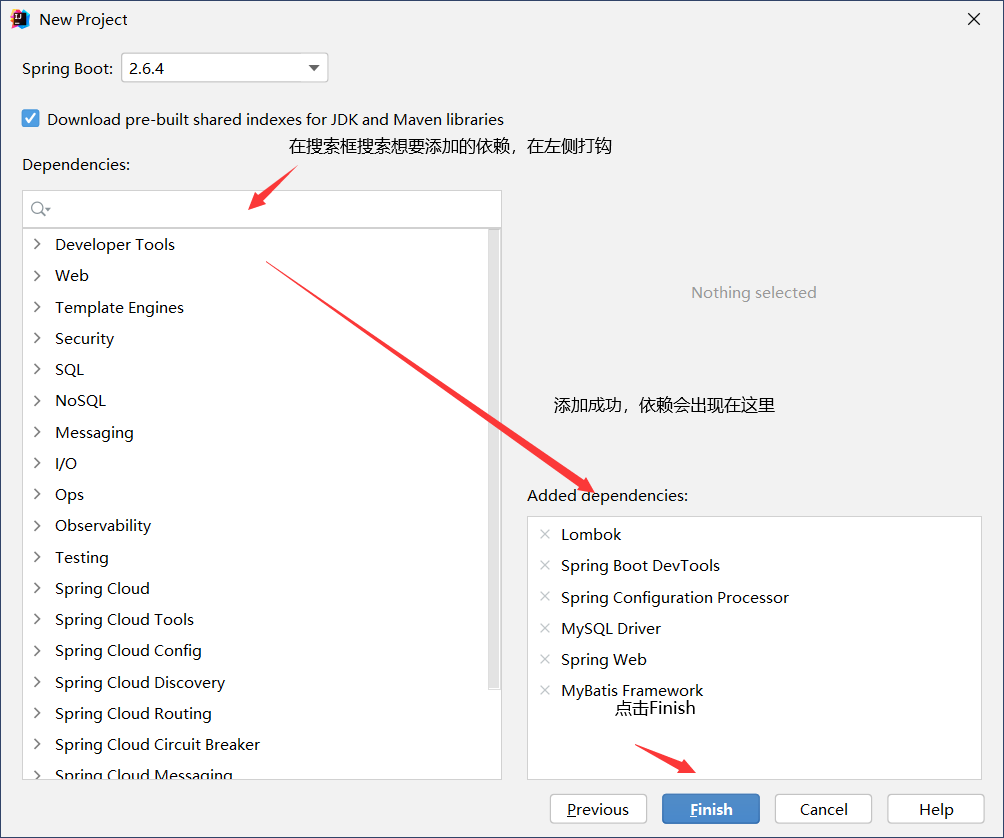


**后端初始化**

**先安装MySQL**

**初始化后端项目**

直接在IDEA开发工具中生成



**idea 连接本地数据库**

点击 database

添加MySQL

测试连接后，添加

**引入 mybatis-plus**

首先新建一个数据库中的表，（把官网的复制了，后期根据需求在补充修改）

DROP TABLE IF EXISTS user;

CREATE TABLE user

(

id BIGINT(20) NOT NULL COMMENT '主键ID',

name VARCHAR(30) NULL DEFAULT NULL COMMENT '姓名',

age INT(11) NULL DEFAULT NULL COMMENT '年龄',

email VARCHAR(50) NULL DEFAULT NULL COMMENT '邮箱',

PRIMARY KEY (id)

);

在idea创建一个数据库

往user表插入数据

引入依赖

<dependency>

<groupId>com.baomidou</groupId>

<artifactId>mybatis-plus-boot-starter</artifactId>

<version>3.5.1<versiopn>

</dependency>

spring:

application:

name: user-center

datasource:

driver-class-name: com.mysql.cj.jdbc.Driver

url: jdbc:mysql://localhost:3306/jayin

username: root

password: 123456

在SpringBoot 启动类中添加 @MapperScan 注解，扫描Mapper 文件夹

@SpringBootApplication

@MapperScan("com.luo.usercenter.mapper")

public class UserCenterApplication {

public static void main(String[] args) {

SpringApplication.run(UserCenterApplication.class, args);

}

}

编码

创建实体类，Mapper 类

开始使用

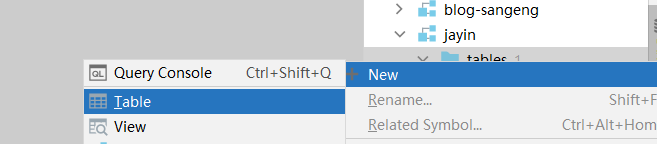
**中**

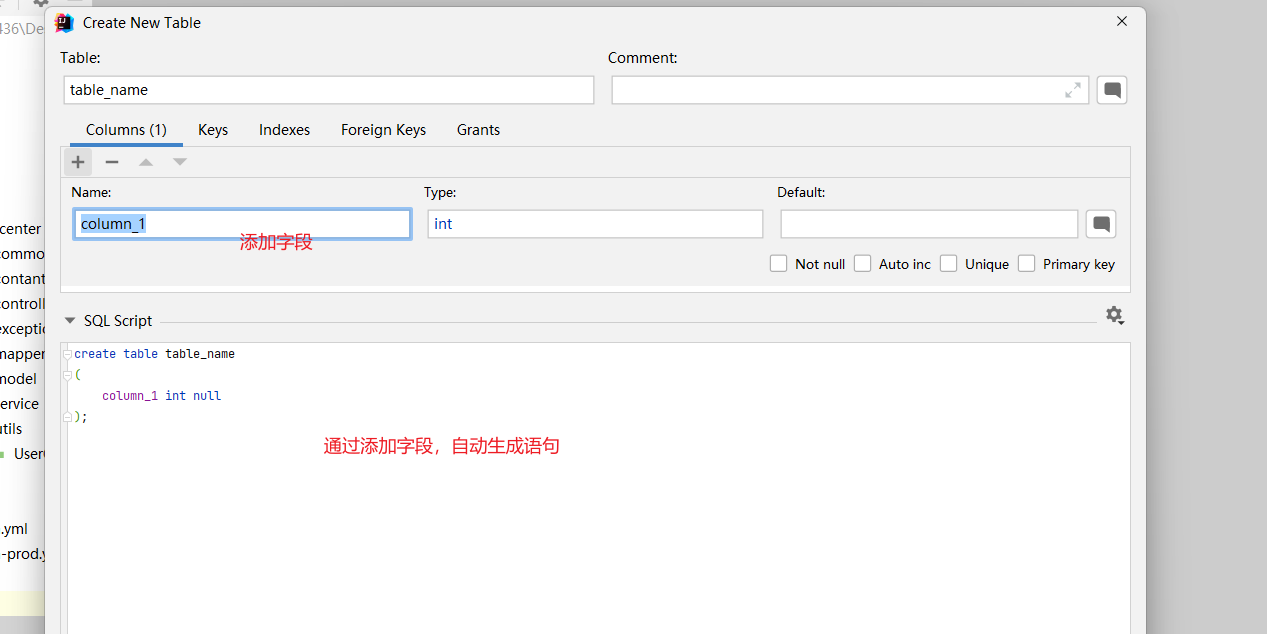
**数据库设计**

**表设计**



**建表**





什么是自动递增

id随着表记录的增加自动+1

什么是主键

数据库主键，指的是一个列或多列的组合，其值能唯一地标识表中的每一行，通过它可强制表的实体完整性。主键主要是用于其他表的外键关联，以及本记录的修改与删除（ 主键是数据表的唯一索引，比如学生表里有学号和姓名，姓名可能有重名，但学号却是唯一的）

create table user

(

id bigint auto\_increment comment 'id

'

primary key,

username varchar(256) null comment '用户昵称',

userAccount varchar(256) null comment '登录账号

',

avatarUrl varchar(1024) null comment '用户头像',

gender tinyint null comment '性别

',

userPassword varchar(512) not null comment '密码',

phone varchar(128) null comment '电话',

email varchar(512) null comment '邮箱',

userStatus int default 0 null comment '用户状态 0 正常',

createTime datetime default CURRENT\_TIMESTAMP null comment '创建时间',

updateTime datetime default CURRENT\_TIMESTAMP null comment '更新时间',

isDelete tinyint default 0 null comment '是否删除',

userRole int default 0 not null comment '用户角色 0 - 普通用户 1 - 管理员',

planetCode varchar(512) null comment '星球编号'

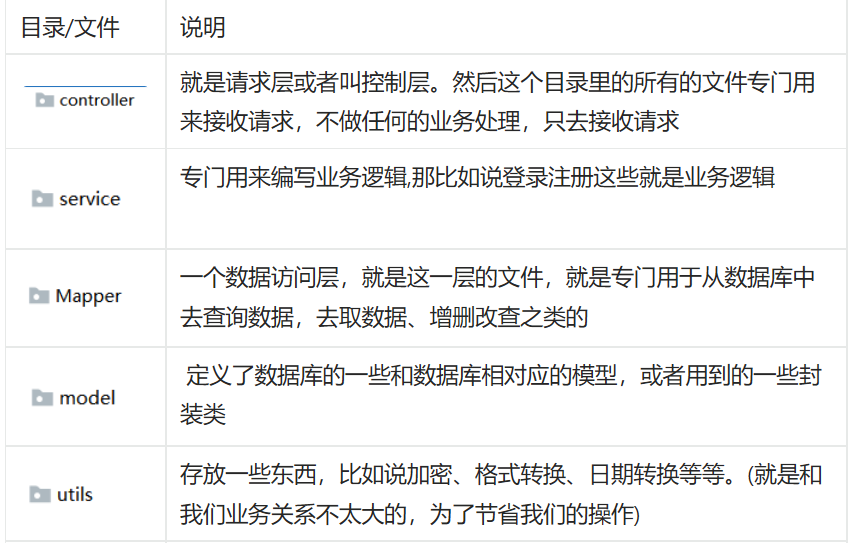
)

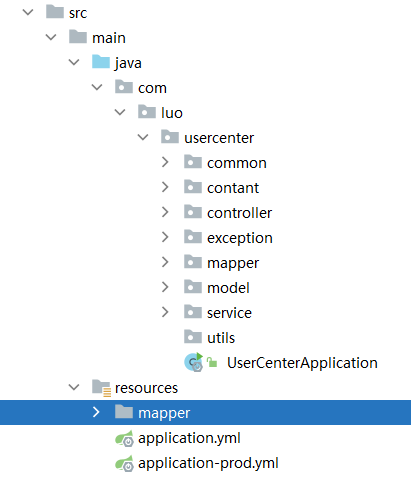
comment '用户';

**后端**

**规整项目目录**

**目录文件的说明**

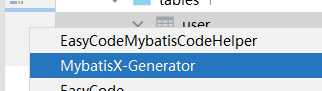


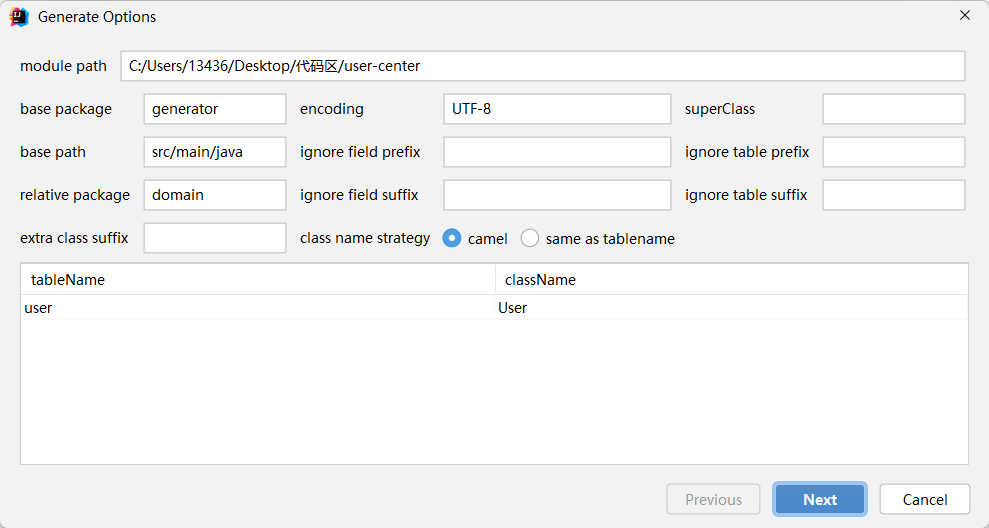


**实现基本数据库操作**

**自动生成器的使用**

MyBatisX插件， 自动根据数据库生成 domain,mapper,mapper.xml,service,serviceImpl

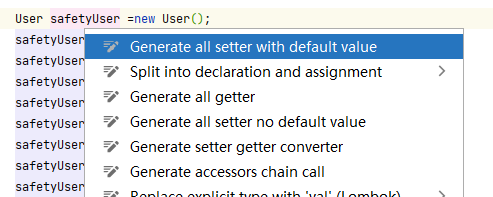




**小插件**

**GenerateAllSeter**

一键帮你生成对象的所有set方法，生成一个充满假数据的类，充满假数据的对象等等



**报错提示**

mybatis-plus 默认会将驼峰转换成下划线，所有出现在”字段列表“中出现未知列”user\_account"，在application.yml关闭默认转换。

