目录

软件工程Project后端说明文档

这里是软件工程Project后端项目的说明文档，本文档介绍以下内容

- 后端数据组织
- 后端设计思路
- 后端提供的api以及其实现思路
- 后端身份验证
- 项目目前存在的问题

后端用一个SpringBoot项目实现，请确保在阅读这份文档之前了解过SpringBoot的相关知识，至少要明白项目架构以及最基本的注解比如 `@Compent`、`@PostMapping`是干啥的

这里不介绍接口设计，详情请查看接口文档。

1. 数据组织

在后端中需要存储的信息有三类

- 用户信息
- 师生关系
- 申请信息

这些信息存放在 `/webapp`这一 `sqlite`数据库中

1.1 用户信息

这个存放在 `users`表中。

在后端，用户信息需要存储每个用户的信息与数据库表字段的对应关系如下

需要存储的信息	表中对应字段	类型	说明
用户UID	uid	char(20)	9位纯数字，后端自动生成，为user表的主键
用户登录密码	pwd	char(100)	无要求

需要存储的信息	表中对应字段	类型	说明
用户手机号	phone	char(20)	唯一
用户邮箱	email	char(40)	为邮箱格式，前端校验格式是否正确
用户姓名	name	varchar(20)	不唯一
用户昵称	nickname	varchar(20)	唯一
用户头像地址	usr_pic	varchar(120)	
用户个人主页网址	mypage	char(100)	为一个url，前端校验格式是否正确

在用户注册时，后端根据该用户是第几个注册的来自动生成一个 `uid`，因此还要维护用户注册人数这一信息。这里用另一张 `envvalue`表来存。

这张表本来是想存后端运行用到一些环境变量的，但是写完后发现只存储了注册人数这一信息。为什么不直接根据users中的条目数来判断第几个注册的呢？因为考虑到实际情况，会存在用户注销这一操作，所以直接根据users中的条目数来判断第几个注册会出现错误。~~（虽然这里没做注销的功能吧）~~

其中有两个字段 `env`和 `value`，都是 `char(50)`。

注册人数对应的 `env`为 `id_cont`，其 `value`值就是当前注册人数。

1.2 师生关系

这个存放在 `tree`表中

师生关系需要存储的信息与数据库表字段对应关系如下

需要存储的信息	表中对应字段	类型	说明
师生关系的老师	teacher_uid	char(20)	这里存储的是老师对应用户的UID
师生关系的学生	student_uid	char(20)	这里存储的是学生对应用户的UID
师生关系的层次	level	char(1)	0-本科、1-硕士、2-博士
师生关系开始时间	start_time	char(7)	格式为YYYYMM
师生关系结束时间	end_time	char(7)	格式为YYYYMM

1.3 申请信息

这个存放在 `request`表中

师生关系需要存储的信息与数据库表字段对应关系如下

需要存储的信息	表中对应字段	类型	说明
申请的id	rid	char(20)	用一个时间戳最为唯一标识
该申请被发送到谁	uid	char(20)	
该申请由谁提出	from_uid	char(20)	
该申请修改的结点	to_uid	char(20)	
修改类型	rtype	char(3)	由三位组成第一位代表该申请是由于对自己树修改-0而被提出，还是对别人树的修改-1而提出；第二位代表该申请的类型，删除-0，添加-1，编辑-2；第三位代表该申请操作的是一个老师-1还是学生-2，对于编辑操作不区分统一为0
新设置的层次	level	char(1)	操作的师生关系层次
新设置的	start_time	char(7)	只有添加、编辑操作此字段内容有效

需要存储的信息	表中对应字段	类型	说明
开始时间			
新设置的结束时间	end_time	char(7)	只有添加、编辑操作此字段内容有效

可能有些乱，举几个例子

假设张三想要添加李四为自己的老师

- uid字段：由于这个操作被提交到李四那里，所以值为李四的uid
- from_uid字段：由于这个操作是由张三提出的，所以值为张三的uid
- to_uid字段：如果该申请被通过，会在李四的树上添加一个老师结点张三，所以这里修改的结点是张三
    - 但是张三的树上也会添加李四为学生结点，为什么是张三而不是李四呢？这是因为这里修改的结点是指该**申请被提交到的用户的树对应的节点**。这个申请被提交到李四处，修改的是李四树上的张三节点，所以是张三而不是李四
- rtype字段：
    - 第一位：对自己树的修改，所以是0
    - 第二位：添加操作，所以是1
    - 第三位：添加的是老师，所以是1
- 其余字段该是啥是啥

