**伙伴匹配系统**

**需求分析**

用户去添加标签，标签的分类（要有哪些标签，怎么把标签进行分类）

主动搜索： 允许用户根据标签去搜索其他用户

Redis缓存

组队

创建队伍

加入队伍

根据标签查询队伍

邀请其他人

允许用户去修改标签

推荐

相似度计算算法 + 本地分布式计算

**技术选型**

**前端**

Vue3 （开发框架，提高页面开发效率）

Vant UI 组件库 （基于Vue 的移动端组件库）（React 版 Zent )

Vite 脚手架 (打包工具，快！）

Axios 请求库

Nginx 单机部署

**后端**

Java SpringBoot 框架

MySQL 数据库

MyBatis-Plus

MyBatis X 自动生成

Redis 缓存 （Spring Data Redis 等多种实现方式）

Redisson 分布式锁

Easy Excel 数据导入

Spring Scheduler 定时任务

Swagger + Knife4j 接口文档

Gson : JSON 序列化库

相似度匹配算法

**部署**

Serverless 服务

云原生容器平台

**直播1**

前端项目初始化

前端主页 + 组件概览

数据库表设计

标签表

用户表

开发后端 - 根据标签搜索用户

**前端项目初始化**

**初始化项目，建议用脚手架**

Vite 脚手架：<https://vitejs.cn/guide/#scaffolding-your-first-vite-project>

整合组件库vant:

安装vant

按需引入

**开发页面经验**

多参考

从整体到局部

先想清楚页面要做成什么样子，再写代码

**在需要创建项目的目录输入cmd 来初始化搭建项目**

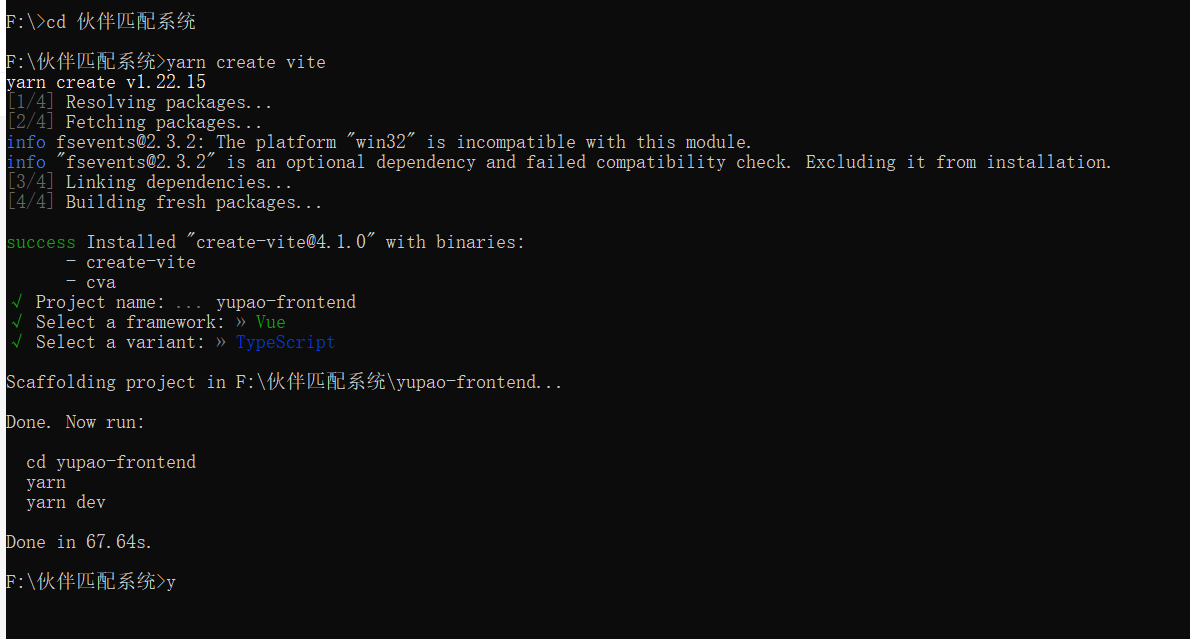
yarn create vite

第一个输入项目的名称

第二个选择脚手架类型 ----vue

第三个选择语言------ts

初始化总截图



**初始化完善**

用WebStorm打开已构建的项目

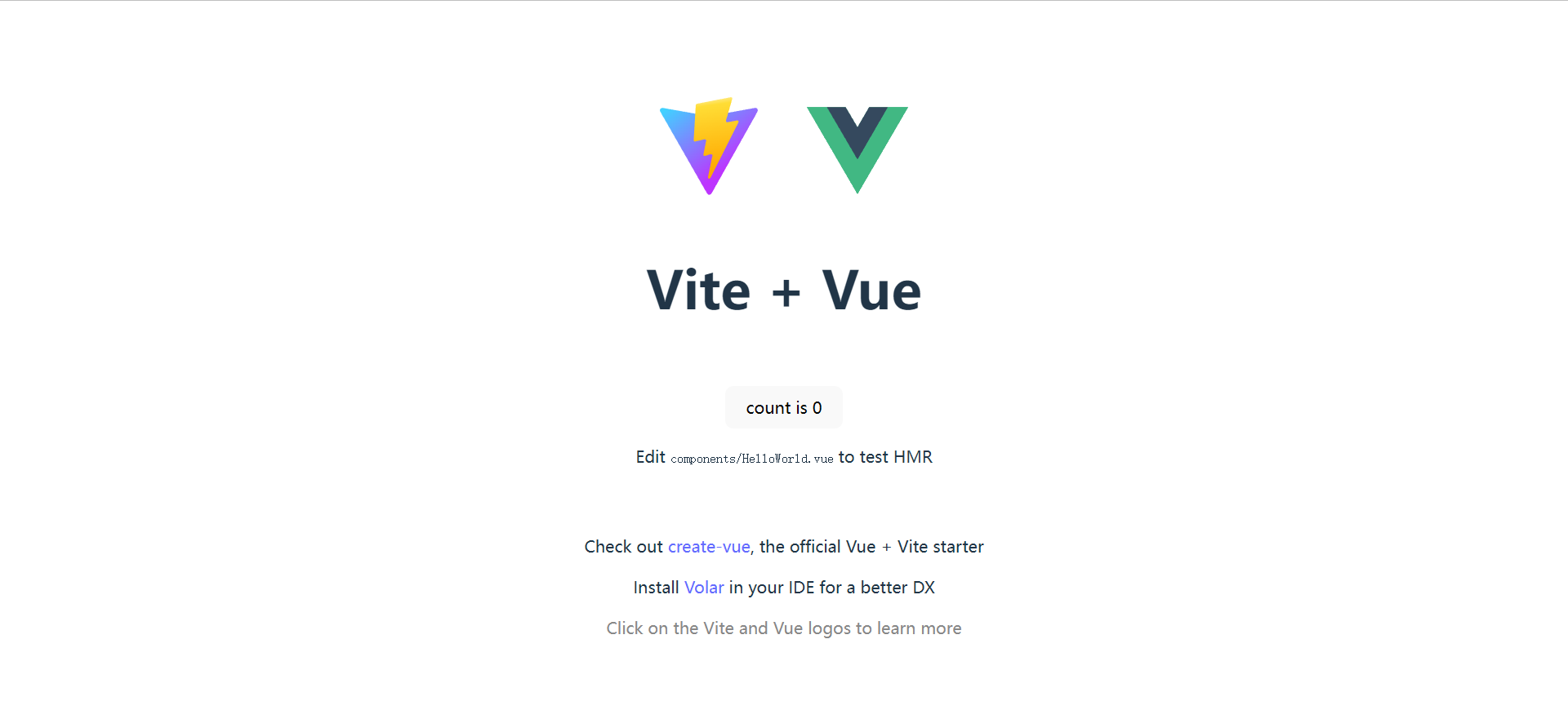
在终端中 输入 yarn 或者 npm install 导入依赖

在package.json 中点击 dev 启动

如果发现端口不是3000 ，为了后续方便，可以修改vite.config.ts文件



启动



**引入vant组件库**

安装插件

npm i unplugin-vue-components -D //通过 npm 安装

配置插件

安装完成后，在 vite.config.js 文件中 配置插件

import { defineConfig } from 'vite'

import vue from '@vitejs/plugin-vue'

import Components from 'unplugin-vue-components/vite';

import { VantResolver } from 'unplugin-vue-components/resolvers';

// https://vitejs.dev/config/

export default defineConfig({

plugins: [vue(),

Components({

resolvers:[VantResolver()],

})],

})

引入组件

在main.ts

import { createApp } from 'vue'

import './style.css'

import App from './App.vue'

import {Button} from 'vant'

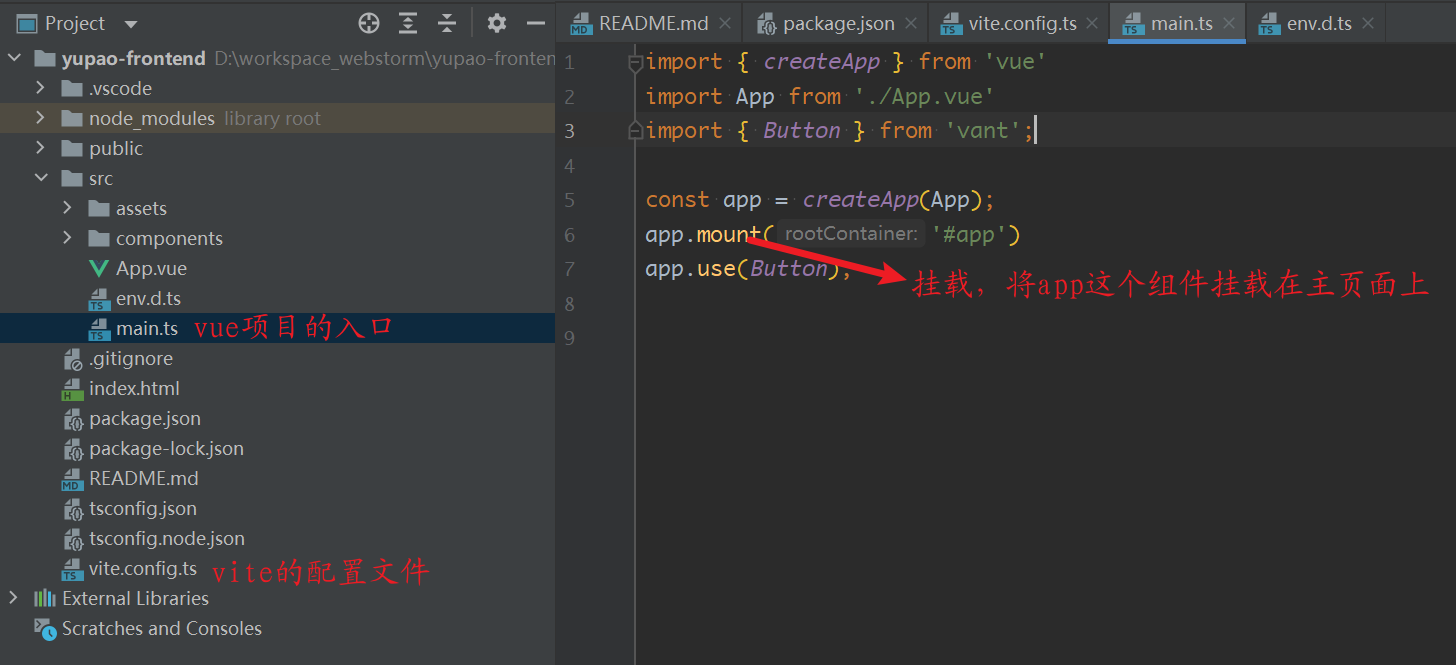
const app=createApp(App);

app.use(Button);

app.mount('#app')

安装vant组件

npm i vant



测试 引入是否成功

<van-button type="primary">主要按钮</van-button>

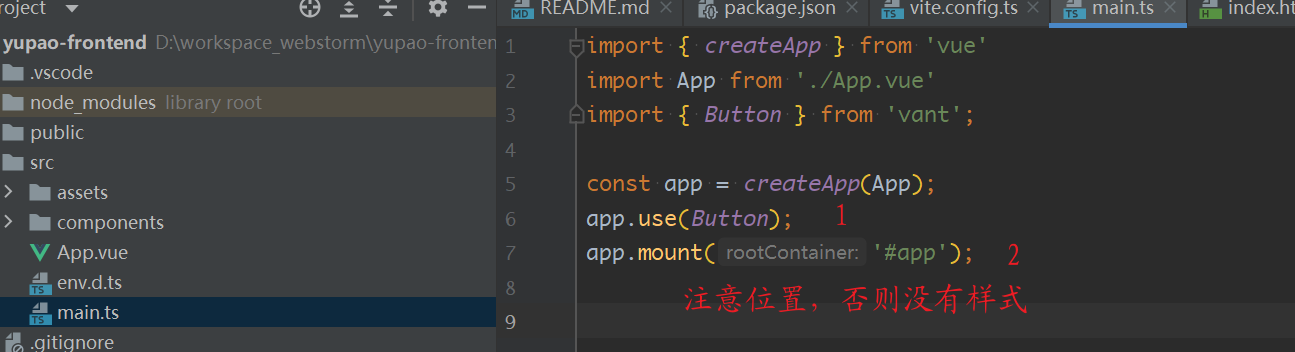
<van-button type="success">成功按钮</van-button>

<van-button type="default">默认按钮</van-button>

<van-button type="warning">警告按钮</van-button>

<van-button type="danger">危险按钮</van-button>

注意



修改vite.config.ts代码，增加如下代码（更改路径）

export default defineConfig({

plugins: [

vue(),

styleImport({

resolves: [VantResolve()],

//增加的代码

libs: [

{

libraryName: 'vant',

esModule: true,

resolveStyle: name => `../es/${name}/style`

}

]

}),

],

server: {

port: 3000,

}

})

启动

**前端主页 + 组件概览**

**设计**

导航条： 展示当前页面名称

主页搜索框 ==> 搜索页==> 搜索结果页（标签筛选页）

内容

tab栏：

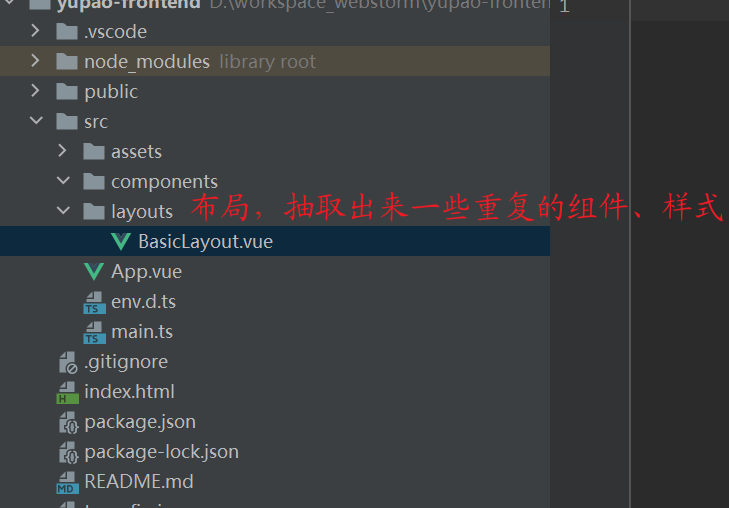
主页（推荐页+广告 ）

队伍页

用户页 （消息 - 暂时考虑邮件发送）

**开发**

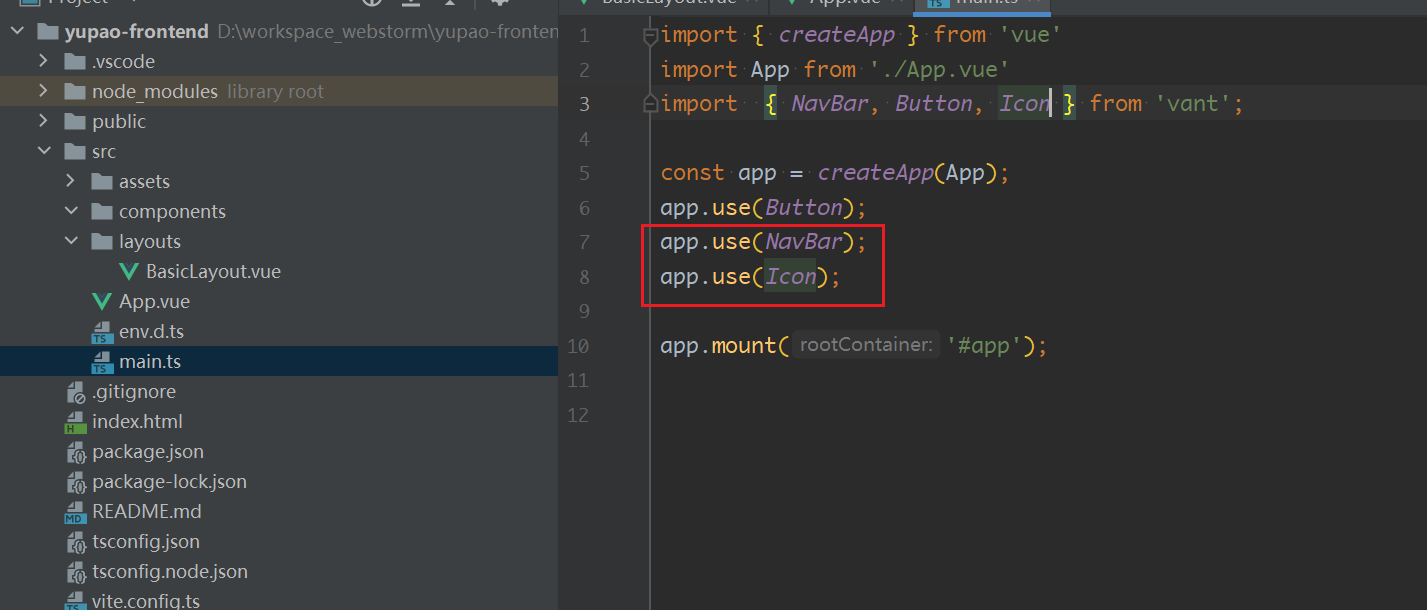
很多页面要重用组件 / 样式， 重复写 很麻烦， 不利于维护， 所以抽象一个通用的布局 （ Layout ）