在配置文件中添加：

mybatis-plus:

configuration:

map-underscore-to-camel-case: false

**注册逻辑**

用户在前端输入账户和密码，以及校验码

校验用户的账户，密码，校验密码，是否符合要求

非空

账户长度不小于4

密码不小于8

账户不能重复

账户不包含特殊字符

密码和校验密码相同

对密码进行加密

向数据库插入用户数据

**写**

UserService

/\*\*

\* 用户注册

\*

\* @param userAccount 用户账户

\* @param userPassword 用户密码

\* @param checkPassword 校验密码

\* @param planetCode 星球编号

\* @return id 新用户

\*/

long userRegister(String userAccount,String userPassword ,String checkPassword,String planetCode);

UserServiceImpl

/\*\*

\* 用户注册

\* @param userAccount 用户账户

\* @param userPassword 用户密码

\* @param checkPassword 校验密码

\* @return

\*/

@Override

public long userRegister(String userAccount, String userPassword, String checkPassword,String planetCode) {

// 1.校验

if (StringUtils.isAnyBlank(userAccount,userPassword,checkPassword,planetCode)){

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"参数为空");

}

if (userAccount.length()<4){

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"用户账户过短");

}

if (userPassword.length()<8 || checkPassword.length()<8){

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"用户密码过短");

}

if (planetCode.length()>5){

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"星球编号过长");

}

//账户不能包含特殊字符

String validPattern="[`~!@#$%^&\*()+=|{}':;',\\\\[\\\\].<>/?~！@#￥%……&\*（）——+|{}【】‘；：”“’。，、？]";

Matcher matcher = Pattern.compile(validPattern).matcher(userAccount);

if (matcher.find()) {

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"账户出现特殊字符");

}

//密码和校验密码相同

if(!userPassword.equals(checkPassword)){

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"校验密码与密码不相同");

}

// 账户不能重复

QueryWrapper<User> queryWrapper=new QueryWrapper<>();

queryWrapper.eq("userAccount",userAccount);

long count=userMapper.selectCount(queryWrapper);

if (count>0){

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"账号重复");

}

// 星球标号不能重复

queryWrapper=new QueryWrapper<>();

queryWrapper.eq("planetCode",planetCode);

count=userMapper.selectCount(queryWrapper);

if (count>0){

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"编号重复");

}

// 2.加密

String encryptPassword = DigestUtils.md5DigestAsHex((SALT+userPassword).getBytes());

// 3.插入数据

User user = new User();

user.setUserAccount(userAccount);

user.setUserPassword(encryptPassword);

user.setPlanetCode(planetCode);

boolean saveResult = this.save(user);

if (!saveResult){

throw new BusinessException(ErrorCode.SAVE\_ERROR,"插入数据失败");

}

return user.getId();

}

UserController

@PostMapping("/register")

public BaseResponse<Long> userRegister(@RequestBody UserRegisterRequest userRegisterRequest){

if(userRegisterRequest==null){

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

String userAccount = userRegisterRequest.getUserAccount();

String userPassword = userRegisterRequest.getUserPassword();

String checkPassword = userRegisterRequest.getCheckPassword();

String planetCode=userRegisterRequest.getPlanetCode();

if (StringUtils.isAnyBlank(userAccount,userPassword,checkPassword,planetCode)){

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

long result=userService.userRegister(userAccount, userPassword, checkPassword,planetCode);

return ResultUtils.success(result);

}

**下**

**后端**

**用户登录**

**service**

/\*\*

\* 用户登录

\* @param userAccount 用户账户

\* @param userPassword 用户密码

\* @param request

\* @return 脱敏后的用户信息

\*/

User userLogin(String userAccount, String userPassword, HttpServletRequest request);

**serviceImpl**

/\*\*

\* 用户登录

\* @param userAccount 用户账户

\* @param userPassword 用户密码

\* @param request

\* @return

\*/

@Override

public User userLogin(String userAccount, String userPassword, HttpServletRequest request) {

// 1.校验

if (StringUtils.isAnyBlank(userAccount,userPassword)){

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"参数为空");

}

if (userAccount.length()<4){

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"用户账户过短");

}

if (userPassword.length()<8){

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"用户密码过短");

}

//账户不能包含特殊字符

String validPattern="[`~!@#$%^&\*()+=|{}':;',\\\\[\\\\].<>/?~！@#￥%……&\*（）——+|{}【】‘；：”“’。，、？]";

Matcher matcher = Pattern.compile(validPattern).matcher(userAccount);

if (matcher.find()) {

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"账户出现特殊字符");

}

// 2.加密

String encryptPassword = DigestUtils.md5DigestAsHex((SALT+userPassword).getBytes());

//查询用户是否存在

QueryWrapper<User> queryWrapper=new QueryWrapper<>();

queryWrapper.eq("userAccount",userAccount);

queryWrapper.eq("userPassword",encryptPassword);

User user=userMapper.selectOne(queryWrapper);

//用户不存在

if (user==null){

log.info("user login failed, userAccount cannot match userPassword");

throw new BusinessException(ErrorCode.NULL\_ERROR,"用户不存在");

}

// 3. 用户脱敏

User safetyUser=getSafteyUser(user);

// 4. 记录用户的登录态

request.getSession().setAttribute(USER\_LOGIN\_STATE,safetyUser);

return safetyUser;

}

**登录和注册加密都有使用 final String SALT=“jayin” , 可以直接提取出来，让它称为公共值，顺便定义一下，加上注释**

/\*\*

\* 盐值，混淆密码

\*/

private static final String SALT="yupi";

**用户脱敏**

// 3. 用户脱敏

User safetyUser=getSafteyUser(user);

/\*\*

\* 用户脱敏

\* @param originUser

\* @return

\*/

@Override

public User getSafteyUser(User originUser){

if (originUser==null){

throw new BusinessException(ErrorCode.NULL\_ERROR,"用户为空");

}

User safetyUser =new User();

safetyUser.setId(originUser.getId());

safetyUser.setUsername(originUser.getUsername());

safetyUser.setUserAccount(originUser.getUserAccount());

safetyUser.setAvatarUrl(originUser.getAvatarUrl());

safetyUser.setGender(originUser.getGender());

safetyUser.setPhone(originUser.getPhone());

safetyUser.setEmail(originUser.getEmail());

safetyUser.setPlanetCode(originUser.getPlanetCode());

safetyUser.setUserRole(originUser.getUserRole());

safetyUser.setUserStatus(originUser.getUserStatus());

safetyUser.setCreateTime(originUser.getCreateTime());

return safetyUser;

}

**逻辑删除**

如果说一个用户他的delete状态是已经被删除了，我们是否能查出来？

MyBatis-Plus 框架它有一个逻辑删除，默认会帮助我们查询出没有被删的用户。

mybatis-plus:

global-config:

db-config:

logic-delete-field: isDelete # 全局逻辑删除的实体字段名(since 3.3.0,配置后可以忽略不配置步骤2)

logic-delete-value: 1 # 逻辑已删除值(默认为 1)

logic-not-delete-value: 0 # 逻辑未删除值(默认为 0)

/\*\*

\* 是否删除

\*/

@TableLogic

private Integer isDelete;

**Controller**

@RestController 注解 适用于编写 restful 风格的api 返回值默认为json 类型

这个类里面 所有的请求的接口返回值，响应的数据类型都是 application json

@RestController

@RequestMapping("/user")

public class UserController {

}

@PostMapping("/login")

public BaseResponse<User> userLogin(@RequestBody UserLoginRequest userLoginRequest, HttpServletRequest request ){

if(userLoginRequest==null){

return ResultUtils.error(ErrorCode.PARAMS\_ERROR);

}

String userAccount = userLoginRequest.getUserAccount();

String userPassword = userLoginRequest.getUserPassword();

if (StringUtils.isAnyBlank(userAccount,userPassword)){

return ResultUtils.error(ErrorCode.PARAMS\_ERROR);

}

User user=userService.userLogin(userAccount, userPassword,request);

return ResultUtils.success(user);

}

@RequestBody 没打上的话，springmvc框架不知道怎么把前端传来的json参数去对合适的对象做一个关联

**序列化**

这里需要先封装一个对象，专门用来接收请求参数

/\*\*

\* 用户注册请求体

\* @author 13436

\*/

@Data

public class UserRegisterRequest implements Serializable {

public static final long serialVersionUID = 319124716373120793L;

private String userAccount;

private String userPassword;

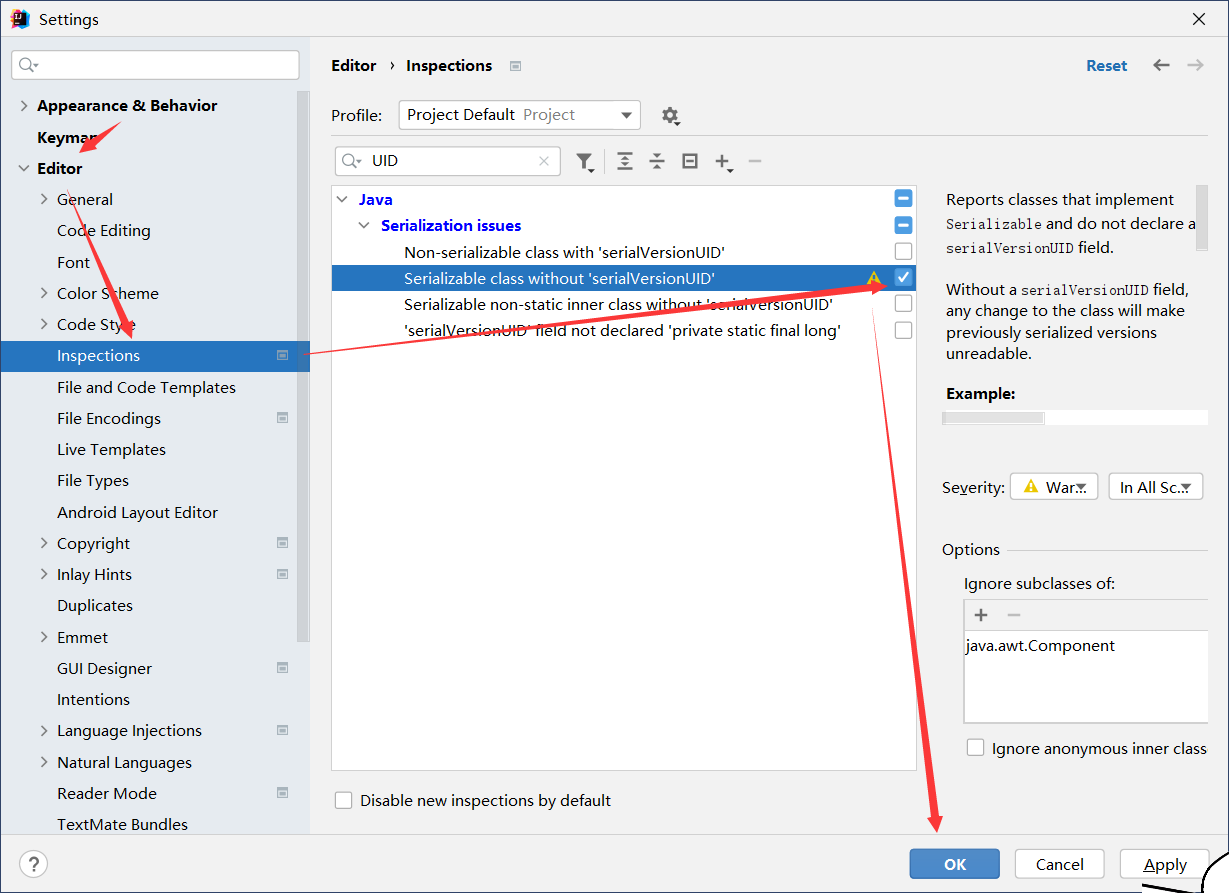
private String checkPassword;

private String planetCode;

}

继承Serializable 就是 序列化

实现序列化接口





**序列化**

对象的寿命通常随着生成该对象的程序的终止而终止，有时候需要把在内存中的各种对象的状态（也就是实例变量，不是方法）保存下来，并且可以在需要时再将对象恢复。虽然你可以用你自己的各种各样的方法来保存对象的状态，但是Java给你提供一种应该比你自己的好的保存对象状态的机制，那就是序列化。

**总结**

Java 序列化技术可以使你将一个对象的状态写入一个Byte 流里（系列化），并且可以从其它地方把该Byte 流里的数据读出来（反序列化）。

**用途**

想把 内存中的对象状态保存到一个文件中或者数据库中的时候

想把 对象通过网络进行传播的时候

**用户管理**

**查询用户**

@GetMapping("/search")

public BaseResponse<List<User>> searchUsers(String username, HttpServletRequest request){

if(!isAdmin(request)){

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

QueryWrapper<User> queryWrapper =new QueryWrapper<>();

if (StringUtils.isNotBlank(username)){

queryWrapper.like("username",username);

}

List<User> userList=userService.list(queryWrapper);