再假设张三想要添加王五为李四的学生

- uid字段：由于这个操作被提交到李四那里，所以值为李四的uid
- from_uid字段：这个申请是由张三提出的，所以值为张三的uid
- to_uid字段：有了上面的解释，这里就应该是王五的uid
- 其余的不在赘述

2.设计思路

采用SpringBoot框架，整体分为controller层、service层、mapper层

每一层根据操作的数据库表的不同进行相应的划分，在controller层暴露api，service层提供具体的服务，mapper层提供数据操作

下面说明每个controller实现了哪些接口。

controller	api
UserController	1.1-1.8
RequestController	2.1-2.13
TreeController	3.1

2.1 设计的实体类

2.1.1 用户实体类Users

存储一个用户的所有信息，字段与数据库相对于

2.1.2 请求实体类Requests

存储一个请求的所有信息。要注意的是，存储申请类型时，将数据库的三位拆开存储。并且添加对应的姓名字段和申请描述字段以及构建描述的方法。这个挺简单的，自己看代码吧。

2.1.3 师承树节点实体类Node

存储一个树结点的所有信息，relationships存储该节点与用户的所有层次的师生关系。type指明该节点是老师节点还是学生节点还是用户个人节点。

2.1.4 响应数据实体类Result

这个我跟着视频学的，泛型啥的我也没整明白。

3.API实现思路

这里就按照接口文档中的顺序来了，具体介绍controller层、service层、mapper层各个层的操作

在接口文档中的申请相关接口中，有很多逻辑上相似的接口，这里就只讲几个不全讲。

3.1 注册

类注解

```
@RestController
@RequestMapping("/user")
public class UserController
```

第一个注解将这个类注册为一个 `RestController`， 第二个注解为这个 `Controller`的所有Api都加上一个前缀 `/user`

3.1.1 Controller层

代码

```
@PostMapping("/register")
public Result register(String userPhone,String password){
    //查询手机号是否被注册
    Users u=userService.findByPhone(userPhone);
    if(u==null){
        //没有占用
        // 注册
        userService.register(userPhone,password);
        return Result.success();
    }
    else {
        //占用
        return Result.error("手机号已被注册");
    }
}
```

其中 `@PostMapping("/register")`绑定这个方法处理 `/user/register`路径下的 `Post`请求，通过参数自动解析请求数据中的手机号和密码。

之后调用Service层的 `findBtPhone`方法查询该手机号是否已经被注册，如果被注册则返回一个带有错误提示的响应数据。

如果没有被注册则调用Service层的 `register`方法注册一个用户，并返回成功的响应数据。

3.1.2 Service层

在这里调用的Service层的方法有两个，分别是根据手机号查询用户的 findByPhone方法，和注册新用户的 register方法，下面分别对这两个方法进行讲解。

findByPhone方法

代码如下

```
@Override
public Users findByPhone(String userPhone) {
    Users u=userMapper.findByPhone(userPhone);
    return u;
}
```

直接调用Mapper层的 findByPhone方法获取用户实例并返回

register方法

代码如下

```
@Override
public void register(String userPhone, String password) {
    //加密密码
    String md5String=MD5Util.encrypt(password);
    //获取uid
    String cnt=envMapper.findVarByEnv("id_cnt");
    int new_cnt=Integer.parseInt(cnt)+1;
    envMapper.updateByEnv("id_cnt",String.valueOf(new_cnt));
    String uid= String.valueOf(100000000+new_cnt);
    //调用Mapper添加用户
    userMapper.addWithPhonePwdAndUid(userPhone,md5String,uid);
}
```

这里为了数据安全，在数据库中存储的并不是密码明文，而是经过MD5处理过后的密码。之后获得当前注册的用户数，并将其加一重新存储至数据库。然后根据构造 Uid并调用 Mapper层的 addWithPhonePwdAndUid方法将该用户添加至用户表中。

3.1.3 Mapper层

这里涉及到的Mapper层操作除了 UserMapper中的操作还有环境变量表的 EnvMapper操作。

先讲 EnvMapper。

findVarByEnv方法

代码如下

```
@Override
public String findVarByEnv(String env) {
    try {
        String sql = "select value from envvalue where env='" + env + "'";
        dbUtil.getConnection();
        ResultSet rs = dbUtil.executeQuery(sql);
        String value = "";
        while (rs.next()) {
            value = rs.getString("value");
        }
        dbUtil.close();
        return value;
    } catch (SQLException e) {
        dbUtil.close();
        throw new RuntimeException(e);
    }
}
```