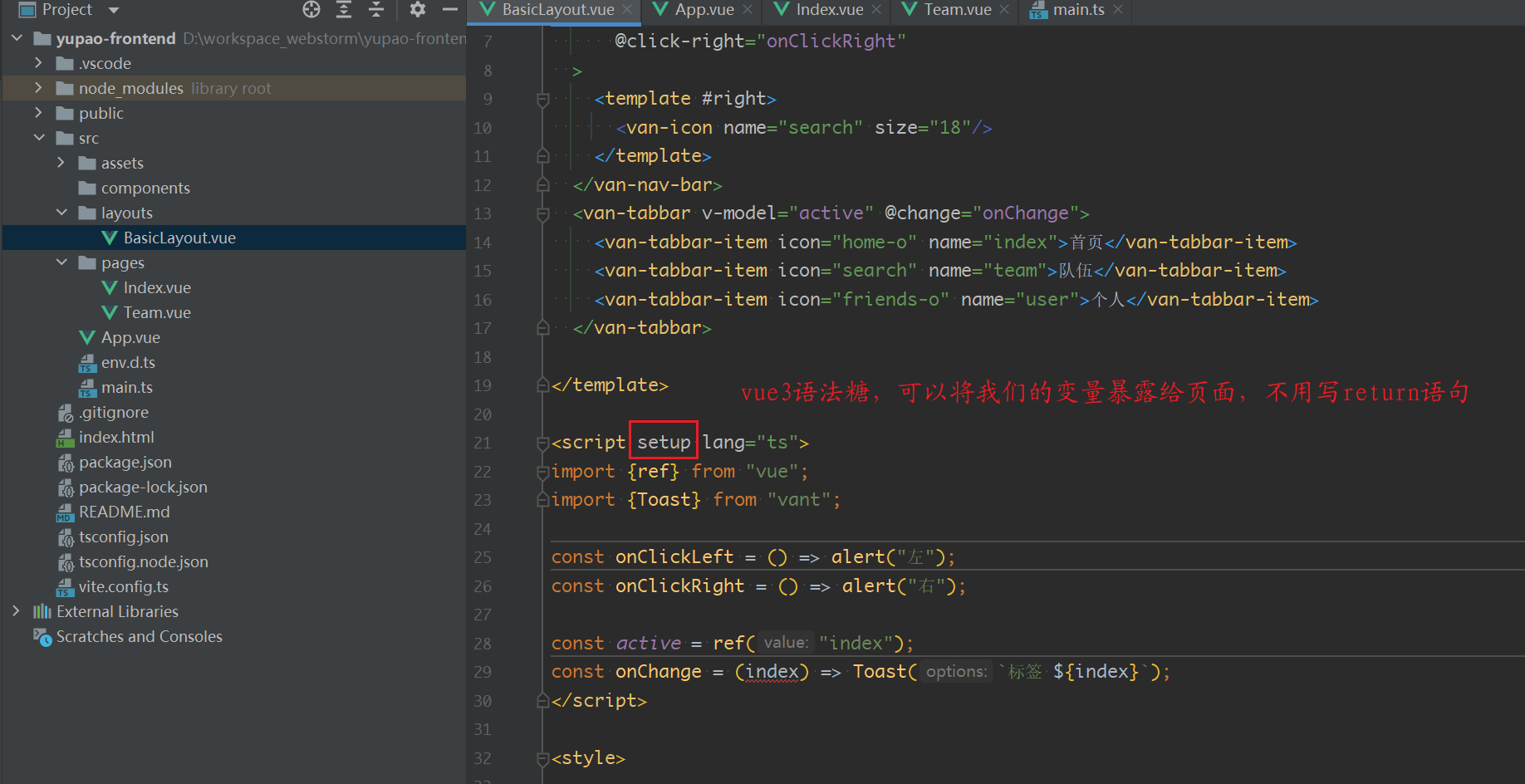


**删除App.vue中无用的代码，只剩下框架；删除components中的hello文件，建立layouts文件夹（复用，通用布局）**

**NavBar 导航栏**

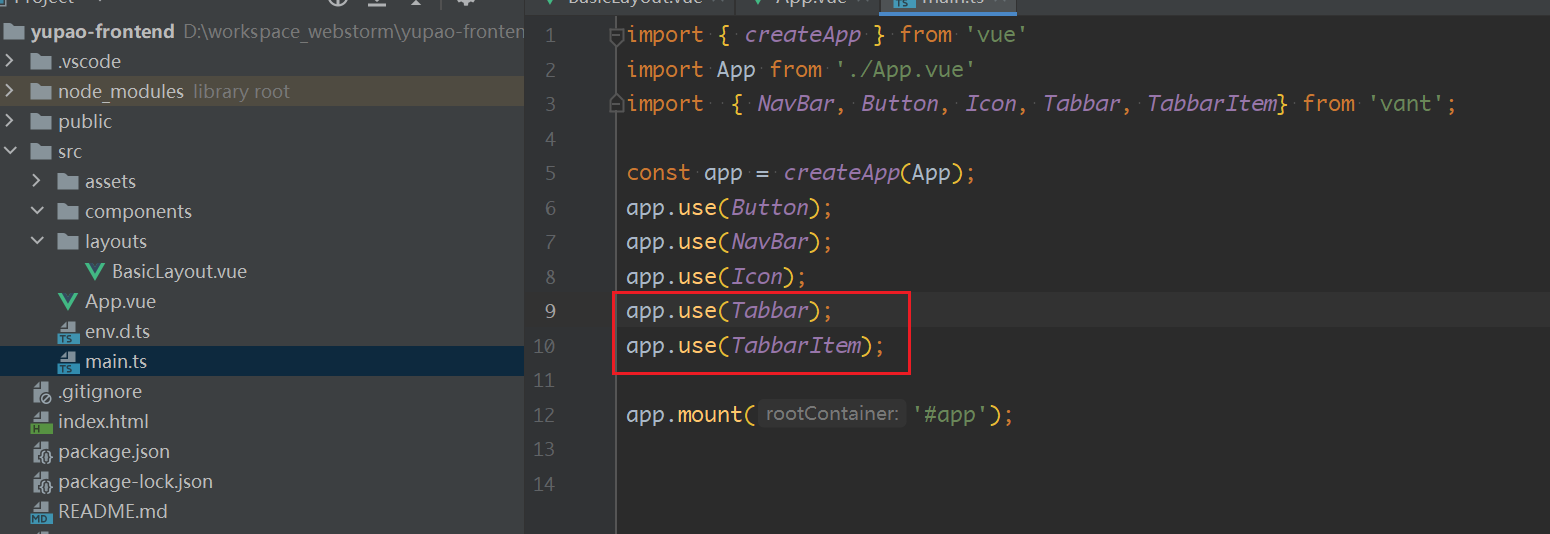
引入组件 （ **按需引入** ）

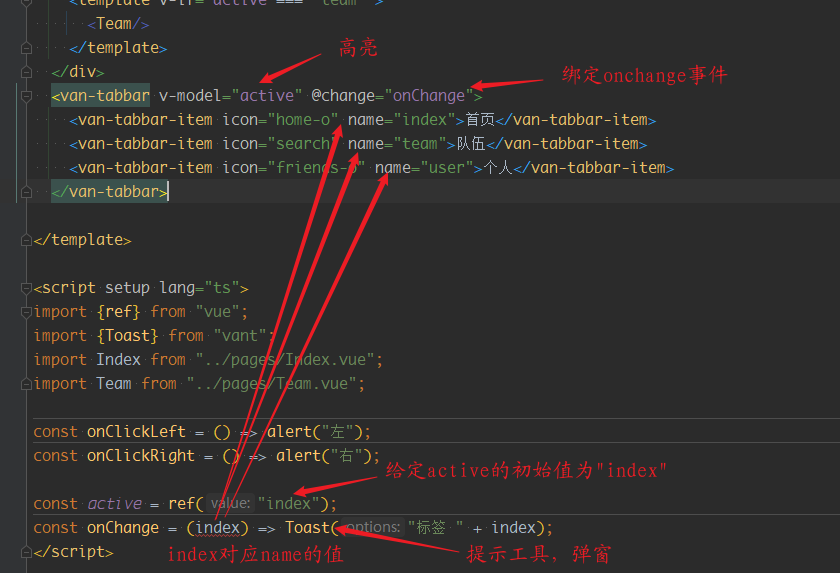




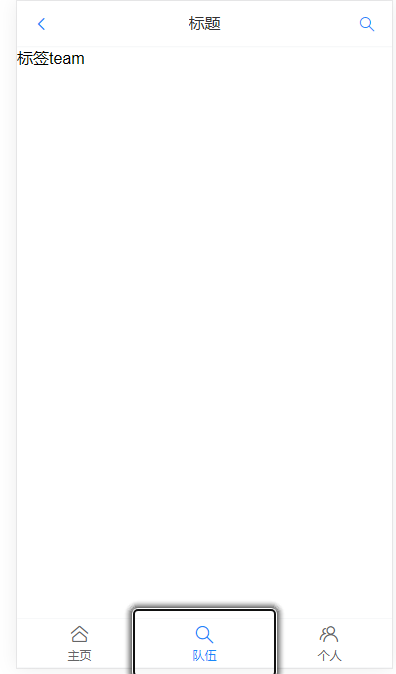
**Tabbar 标签栏**

引入组件

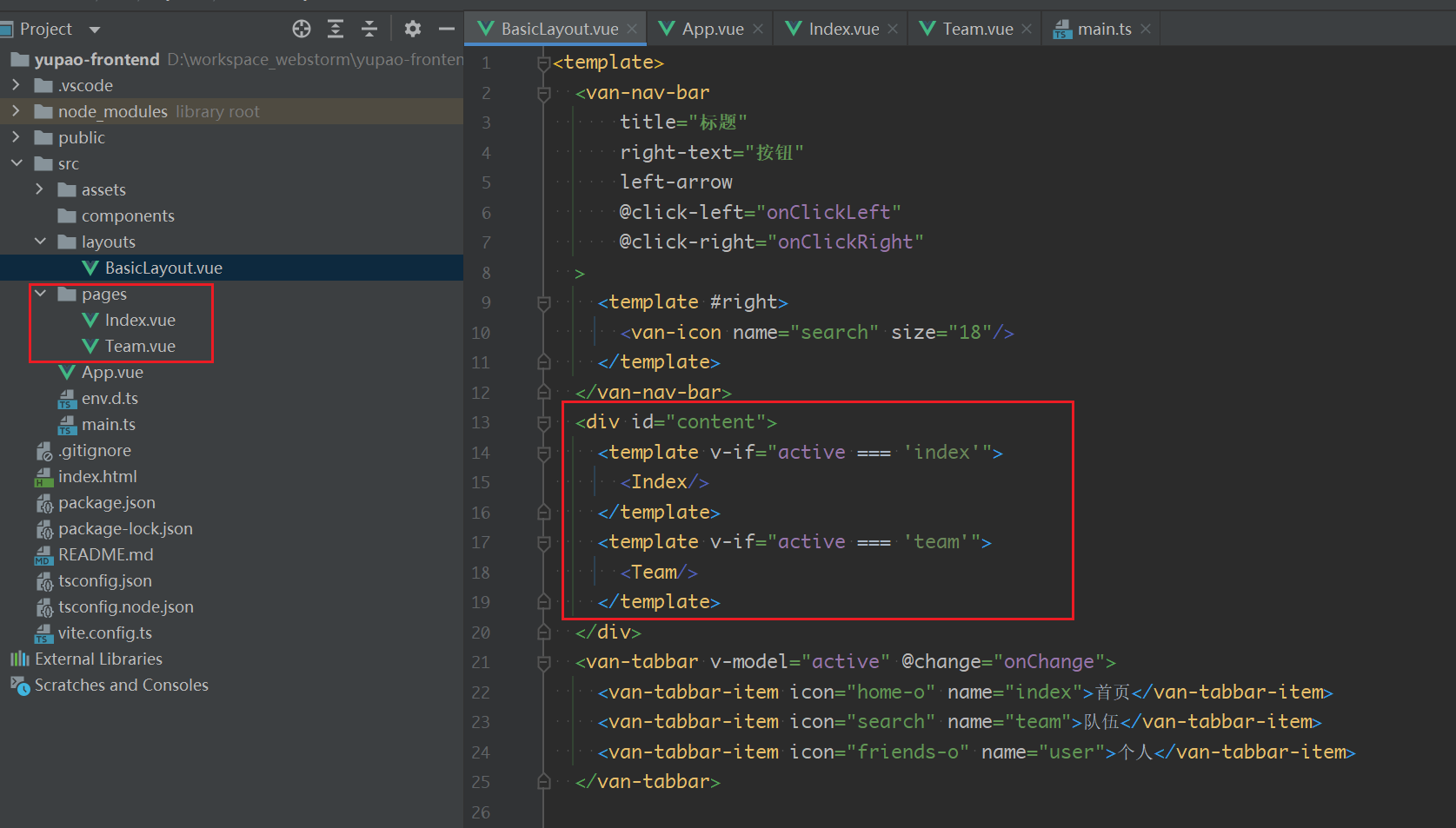




效果：



组件化：



**数据库表设计**

标签的分类 （ 要有哪些标签，怎么把标签进行分类 ）

**新增标签表 （ 分类表 ）**

建议用标签，不要用分类，更灵活

建议用标签，不要用分类，更灵活。

性别：男、女

方向：Java、C++、Go、前端

正在学：Spring

目标：考研、春招、秋招、社招、考公、竞赛（蓝桥杯）、转行、跳槽

段位：初级、中级、高级、王者

身份：小学、初中、高中、大一、大二、大三、大四、学生、待业、已就业、研一、研二、研三

状态：乐观、有点丧、一般、单身、已婚、有对象

【用户自定义标签】？

字段：

id int 主键

标签名 varchar 非空 （ 必须唯一，唯一索引 )

上传标签的用户 userId int ( 如果要根据 userId 查已上传标签的化，最好加上，普通索引 )

父标签id parentId tinyint ( 0 , 1 ) （分类）

创建时间 createTime datatime

更新时间 updateTime datatime

是否删除 isDelete tinyint ( 0 , 1 )

怎么查询所有标签，并且把标签分好组 ？ 查询所有，然后根据父标签id 进行分组 √

根据父标签查询子标签 ？ 根据id 查询 √

建表

create table tag

(

id bigint auto\_increment comment 'id' primary key,

tagName varchar(256) null comment '标签名称',

userId bigint null comment '用户 id',

parentId bigint null comment '父标签 id',

isParent tinyint null comment '0 - 不是， 1 - 父标签',

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

updateTime datetime default CURRENT\_TIMESTAMP null on update CURRENT\_TIMESTAMP,

isDelete tinyint default 0 null comment '是否删除'

)

comment '标签';

**SQL语言分类：**

DDL define 建表， 操作表

DML manage 更新删除数据，影响实际表的内容

DCL controller 控制，权限

DQL query 查询

[DQL、DML、DDL、DCL的概念与区别 - 范兵 - 博客园 (cnblogs.com)](https://www.cnblogs.com/fan-yuan/p/7879353.html)