List<User> list=userList.stream().map(user -> userService.getSafteyUser(user)).collect(Collectors.toList());

return ResultUtils.success(list);

}

**删除用户**

@PostMapping("/delete")

public BaseResponse<Boolean> deleteUsers(@RequestBody long id,HttpServletRequest request){

if(!isAdmin(request)){

throw new BusinessException(ErrorCode.NO\_AUTH);

}

if(id<=0){

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

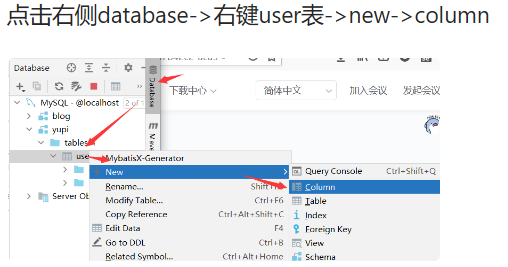
}

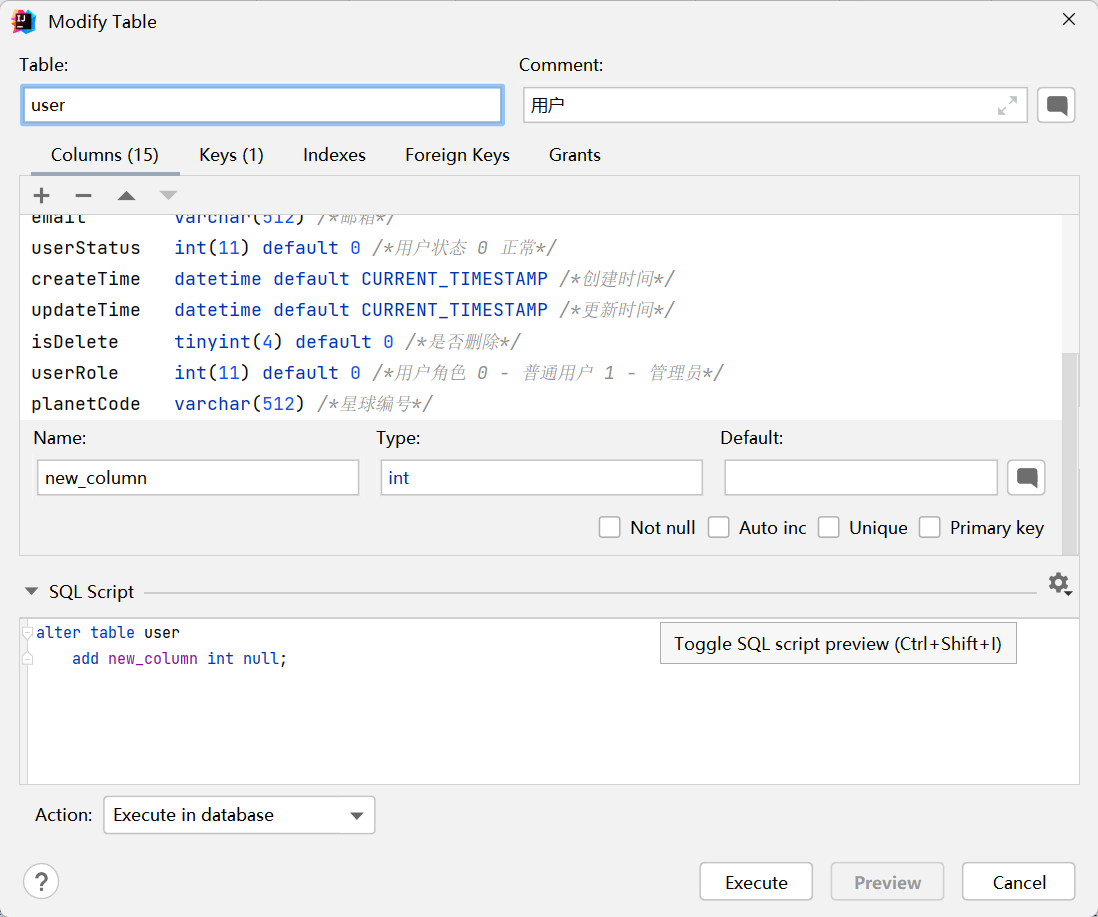
boolean b =userService.removeById(id);

return ResultUtils.success(b);

}

**添加字段**





**定义常量类**

/\*\*

\* 用户常量

\* @author 13436

\*/

public interface UserConstant {

/\*\*

\* 用户登录态键

\*/

String USER\_LOGIN\_STATE="userLoginState";

// -------权限--------------

/\*\*

\* 默认权限 - 普通用户

\*/

int DEFAULT\_ROLE=0;

/\*\*

\* 管理员权限

\*/

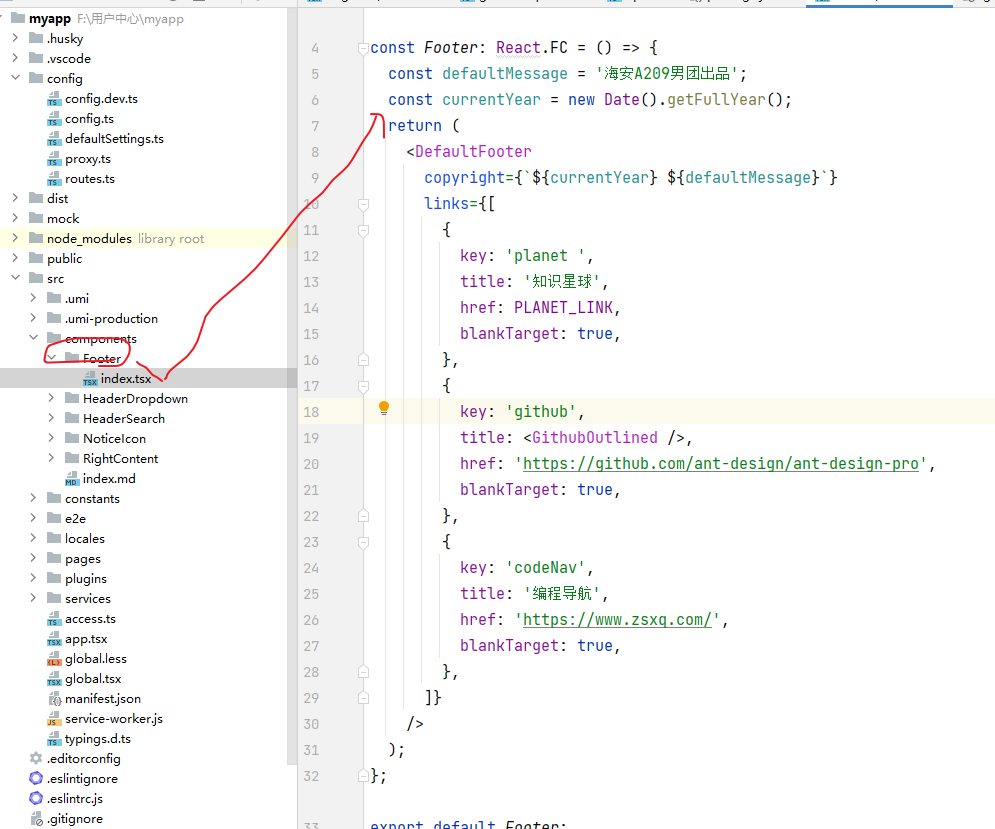
int ADMIN\_ROLE=1;

}

**前端**

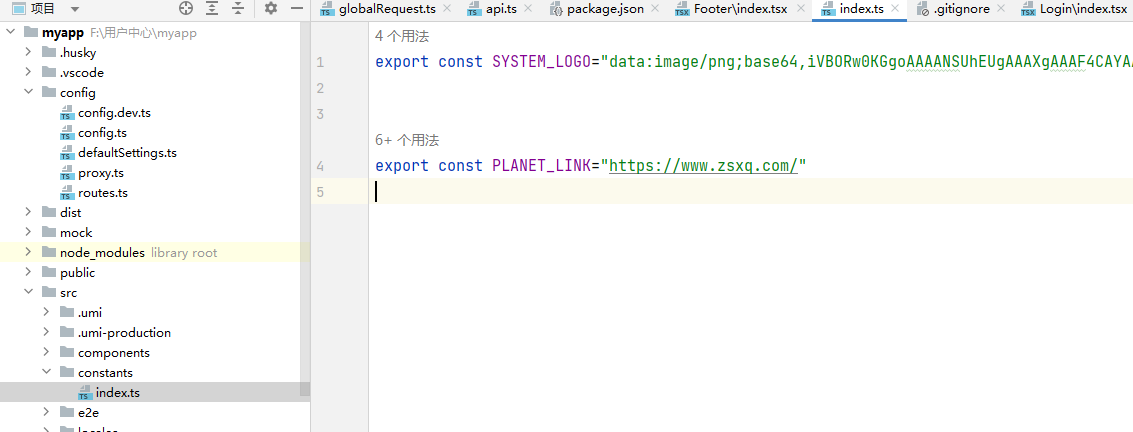
**修改底部版权信息**





**修改logo**

定义一个全局的变量，在src包下建 constant包



**修改标题和副标题**



**删除多余代码**

对Login -> index.tsx :

删除 其他登录方式

删除 手机号登录

删除 自动提示错误的用户和密码

用户名 全局 改成 账号 （ Ctrl + F） （Ctrl + R）

密码 修改为 请输入密码

验证码，把手机号登录的代码折叠删除

设置忘记密码的链接



**对接后台接口**

找到handleSubmit 点进去

点击 LoginParams

进入 Typing

点击username 按 F2 重构为 userAccount password 重构为 userPassword

返回 index 点击login

进入api

修改reques的地址。 搜索官方文档 得到以下配置文件 用于修改，提示我们去app.ts 配置

import {RequestConfig} from 'umi'

export const request: RequestConfig={

timeout:1000000, //建议设置比较大的数不然会加载超时。

}

api.ts 的登录接口，改为后端的接口， “ /api/user/login"

**问题 ：** 遇到了跨域错误，为什么会跨域错误，因为我们当前前端的浏览器里的地址，是8000，而后端是 8080 ，端口不一样就是跨域。

跨域的解决方法，本次使用了 “代理”

去修改 proxy.ts 代理



在后端 application.yml 指定接口全局api

server:

servlet:

context-path: /api

测试 请求成功

在index.tsx 把 username 替换成 userAccount password替换成 userPassword (区分大小写匹配）

改为 user

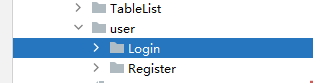


**终**

**注册功能的前端开发**

**增加注册页面**

直接复制登录页面，重命名为Register



把组件的名称改一下 （Login --> Register）

**定义注册地址**

定义以下，用户在地址栏里面输入什么网址，才能访问到新建的页面，新建路由

现在用了 react router 这样一个路由组件，可以帮助我们把url和写的页面组件关联上，这样的ua用户访问某一个地址。比如说 user/register ，然后访问它的时候，后面去指定一个我们写的组件位置， 比如说 user/register 它就会找到user 配置目录下的register



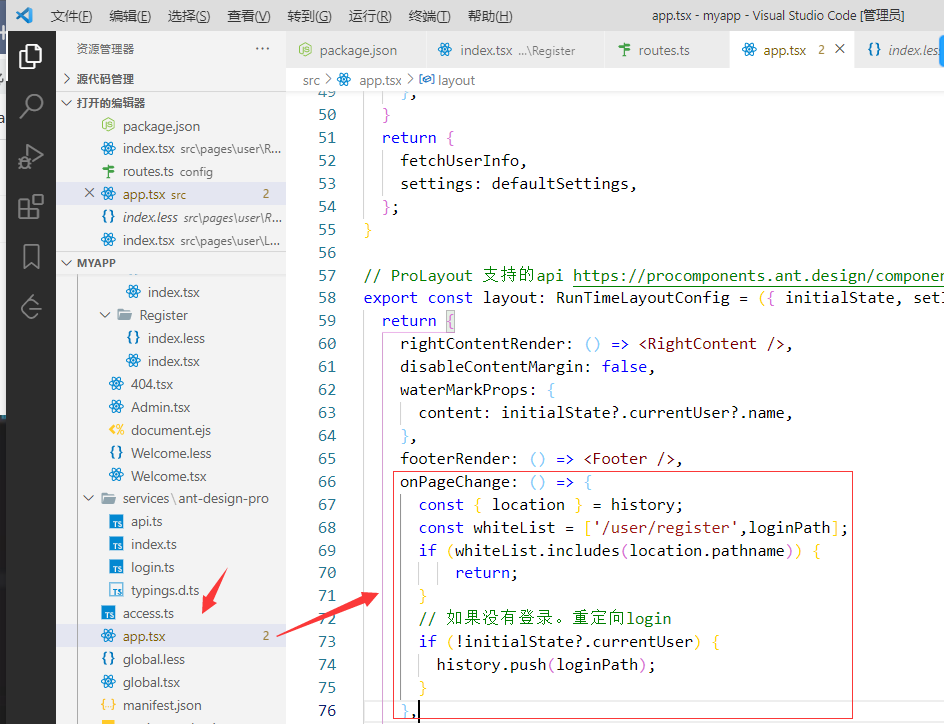
删除注册页面不需要的信息

还要到app.tsx 修改一些内容

把这段代码



改为：



为什么会出发这种自动重定向到登录页的逻辑？

主要就是因为 app.tsx 它是咱们这个前端项目的全局文件，里面定义了刚进入一个时要执行的逻辑，比如说我们刚进入页面要获取用户 信息，如果没获取到用户信息，就重定向到登录页面让你登录，那同时，因为使用的这个框架，更偏向于后台管理系统，所以说每个页面都会去校验， 如果说用户没登陆，没有登录状态，那我们就把它重定向到登录页