首先构建查询指定环境变量的sql语句，之后调用数据库操作类获得查询结果并获取值，最后关闭数据库连接并返回对应的值。

updateByEnv方法

代码如下

```
@Override
public void updateByEnv(String env, String value) {
    dbUtil.getConnection();
    String sql = "update envvalue set value='" + value + "' where env='" + env +
"'";
    System.out.println(sql);
    dbUtil.executeUpdate(sql);
    dbUtil.close();
}
```

首先获取数据库连接并构建更新对应环境变量的sql语句，最后调用数据库操作类进行数据库更新操作，并关闭数据库。

之后讲解UserMapper中涉及的方法。一共有两个个 findByPhone和addWithPhonePwdAndUid。

findByPhone方法

代码如下

```
@Override
public Users findByPhone(String userPhone) {
    try {
        String sql = "select * from users where phone='" + userPhone + "'";
        return getUsers(sql);
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}

private Users getUsers(String sql) throws SQLException {
    System.out.println(sql);
    String uid;
    String pwd;
    String phone;
    String email;
    String name;
    String nickname;
    String usrPic;
    String myPage;
    dbUtil.getConnection();
    try (ResultSet rs = dbUtil.executeQuery(sql)) {
        uid = "";
        pwd = "";
        phone = "";
        email = "";
        name = "";
        nickname = "";
        usrPic = "";
        myPage = "";
        while (rs.next()) {
            uid = rs.getString("uid");
            pwd = rs.getString("pwd");
            phone = rs.getString("phone");
            email = rs.getString("email");
            name = rs.getString("name");
            nickname = rs.getString("nickname");
            usrPic = rs.getString("usr_pic");
            myPage = rs.getString("mypage");
        }
    }
    if (Objects.equals(uid, "")) {
        return null;
    }
    dbUtil.close();
    return new Users(uid, pwd, phone, email, name, nickname, usrPic, myPage);
}
```

首先构建查询的sql语句，之后调用 `getUsers`方法获取指定的用户实例。

在 `getUsers`中，根据查询到的结果构建对应的用户实例，如果不存在则直接返回 `null`。

addWithPhonePwdAndUid方法

代码如下

```
@Override
public void addWithPhonePwdAndUid(String userPhone, String password, String uid) {
    String sql = "insert into users(phone, pwd, uid) values('" + userPhone + "', '"
+ password + "','" + uid + "')";
    System.out.println(sql);
    dbUtil.getConnection();
    dbUtil.executeUpdate(sql);
    dbUtil.close();
}
```

很简单没啥要讲的。

3.2 登录

3.2.1 Controller层

代码如下

```
@PostMapping("/login")
public Result<String> login(String id, String password, int pori) {
    System.out.println(id + password + pori);
    Users u;
    //查询用户
    if (pori == 0) {
        u = userService.findByPhone(id);
    } else if (pori == 1) {
        u = userService.findByUid(id);
    } else {
        return Result.error("请求参数错误，pori只能为0或1");
    }
    //判断用户是否存在
    if (u == null) {
        return Result.error("用户名错误");
    }
    //判断密码是否正确
    if (MD5Util.encrypt(password).equals(u.getPwd())) {
        //生成JWT令牌
        Map<String, Object> claims = new HashMap<>();
        claims.put("uid", u.getUid());
        claims.put("phone", u.getPhone());
        String token = JwtUtil.genToken(claims);
        return Result.success(token);
    }
```

```
        return Result.error("密码错误");
    }
```

首先根据用户登录凭证类型查询用户，并检查前端传参时候合法。之后根据查询结构判断用户是否存在。如果不存在则返回一个错误响应通知前端**用户名错误**。如果存在则判断登录密码是否正确。如果不正确则返回一个错误响应通知前端**密码错误**。 如果密码正确，则根据登录用户的uid和phone生成一个JWT令牌返回前端。

3.2.2 Service层

涉及的Service层方法有两个，分别为 `findByPhone`和 `findByUid`。`findByPhone`前面已经讲过，这里只讲另一个。

`findByUid`方法

代码如下

```
public Users findByUid(String id) {
    Users u = userMapper.findByUid(id);
    return u;
}
```