**修改用户表**

用户有那些标签？

**根据自己的实际需求来！ 选择**

直接在用户表补充 tags 字段 ['java','男'] 存 json 字符串

优点 ： 查询方便，不用新建关联表， 标签是用户的固有属性 （ 除了该系统， 其他系统可能要用到） 节省开发成本

**查询用户列表，查关系表拿到 100 个用户有的所有标签id 再根据标签id 去查询标签表。**

哪怕性能低，可以用缓存。

缺点： 用户表多一列，会有点。。。

加一个关联表，记录用户和标签的关系

关联表的应用场景： 查询灵活，可以正查反查

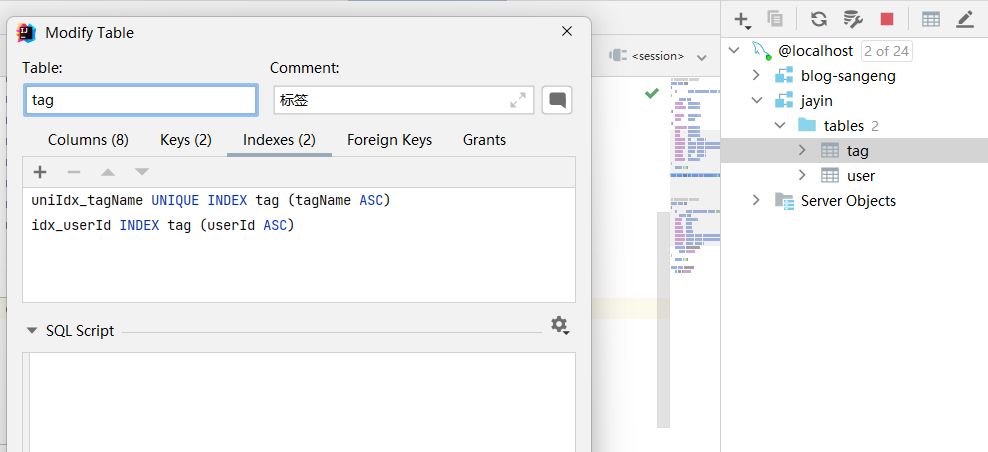
缺点 ： 要多建一个表，多维护一个表

**重点 ：企业大项目开发中尽量减少关联查询，很影响扩展性，而且会影响查询性能**

选择第一种。 在用户表添加tags 字段

alter table user add COLUMN tags varchar(1024) null comment '标签列表'

添加索引



-- auto-generated definition

create table tag

(

id bigint auto\_increment comment 'id'

primary key,

tagName varchar(256) null comment '标签名称',

userId bigint null comment '用户 id',

parentId bigint null comment '父标签 id',

isParent tinyint null comment '0 - 不是， 1 - 父标签',

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

updateTime datetime default CURRENT\_TIMESTAMP null on update CURRENT\_TIMESTAMP,

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

constraint uniIdx\_tagName

unique (tagName)

)

comment '标签';

create index idx\_userId

on tag (userId);

**开发后端接口**

**笔记**

**1.搜索标签**

允许用户传入多个标签，多个标签都存在才搜索出来 and。like '%Java%' and like '%C++%'。

允许用户传入多个标签，有任何一个标签存在就能搜索出来 or。like '%Java%' or like '%C++%'

**两种方式：**

SQL 查询（实现简单，可以通过拆分查询进一步优化）

内存查询（灵活，可以通过并发进一步优化）

如果参数可以分析，根据用户的参数去选择查询方式，比如标签数

如果参数不可分析，并且数据库连接足够、内存空间足够，可以并发同时查询，谁先返回用谁。

还可以 SQL 查询与内存计算相结合，比如先用 SQL 过滤掉部分 tag

建议通过实际测试来分析哪种查询比较快，数据量大的时候验证效果更明显！

**2.解析 JSON 字符串**

序列化：java对象转成 json反序列化：把 json 转为 java 对象java  json 序列化库有很多：

gson（google 的）

fastjson alibaba（ali 出品，快，但是漏洞太多）

jackson

kryo

**初始化后端**

**查询带有标签用户的数量时，利用到user表，所以在用户中心项目编写代码**

用户中心 来 集中 提供用户的检索操作，注册，登录，鉴权

直接在用户中心的基础上写。

service

/\*\*

\* 根据标签搜索用户

\* @param tagNameList 用户拥有的标签

\* @return

\*/

List<User> searchUsersByTags(List<String> tagNameList);

serviceImpl

@Override

public List<User> searchUsersByTags(List<String> tagNameList) {

if (CollectionUtils.isEmpty(tagNameList)){

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

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

//拼接 and 查询

//like '%java%' and like '%Python%'

for(String tagName : tagNameList){

queryWrapper= queryWrapper.like("tags",tagName);

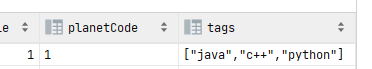
}

List<User> userList = userMapper.selectList(queryWrapper);

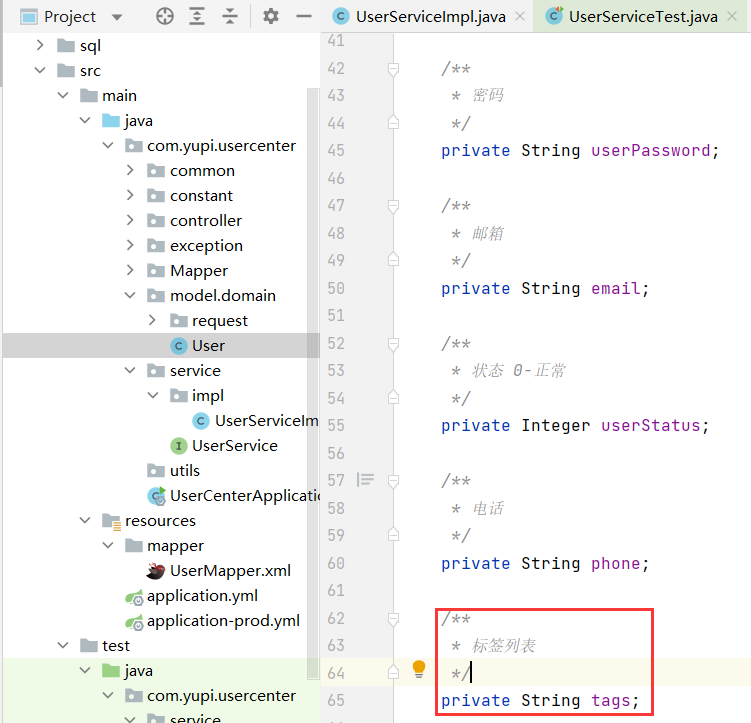
return userList.stream().map(this::getSafteyUser).collect(Collectors.toList());

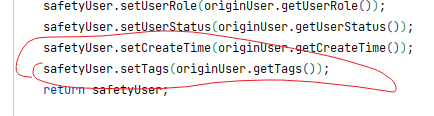
}

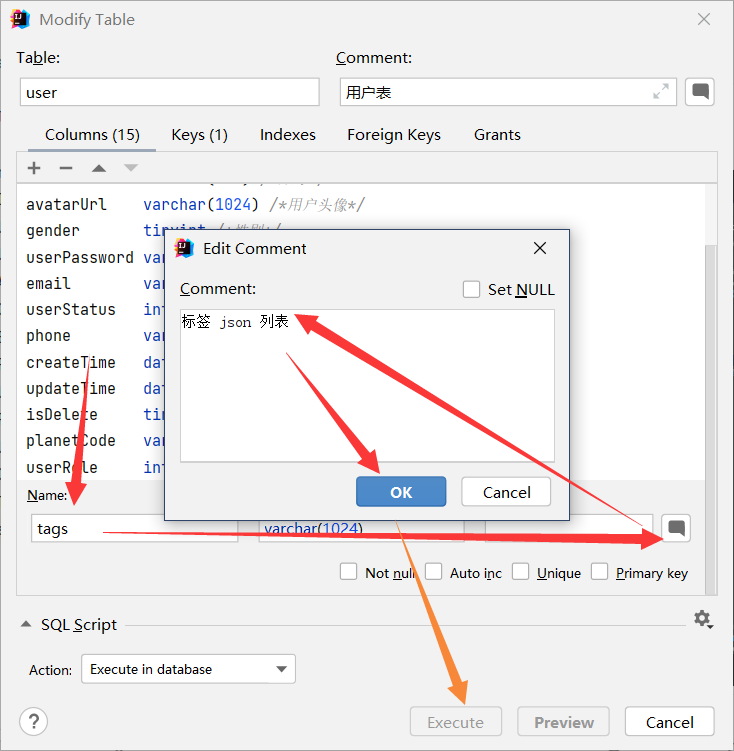
在数据库中添加信息



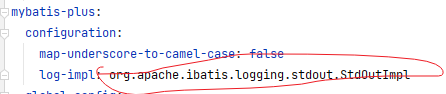
在User添加 tag字段， 在脱敏代码处 添加 tag... , 在database 新增字段处， 注解写详细点。

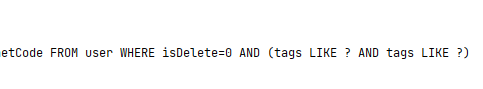






添加sql日志





**根据标签搜索用户**

**SQL 查询**

实现简单，可以通过拆分查询进一步优化

允许用户传入多个标签，多个标签都存在才搜索出来。 and . like '%Java%' and like '%C++%'

允许用户传入多个标签，有任何一个标签存在就能搜索出来 or 。 like '%Java%' or like 'like%C++%'

//SQL 查询

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

//拼接 and 查询

//like '%java%' and like '%Python%'

for(String tagName : tagNameList){

queryWrapper= queryWrapper.like("tags",tagName);

}

List<User> userList = userMapper.selectList(queryWrapper);

return userList.stream().map(this::getSafteyUser).collect(Collectors.toList());

**内存查询**

灵活，可以通过并发进一步优化

//内存查询

//1. 先查询所有用户

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

userMapper.selectCount(null);

List<User> userList=userMapper.selectList(queryWrapper);

Gson gson = new Gson();

//2. 在内存中判断是否包含要求的标签

return userList.stream().filter(user ->{

String tagsStr=user.getTags();

if (StringUtils.isBlank(tagsStr)){

return false;

}

Set<String> tempTagNameSet=gson.fromJson(tagsStr,new TypeToken<Set<String>>(){}.getType());

for (String tagName : tagNameList){

if (!tempTagNameSet.contains(tagName)){

return false;

}

}

return true;

}).map(this::getSafteyUser).collect(Collectors.toList());

优化思路：

如果参数可以分析，根据用户的参数去选择查询方式，比如标签数

如果参数不可以分析，并且数据库连接足够，内存空间足够，可以并发同时查询，谁先返回用谁

还可以SQL 查询 与内存计算相结合，比如先用SQL 过滤掉部分 tag

建议通过实际测试来分析那种查询比较快，数据量大的时候验证效果更明显

**解析JSON 字符串**

序列化 ： Java对象 转换成 json

反序列化： 把json 转换为 Java 对象

Java json 序列化库有很多：

gson (谷歌的）

fastjson alibaba （快，但漏洞多）

jackson

kryo

**直播2**

上次的标签接口调试

前端整合路由

前端开发 （搜索页面 ，用户信息页，用户信息修改页）

**Java 8**

[java8新特性之一：流式数据处理（包含list，map数据处理）。 - 百度文库 (baidu.com)](https://wenku.baidu.com/view/ebdb9bce142ded630b1c59eef8c75fbfc77d9410.html?_wkts_=1678869300852)

Java parallelStream 流失处理 陷阱 [(5条消息) java基础总结(七十)--Java8中的parallelStream的坑\_秋天的猿的博客-CSDN博客](https://blog.csdn.net/lsx2017/article/details/105749984)

Optinal类 （判空）

// if (StringUtils.isBlank(tagsStr)){

// return false;

// }

//判断非空， java 8 新特性

tempTagNameSet= Optional.ofNullable(tempTagNameSet).orElse(new HashSet<>());

作用： 判空

**前端整合路由**

Vue-Router 其实就是帮助你根据不同的url来展示不同的页面 （组件） ，不用自己写 if /else

路由配置影响整个项目，所以建议单独用config 目录，单独的配置文件去集中定义和管理。

有些组件可能自带了和Vue -Router 的整合，所以尽量先看组件文档，省去自己写的时间

到 Vue Router 官方文档 <https://router.vuejs.org/zh/> 安装

我跟着官网做，系统报错，因此，我百度搜索其他方式。

**上次的标签接口调试**

将SQL 查询 与 内存查询 分开两个方法，目前先用 内存查询 （ 灵活 ）

**更改代码**

上次的searchUsersByTags方法里写了两种查询方式，这次就把它们分开，写成两个方法

整理如下：

/\*\*

\* 根据标签搜索用户(内存过滤)

\*

\* @param tagNameList 用户拥有的标签

\* @return

\*/

@Override

public List<User> searchUsersByTags(List<String> tagNameList) {

if (CollectionUtils.isEmpty(tagNameList)) {

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

//1.先查询所有用户

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

List<User> userList = userMapper.selectList(queryWrapper);

Gson gson = new Gson();

//2.在内存中判断是否包含要求的标签

return userList.stream().filter(user -> {

String tagsStr = user.getTags();

Set<String> tempTagNameSet = gson.fromJson(tagsStr, new TypeToken<Set<String>>() {

}.getType());

tempTagNameSet = Optional.ofNullable(tempTagNameSet).orElse(new HashSet<>());

for (String tagName : tagNameList) {

if (!tempTagNameSet.contains(tagName)) {

return false;

}

}

return true;

}).map(this::getSafetyUser).collect(Collectors.toList());

}

/\*\*

\* 根据标签搜索用户(SQL查询)

\* @param tagNameList

\* @return

\*/

@Deprecated

private List<User> searchUsersByTagsBySQL(List<String> tagNameList) {

if (CollectionUtils.isEmpty(tagNameList)) {

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

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

//拼接 and 查询

//like '%Java%' and like '%Python%'

for (String tagName : tagNameList) {

queryWrapper = queryWrapper.like("tags", tagName);

}

List<User> userList = userMapper.selectList(queryWrapper);

return userList.stream().map(this::getSafetyUser).collect(Collectors.toList());

}

**测试**

**前端路由整合**

vue路由组件库地址：

**安装命令**

yarn add vue-router@4

[(5条消息) vue-router的安装使用\_router安装教程\_傻小胖的博客-CSDN博客](https://blog.csdn.net/qq_34645412/article/details/78969279)

**路由引入**

找到前端项目主页面 进行修改

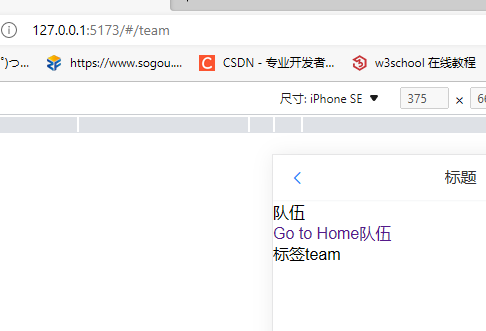


新建一个配置文件，在src目录下创建config 目录， 在此目录下 创建 route.ts文件 导入



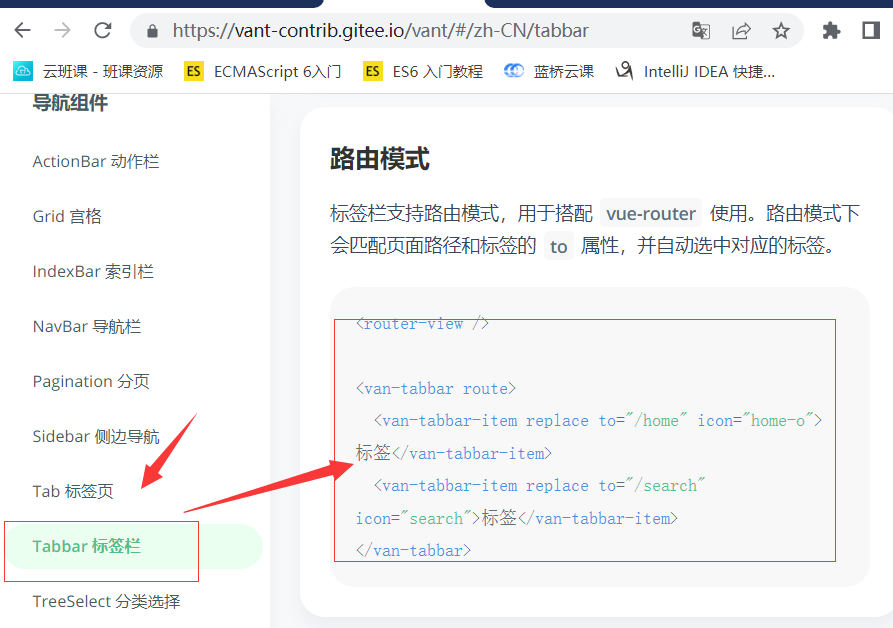
找到布局文件 BasicLayout.vue，删除 v-if 引入 router-view ，这个是根据不同的页面展示不同的内容，再搭配 router-link创建链接