**写前端代码**

删 没用的代码

修改代码 （界面） 账号登录 改为 账号注册

界面上只有两个文本框， 要需要一个确认密码的密码框， 复制 写入密码的 代码， 修改内容。

Ctrl+F 把登录全局替换成注册

遇到问题： 修改后 你会发现的登录按钮 的文字 没有变成注册 （因为我们使用这个现成的loginform组件的原因）

点进去看LoginForm 源码， 进入对应的文件。点击js，看到组件源码 中有 submitText

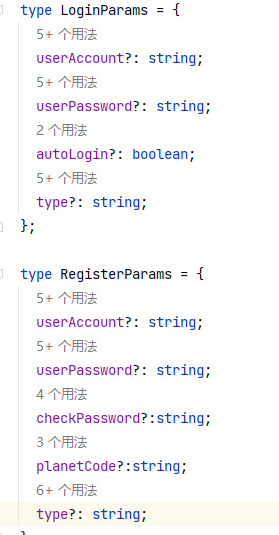
我们仿照源码，在注册页面自己写一个，将其覆盖。

在app.tsx 新增代码



点进去LoginParams 去补充注册参数

复制 LoginParams 改为 RegisterParams

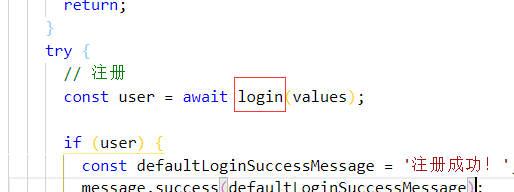


回到index 把 LoginParams 全部替换成 RegisterParams

在提交前写一些校验：



写一个注册方法，点进去login



api



回到 index 将 user 改为 id login 改为 register

修改 注册成功跳转的位置和注册失败提示的信息

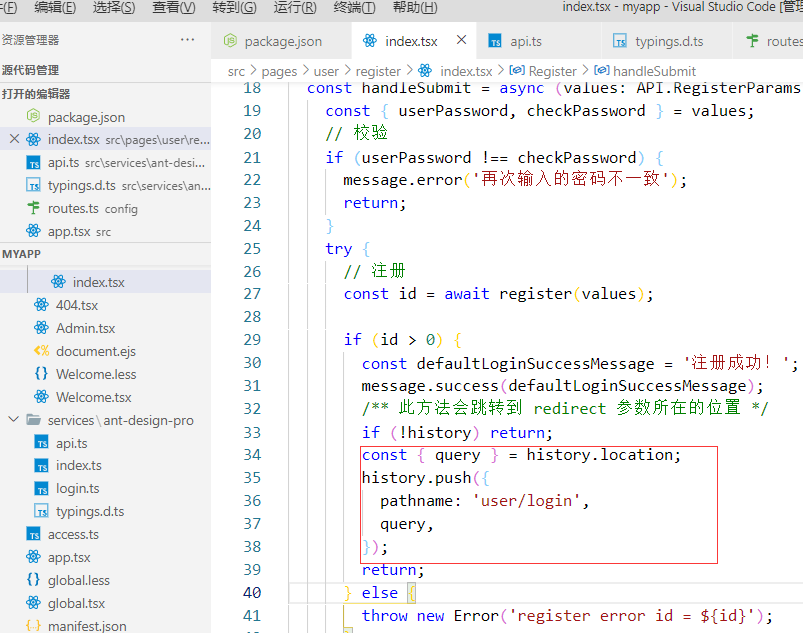


删除没用的代码

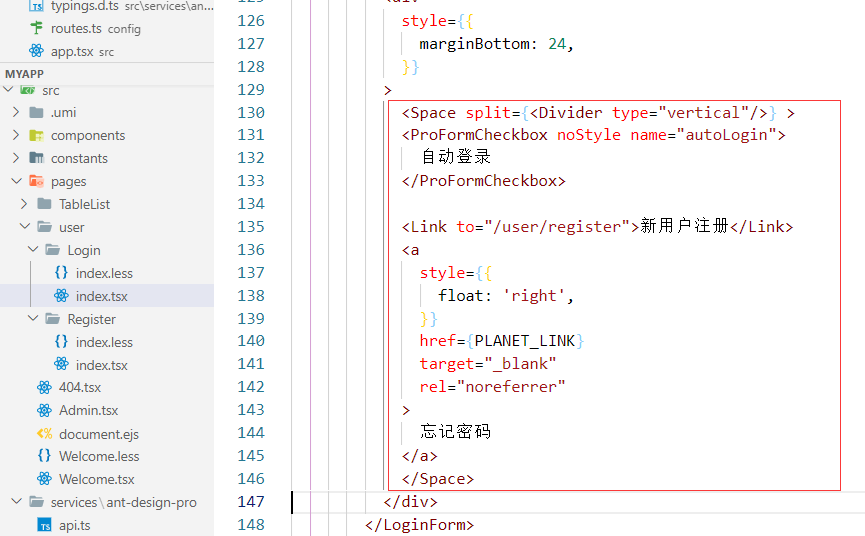
测试

问题： 登录成功是跳转到登录页，但是这个 redirectundefined 地址， 有点小bug

修改之前的代码

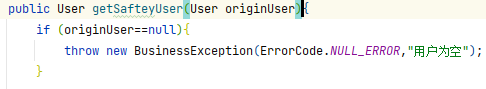


登录页面没有注册跳转的链接，去登录页面加一下



**获取当前用户登录态度，信息接口**

在UserServiceImpl 做一个判断



在UserController 中添加获取当前用户登录态，信息接口

@GetMapping("/current")

public BaseResponse<User> getCurrentUser(HttpServletRequest request){

Object userObj=request.getSession().getAttribute(USER\_LOGIN\_STATE);

User currentUser= (User) userObj;

if(currentUser==null){

throw new BusinessException(ErrorCode.NOT\_LOGIN);

}

Long userId = currentUser.getId();

//todo 检验用户是否合法

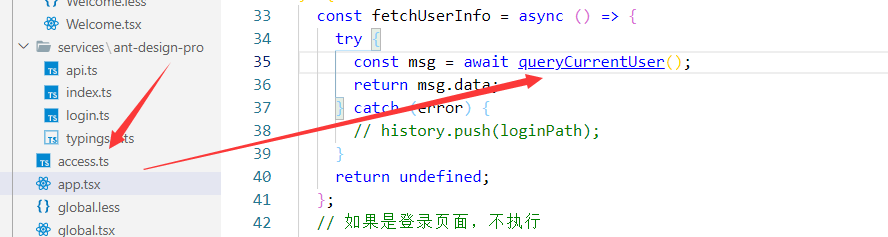
User user = userService.getById(userId);

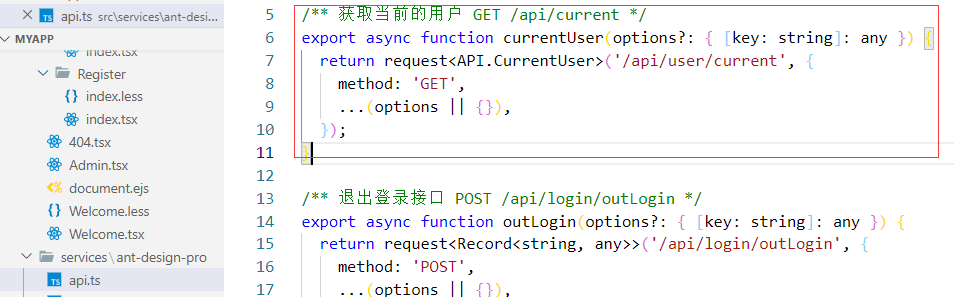
User safeUser = userService.getSafteyUser(user);

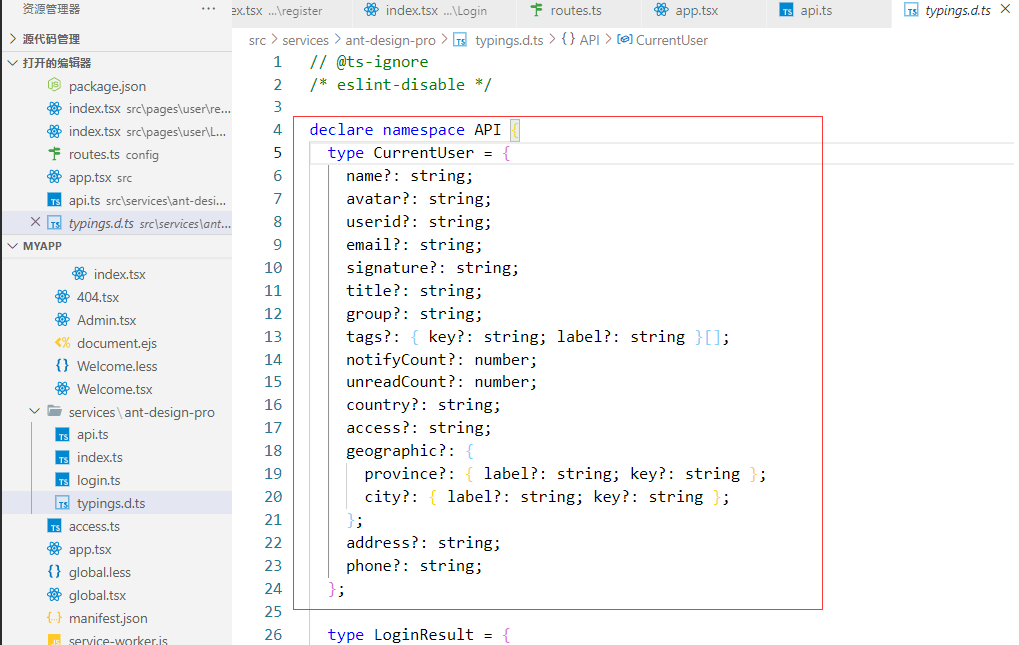
return ResultUtils.success(safeUser);