和 `findByPhone`方法类似。

3.2.3 Mapper层

只有一个 `findByUid`方法要讲，和前面Mapper层讲过的 `findByPhone`方法类似。

3.3 获取用户信息

3.3.1 Controller层

代码如下

```
@GetMapping("/userinfo")
public Result<Users> userInfo() {
    //根据用户名查询用户
    // 优化：在拦截器中解析了token，在这里使用解析结果，
    // 拦截器解析时使用ThreadLocal存储线程局部变量
    // 后续在Control、Service、Mapper层使用时使用ThreadLocal获取局部变量
    Map<String, Object> map = ThreadLocalUtil.get();
```

```
    String uid = (String) map.get("uid");
    Users user = userService.findByUid(uid);
    return Result.success(user);
}
```

首先从 `ThreadLocal`中获取获取解析请求中 `JWT`令牌得到的用户 `uid`。之后调用Service层的 `findByUid`方法获取用户实例。由于Service层和Mapper层并没有新的处理逻辑，所以后面两项直接略过。

`ThreadLocal`这一部分怎么实现的放到4.用户验证讲。

3.3.2 Service层

略

3.3.3 Mapper层

略

3.4 更新用户信息

3.4.1 Controller层

代码如下

```
@PutMapping("/update")
public Result update(@RequestBody Users user) {
    //判断user是否存在
    if (userService.findByUid(user.getUid()) == null) {
        return Result.error("用户不存在");
    }
    //进行更新操作
    int flag = userService.update(user);
    //判断是否出错
    if (flag == 0) {
        return Result.success();
    } else if (flag == 1) {
        return Result.error("手机号已被占用");
    } else if (flag == 2) {
        return Result.error("昵称已被占用");
    } else {
        return Result.error("未知错误：更新用户信息-1");
    }
}
```

首先判断要更新的用户是否存在，这一步可以省略因为前端只有在用户登录后才会发送这个请求，且后端服务端口并不对外暴露。之后调用Service层的update方法更新用户并获取返回值。最后根据返回值判断更新是否成功，如果成功返回成功响应。否则返回带有错误提示信息的响应数据。

3.4.2 Service层

涉及的新的Service层方法只有一个 `update`方法。

代码如下

```
@Override
public int update(Users user) {
    return userMapper.update(user);
}
```

没啥好讲的。

3.4.3 Mapper层

只涉及一个 `update`方法

代码如下

```
public int update(Users user) {
    dbUtil.getConnection();
    String uid = user.getUid();
    String phone = user.getPhone();
    String email = user.getEmail();
    String name = user.getName();
    String nickname = user.getNickname();
    String myPage = user.getMyPage();
    String sql = "select * from users where phone='" + phone + "' and uid!='" + uid
+ "'";
    ResultSet rs = dbUtil.executeQuery(sql);
    //出错标识 0-正常，1-重复手机号，2-重复昵称
    try {
        if (rs.next()) {
            dbUtil.close();
            return 1;
        }
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
    sql = "select * from users where nickname='" + nickname + "' and uid!='" + uid
+ "'";
    rs = dbUtil.executeQuery(sql);
```

```
    try {
        if (rs.next()) {
            dbUtil.close();
            return 2;
        }
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
    sql = "UPDATE users SET "
            + "phone = '" + phone + "', "
            + "email = '" + email + "', "
            + "name = '" + name + "', "
            + "nickname = '" + nickname + "',"
            + "mypage = '" + myPage + "' "
            + "WHERE uid = '" + uid + "'";
    System.out.println(sql);
    dbUtil.executeUpdate(sql);
    dbUtil.close();
    return 0;
}
```

首先检查设置的手机号是否与其他人的重复。之后检查设置的昵称是否与其他人重复。（这两部分应该在Service层实现的）

检查无误后，根据传入的Users类实例构建更新sql语句并执行更新操作。

3.5 更新用户头像地址

这个和3.4类似，不讲了。

3.6 更新密码

3.6.1 Controller层

代码如下

```
@PatchMapping("updatePwd")
public Result updatePwd(@RequestBody Map<String, String> params) {
    int flag = userService.updatePwd(params.get("old_pwd"), params.get("new_pwd"));
    if (flag == 0) {
        return Result.success();
    } else {
        return Result.error("密码错误");
    }
}
```

首先调用Service层的 updatePwd方法更新密码，并获取其返回值。如果没有问题直接返回成功响应，否则返回失败响应提示前端**密码错误**。

3.6.2 Service层

updatePwd方法

代码如下

```
public int updatePwd(String oldPwd, String newPwd) {
    String md5OldPwd = MD5Util.encrypt(oldPwd);
    String md5NewPwd = MD5Util.encrypt(newPwd);
    return userMapper.updatePwd(md5OldPwd, md5NewPwd);
}
```