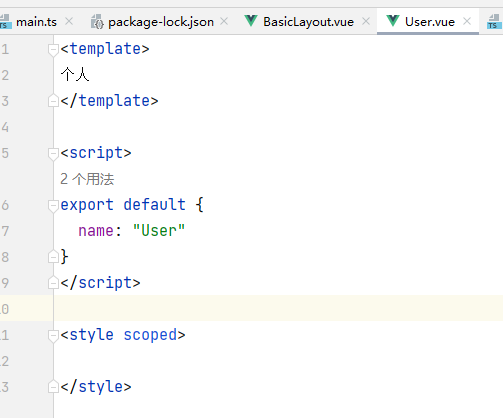
点击队伍显示队伍，发现路径会有改变！

继续修改 BasicLayout.vue 在 官方文档。

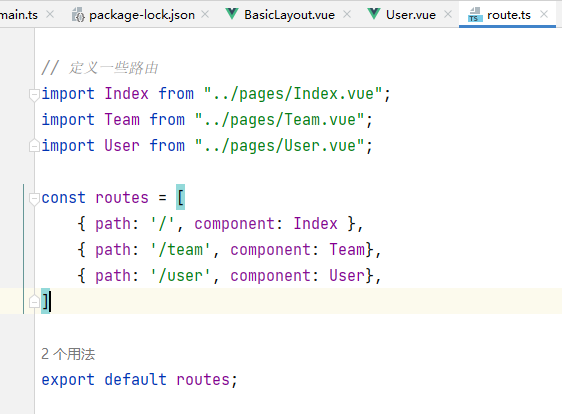




在pages目录下 创建 User.vue



加上个人页的路由

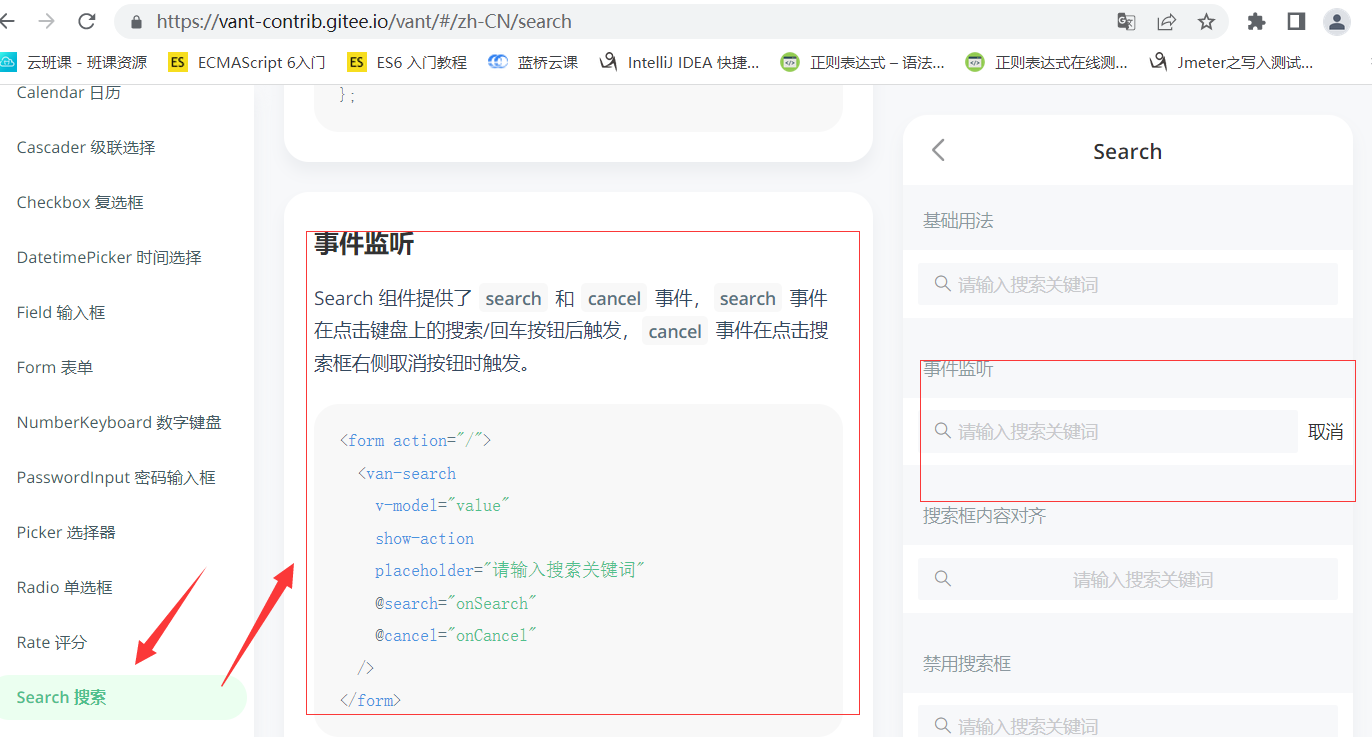


测试成功，把 router-link 删掉

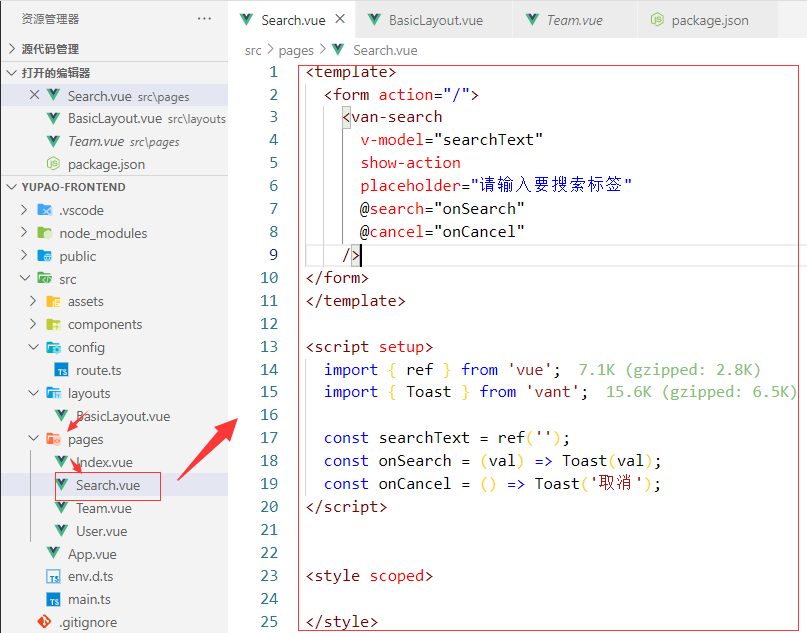
**前端开发**

**搜索页面**

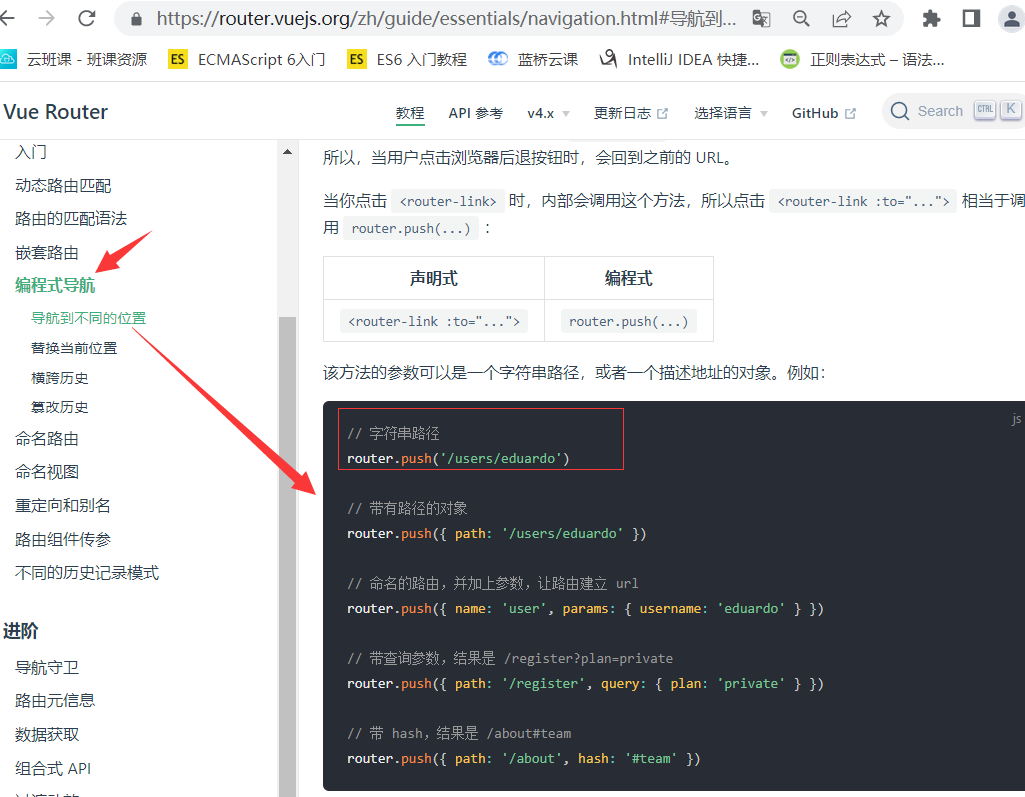
到vant3 官方文档找 现成模板



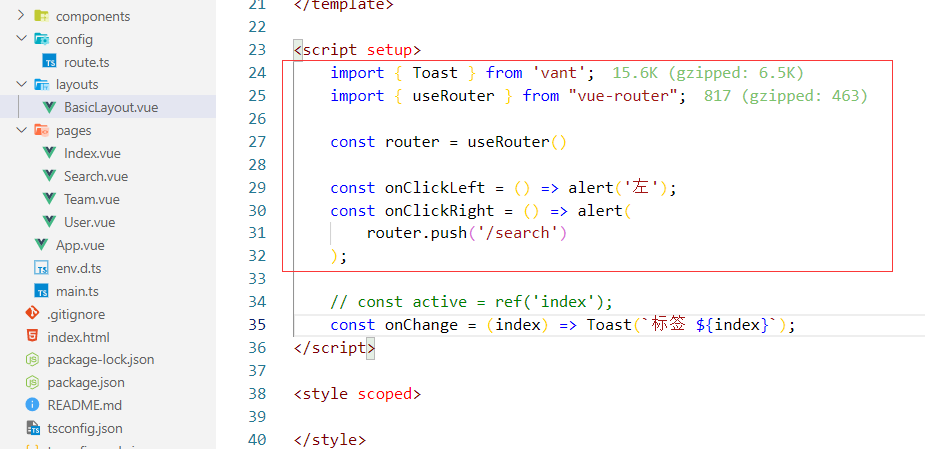
创建一个搜索页 SearchPage.vue 把代码粘过去



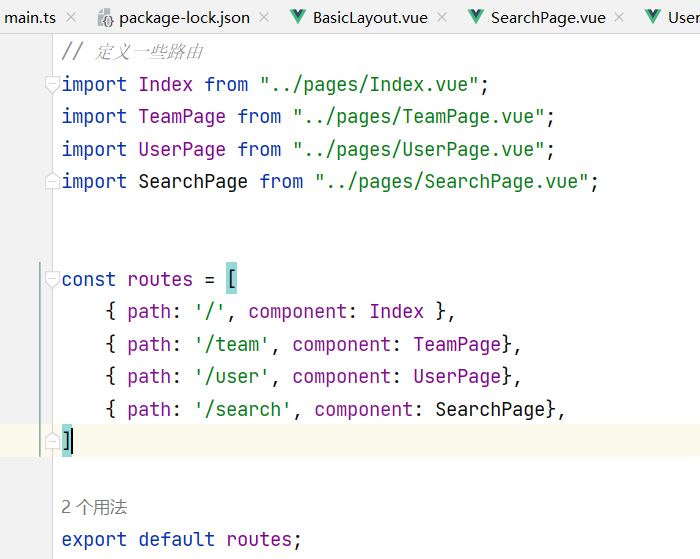
然后做一个点击搜索按钮，跳转搜索页面。 查看 Vue Router的官方文档，找一下导航到不同位置的代码。



粘到 BasicLayout.vue ，修改一下



挂载一下 搜索路由



测试

点击搜索后，还想让它展示当前已选标签，去 vant3 找一下tag标签



还要写一个列表给它分组，类似哔哩哔哩。 再去vant3 找类似的



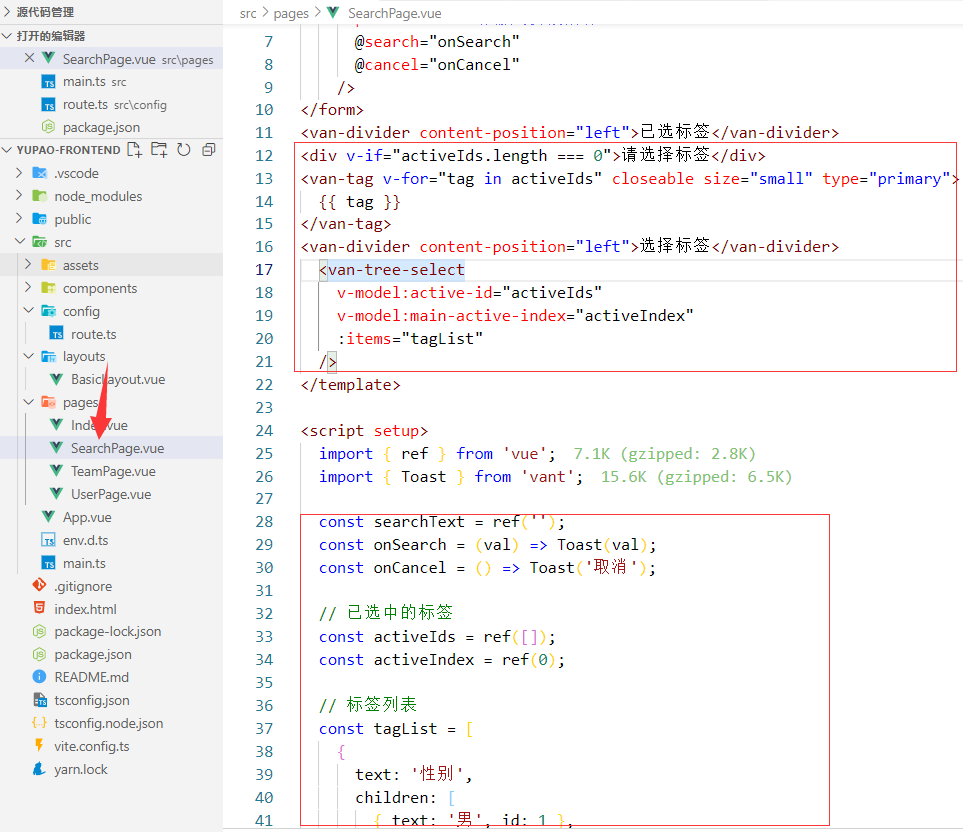
修改 “导入所有组件”



“选择标签” 用 TreeSelect 分类选择

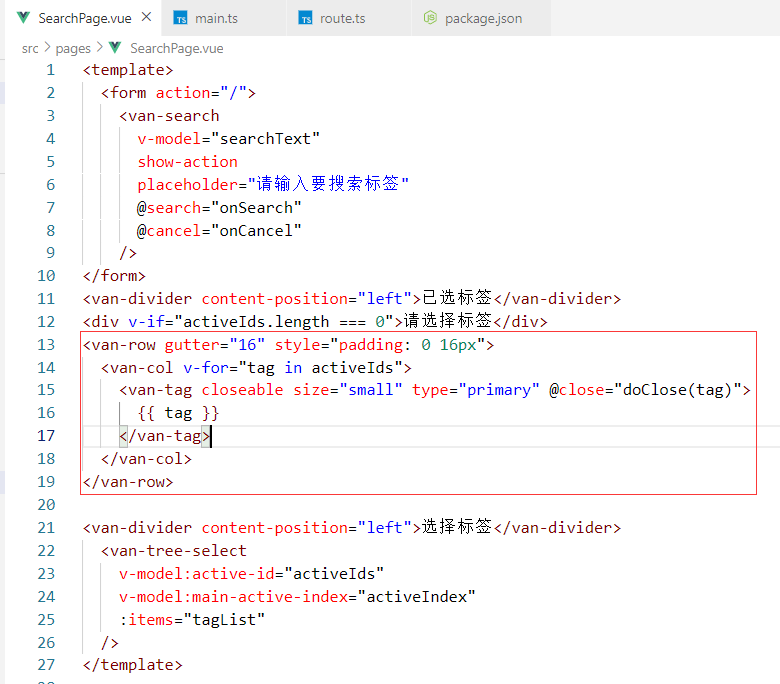


粘贴到 SearchPage,修改一下



已选标签挤在一起，改一下，选择Layout布局来设计 间距





还差一个过滤标签

可以直接在前端过滤，因为标签的数据量并不大，没必要向后台发请求

<template>

<form action="/">

<van-search

v-model="searchText"

show-action

placeholder="请输入搜索关键词"

@search="onSearch"

@cancel="onCancel"

/>

</form>

<van-divider>已选标签</van-divider>

<div v-if="activeIds.length === 0">请选择标签</div>

<van-row gutter="16" style="padding: 0 16px">

<van-col v-for="tag in activeIds">

<van-tag closeable size="medium" type="primary" @close="close(tag)">

{{ tag }}

</van-tag>

</van-col>

</van-row>

<van-divider>全部标签</van-divider>

<van-tree-select

v-model:active-id="activeIds"

v-model:main-active-index="activeIndex"

:items="tagList"