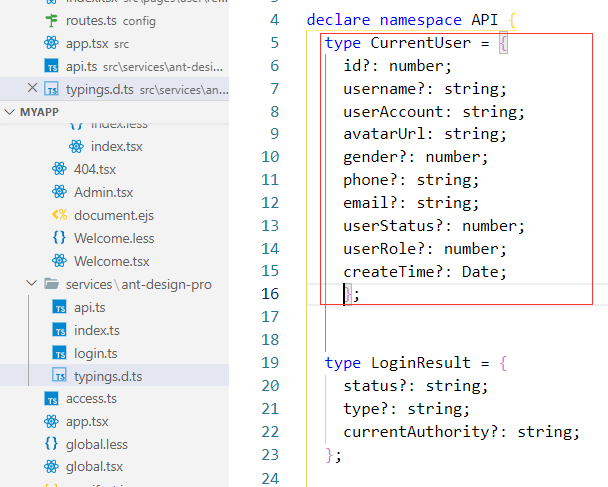
}

回到前端，点击 queryCurrentUser 修改

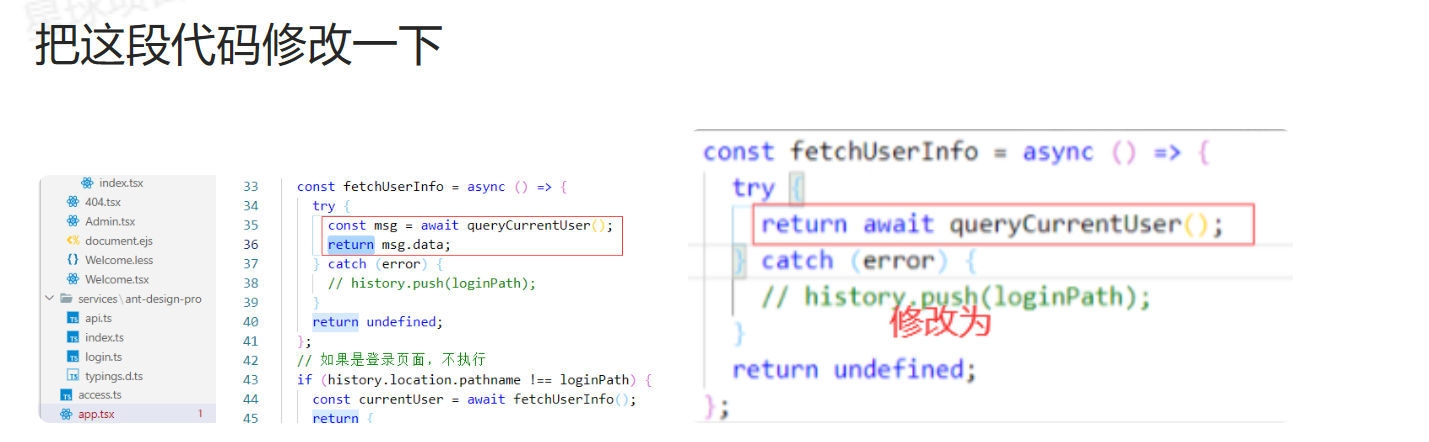


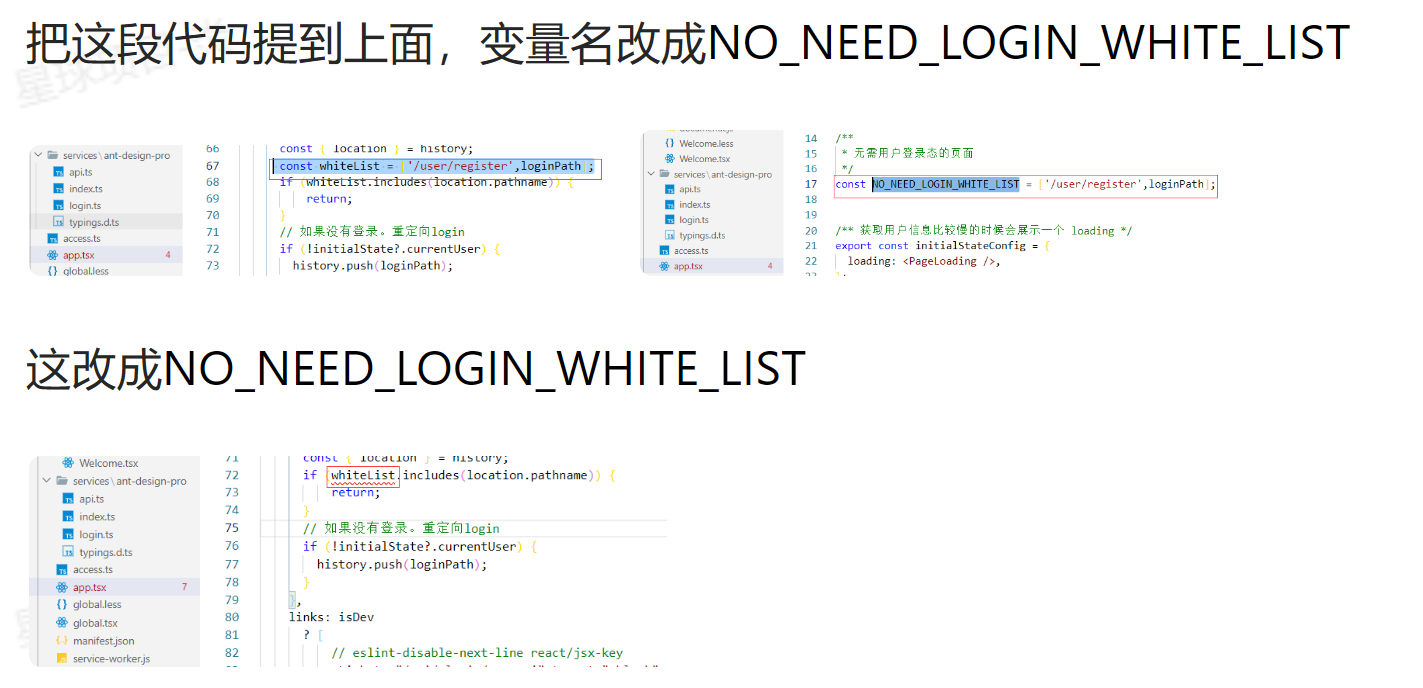
















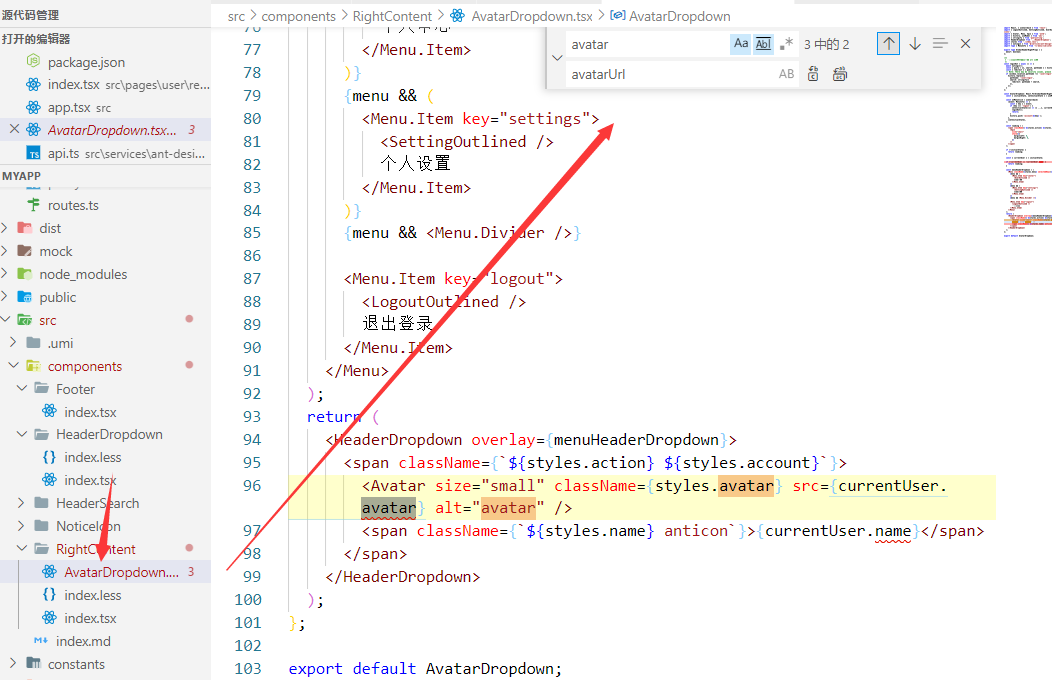
测试

登录成功后，为什么右上角的角色和用户名一直在加载呢？ 因为 新注册的用户，没有 写上用户名和头像，去数据库添加一下就可以显示了

添加后

水印出来了，头像没出来， ---> ? 我们取的头像变量名和它原来的对不上，所以没显示

找到AvatarDropdown.tsx 修改头像变量名

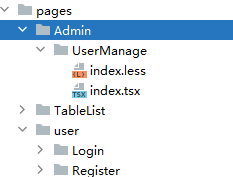




**完成用户管理后台的前端开发**

**创建用户管理页面**

在page文件夹中新建Admin文件夹，把user文件夹下的Register 文件复制，粘贴到Admin文件下，改名 UserMange



去route.ts 添加一个路由



访问一下 ---> 无权访问



对比发现， 多了 access:'canAdmin' 也就是管理员才能访问

找到access.ts 修改



**开发管理页面**

把二级管理员的路由删掉

运行发现，页面显示出问题， 我们复制的是注册的代码，但是显示了其他的。。。 它里面根本米有根据路由去先择到底要显示哪个页面，所以 需要修改

问题在 component ： './Admin'

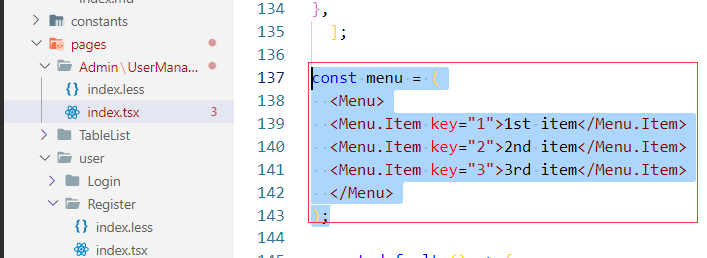
找到Admin.tsx内的文本内容和页显示相同，说明这个页面写死了， 修改一下内容。

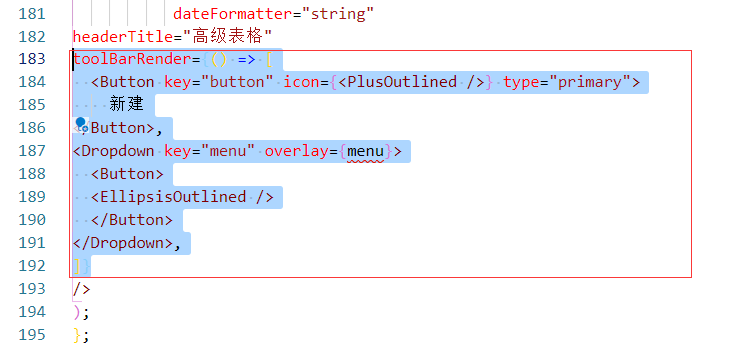


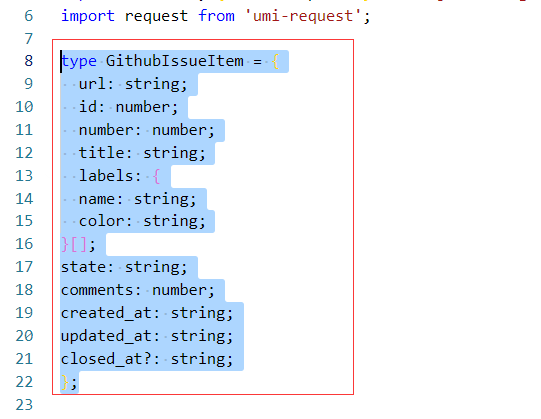
**瘦身**

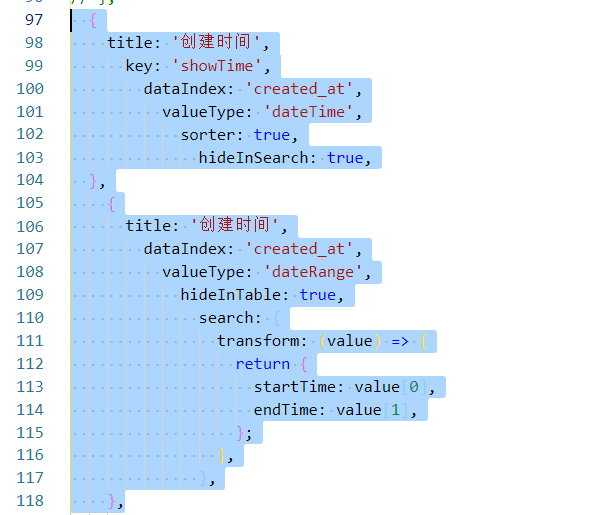
去官方文档直接用现有的表格 [Pro Components (ant.design)](https://procomponents.ant.design/components/form/)

覆盖 index，删掉没用的代码







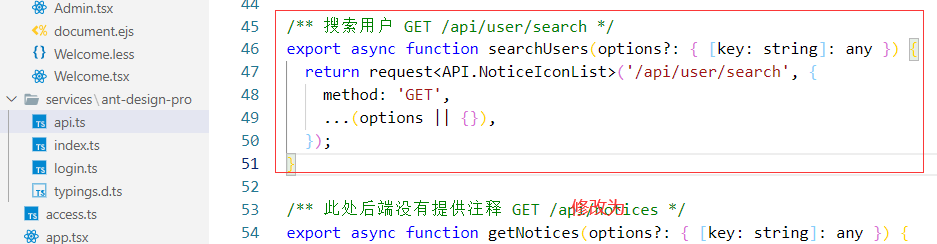


修改





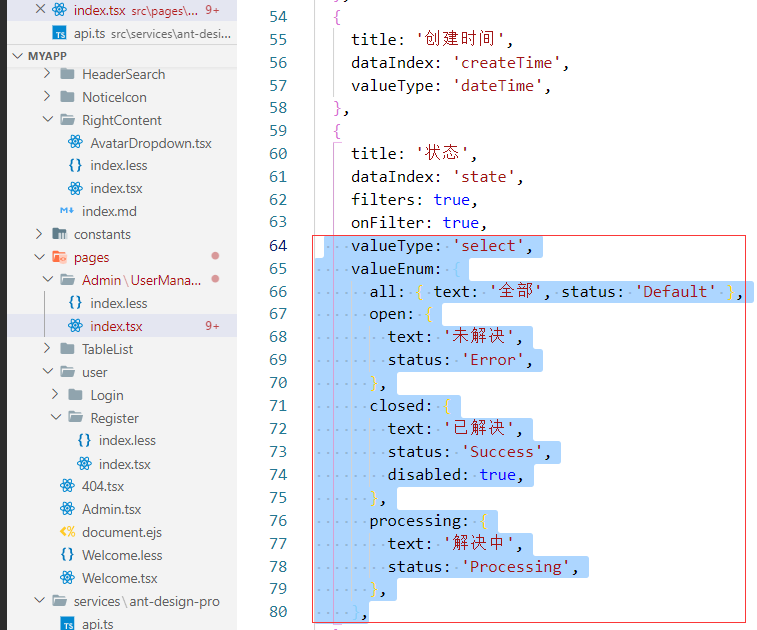
在api.ts 增加一个搜索用户的接口



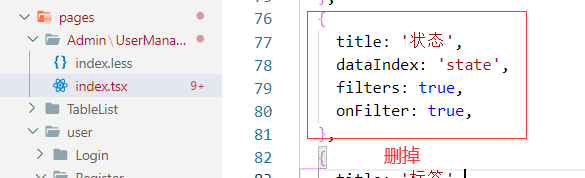
测试 ---> 头像显示有问题， 应该直接显示图片的



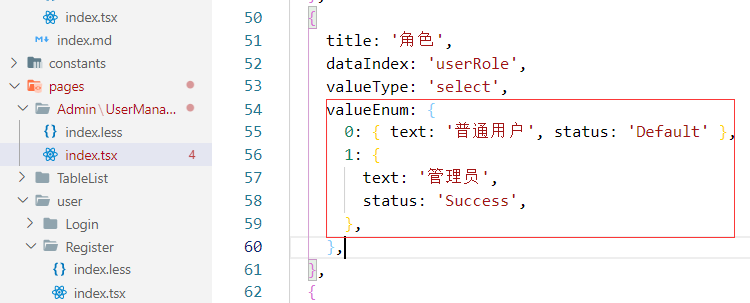
删 改











**末**

**开发用户注销前后端**

**后端**

先写业务逻辑层，业务逻辑层就是service

用户注销需要接收什么参数？ 看一下用户登录的时候，是填了什么参数，执行了什么擦欧总。

之前写过一个用户登录的逻辑，当用户登录成功，我们要把用户的登录状态存到服务器的一个session中，我们要去这个session的时候，就从request 对象里面去取