对原密码和新密码经过MD5处理后调用Mapper层的 updatePwd方法并返回其返回值。

3.6.3 Mapper层

updatePwd方法

代码如下

```
public int updatePwd(String oldPwd, String newPwd) {
    Map<String, Object> map = ThreadLocalUtil.get();
    String uid = (String) map.get("uid");
    System.out.println(uid);
    String sql = "select * from users where uid='" + uid + "' and pwd='" + oldPwd +
"'";
    dbUtil.getConnection();
    ResultSet rs = dbUtil.executeQuery(sql);
    try {
        if (rs.next()) {
            sql = "update users set pwd='" + newPwd + "'";
            dbUtil.executeUpdate(sql);
            dbUtil.close();
            return 0;
        } else {
            dbUtil.close();
            return 1;
        }
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
```

首先获得uid并检查原密码是否正确。如果错误则返回1。正确则执行更新操作。

3.7 上传头像

这一部分直接在Controller层实现了。

逻辑我是直接用gpt写的，头像存储和返回新路径都没问题，但是前端就是访问不到新的头像，直接报404。

这部分就是接受头像文件并保存到本地。这里用了一个usrPicDir类，主要是为了获取配置文件中设置的存储路径和访问路径。

存储逻辑如下：

- 在存储时，文件名为uid+两位标识
- 如果这是用户第一张头像，则标识为00
- 之后每次更新头像都将两位标志加一存储并删除旧头像。99之后变为00

这样做主要是为了避免前端的缓存导致头像更新后的回显出问题。

返回的响应信息中携带新的用户头像访问地址以便头像回显。

这里存在问题，由于后端Spring项目中无法访问到设置的静态资源导致回显出现问题。

3.8 搜索用户

3.8.1 Controller层

代码如下

```
@PostMapping("/searchUsers")
public Result searchUsers(String id, String type) {
    ArrayList<Users> u = new ArrayList<>();
    if (Objects.equals(type, "0")) {
        u = userService.findByName(id);
    } else if (Objects.equals(type, "1")) {
        Users user = userService.findByUid(id);
        u.add(user);
    } else if (Objects.equals(type, "2")) {
        Users user = userService.findByPhone(id);
        u.add(user);
    } else {
        return Result.error("type字段不符合要求");
    }
```

```
    return Result.success(u);
}
```

首先根据前端返回的搜索类型调用Service层的不同的方法获取用户实例，之后返回携带这些用户的响应数据。

3.8.2 Service层

`findByName`方法

代码如下

```
public ArrayList<Users> findByName(String id) {
    return userMapper.findByName(id);
}
```

直接调的Mapper层方法，并返回其返回值。

3.8.3 Mapper层

`findByName`方法

代码如下

```
public ArrayList<Users> findByName(String id) {
    ArrayList<Users> u = new ArrayList<>();
    dbUtil.getConnection();
    String sql = "select * from users where name='" + id + "'";
    ResultSet rs = dbUtil.executeQuery(sql);
    String uid;
    String pwd;
    String phone;
    String email;
    String name;
    String nickname;
    String usrPic;
    String myPage;
    while (true) {
        try {
            if (!rs.next()) break;
            uid = rs.getString("uid");
            pwd = rs.getString("pwd");
            phone = rs.getString("phone");
            email = rs.getString("email");
            name = rs.getString("name");
            nickname = rs.getString("nickname");
            usrPic = rs.getString("usr_pic");
```

```
                myPage = rs.getString("mypage");
                Users user = new Users(uid, pwd, phone, email, name, nickname, usrPic,
myPage);
                u.add(user);
            } catch (SQLException e) {
                throw new RuntimeException(e);
            }
        }
        dbUtil.close();
        return u;
    }
```

首先构建查询sql语句，之后根据查询结果构建对应的Users类的实例并用一个ArrayList组织起来。最后返回这个ArrayList。

3.9 添加为自己的学生

3.9-3.17都是在讲与申请有关的API。代码文件为 `RequestController`、`RequestServiceImpl`、`RequestMapperImpl`。

3.9.1 Controller层

代码如下：

```
@PostMapping("/addMyStudent")
public Result addMyStudent(String addUid, String level, String startTime, String
endTime) {
    int flag = requestService.addMyStudent(addUid, level, startTime, endTime);
    if (flag == 0) {
        return Result.success();
    } else if (flag == 1) {
        return Result.error("已经有相同操作");
    } else if (flag == 2) {
        return Result.error("已经是您的学生");
    } else {
        return Result.error("不能添加老师为学生");
    }
}
```