/>

</template>

<script setup>

import {ref} from 'vue';

const searchText = ref('');

const originTagList = [

{

text: '性别',

children: [

{text: '男', id: '男'},

{text: '女', id: '女'},

],

},

{

text: '年级',

children: [

{text: '大一', id: '大一'},

{text: '大二', id: '大二'},

{text: '大3', id: '大3'},

{text: '大4', id: '大4'},

{text: '大5', id: '大5aaaaaa'},

{text: '大6', id: '大6aaaaaa'},

],

},

];

let tagList = ref(originTagList)

const onSearch = (val) => {

tagList.value = originTagList.map(parentTag => {

const tempChildrenTag = [...parentTag.children];

const tempParentTag = {...parentTag};

tempParentTag.children = tempChildrenTag.filter(item => item.text.includes(searchText.value)

)

return tempParentTag;

})

};

const onCancel = () => {

searchText.value = '';

tagList.value = originTagList

};

const activeIds = ref([]);

const activeIndex = ref(0);

const close = (tag) => {

activeIds.value = activeIds.value.filter(id => id !== tag)

}

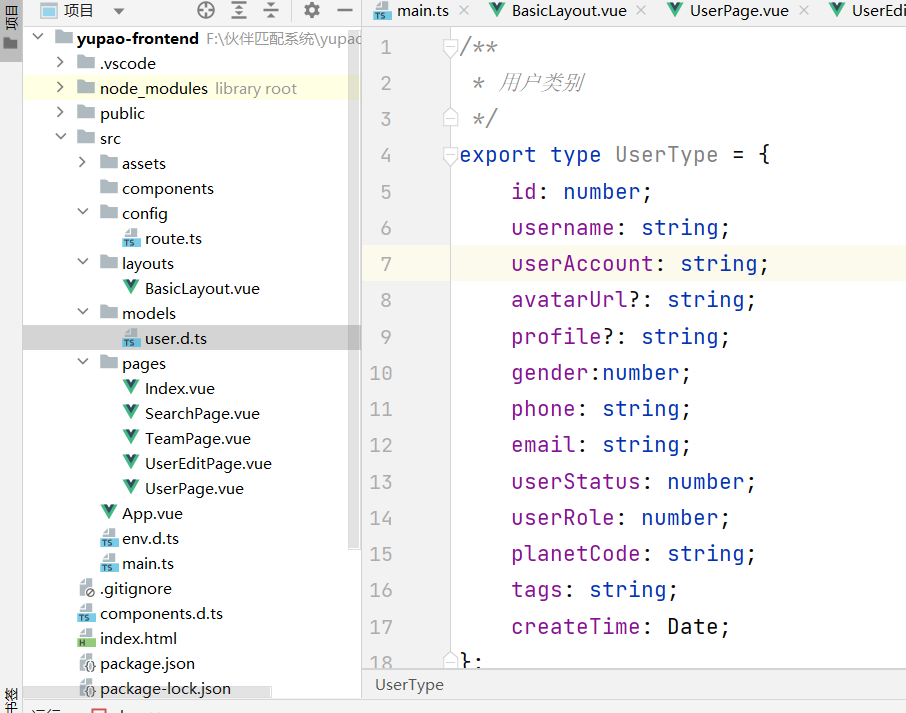
</script>

<style scoped>

</style>

**用户信息页**

定义用户数据的类别



UserPage

<template>

<van-cell title="头像" is-link>

<img :src="user.avatar" alt="" style="height: 46px">

</van-cell>

<van-cell title="昵称" is-link :value="user.userName" @click="toEdit('userName', user.userName, '昵称')"/>

<van-cell title="账号" :value="user.userAccount"/>

<van-cell title="性别" is-link :value="user.gender" @click="toEdit('gender', user.gender, '性别')"/>

<van-cell title="电话" is-link :value="user.phone" @click="toEdit('phone', user.phone, '电话')"/>

<van-cell title="邮箱" is-link :value="user.email" @click="toEdit('email', user.email, '邮箱')"/>

<van-cell title="星球编号" :value="user.planetCode"/>

<van-cell title="注册时间" :value="user.gmtCreate"/>

</template>

<script setup lang="ts">

import {ref} from "vue";

import {useRouter} from "vue-router";

const user = ref({

id: 1,

userName: 'tt110617',

userAccount: 'hzz',

gender: '男',

phone: '12345',

email: '1253728@qq.com',

planetCode: '123',

avatar: 'https://img2.baidu.com/it/u=1790834130,1952230725&fm=253&fmt=auto&app=138&f=JPEG?w=500&h=500',

gmtCreate: new Date().toDateString()

})

const router = useRouter();

const toEdit = (editKey: string, currentValue: string, editName: string) => {

router.push({

path: '/user/edit',

query: {

editKey,

currentValue,

editName

}

})

}

</script>

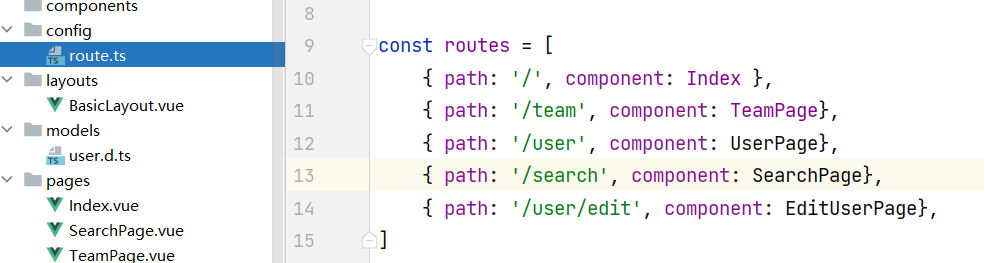
<style scoped>

</style>

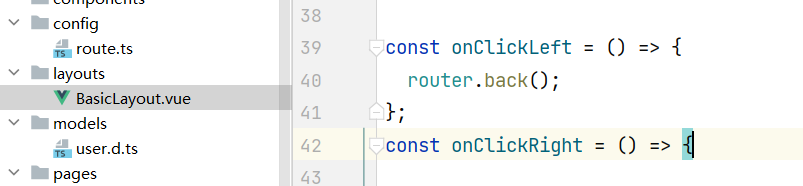
**用户信息修改页**

在Pages 目录下，创建UserEditPage

新增路由



提交后返回直接跳回主页面了，这里应该返回上一个页面



UserEditPage

<template>

<van-form @submit="onSubmit">

<van-cell-group inset>

<van-field

v-model="editUser.currentValue"

:name="editUser.editKey"

:label="editUser.editName"

:placeholder="`请输入${editUser.editName}`"

/>

</van-cell-group>

<div style="margin: 16px;">

<van-button round block type="primary" native-type="submit">

提交

</van-button>

</div>

</van-form>

</template>

<script setup>

import {useRoute} from "vue-router";

import {ref} from "vue";

const route = useRoute()

const editUser = ref({

editKey: route.query.editKey,

currentValue: route.query.currentValue,

editName: route.query.editName

})

const onSubmit = (values) => {

// todo 将 editKey,currentValue,editName 提交给后台

console.log(values)

};

</script>

<style scoped>

</style>

**直播3**

Java 后端整合 Swagger + Knife4j 接口文档

存量用户信息导入及同步 （爬虫)

**后端整合 Swagger + Knife4j 接口文档**

**什么是接口文档？**

写接口信息的文档，每条接口包括：

请求参数

响应参数

错误码

接口地址

接口名称

请求类型

请求格式

备注

谁用？ 一般是后端或者负责人来提供，后端和前端都要使用

**为什么需要接口文档 ？**

有个书面内容（背书或归档），便于大家参考和查阅，便于沉淀和维护，拒绝口口相传

接口文档便于前端和后端对接，前后端联调的 **介质**

好的接口文档支持在线调试，在线测试，可以作为工具提高我们的开发测试效率

**怎么做接口文档 ？**

手写 （ 腾讯文档，MarkDown ）

自动生成 ： 自动根据项目代码生成完整的文档或在线调试网页： Swagger , Postman （侧重接口管理） （国外）

apifox , apipost , enlink （国产）

**接口文档有哪些技巧？**

Swagger 原理：

引入依赖（Swagger 或 Knife4j：<https://doc.xiaominfo.com/knife4j/documentation/get_start.html>）

自定义 Swagger 配置类

定义需要生成接口文档的代码位置 （Controller ） ，

注意： 线上环境不要把接口暴露出去，可以通过在 SwaggerConfig 配置文件开头加上 @Profile({"dev", "test"}) 限定配置仅在部分环境开启

启动即可

可以通过在 controller 方法上添加 @Api、@ApiImplicitParam(name = "name",value = "姓名",required = true)    @ApiOperation(value = "向客人问好") 等注解来自定义生成的接口描述信息

**步骤**

在maven 引入 Knife4j 依赖

<dependency>

<groupId>com.github.xiaoymin</groupId>

<artifactId>knife4j-spring-boot-starter</artifactId>

<version>2.0.7</version>

</dependency>

创建Swagger配置依赖

@Configuration

@EnableSwagger2WebMvc

public class Knife4jConfiguration {

@Bean(value = "defaultApi2")

public Docket defaultApi2() {

Docket docket=new Docket(DocumentationType.SWAGGER\_2)

.apiInfo(new ApiInfoBuilder()

//.title("swagger-bootstrap-ui-demo RESTful APIs")

.description("# swagger-bootstrap-ui-demo RESTful APIs")

.termsOfServiceUrl("http://www.xx.com/")

.contact("xx@qq.com")

.version("1.0")

.build())

//分组名称

.groupName("2.X版本")

.select()

//这里指定Controller扫描包路径

.apis(RequestHandlerSelectors.basePackage("com.luo.yupao.controller"))

.paths(PathSelectors.any())

.build();

return docket;

}

}

如果 SpringBoot version >= 2.6 需要在yml添加以下配置：

spring:

mvc:

pathmatch:

matching-strategy: ant\_path\_matcher

访问

http://localhost:8080/api/doc.html

**存量用户信息导入及同步 （爬虫)**

把所有星球用户的信息导入

把写了自我介绍的同学的标签信息导入

**看上了网页信息，怎么抓？**

分析原网站是怎么获取这些数据的？ 哪个 接口？

curl "https://api.zsxq.com/v2/hashtags/48844541281228/topics?count=20" ^

-H "authority: api.zsxq.com" ^

-H "accept: application/json, text/plain, \*/\*" ^

-H "accept-language: zh-CN,zh;q=0.9,en;q=0.8,en-GB;q=0.7,en-US;q=0.6" ^

-H "origin: https://wx.zsxq.com" ^

-H "referer: https://wx.zsxq.com/" ^

-H "sec-ch-ua: ^\^"Not?A\_Brand^\^";v=^\^"8^\^", ^\^"Chromium^\^";v=^\^"108^\^", ^\^"Microsoft Edge^\^";v=^\^"108^\^"" ^

-H "sec-ch-ua-mobile: ?0" ^

-H "sec-ch-ua-platform: ^\^"Windows^\^"" ^

-H "sec-fetch-dest: empty" ^

-H "sec-fetch-mode: cors" ^

-H "sec-fetch-site: same-site" ^

-H "user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/108.0.0.0 Safari/537.36 Edg/108.0.1462.54" ^

--compressed

用程序去调用接口

处理（清洗）一下数据，之后就可以写到数据库里

**流程**

从excel 中导入全量用户数据，判重。 easyexcel ：<https://easyexcel.opensource.alibaba.com/>

抓取写了自我介绍的同学信息，提取出用户昵称，用户唯一id，自我介绍信息

从自我介绍中提取信息，写入数据库

**EasyExcel**

两种读方式：

确定表头：建立对象

不确定表头： 每一行数据映射为 Map<String,Object>

两种读取模式：

监听器：先创建监听器，在读文件时绑定监听器。单独抽离处理逻辑，代码清晰易于维护；一条一条处理，适用于数据量大的场景

同步读：无需创建监听器，一次性获取完整数据。方便简单，但是数据量大的时候会有等待时长，也可能内存溢出。

**步骤**

创建读对象

package com.yupi.usercenter.once;

import com.alibaba.excel.annotation.ExcelProperty;

import lombok.Data;

import lombok.EqualsAndHashCode;

@Data

@EqualsAndHashCode

public class XingQiuTableUserInfo {

/\*\*

\* 星球编号

\*/

@ExcelProperty("成员编号")

private String planetCode;

/\*\*

\* 用户昵称

\*/

@ExcelProperty("昵称")

private String userName;

}

创建监听器

package com.yupi.usercenter.once;

import com.alibaba.excel.context.AnalysisContext;

import com.alibaba.excel.read.listener.ReadListener;

import lombok.extern.slf4j.Slf4j;

// 有个很重要的点 XingQiuTableUserInfoListener 不能被spring管理，要每次读取excel都要new,然后里面用到spring可以构造方法传进去

@Slf4j

public class XingQiuTableUserInfoListener implements ReadListener<XingQiuTableUserInfo> {

/\*\*

\* 这个每一条数据解析都会来调用

\*

\* @param data one row value. Is is same as {@link AnalysisContext#readRowHolder()}

\* @param context

\*/

@Override

public void invoke(XingQiuTableUserInfo data, AnalysisContext context) {

log.info("" + data);

}

/\*\*

\* 所有数据解析完成了 都会来调用

\*

\* @param context

\*/

@Override

public void doAfterAllAnalysed(AnalysisContext context) {

// 这里也要保存数据，确保最后遗留的数据也存储到数据库

log.info("所有数据解析完成！");

}

}

使用

package com.yupi.usercenter.once;

import com.alibaba.excel.EasyExcel;

import java.util.List;

/\*\*

\* 导入 Excel

\* @author hzz

\* @create 2022-06-13 13:04

\*/

public class ImportExcel {

public static void main(String[] args) {

String fileName = "D:\\workspace\_idea\\user\_center\\src\\main\\resources\\星球数据.xlsx";

// listenerRead(fileName);

synchronousRead(fileName);

}

public static void listenerRead(String fileName) {

// 这里 需要指定读用哪个class去读，然后读取第一个sheet 文件流会自动关闭

// 这里每次会读取100条数据 然后返回过来 直接调用使用数据就行

EasyExcel.read(fileName, XingQiuTableUserInfo.class, new XingQiuTableUserInfoListener()).sheet().doRead();

}

public static void synchronousRead(String fileName) {

// 这里 需要指定读用哪个class去读，然后读取第一个sheet 同步读取会自动finish

List<XingQiuTableUserInfo> list = EasyExcel.read(fileName).head(XingQiuTableUserInfo.class).sheet().doReadSync();

for (XingQiuTableUserInfo data : list) {

System.out.println(data);

}

}

}

**直播4**

页面和功能开发 （ 搜索，信息，修改页面）

改造用户中心，单机 -> session

标签的整理，细节的优化

**笔记**

**前端页面跳转传值**

query => url searchParams，url 后附加参数，传递的值长度有限

vuex（全局状态管理），搜索页将关键词塞到状态中，搜索结果页从状态取值

**Session 共享**

种 session 的时候注意范围，cookie.domain比如两个域名：aaa.yupi.combbb.yupi.com如果要共享 cookie，可以种一个更高层的公共域名，比如 yupi.com

**为什么服务器 A 登录后，请求发到服务器 B，不认识该用户？**

用户在 A 登录，所以 session（用户登录信息）存在了 A 上结果请求 B 时，B 没有用户信息，所以不认识。

解决方案：**共享存储** ，而不是把数据放到单台服务器的内存中

如何共享存储？

Redis（基于内存的 K / V 数据库）此处选择 Redis，因为用户信息读取 / 是否登录的判断极其**频繁** ，Redis 基于内存，读写性能很高，简单的数据单机 qps 5w - 10w

MySQL

文件服务器 ceph

Session 共享实现

**安装 Redis**

官网：<https://redis.io/windows>

windows 下载：

Redis 5.0.14 [下载：链接：https://pan.baidu.com/s/1XcsAIrdeesQAyQU](https://pan.baidu.com/s/1XcsAIrdeesQAyQU2lE3cOg)2lE3cOg

提取码：vkoi redis 管理工具

quick [redis：https://quick1](https://quick123.net/)

引入 redis，能够操作 redis：

<!-- <https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-data-redis> -->

<depe[ndency>](https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-data-redis)

   <[groupId>org.springframework.boot</groupId>](https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-data-redis)

   <[artifactId>spring-boot-starter-data-redis</artifactId>](https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-data-redis)

   <[version>2.6.4</version>](https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-data-redis)

</dep[endency>](https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-data-redis)

引入 spring-session 和 redis 的整合，使得自动将 session 存储到 redis 中：

<!-- <https://mvnrepository.com/artifact/org.springframework.session/spring-session-data-redis> -->

<depe[ndency>](https://mvnrepository.com/artifact/org.springframework.session/spring-session-data-redis)

   <[groupId>org.springframework.session</groupId>](https://mvnrepository.com/artifact/org.springframework.session/spring-session-data-redis)

   <[artifactId>spring-session-data-redis</artifactId>](https://mvnrepository.com/artifact/org.springframework.session/spring-session-data-redis)

   <[version>2.6.3</version>](https://mvnrepository.com/artifact/org.springframework.session/spring-session-data-redis)

</dep[endency>](https://mvnrepository.com/artifact/org.springframework.session/spring-session-data-redis)

修改 spring-session 存储配置 spring.session.store-type默认是 none，表示存储在单台服务器store-type: redis，表示从 redis 读写 session