当用户登录成功后，这个请求体的session中， 就是说这个请求对应的session 中保存了用户的登录状态，之后 我们判断用户是否登录，只要看看这个session里面有没有这个标识，就能知道用户是否登录。

那么 如果想用用户注销的话，反其道而行之，用户登录需要设置一个登录状态，注销不就是移除一个登录状态吗？ 把同样的一个登录态的标识移除掉就可以了

**service**

/\*\*

\* 用户注销

\* @param request

\* @return

\*/

int userLogout(HttpServletRequest request);

**serviceImpl**

/\*\*

\* 用户注销

\* @param request

\* @return

\*/

@Override

public int userLogout(HttpServletRequest request) {

//移除登录态

request.getSession().removeAttribute(USER\_LOGIN\_STATE);

return 1;

}

**Controller**

@PostMapping("/logout")

public BaseResponse<Integer> userLogout(HttpServletRequest request){

if(request==null){

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

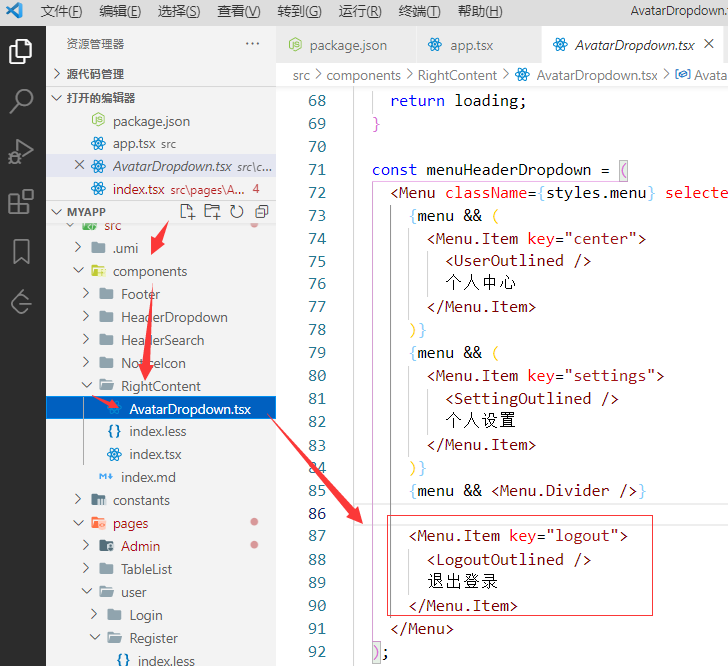
int result=userService.userLogout(request);

return ResultUtils.success(result);

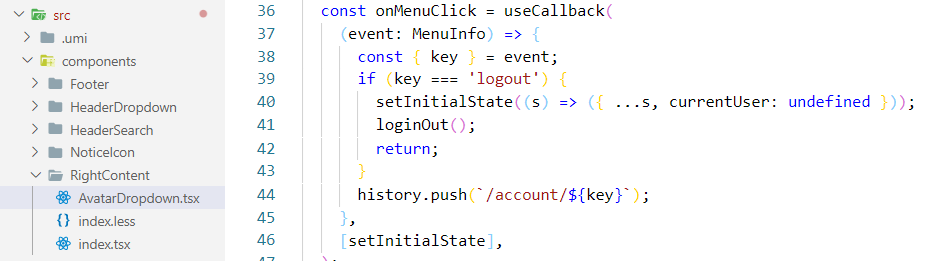
}

**前端**

先找到页面触发注销的位置，看看再哪里去写注销，这个注销按钮是不是在导航上，导航条是所有页面都有的一个组件，所以说这个导航条应该在component目录中，这里有个RightContent目录，里面有个AvatarDropdown



这有一个关于logout的事件绑定，直接用 。

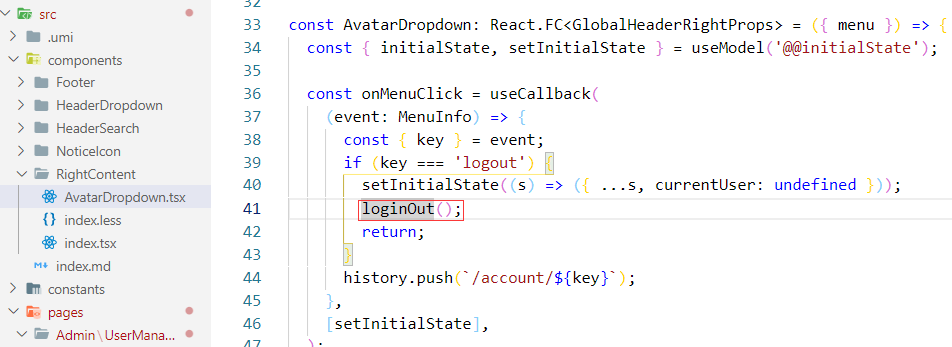


如果说要注销，肯定要把前端已经存在的登录态设置为空，我们前端登录存在了app.tsx 这里定义了一个currentUser,尤其是 getInitialState,每次刷新页面或2刚进入新页面时，最先执行的一个方法。

在这个方法中，定义了一些要初始化的操作，比如获取已登录用户的信息，会把用户信息放到InitialState里，如果我们要注销，就要把InitialState 里已存在的用户信息清空掉



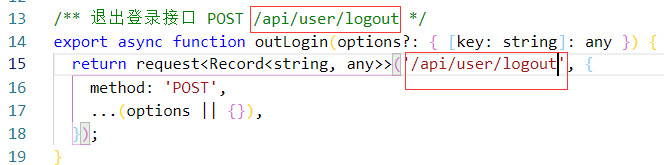
点击logOut



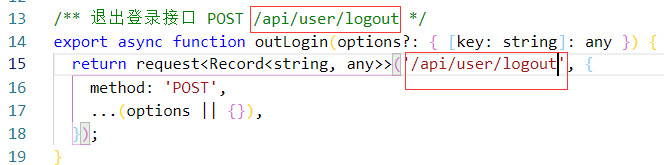
点击outLogin ，这里对应我们后端的接口调用



来到



改个注释，改个接口



测试，刷新。再次退出登录。

**补充用户校验前后端逻辑**

之前我们用户注册的时候，只让用户填了账号密码 ，我们现在要补充一些逻辑来校验这个用户必须是星球里的用户。

可以先让用户自己定，后期再编写一个脚本什么的。输入所有已加入星球的用户列表，然后让服务器去做一次校验，把不符合要求的用户封掉。

**后端**

添加字段

String planetCode

修改相应的类 代码。 在service方法中补充字段。

补充对于用户编号的校验，不能为空，不能大于5 还要校验编号不能重复

if (planetCode.length()>5){

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"星球编号过长");

}

// 星球标号不能重复

queryWrapper=new QueryWrapper<>();

queryWrapper.eq("planetCode",planetCode);

count=userMapper.selectCount(queryWrapper);

if (count>0){

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"编号重复");

}

添加一个插入的数据

**完整**

/\*\*

\* 用户注册

\*

\* @param userAccount 用户账户

\* @param userPassword 用户密码

\* @param checkPassword 校验密码

\* @param planetCode 星球编号

\* @return id 新用户

\*/

long userRegister(String userAccount,String userPassword ,String checkPassword,String planetCode);

/\*\*

\* 用户注册

\* @param userAccount 用户账户

\* @param userPassword 用户密码

\* @param checkPassword 校验密码

\* @return

\*/

@Override

public long userRegister(String userAccount, String userPassword, String checkPassword,String planetCode) {

// 1.校验

if (StringUtils.isAnyBlank(userAccount,userPassword,checkPassword,planetCode)){

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"参数为空");

}

if (userAccount.length()<4){

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"用户账户过短");

}

if (userPassword.length()<8 || checkPassword.length()<8){

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"用户密码过短");

}

if (planetCode.length()>5){

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"星球编号过长");

}

//账户不能包含特殊字符

String validPattern="[`~!@#$%^&\*()+=|{}':;',\\\\[\\\\].<>/?~！@#￥%……&\*（）——+|{}【】‘；：”“’。，、？]";

Matcher matcher = Pattern.compile(validPattern).matcher(userAccount);

if (matcher.find()) {

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"账户出现特殊字符");

}

//密码和校验密码相同

if(!userPassword.equals(checkPassword)){

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"校验密码与密码不相同");

}

// 账户不能重复

QueryWrapper<User> queryWrapper=new QueryWrapper<>();

queryWrapper.eq("userAccount",userAccount);

long count=userMapper.selectCount(queryWrapper);

if (count>0){

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"账号重复");

}

// 星球标号不能重复

queryWrapper=new QueryWrapper<>();

queryWrapper.eq("planetCode",planetCode);

count=userMapper.selectCount(queryWrapper);

if (count>0){

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"编号重复");

}

// 2.加密

String encryptPassword = DigestUtils.md5DigestAsHex((SALT+userPassword).getBytes());

// 3.插入数据

User user = new User();

user.setUserAccount(userAccount);

user.setUserPassword(encryptPassword);

user.setPlanetCode(planetCode);

boolean saveResult = this.save(user);

if (!saveResult){

throw new BusinessException(ErrorCode.SAVE\_ERROR,"插入数据失败");

}

return user.getId();

}

@PostMapping("/register")

public BaseResponse<Long> userRegister(@RequestBody UserRegisterRequest userRegisterRequest){

if(userRegisterRequest==null){

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

String userAccount = userRegisterRequest.getUserAccount();

String userPassword = userRegisterRequest.getUserPassword();

String checkPassword = userRegisterRequest.getCheckPassword();

String planetCode=userRegisterRequest.getPlanetCode();

if (StringUtils.isAnyBlank(userAccount,userPassword,checkPassword,planetCode)){

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

long result=userService.userRegister(userAccount, userPassword, checkPassword,planetCode);

return ResultUtils.success(result);

}

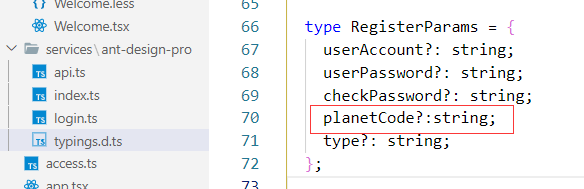
**前端**

前端补充一个输入框，找一下用户注册的文件

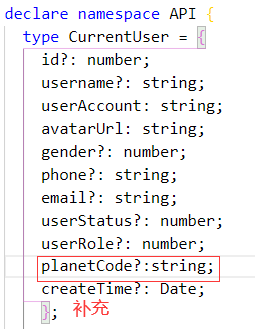
复制账号框，粘贴到确认密码框的下面，修改相应信息



在RegisterParams 补充一个 planetCode



在CurrentUser 补充一个星球编号



在表格内增加一列星球编号

**后端代码优化**

**通用返回对象**

为什么要做 通用返回对象？

后端直接返回一个对象给前端，如果这个数据 出问题了，后端处理报错了，查不到用户，前端如果不可以区别的话，也不知道它到底为什么报错。

比如说一个测试，它返回了一个有六项的列表，加入后台返回了一个空数组，前端可能也不会意识到它是错误的，如果说后台因为一些异常，强制给他返回空数组，但前端可能不知道对不对，他还以为这个知识一个本来就是数据库里没有用户的一个值

所以需要定义一个通过用的返回对象

新建common包

存项目公用的一些类

新建BaseResponse

/\*\*

\* 通用返回类

\* @param <T>

\* @author jayin

\*/

@Data

public class BaseResponse<T> implements Serializable {

private int code;

private T data;

private String message;

private String description;

public BaseResponse(int code, T data, String message,String description) {

this.code = code;

this.data = data;

this.message = message;

this.description = description;

}

public BaseResponse(int code, T data, String message) {

this(code, data, message, "");