首先调用Service层的addMyStudent方法并获取其返回值。之后根据返回值判断该申请是否被添加，如果出错则返回对应的错误响应否则返回成功响应。

在这里需要注意申请的同类检查，即判断当前待处理申请中是否存在与该申请进行的操作相同的申请，这一部分在下面的Service层和Mapper层详细讲解。

3.9.2 Service层

`addMyStudent`方法

代码如下

```
public int addMyStudent(String addUid, String level, String startTime, String
endTime) {
    Map<String, Object> map = ThreadLocalUtil.get();
    String uid = (String) map.get("uid");
    //检查是否已经是自己的学生
    if (treeMapper.IsTeacherOfWhomInLevel(uid, addUid, level)) {
        return 2;
    }
    //检查是否为自己的老师
    if (treeMapper.IsTeacherOfWhomInAnyLevel(addUid, uid)) {
        return 3;
    }
    return requestMapper.addMyStudent(addUid, level, startTime, endTime);
}
```

在Service层，需要进行的检查有要添加的学生是否已经是自己该层次的学生，以及该学生是否为自己的老师。通过检查后调用Mapper层的 `addMyStudent`方法进行同类申请检查并添加申请。

3.9.3 Mapper层

调用的Mapper层方法中有TreeMapper的 `IsTeacherOfWhomInLevel`方法和requestMapper的 `addMyStudent`方法。

`IsTeacherOfWhomInLevel`方法

代码如下

```
public boolean IsTeacherOfWhomInLevel(String tUid, String sUid, String level) {
    treeDbUtil.getConnection();
    String sql = "select * from tree where teacher_uid='" + tUid + "' and
student_uid='" + sUid + "' and level='" + level + "'";
    ResultSet rs = treeDbUtil.executeQuery(sql);
    try {
        if (rs.next()) {
            treeDbUtil.close();
            return true;
        }
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
```

```
        treeDbUtil.close();
        return false;
    }
```

就一个非常简单的查询操作。

addMyStudent方法

代码如下

```
@Override
public int addMyStudent(String addUid, String level, String startTime, String
endTime) {
    dbUtil.getConnection();
    Map<String, Object> map = ThreadLocalUtil.get();
    String uid = (String) map.get("uid");
    //检查是否有同样作用的请求，自己已经申请过或者别人给自己申请过
    if (isRepeatOfMyRequest(addUid, "11", level) == 1) {
        dbUtil.close();
        return 1;
    }
    String rid = TimeStamp.getTimeStamp();
    String rType = "011";
    String sql = "insert into request
(rid,uid,from_uid,to_uid,rtype,level,start_time,end_time) " +
            "values ('" + rid + "','" + addUid + "','" + uid + "','" + uid + "','"
+ rType + "','" + level + "','" + startTime + "','" + endTime + "')";
    dbUtil.executeUpdate(sql);
    dbUtil.close();
    return 0;
}
```

首先获取当前发出申请的用户uid。之后进行同类申请检查，如果检查通过则添加对应的申请。申请的各个字段含义见1.3 申请信息。

下面讲解同类申请判断以及代码实现。

对于一个申请，其的同类申请是当申请通过后对师承树进行的操作与该申请相同的操作。

比如，张三添加李四为老师和李四添加张三为学生就是同类申请。又因为，这里允许他人对自己的树进行修改。所以，张三添加李四为老师也与其他人添加李四为张三的老师为同类申请。类似的，李四添加张三为学生也与他人添加张三为李四的学生为同类申请。

所以这四个申请就是一个同类的申请族。注意，进行判断时还要判断添加和删除的师生关系层次是否一致。对于编辑申请，只关注是否对某一层次进行修改，而不管修改后的

起始时间是否一致。也就是说，只要申请中存在对该层次的修改申请，那么另一个修改申请就不能添加进去，即使他们两个申请修改的时间不一样。

那么该如何进行判断呢。这里为了方便统一对自己树的修改和对他人树的修改的判断，统一用uid、from_uid、to_uid、level来进行讲解，对自己树的修改就可以看作from_uid和to_uid相同的申请。同时，同类判断只关注当前申请想要修改的树。在设计思路时不考虑同意他人申请后会转变为自己申请这一改变。这样并不会导致检查不全，因为在考虑被修改结点是否存在同样申请时就会一并将转变后的同类申请检查了。