JWT 的优缺点：<https://zhuanlan.zhihu.com/p/108999941>

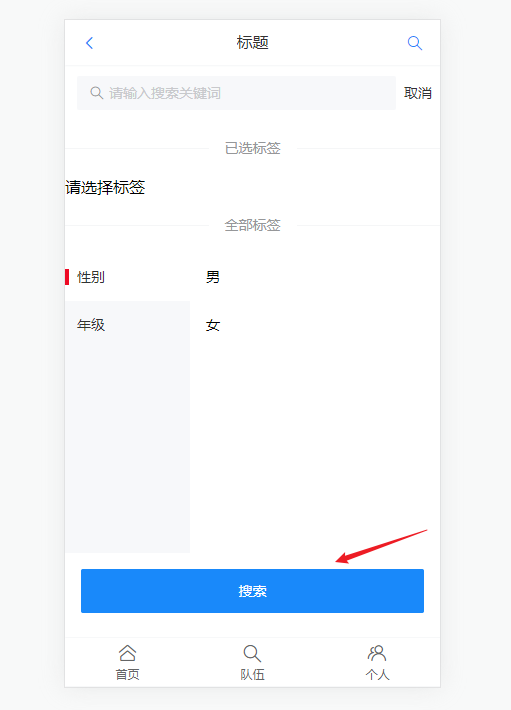
**todo 待优化**

前端：动态展示页面标题、微调格式

**页面和功能开发**

**搜索结果页面**

添加搜索按钮



在SearchPage 添加代码：

<div style="padding: 16px">

<van-button block type="primary" @click="doSearchResult">搜索</van-button>

</div>

const doSearchResult = () => {

router.push({

path: '/user/list',

query: {

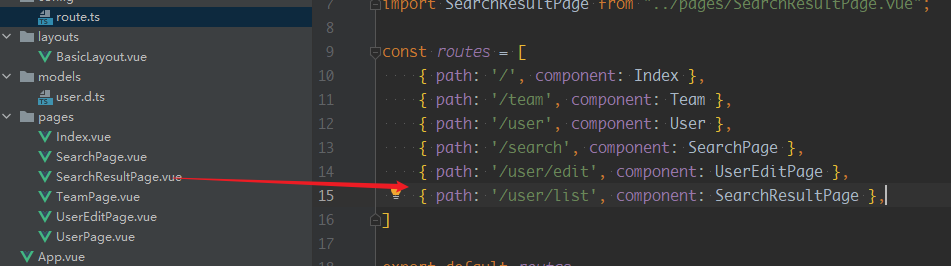
tags: activeIds.value

}

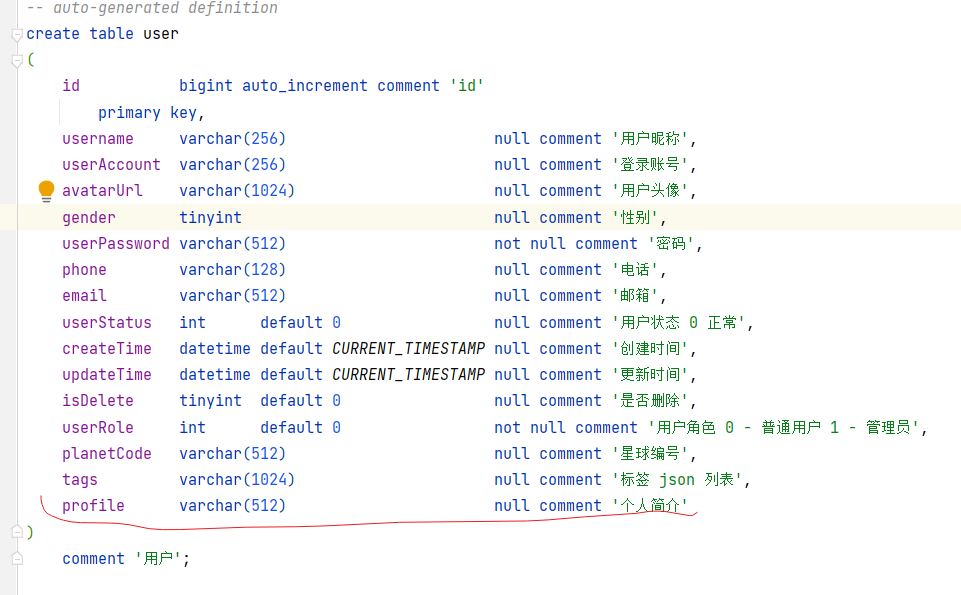
})

}

添加路由和新页面



用户表添加个人简介字段



在SearchResultPage 引入组件 展示数据

axios 版本问题 ： [解决AxiosError: options must be an object ERR\_BAD\_OPTION\_VALUE - yzeng - 博客园 (cnblogs.com)](https://www.cnblogs.com/yzeng/p/16837606.html)

<template>

<van-card

v-for="user in userList"

:desc="user.profile"

:title="`${user.userName} (${user.planetCode})`"

:thumb="user.avatarUrl"

>

<template #tags>

<van-tag plain type="danger" v-for="tag in user.tags" style="margin-right: 8px; margin-top: 8px" >

{{tag}}

</van-tag>

</template>

<template #footer>

<van-button size="mini">联系我</van-button>

</template>

</van-card>

<van-empty v-if="!userList || userList.length < 1" description="搜索结果为空" />

</template>

<script setup>

import {onMounted, ref} from 'vue';

import {useRoute} from "vue-router";

import myAxios from "../plugins/myAxios.js";

import {Toast} from "vant";

import qs from 'qs';

const route=useRoute();

const {tags} =route.query;

const userList = ref([]);

onMounted(async () => {

const userListData= await myAxios.get('/user/search/tags', {

params: {

tagNameList: tags

},

// axios版本升级导致，0.x升级为1.x，在请求参数数组序列化时出现

// paramsSerializer: params => {

// return qs.stringify(params,{indices: false})

// }

paramsSerializer: {

serialize: function (params) {

return qs.stringify(params, {indices: false})

}

}

})

.then(function (response){

console.log('/user/search/tags success',response);

Toast.success('请求成功');

return response.data?.data;

})

.catch(function (error){

console.error('/user/search/tags error',error);

Toast.fail('请求失败');

})

console.log(userListData);

if (userListData){

userListData.forEach(user => {

if (user.tags) {

user.tags = JSON.parse(user.tags);

}

})

userList.value = userListData;

}

})

</script>

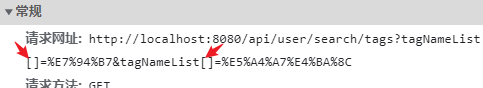
<style scoped>

</style>

出现跨域问题，后端解决：在Controller 添加注解

@CrossOrigin（orgins={xxx}

参数格式不对



引入 qs

**qs 是一个增加了一些安全性的查询字符串解析和序列化字符串的库。**

// axios版本升级导致，0.x升级为1.x，在请求参数数组序列化时出现

// paramsSerializer: params => {

// return qs.stringify(params,{indices: false})

// }

paramsSerializer: {

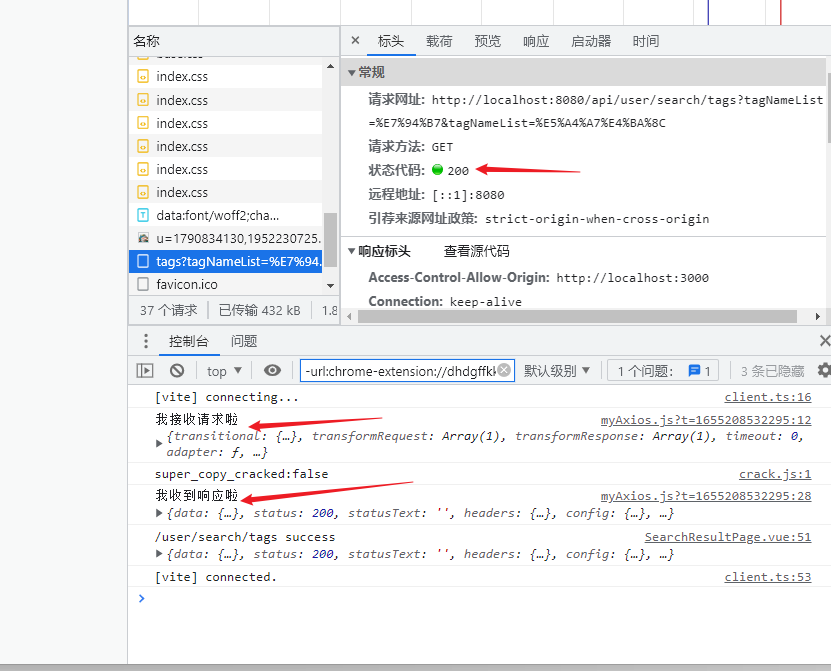
serialize: function (params) {

return qs.stringify(params, {indices: false})

}

}

请求成功



在数据库造假数据

前端展示数据

//存放用户列表

const userList = ref([]);

//等待异步方法执行完成

onMounted(async () => {

const userListData= await myAxios.get('/user/search/tags', {

params: {

tagNameList: tags

},

// axios版本升级导致，0.x升级为1.x，在请求参数数组序列化时出现

// paramsSerializer: params => {

// return qs.stringify(params,{indices: false})

// }

paramsSerializer: {

serialize: function (params) {

return qs.stringify(params, {indices: false})

}

}

})

.then(function (response){

console.log('/user/search/tags success',response);

Toast.success('请求成功');

return response.data?.data; //返回数据， 可选链操作符，避免数据为 null 或 undefined 时 报错

})

.catch(function (error){

console.error('/user/search/tags error',error);

Toast.fail('请求失败');

})

console.log(userListData);

//数据序列化

if (userListData){

userListData.forEach(user => {

if (user.tags) {

user.tags = JSON.parse(user.tags);

}

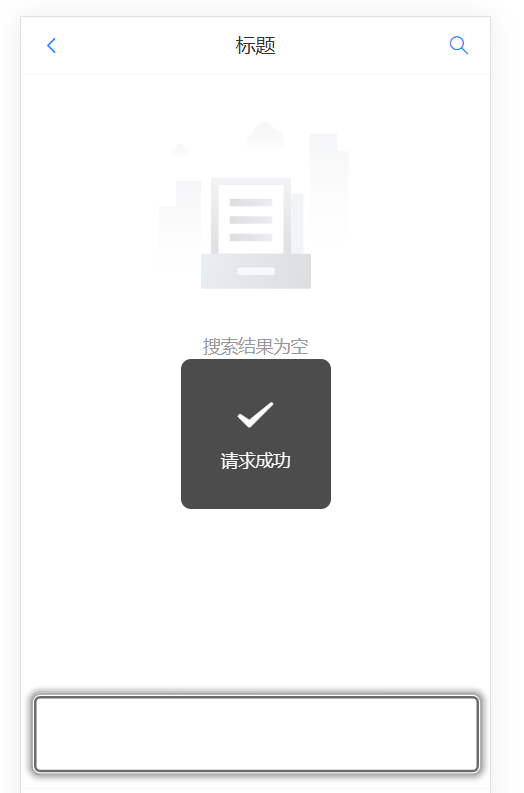
})

userList.value = userListData;

}

})





**改造用户中心，把单机登录改为分布式 session 登录**

session 共享

种 session 的时候注意范围，cookie.domiain

比如两个域名 :