}

public BaseResponse(int code, T data) {

this(code, data,"", "");

}

public BaseResponse(ErrorCode errorCode) {

this(errorCode.getCode(),null,errorCode.getMessage(),errorCode.getDescription());

}

}

新建ResultUtil

/\*\*

\* 返回工具类

\*/

public class ResultUtils {

/\*\*

\* 成功

\* @param data

\* @param <T>

\* @return

\*/

public static <T> BaseResponse<T> success(T data){

return new BaseResponse<>(0,data,"ok");

}

/\*\*

\* 失败

\* @param errorCode

\* @return

\*/

public static BaseResponse error(ErrorCode errorCode){

return new BaseResponse<>(errorCode);

}

/\*\*

\* 失败

\* @param code

\* @param message

\* @param description

\* @return

\*/

public static BaseResponse error(int code,String message,String description){

return new BaseResponse<>(code,null,message,description);

}

/\*\*

\* 失败

\* @param errorCode

\* @param message

\* @param description

\* @return

\*/

public static BaseResponse error(ErrorCode errorCode,String message,String description){

return new BaseResponse<>(errorCode.getCode(),null,message,description);

}

/\*\*

\* 失败

\* @param errorCode

\* @param description

\* @return

\*/

public static BaseResponse error(ErrorCode errorCode,String description){

return new BaseResponse(errorCode.getCode(),errorCode.getMessage(),description);

}

}

自定义错误码 创建 ErrorCode

public enum ErrorCode {

SUCCESS(0,"ok",""),

PARAMS\_ERROR(40000,"请求参数错误",""),

NULL\_ERROR(40001,"请求数据为空",""),

NOT\_LOGIN(40100,"未登录",""),

NO\_AUTH(40101,"无权限",""),

SYSTEM\_ERROR(50000,"系统内部异常",""),

SAVE\_ERROR(40111,"保存失败","");

private final int code;

/\*\*

\* 状态码信息

\*/

private final String message;

/\*\*

\* 状态码描述

\*/

private final String description;

ErrorCode(int code, String message, String description) {

this.code = code;

this.message = message;

this.description = description;

}

public int getCode() {

return code;

}

public String getMessage() {

return message;

}

public String getDescription() {

return description;

}

}

**封装全局异常处理**

新建exception包，创建 BusinessException

/\*\*

\* 自定义异常类

\*/

public class BusinessException extends RuntimeException{

private final int code;

private final String description;

public int getCode() {

return code;

}

public String getDescription() {

return description;

}

public BusinessException( String message,int code, String description) {

super(message);

this.code = code;

this.description = description;

}

public BusinessException(ErrorCode errorCode) {

super(errorCode.getMessage());

this.code = errorCode.getCode();

this.description = errorCode.getDescription();

}

public BusinessException(ErrorCode errorCode, String description) {

super(errorCode.getMessage());

this.code = errorCode.getCode();

this.description = description;

}

}

全局异常处理器的作用： 捕获代码中所有的异常集中处理

新建 GlobalExceptionHandler

@RestControllerAdvice

@Slf4j

public class GlobalExceptionHandler {

@ExceptionHandler(BusinessException.class)

public BaseResponse businessExceptionHandler(BusinessException e){

log.error("businessException:"+e.getMessage(),e);

return ResultUtils.error(e.getCode(),e.getMessage(),e.getDescription());

}

@ExceptionHandler(RuntimeException.class)

public BaseResponse runtimeExceptionHandler(BusinessException e){

log.error("runtimeException:"+e);

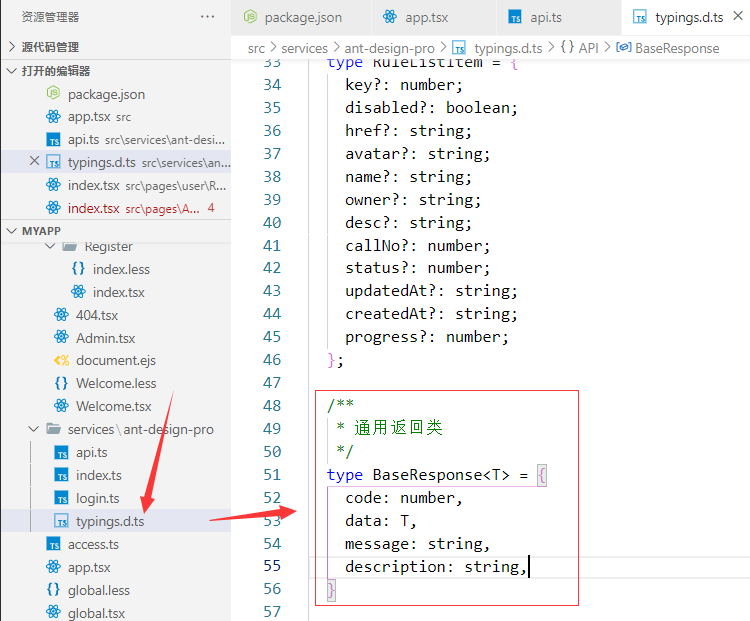
return ResultUtils.error(ErrorCode.SYSTEM\_ERROR,e.getMessage(),"");