当我们拿到一个申请Uid,FromUid,ToUid，Level时。首先要检查被修改的树的用户是否已经提出相同的申请，这里又分为对自己的修改即uid=ToUid,from_uid=Uid,to_uid=Uid,level=Level，类型做类似的修改和他人的修改即uid=Uid,to_uid=ToUid，类型做类似修改。以及被修改的结点用户是否提出相同的申请，考虑其自己提出申请即uid=uid,from_uid=ToUid,to_uid=ToUid，类型做类似修改和他人对其提出的申请即uid=ToUid,to_uid=Uid，类型做类似修改。

下面证明**在考虑被修改结点是否存在同样申请时就会一并将转变后的同类申请检查了**这一论断。

假设有这样的申请：张三想要添加李四为王五的学生。

根据上面的检查，要检查的申请有：王五要添加李四为自己的学生，其他人要添加李四为王五的学生，李四要添加王五为自己的老师，其他人想要添加王五为李四的老师。

那么当王五同意了这个申请之后，这个申请就会变成：王五想要添加李四为自己的学生。可以发现，转变后的申请的同类申请在没有转变之前都进行了检查。

代码这里就不贴了，存在bug，只考虑了**被修改的树的用户是否已经提出相同的申请**没有**考虑被修改的结点用户是否提出相同的申请**。

把这块整明白了其他的树操作申请就只是改变rtype，逻辑是相同的。所以其他的申请操作就不讲了。

3.10 获取所有请求

3.10.1 Controller层

代码如下

```
@GetMapping("/getAllRequests")
public Result<ArrayList<Requests>> getAllRequests() {
    ArrayList<Requests> r = requestService.getAllRequests();
    return Result.success(r);
}
```

就只是单纯的调用Service层方法后获取所有申请并封装到响应数据中返回

3.10.2 Service层

`getAllRequests`方法

代码如下

```
public ArrayList<Requests> getAllRequests() {
    ArrayList<Requests> requests = requestMapper.getAllRequests();
    //解析 fromName toName description
    for (Requests r : requests) {
        String uid = r.getFromUid();
        Users u = userMapper.findByUid(uid);
        r.setFromName(u.getName());
        uid = r.getToUid();
        u = userMapper.findByUid(uid);
        r.setToName(u.getName());
        r.genDescription();
    }
    return requests;
}
```

首先调用Mapper层的方法获取申请，之后填充其中的fromName和toName，并生成该申请的描述。

3.10.3 Mapper层

`getAllRequests`方法

代码如下

```
@Override
public ArrayList<Requests> getAllRequests() {
    Map<String, Object> map = ThreadLocalUtil.get();
    String uid = (String) map.get("uid");
    ArrayList<Requests> requests = new ArrayList<>();
    dbUtil.getConnection();
    String sql = "select * from request where uid='" + uid + "'";
```

```
    try (ResultSet rs = dbUtil.executeQuery(sql)) {
        String rid;
        String fromUid;
        String fromName;
        String toUid;
        String toName;
        String level;
        String meRoOthers;//0-对自己树的申请，1-对他人树的申请
        String type;//0-删除，1-增加，2-修改
        String sOrt;//1-学生，2-老师
        String startTime;
        String endTime;
        while (rs.next()) {
            rid = rs.getString("rid");
            fromUid = rs.getString("from_uid");
            fromName = "";
            toUid = rs.getString("to_uid");
            toName = "";
            level = rs.getString("level");
            String rtype = rs.getString("rtype");
            meRoOthers = String.valueOf(rtype.charAt(0));
            type = String.valueOf(rtype.charAt(1));
            sOrt = String.valueOf(rtype.charAt(2));
            startTime = rs.getString("start_time");
            endTime = rs.getString("end_time");
            requests.add(new Requests(rid, fromUid, fromName, toUid, toName, level,
meRoOthers, type, sOrt, startTime, endTime, ""));
        }
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
    dbUtil.close();
    return requests;
}
```