aaa.yupi.com

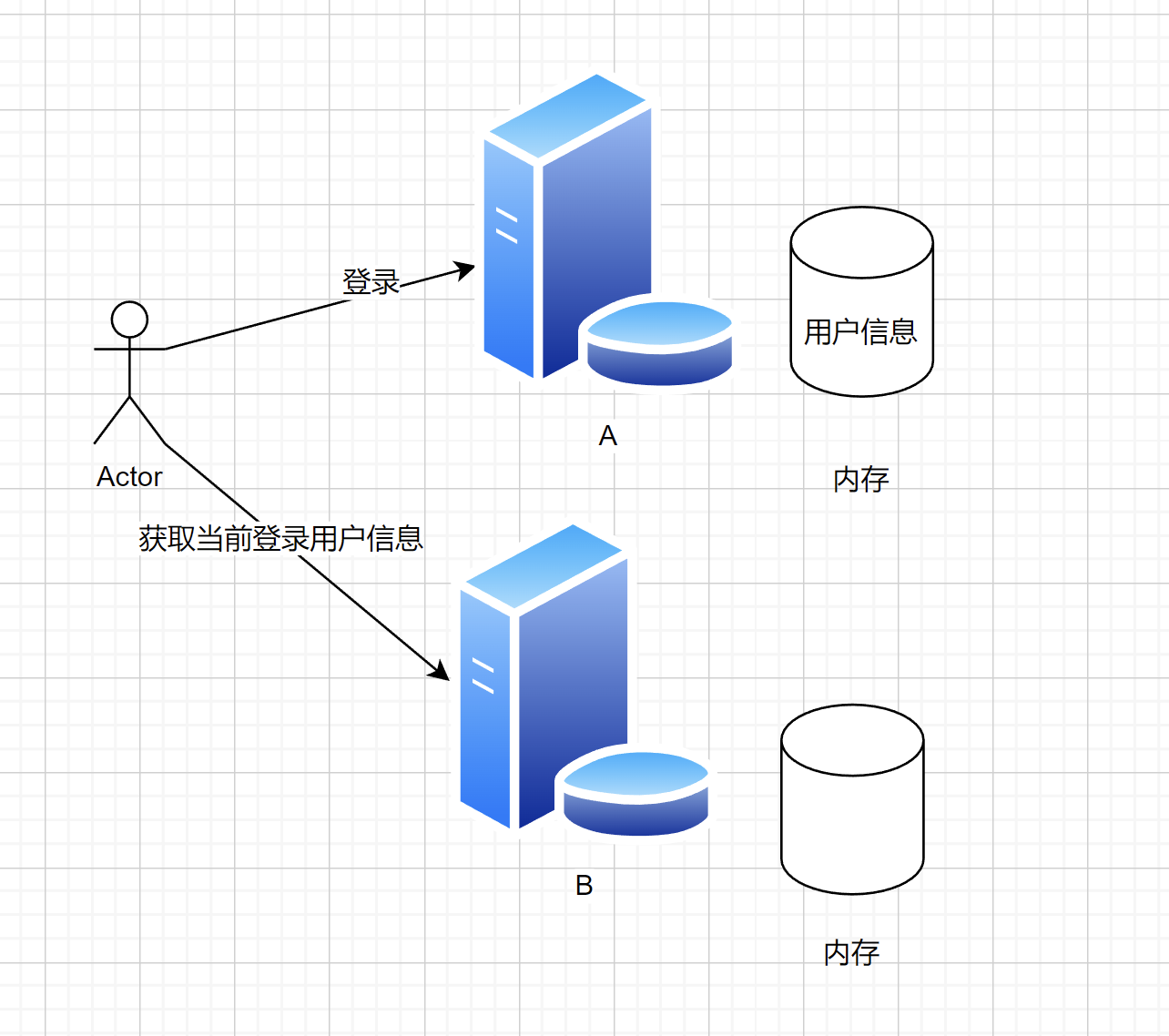
bbb.yupi.com

如果要共享cookie 可以种一个更高层的公共域名，比如 yupi.com

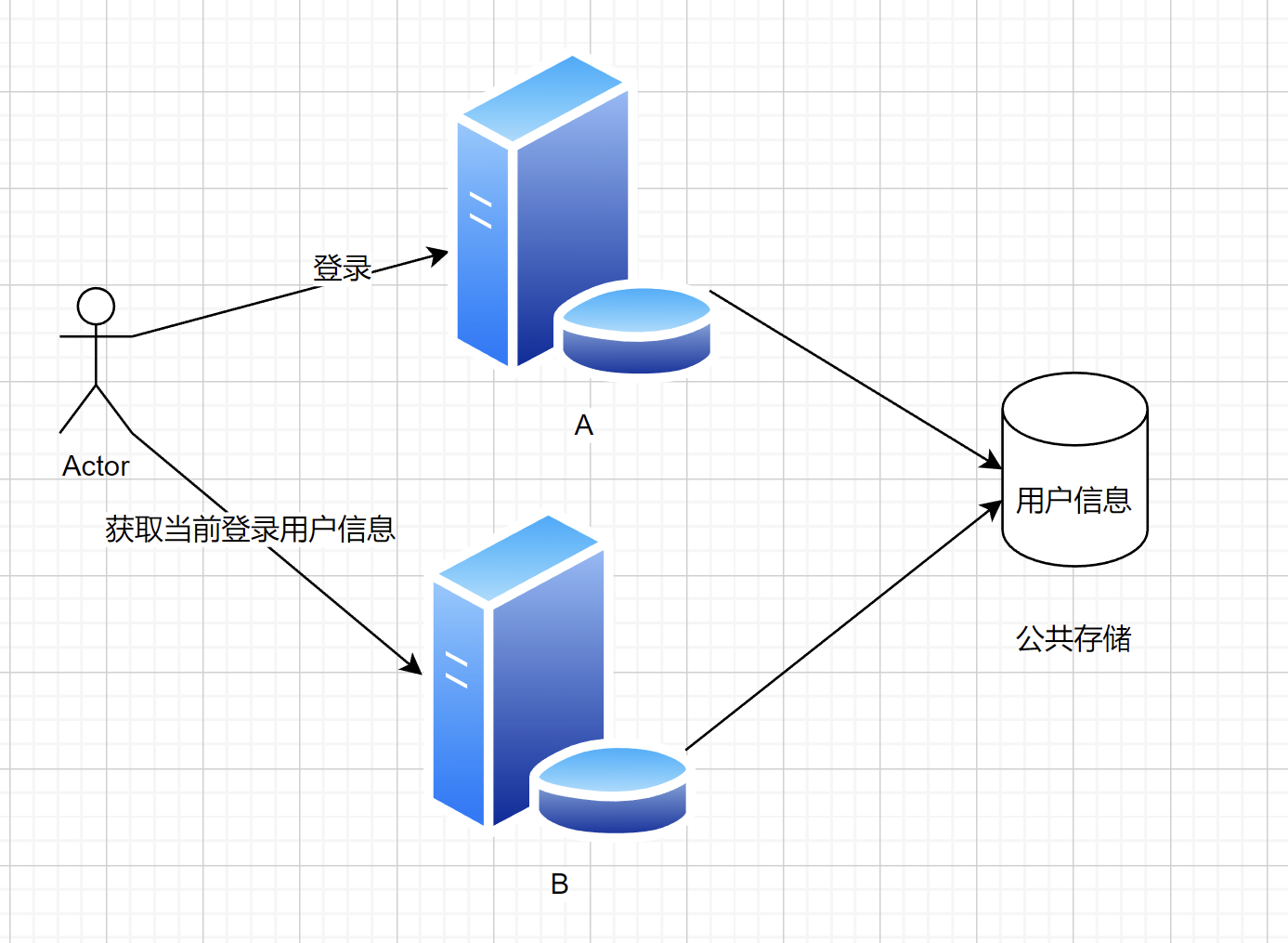
为什么服务器A 登录后，请求发到 服务器B ， 不认识该用户？

用户 在 A 登录， 所以 session （用户登录信息） 存在了 A 上

结果请求B 时，B 没有用户信息， 所以不认识



解决方案：  **共享存储** 而不是把数据放到单台服务器的内存中



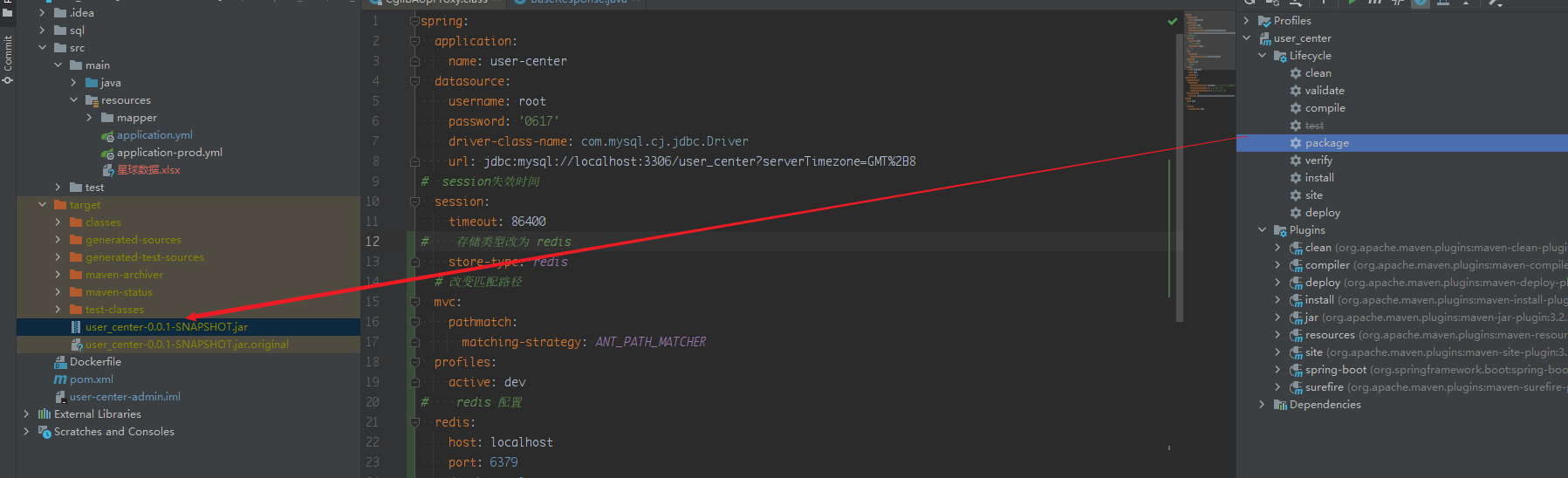
如何共享存储？

Redis （基于内存 的 K / V 数据库） 此处选择 Redis ,因为用户信息 读取 / 是否登录的判断极其频繁， Redis 基于内存，读写性能高，简单的数据单机 qps 5w - 10w

MySQL

文件服务器 ceph

**本地模拟多台服务器**



输入命令行 8081 端口启动

java -jar .\user\_center-0.0.1-SNAPSHOT.jar --server.port=8081

**步骤**

安装 Redis

Redis 5.0.14 下载：

链接：<https://pan.baidu.com/s/1XcsAIrdeesQAyQU2lE3cOg>

提取码：vkoi

redis管理工具， quick redis <https://quick123.net/>

引入启动依赖

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-redis</artifactId>

<version>2.7.6</version>

</dependency>

引入spring-session 和 redis 的 整合，使得自动将session 存储到redis中

<dependency>

<groupId>org.springframework.session</groupId>

<artifactId>spring-session-data-redis</artifactId>

<version>2.7.0</version>

</dependency>

修改 spring-session 存储配置 spring .session .store -type

默认是 none 表示存储在单台服务器

store-type: redis，表示从 redis 读写 session

spring:

session:

store-type: redis

效果



**直播5**

用户修改页面前端，后端开发和联调

**修改用户信息**

**后端接口**

UserController

@PostMapping("/update")

public BaseResponse<Integer> updateUser(@RequestBody User user, HttpServletRequest request) {

// 校验参数是否为空

if (user == null) {

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

User loginUser = userService.getLoginUser(request);

int result = userService.updateUser(user, loginUser);

return ResultUtils.success(result);

}

UserService（由于项目中多次用户鉴权，所以我们在service层编写isAdmin方法（包括复写））

/\*\*

\* 更新用户信息

\* @param user

\* @return

\*/

int updateUser(User user,User loginUser);

/\*\*

\* 获取当前登录用户信息

\* @param request

\* @return

\*/

User getLoginUser(HttpServletRequest request);

/\*\*

\* 是否为管理员

\* @param request

\* @return

\*/

boolean isAdmin(HttpServletRequest request);

/\*\*

\* 是否为管理员

\* @param loginUser

\* @return

\*/

boolean isAdmin(User loginUser);

UserServiceImpl

@Override

public int updateUser(User user, User loginUser) {

long userId = user.getId();

if (userId <= 0) {

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

if (user==loginUser){

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

// todo 补充校验，如果用户没有传任何要更新的值，就直接报错，不用执行 update 语句

// 如果是管理员，允许更新任意用户

// 如果不是管理员，只允许更新当前（自己的）信息

if (!isAdmin(loginUser) && userId != loginUser.getId()) {

throw new BusinessException(ErrorCode.NO\_AUTH);

}

User oldUser = userMapper.selectById(userId);

if (oldUser == null) {

throw new BusinessException(ErrorCode.NULL\_ERROR);

}

return userMapper.updateById(user);

}

/\*\*

\* 获取当前登录用户信息

\* @param request

\* @return

\*/

@Override

public User getLoginUser(HttpServletRequest request) {

if (request==null){

return null;

}

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

if (user==null){

throw new BusinessException(ErrorCode.NO\_AUTH);

}

return (User) user;

}

/\*\*

\* 是否为管理员

\* @param request

\* @return

\*/

@Override

public boolean isAdmin(HttpServletRequest request) {

//仅管理员可查询

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

User user = (User) userObj;

return user != null && user.getUserRole() == ADMIN\_ROLE;

}

/\*\*

\* 是否为管理员

\* @param loginUser

\* @return

\*/

@Override

public boolean isAdmin(User loginUser) {

return loginUser != null && loginUser.getUserRole() == ADMIN\_ROLE;

}

**前端页面**

**添加登录页面**

在route.ts 添加路由

import UserLoginPage from "../pages/UserLoginPage.vue";

{ path: '/user/login', component: UserLoginPage },

UserLoginPage

<template>

<van-form @submit="onSubmit">

<van-cell-group inset>

<van-field

v-model="userAccount"

name="userAccount"

label="账号"

placeholder="请输入账号"

:rules="[{ required: true, message: '请填写账号' }]"

/>

<van-field

v-model="userPassword"

type="password"

name="userPassword"

label="密码"

placeholder="请输入密码密码"

:rules="[{ required: true, message: '请填写密码' }]"

/>

</van-cell-group>

<div style="margin: 16px;">

<van-button round block type="primary" native-type="submit">

提交

</van-button>

</div>

</van-form>

</template>

<script setup lang="ts">

import myAxios from "../plugins/myAxios";

import {Toast} from "vant";

import {ref} from "vue";

import {useRouter} from "vue-router";

const router = useRouter();

const userAccount = ref();

const userPassword = ref();

const onSubmit = async () => {

console.log("用户登录");

const res = await myAxios.post("/user/login", {

userAccount: userAccount.value,

userPassword: userPassword.value

});

if (res.code === 0 && res.data != null) {

Toast.success("登录成功");

router.replace('/');

} else {

Toast.fail("登录失败");

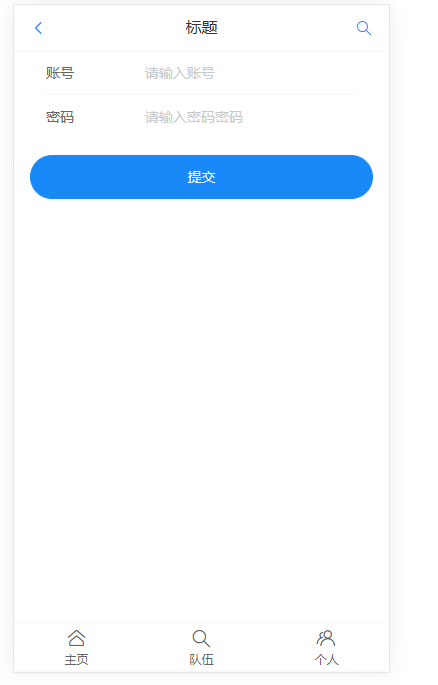
}

};

</script>

<style scoped>

</style>



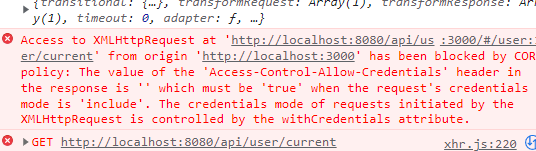
登录成功之后，发现请求没带上cookie



在myAxios.ts 添加配置

myAxios.defaults.withCredentials = true; // 允许携带 cookie

出现新问题



给后端接口添加允许携带cookie的配置



后端设置cookie的作用域



**获取和修改用户信息**

提取 获取用户信息的方法

在src 目录下 新建 service目录 --> user.ts

/\*\*

\* 获取用户信息

\* @returns {Promise<null|any>}

\*/

export const getCurrentUser = async () => {

// const user = getCurrentUserState();

// if (user) {

// return user;

// }

//从远程处获取用户信息

const res = await myAxios.get('/user/current');

if (res.code === 0 ) {

setCurrentUserState(res.data);

return res.data;

}

return null;

}

UserPage 获取用户信息

<template>

<template v-if="user">

<van-cell title="头像" is-link to="/user/edit">

<img :src="user.avatarUrl" style="height: 48px">

</van-cell>

<van-cell title="昵称" is-link :value="user.username" @click="toEdit('username', user.username, '昵称')"/>

<van-cell title="账号" :value="user.userAccount"/>

<van-cell title="性别" is-link :value="user.gender" @click="toEdit('gender', user.gender, '性别')"/>

<van-cell title="电话" is-link :value="user.phone" @click="toEdit('phone', user.phone, '电话')"/>

<van-cell title="邮箱" is-link :value="user.email" @click="toEdit('email', user.email, '邮箱')"/>

<van-cell title="星球编号" :value="user.planetCode"/>

<van-cell title="注册时间" :value="user.createTime"/>

</template>

</template>

<script setup lang="ts">

import {onMounted, ref} from "vue";

import {useRouter} from "vue-router";

import {getCurrentUser} from "../services/user";

const user=ref();

const router = useRouter();

onMounted( async ()=>{

user.value= await getCurrentUser();

})

const toEdit = (editKey: string, currentValue: string, editName: string) => {

router.push({

path: '/user/edit',

query: {

editKey,

currentValue,

editName

}

})

}

</script>

<style scoped>

</style>

UserEditPage 发送修改用户信息请求

<template>

<van-form @submit="onSubmit">

<van-field

v-model="editUser.currentValue"

:name="editUser.editKey"

:label="editUser.editName"

:placeholder="`请输入${editUser.editName}`"

/>

<div style="margin: 16px;">

<van-button round block type="primary" native-type="submit" @click="onSubmit">

提交

</van-button>

</div>

</van-form>

</template>

<script setup lang="ts">

import {useRoute, useRouter} from "vue-router";

import { ref} from 'vue';

import myAxios from '../plugins/myAxios';

import {Toast} from "vant";

import {getCurrentUser} from '../services/user';

const route = useRoute();

const router=useRouter();

const editUser = ref({

editKey: route.query.editKey,

currentValue: route.query.currentValue,

editName: route.query.editName

})

// 不可以写在外面，否则页面不显示内容，还没有报错信息

// const currentUser= await getCurrentUser();

const onSubmit = async () => {

//异步方法必须添加 await 才可以拿到数据，否则拿到的是 promise 对象

const currentUser= await getCurrentUser();

if (!currentUser) {

Toast.fail('用户未登录');

return;

}

console.log("-------UserEditPage",currentUser);

const res= await myAxios.post('/user/update',{

'id': currentUser.id,

[editUser.value.editKey as string ]: editUser.value.currentValue, //动态取值

})

console.log("修改用户信息",res);

if (res.code === 0 && res.data >0 ){

Toast.success('修改成功');

router.replace('/user');

}else {

Toast.fail(' 修改失败');

}

};

</script>

<style scoped>

</style>

缓存当前用户信息

在src 目录下 新建states目录 --> user.ts

import {UserType} from "../models/user";

let currentUser: UserType;

const setCurrentUserState = (user:UserType) =>{

currentUser =user;

}

const getCurrentUserState = (): UserType =>{

return currentUser;

}

export {

setCurrentUserState,

getCurrentUserState,

}

**直播6**

开发主页 （ 默认推荐和自己兴趣相当的用户 ）

优化主页的性能 （ 缓存 + 定时任务 + 分布式锁 ）

**笔录**

**开发主页**

最简单：直接 List 列表

模拟 1000 万个用户，再去查询

**导入数据**

可视化界面 ： 适合一次性导入，数据量可控

写程序： for 循环，建议分批，不要一把梭哈 （可以用接口来控制） **要保证可控，幂等，注意线上环境和测试环境是有区别的**  导入 1000 万条 ， for i 1000 w

执行SQL 语句：适用于小数据量

**编写一次性任务**

for 循环插入数据的问题：

建立和释放数据库连接（批量查询解决）

for 循环是绝对线性的 （并发）

并发要注意执行的先后顺序无所谓，不要用到非并发类的集合

//CPU 密集型： 分配的核心线程数 = CPU -1

// IO 密集型： 分配的核心线程数可以大于 CPU 核数

private ExecutorService executorService=new ThreadPoolExecutor(40,1000,10000, TimeUnit.MINUTES,new ArrayBlockingQueue<>(10000));

数据库慢？ 预先把数据查出来，放到一个更快读取的地方，不用再查数据库了（缓存）

预加载 缓存， 定时更新缓存 （定时任务）

多个机器都要执行任务么？ （分布式锁： 控制同一时间只有一台机器去执行定时任务，其他机器不用重复执行了）

**开发页面**

**编写主页 （直接list列表）**

Controller

@GetMapping("/recommend")

public BaseResponse<Page<User>> recommendUsers(long pageSize,long pageNum,HttpServletRequest request){