}

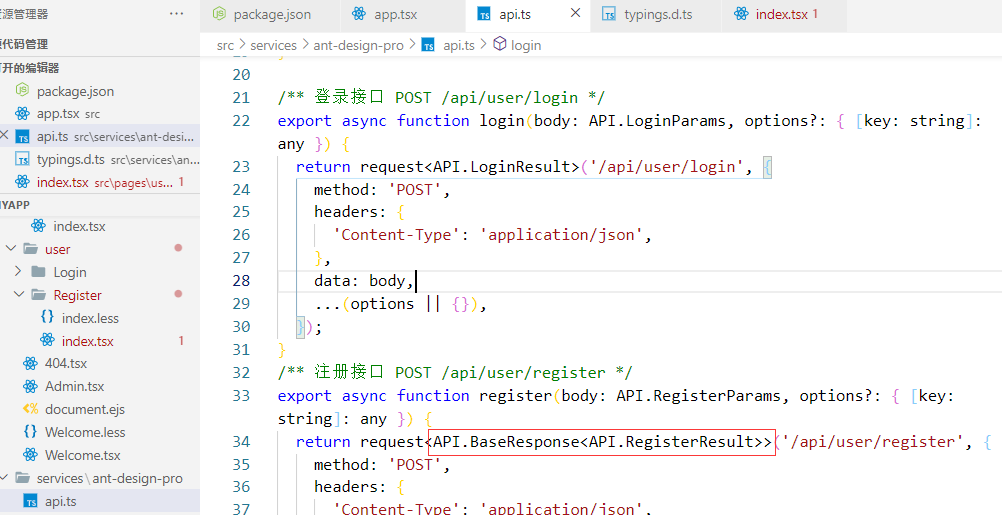
}

**前端优化**

在前端定义通用返回对象



给register 的 相应类型封装一下

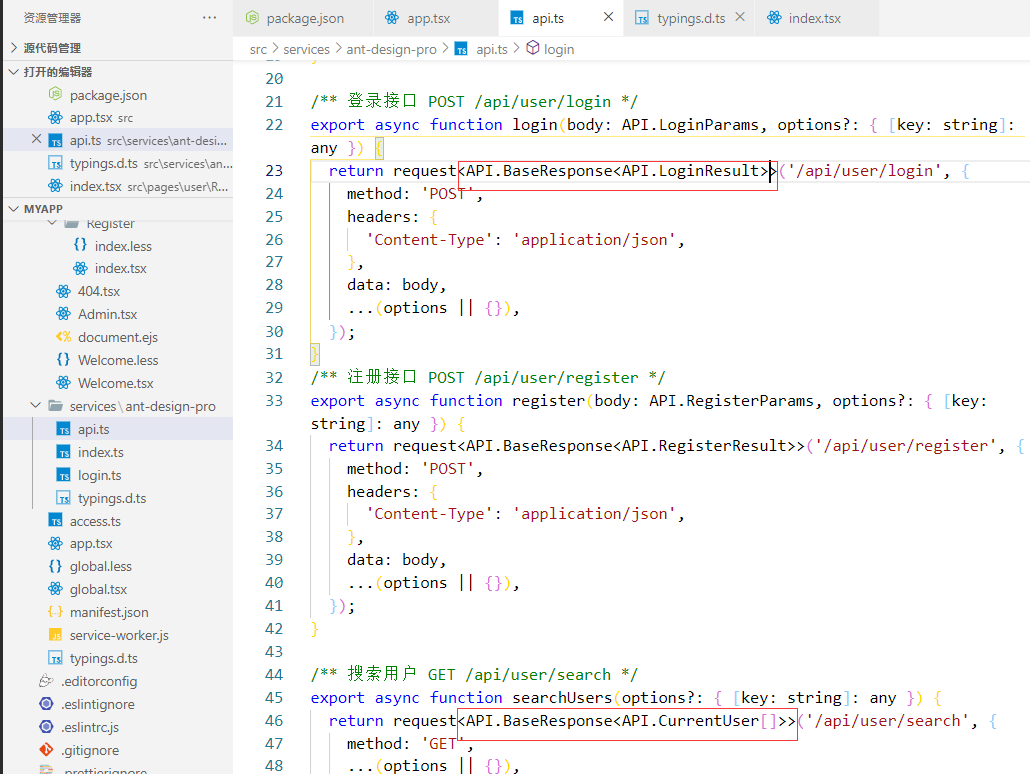


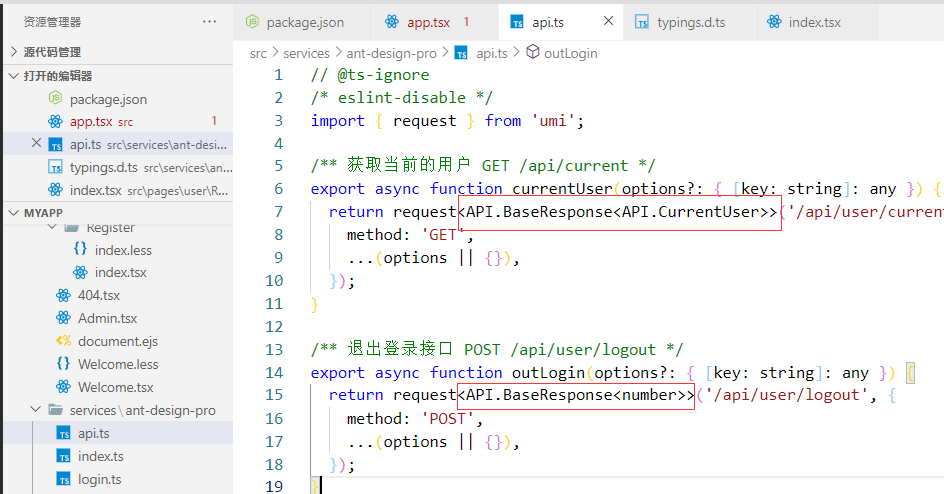
测试



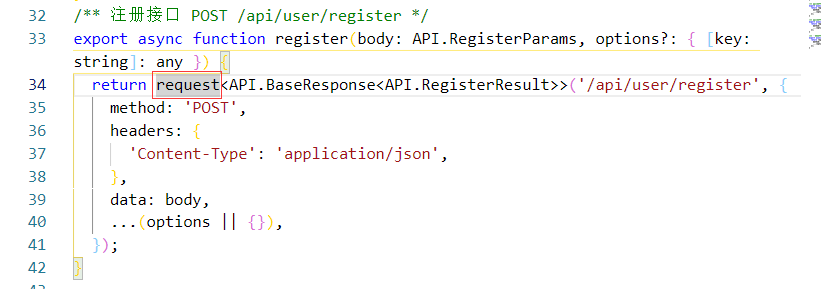
让前端和后端做适配

把API，全修改

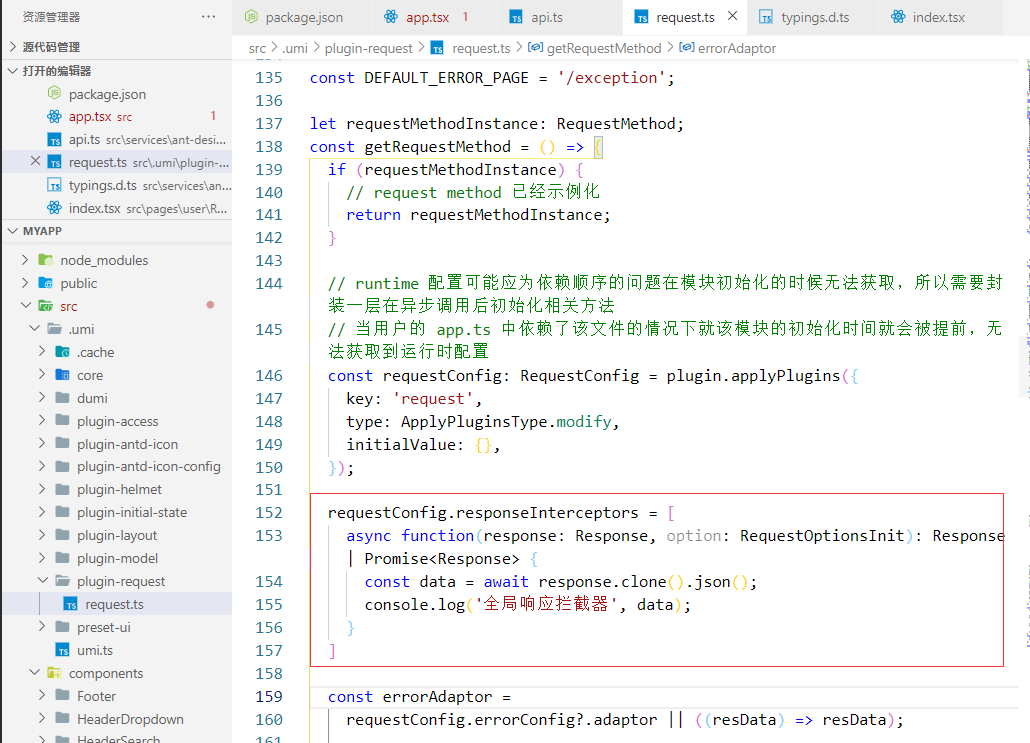




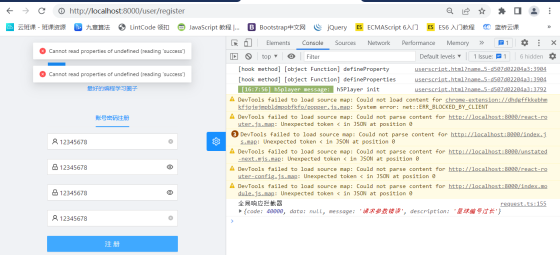
CTRL 点击 request



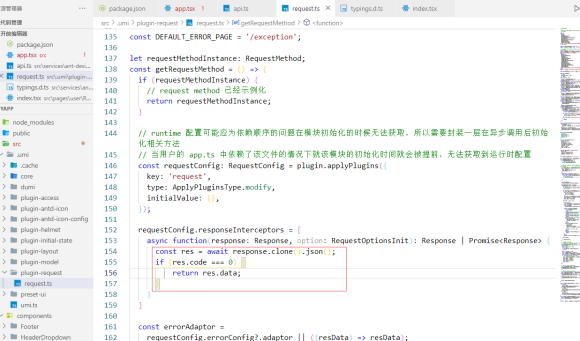
写一个全局响应拦截器



测试



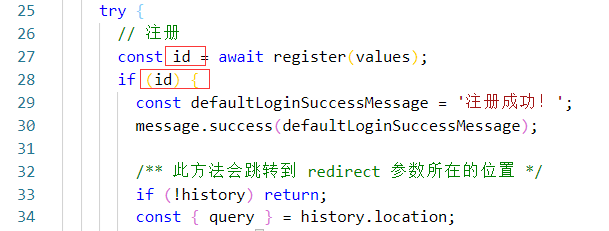
返回了后端传递出来的值，现在直接把所有接口中响应的data全部取出来

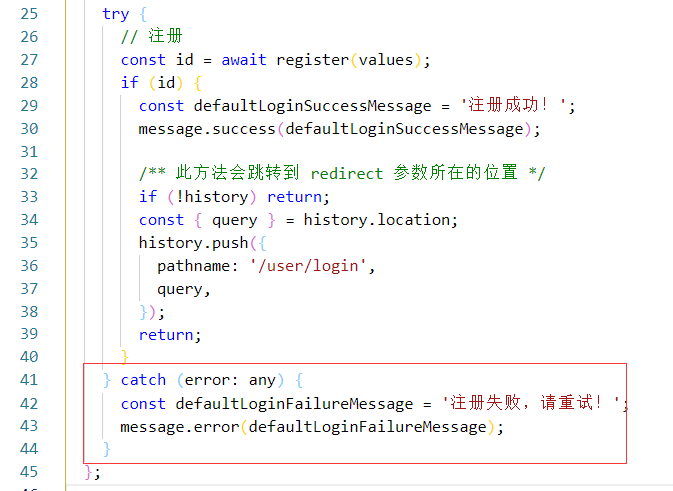


如果它等于 0 表示响应成功， 响应成功之后， 直接 return res.data ;

这样的话，前端不用做任何修改，之前我们把注册手动去判断 res.code 等于 0 判断data 是否为0， 现在直接取出来就是data

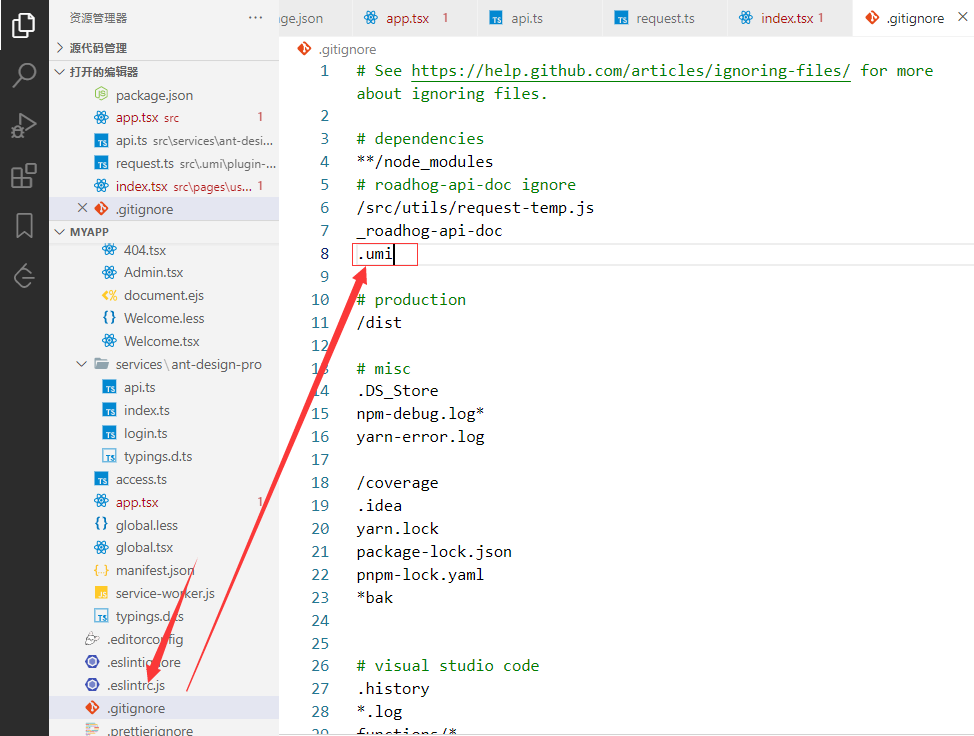
把 Register 包 下 的 index.tsx 代码修改为：





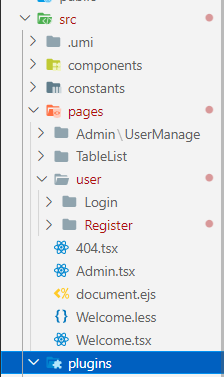
request.ts 属于 .umi 它是框架自动帮我们生成的文件，不是自己的，有可能被覆盖掉。

在.gitignore 添加上 .umi 让编辑器帮我们识别出来，它就是项目自动生成文件

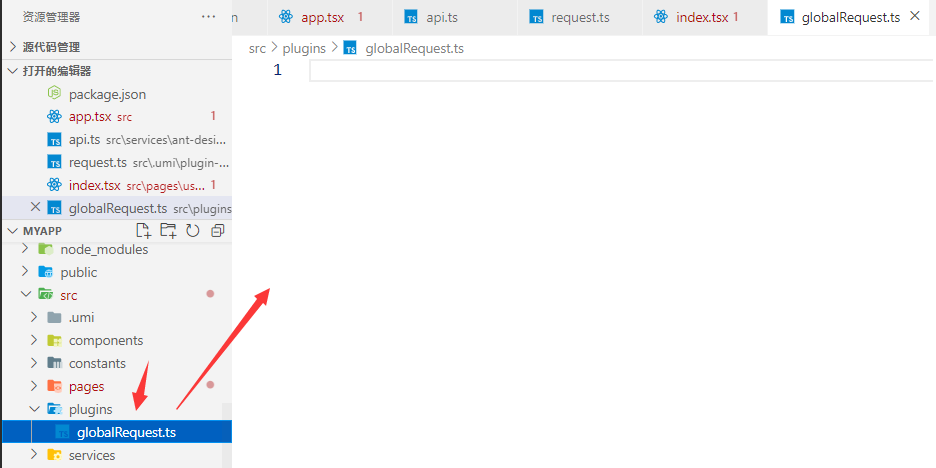


写一个请求类，覆盖默认的request方法

在src文件下新建 plugins文件夹



在plugins文件 新建 globalRequest.ts

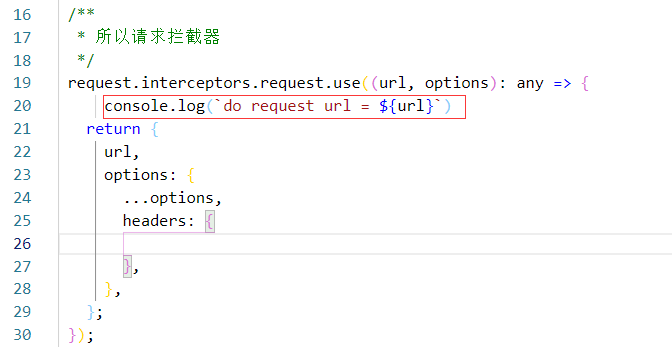


复制[(4条消息) 在Ant Design Pro中Umi-request全局请求响应的处理\_ant design request\_hanxiaozhang2018的博客-CSDN博客](https://blog.csdn.net/huantai3334/article/details/116780020)

到globalRequest.ts

删除多余代码

定义请求拦截器，比如说，我们可以在每次发请求的时候，console.log 一下



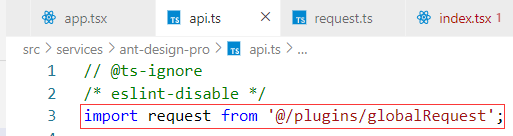
前端请求接收到一个响应之后，我们先获取到这个相应的data值，然后从这个data值中判断一下



把api.ts 引入我们自己写的



改成



测试

/\*\*

\* request 网络请求工具

\* 更详细的 api 文档: https://github.com/umi[js/umi-request](https://github.com/umijs/umi-request)

\*/

import {extend} from 'umi-request';

import {message} from "antd";

import {history} from "umi";

import {stringify} from "querystring";

import \* as process from "process";

/\*\*

\* 配置request请求时的默认参数

\*/

const request = extend({

credentials: 'include', // 默认请求是否带上c[ookie](https://github.com/umijs/umi-request)

prefix: process.env.NODE\_ENV=== 'pro[duction' ? 'http://user-backed.code-](https://github.com/umijs/umi-request)nav.cn' : undefined

// requestType: 'form',

});

/\*\*

\* 所以请求拦截器

\*/

request.interceptors.request.use((url, [options): any => {](https://github.com/umijs/umi-request)

console.log(`do request url= ${url}[`)](https://github.com/umijs/umi-request)

return {

url,

options: {

...options,

headers: {

},

},

};

});

/\*\*

\* 所有响应拦截器

\*/

request.interceptors.response.use(asyn[c (response, options): Promise<any>](https://github.com/umijs/umi-request) => {

const res = await response.clone().j[son();](https://github.com/umijs/umi-request)

if(res.code===0){

return res.data;

}

if (res.code===40100) {

message.error('请先登录');

history.replace({

pathname: '/user/login',

search: stringify({

redirect: location.pathname,

}),

});

}else {

message.error(res.description)

}

return res.data;

});

export default request;

**项目优化点**

功能扩充

管理员创建用户，修改用户信息，删除用户

上传头像

按照更多条件去查询用户

更改权限

修改Bug

项目登录改为分布式 session （单点登录 - redis)

通用性

set-cookie domain 域名更通用， 比如 改为 \*.xxx.com

把用户管理系统 => 用户中心 （周所有服务都请求这个后端）

后台添加全局请求拦截器 （统一去判断用户权限，统一记录请求日志）

**简历写法**

为了提高开发效率，选用 Ant Design Pro 脚手架快速搭建基础页面，并对原始模板进行瘦身，抽象为可复用的公共模板，便于后续同类项目的快速研发

在脚手架自带的 umi-request请求库基础上进行改造和封装，添加了全局请求拦截器和全局异常处理逻辑，自动根据项目启动命令来区分环境，减少重复代码，提升项目可维护性

选用MyBatis + MyBatis-Plus 进行数据访问层开发，复用大多数通用方法，并且通过继承定制了自己的通用操作模板，大幅提高了项目开发效率

为了明确接口的返回，自定义统一的错误码，并且封装全局异常处理类，从而规范了异常返回，屏蔽了项目冗余的报错细节

对于项目中JSON格式化处理对象，采用了双检锁单例模式 进行管理，从而复用对象，避免了重复创建对象的开销，便于集中维护管理（不懂）

采用Nginx 完成项目部署，采用Dokcer完成后端项目部署，并且使用宝塔面板对项目进行运维监控（未完成）