就是获取查询结果并构建申请实例组成一个ArrayList返回

3.11 同意他人请求

3.11.1 Controller层

直接调的Service层方法

3.11.2 Service层

代码如下

```
public void acceptRequest(String rid) {
    Requests request = requestMapper.acceptRequest(rid);
```

```
        String uid = request.getUid();
        String fromUid = request.getFromUid();
        String toUid = request.getToUid();
        String meRoOthers = request.getMeRoOthers();
        String type = request.getType();
        String sOrt = request.getSOrt();
        String tid;
        String sid;
        String level = request.getLevel();
        String startTime = request.getStartTime();
        String endTime = request.getEndTime();
        //根据request中的内容修改tree表
        if (Objects.equals(meRoOthers, "0")) {
            if (Objects.equals(type, "0")) {
                if (Objects.equals(sOrt, "0")) {
                    //删除自己的学生
                    tid = request.getFromUid();
                    sid = request.getUid();
                } else {
                    //删除自己的老师
                    sid = request.getFromUid();
                    tid = request.getUid();
                }
                treeMapper.del(tid, sid, level);
            } else if (Objects.equals(type, "1")) {
                if (Objects.equals(sOrt, "0")) {
                    //增加自己老师
                    tid = request.getFromUid();
                    sid = request.getUid();
                } else {
                    //增加自己的老师
                    sid = request.getFromUid();
                    tid = request.getUid();
                }
                treeMapper.add(tid, sid, level, startTime, endTime);
            } else {
                //修改自己树结点
                tid = request.getFromUid();
                sid = request.getUid();
                if (treeMapper.IsTeacherOfWhomInLevel(tid, sid, level)) {
                    treeMapper.modify(tid, sid, level, startTime, endTime);
                } else {
                    treeMapper.modify(sid, tid, level, startTime, endTime);
                }
            }
            requestMapper.refuseRequest(rid);
        } else {
            requestMapper.changeToMyRequest(rid, uid, fromUid, toUid, "0" + type +
    sOrt);
        }
    }
```

首先调用Mapper层的acceptRequest方法获取申请实例，之后根据申请类型不同对树做出相应的修改。如果是他人对自己的树的修改申请，则需要将其转变为自己的申请并存入申请表。

3.11.3 Mapper层

对树的操作很简单，这里不讲了。acceptRequest方法也只是查询数据库并构建相应的申请实例，也不讲了。

changeToMyRequest方法也只是将他人申请就地转变为自己申请，也很简单，需要注意一下最后一个参数s的构造。

3.12 拒绝他人请求

这个没啥好讲的就单纯的删除申请就行了

3.13 获取指定用户的师承树

3.13.1 Controller层

直接调Service层方法

3.13.2 Service层

```
public ArrayList<Node> getTree(String uid) {
    ArrayList<Node> nodes = treeMapper.getTree(uid);
    //解析nodes中的name和myPage
    for (Node node : nodes) {
        Users u = userMapper.findByUid(node.getUid());
        node.setName(u.getName());
        node.setMyPage(u.getMyPage());
    }
    return nodes;
}
```

调Mapper层方法获取结点列表，之后填充每个结点的name和mypage，最后返回。

3.13.3 Mapper层

```
public ArrayList<Node> getTree(String uid) {
    ArrayList<Node> nodes = new ArrayList<>();
    treeDbUtil.getConnection();
    //获取老师
    String sql = "select * from tree where student_uid='" + uid + "'";
    try (ResultSet rs = treeDbUtil.executeQuery(sql)) {
```

```
            String tUid;
            String type;
            while (rs.next()) {
                tUid = rs.getString("teacher_uid");
                type = "1";
                addRelation(nodes, rs, tUid, type);
            }
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
        //获取学生
        sql = "select * from tree where teacher_uid='" + uid + "'";
        try (ResultSet rs = treeDbUtil.executeQuery(sql)) {
            String sUid;
            String type;
            while (rs.next()) {
                sUid = rs.getString("student_uid");
                type = "2";
                addRelation(nodes, rs, sUid, type);
            }
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
        nodes.add(new Node(uid, "0", "", ""));
        treeDbUtil.close();
        return nodes;
    }
```

首先获取老师结点，之后获取学生结点，最后添加自己的结点。需要注意的是，一条边可能会存在多个层次的师生关系，这些师生关系都要存在一个结点中。

4.用户验证

这里采用JWT令牌的方式实现用户验证，在用户登录时后端返回前端一个JWT令牌，之后每次请求都要携带这个JWT令牌。

为了实现用户验证，使用一个拦截器实现，在请求到来之前检查其是否携带了JWT令牌，并且验证JWT令牌是否有效。如果有效则将JWT令牌中携带的uid和phone保存到ThreadLocal中，以便后续处理直接使用。如果未通过JWT验证，则拦截访问，并返回状态码401。

在每个请求结束后，清除ThreadLocal中存储的uid和phone信息。

ThreadLocal原理，在处理每一次请求时，Spring都会单独开一个线程。在这个线程中可以使用ThreadLocal存储线程局部信息，这个信息是线程独占的其他线程不能访问到这个线程的ThreadLocal中的数据。

5.项目目前存在的问题

数据库连接未使用连接池实现会存在并发处理请求时数据库连接问题。

同类申请判断考虑不全。

头像回显访问不了静态资源。