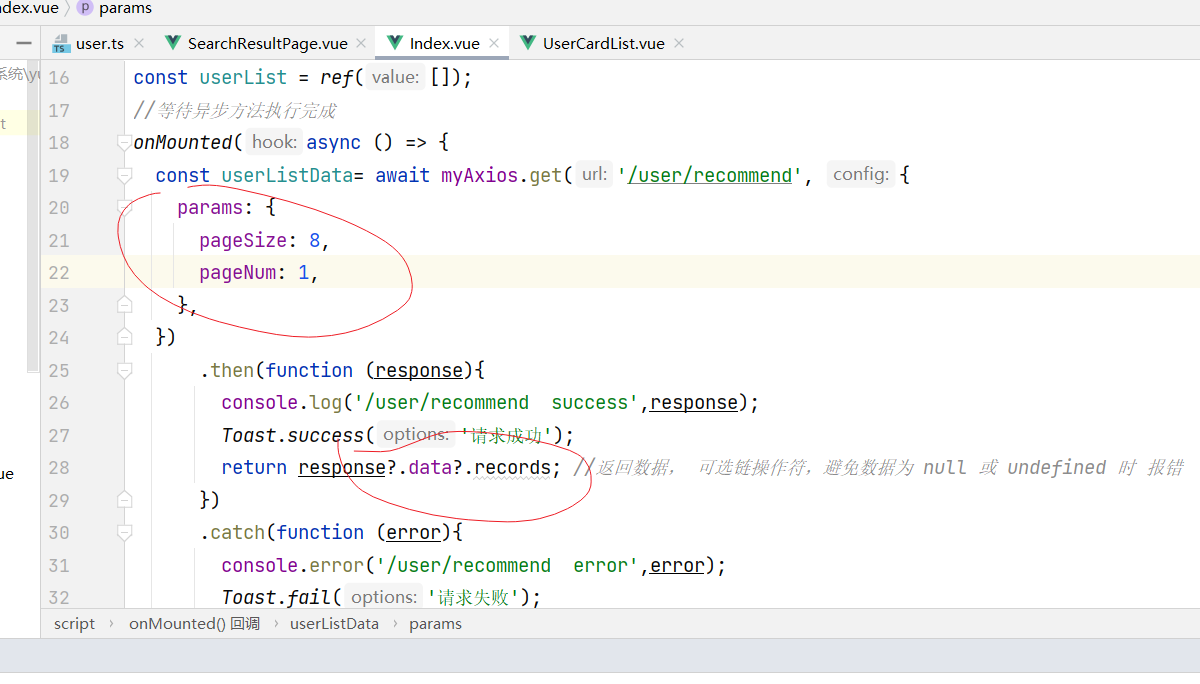
QueryWrapper<User> queryWrapper =new QueryWrapper<>();

Page<User> userList=userService.page(new Page<>(pageNum,pageSize),queryWrapper);

return ResultUtils.success(userList);

}

**前端数据修改**



把用户信息表抽象出来

先修改一下布局（底部无法完全显示）



在 src/components 下创建UserCardList

<template>

<van-card

v-for="user in props.userList"

:desc="user.profile"

:title="`${user.username} (${user.planetCode})`"

:thumb="user.avatarUrl"

>

<template #tags>

<van-tag plain type="danger" v-for="tag in user.tags" style="margin-right: 8px; margin-top: 8px" >

{{tag}}

</van-tag>

</template>

<template #footer>

<van-button size="mini">联系我</van-button>

</template>

</van-card>

</template>

<script setup lang="ts">

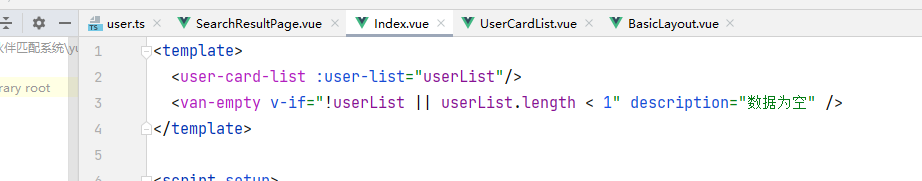
import {UserType} from "../models/user";

interface UserCardListProps{

userList: UserType[];

}

在主页面和搜索结果页面调用





**数据模拟**

我们需要插入数据：

用可视化界面：适合一次性导入、数据量可控由于编码，主键以及某些字段的问题（id，createtime等），演示插入失败，这里不推荐

写程序：for 循环，建议分批，不要一把梭哈,这里演示了两种插入数据的方法首先创建测试方法文件InsertUsersTest，编写批量查询解决

@SpringBootTest

public class InsertUsersTest {

@Resource

private UserService userService;

//private ExecutorService executorService = new ThreadPoolExecutor(40,1000,10000, TimeUnit.MINUTES,new ArrayBlockingQueue<>(10000));

@Test

public void doInsertUsers(){

StopWatch stopWatch = new StopWatch();

stopWatch.start();

final int INSERT\_NUM=100000;

List<User> userList = new ArrayList<>();

for (int i = 0; i < INSERT\_NUM; i++) {

User user = new User();

user.setUsername("假数据");

user.setUserAccount("fakeaccount");

user.setAvatarUrl("https://img1.baidu.com/it/u=1645832847,2375824523&fm=253&fmt=auto&app=138&f=JPEG?w=480&h=480");

user.setGender(0);

user.setUserPassword("231313123");

user.setPhone("1231312");

user.setEmail("12331234@qq.com");

user.setUserStatus(0);

user.setUserRole(0);

user.setPlanetCode("213123");

user.setTags("[]");

userList.add(user);

}

userService.saveBatch(userList,10000);

stopWatch.stop();

System.out.println(stopWatch.getTotalTimeMillis());

}

}

并发执行，这里的线程可自定义或者用idea默认的，两种方法的区别是，自定义可以跑满线程，而默认的只能跑CPU核数-1，代码区别：就是在异步执行处加上自定义的线程名

private ExecutorService executorService = new ThreadPoolExecutor(40,1000,10000, TimeUnit.MINUTES,new ArrayBlockingQueue<>(10000));

@Test

public void doConcurrencyInsertUsers(){

StopWatch stopWatch = new StopWatch();

stopWatch.start();

//分10组

int batchSize = 5000;

int j=0;

List<CompletableFuture<Void>> futureList = new ArrayList<>();

for (int i = 0; i < 20; i++) {

List<User> userList = new ArrayList<>();

while (true){

j++;

User user = new User();

user.setUsername("假数据");

user.setUserAccount("fakeaccount");

user.setAvatarUrl("https://img1.baidu.com/it/u=1645832847,2375824523&fm=253&fmt=auto&app=138&f=JPEG?w=480&h=480");

user.setGender(0);

user.setUserPassword("231313123");

user.setPhone("1231312");

user.setEmail("12331234@qq.com");

user.setUserStatus(0);

user.setUserRole(0);

user.setPlanetCode("213123");

user.setTags("[]");

userList.add(user);

if (j % batchSize==0){

break;

}

}

//异步执行

CompletableFuture<Void> future = CompletableFuture.runAsync(() ->{

System.out.println("threadName:"+Thread.currentThread().getName());

userService.saveBatch(userList,batchSize);

},executorService);

futureList.add(future);

}

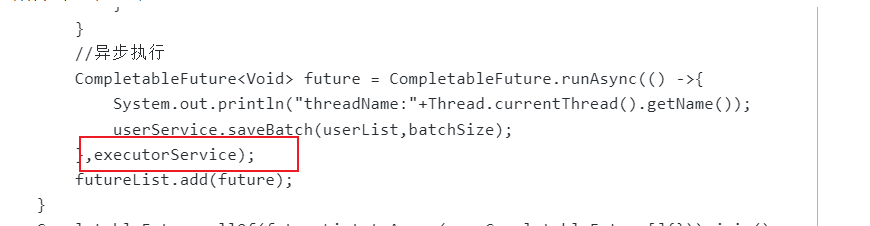
CompletableFuture.allOf(futureList.toArray(new CompletableFuture[]{})).join();

stopWatch.stop();

System.out.println(stopWatch.getTotalTimeMillis());

}

若使用默认线程池，删去



现在启动前后端，查看主页，发现搜查不出，这是因为数据太多需要分页，修改后端接口方法

/\*\*

\* 推荐页面

\* @param request

\* @return

\*/

@GetMapping("/recommend")

public BaseResponse<Page<User>> recommendUsers(long pageSize,long pageNum, HttpServletRequest request){

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

Page<User> userList = userService.page(new Page<>(pageNum, pageSize), queryWrapper);

return ResultUtils.success(userList);

}

同时还要引入mybatis的分页插件配置，直接复制文档到config目录

@Configuration

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

public class MybatisPlusConfig {

/\*\*

\* 新的分页插件,一缓和二缓遵循mybatis的规则,需要设置 MybatisConfiguration#useDeprecatedExecutor = false 避免缓存出现问题(该属性会在旧插件移除后一同移除)

\*/

@Bean

public MybatisPlusInterceptor mybatisPlusInterceptor() {

MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();

interceptor.addInnerInterceptor(new PaginationInnerInterceptor(DbType.H2));

return interceptor;

}

}

主要不要忘了把扫包的路径改为自己的

现在去修改前端主页



**直播7**

**笔录**

**数据库查询慢怎么办？**

用缓存： 提前把数据提出来保存好 （ 通常保存到读写更快的介质，比如内存 ） 就可以更快地读写。

**缓存的实现**

Redis （分布式缓存）

memcache （分布式）

Etcd （云原生架构的一个分布式存储，存储配置，扩充能力）

ehcache （单机）

本地缓存 （Java 内存 Map）

Caffeine （Java 内存缓存，高性能）

Goole Guava

**Redis**

NoSQL 数据库

key - value 存储系统 （区别于MySQL ， 它存储的是键值对）

**Redis 数据结构**

String 字符串类型： name: "yupi"

List 列表 ： names : ["yupi","jayin","yupi"]

Set集合 ： names : ["yupi","jayin"] （值不能重复）

Hash 哈希 ： nameAge : { "yupi":1,"jayin":2}

Zset 集合： names :{ yupi - 9, jayin - 12 } (适合做排行榜)

bloomfilter ( 布隆过滤器，主要从大量的数据中快速过滤值，比如邮件黑名单拦截）

geo （ 计算地理位置）

hyperloglog （ pv / uv)

pub / sub ( 发布订阅，类似消息队列）

BigMap （10101011010101001010101010）

**自定义序列化**

@Configuration

public class RedisTemplateConfig {

@Bean

public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory connectionFactory) {

RedisTemplate<String, Object> redisTemplate = new RedisTemplate<>();

redisTemplate.setConnectionFactory(connectionFactory);

redisTemplate.setKeySerializer(RedisSerializer.string());

return redisTemplate;

}

}

引入一个库时，先写测试类

**Java里的实现方式**

**Spring Data Redis （推荐）**

Spring Data : 通用的数据访问框架，定义了一组 **增删改查** 的接口

mysql , redis , jpa

spring -data -redis

引入

<dependency>

   <groupId>org.springframework.boot</groupId>

   <artifactId>spring-boot-starter-data-redis</artifactId>

   <version>2.6.4</version>

</dependency>

配置Redis 地址

spring:

 # redis 配置

redis:

  port: 6379

  host: localhost

  database: 0

**Jedis**

独立于 Spring 操作 Redis 的Java 客户端

要配合 Jedis Pool 使用

**Lettuce**

高阶的 操作 Redis 的 Java 客户端

异步，连接池

**Redisson**

分布式操作Redis 的Java 客户端，让你像在使用本地的集合一样操作 Redis ( 分布式 Redis 数据网络 ）

**JetCache**

对比

如果你用的是 Spring ，并且没有过多的定制化要求，可以用 Spring Data Redis 最方便

如果你用的不是Spring， 并且追求简单，并且没过高的性能要求， 可以用 Jedis + Jedis Pool

如果你的项目不是 Spring，并且追求高性能、高定制化，可以用 Lettuce，支持异步、连接池

如果你的项目是分布式的，需要用到一些分布式的特性（比如分布式锁、分布式集合），推荐用 redisson

**设计缓存key**

不同用户看到的数据不同

systemId.moduleId.func (不要和别人冲突）

yupao:user:recommend:userId

**redis 内存不能无限增加 ， 一定要设置过期时间**

**缓存预热**

问题： 第一个用户访问还是比较慢 ，一定程度上保护了数据库

缓存预热的优点：

解决上面的问题，可以让用户适中访问呢很快

缺点：

增加开发成本 （额外的开发，设计）

预热的时机和时间如果错了，有可能你缓存的数据不对或者太老

占用 额外 空间

**怎么缓存预热？**

定时

模拟出发 （ 手动触发 ）

**实现**

用定时任务，每天刷新所有用户的推荐列表

注意点：

缓存预热的意义 （ 新增少，总用户多 )

缓存的空间不能太大，要预留给其他缓存 空间

缓存数据的周期（ 此处 每天一次 )

分析优缺点的时候，要打开思路，从整个项目从 0 到 1 的链路上去分析

**定时任务实现**

Spring Scheduler（spring boot 默认整合了）

Quartz（独立于 Spring 存在的定时任务框架）

XXL-Job 之类的分布式任务调度平台（界面 + sdk）

第一种方式：

主类开启 @EnableScheduling

给要定时执行的方法添加 @Scheduling 注解，指定 cron 表达式或者执行频率

不要去背 cron 表达式！！！！！

<https://cron.qqe2.com/>

<https://www.matools.com/crontab/>

**测试Redis**

因为java自带的redistemplate只能查询字符串类型，不够全面，所以我们需要自定义序列化，在config包里创建如下配置类

@Configuration

public class RedisTemplateConfig {

@Bean

public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory connectionFactory) {

RedisTemplate<String, Object> redisTemplate = new RedisTemplate<>();

redisTemplate.setConnectionFactory(connectionFactory);

redisTemplate.setKeySerializer(RedisSerializer.string());

return redisTemplate;

}

}

在测试包里创建测试类RedisTest，测试代码如下

@SpringBootTest

public class RedisTest {

@Resource

private RedisTemplate redisTemplate;

@Test

void test() {

ValueOperations valueOperations = redisTemplate.opsForValue();

// 增

valueOperations.set("yupiString", "dog");

valueOperations.set("yupiInt", 1);

valueOperations.set("yupiDouble", 2.0);

User user = new User();

user.setId(1L);

user.setUsername("yupi");

valueOperations.set("yupiUser", user);

// 查

Object yupi = valueOperations.get("yupiString");

Assertions.assertTrue("dog".equals((String) yupi));

yupi = valueOperations.get("yupiInt");

Assertions.assertTrue(1 == (Integer) yupi);

yupi = valueOperations.get("yupiDouble");

Assertions.assertTrue(2.0 == (Double) yupi);

System.out.println(valueOperations.get("yupiUser"));

valueOperations.set("yupiString", "dog");

redisTemplate.delete("yupiString");

}

}

redis中可查询到缓存的数据

**根据用户开发个性推荐页**

**修改推荐页面的接口**

@GetMapping("/recommend")

public BaseResponse<Page<User>> recommendUsers(long pageSize,long pageNum,HttpServletRequest request){

User loginUser = userService.getLoginUser(request);

String redisKey = String.format("yupao:user:recommend:%s", loginUser.getId());

ValueOperations<String,Object> valueOperations = redisTemplate.opsForValue();

//如果有缓存，直接读缓存

Page<User> userPage = (Page<User>) valueOperations.get(redisKey);

if (userPage!=null){

return ResultUtils.success(userPage);

}

//无缓存 ， 查数据库

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

userPage=userService.page(new Page<>(pageNum,pageSize),queryWrapper);

//写缓存

try {

valueOperations.set(redisKey,userPage,30000, TimeUnit.MILLISECONDS);

}catch (Exception e){

log.error("redis set key error",e);

}

return ResultUtils.success(userPage);

}

刷新页面，有了缓存之后查询速度变快

**定时任务**

但是还存在一个问题：第一个用户访问还是很慢（加入第一个老板），要实现缓存预热这里我们使用了定时的方法

新建一个job包，写入下面代码实现定时预热缓存

/\*\*

\* 缓存预热任务

\*/

@Component

@Slf4j

public class PreCacheJob {

@Resource

private UserService userService;

@Resource

private RedisTemplate<String, Object> redisTemplate;

//重点用户

private List<Long> mainUserList= Arrays.asList(1L);

@Scheduled(cron = "0 59 23 \* \* \*")

public void doCacheRecommendUser(){

for (Long userId : mainUserList){

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

Page<User> userPage=userService.page(new Page<>(1,20),queryWrapper);

String redisKey = String.format("yupao:user:recommend:%s", userId);

ValueOperations<String,Object> valueOperations = redisTemplate.opsForValue();

//写缓存

try {

valueOperations.set(redisKey,userPage,30000, TimeUnit.MILLISECONDS);

}catch (Exception e){

log.error("redis set key error",e);

}

}

}

}

**注意:要在UserCenterApplication添加@EnableScheduling注解，允许定时任务**

删去redis里的缓存，设定一个时间，启动项目（前后端），等待时间到来，发现redis里面已经有了缓存

**直播8**

**笔录**

**控制定时任务的执行**

为什么？

浪费资源，想象 1w台服务器同时 “打鸣”

出现脏数据，比如重复插入

**方案**

**1. 要控制定时任务在同一时间只有一台服务器能执行**

怎么做？

分离定时任务程序和主程序，只在 1 个 服务器 运行 定时任务。成本太大

写死配置，每个服务器都执行定时任务，但是只有ip符合配置的服务器才真实执行业务逻辑，其他的直接返回。成本最低，但是我们的IP 可能是不固定的，把 IP 写太死了。

动态配置，配置是可以轻松，方便地更新 （ 代码无需重启 ） ，但是只有 ip 符合配置的服务器才能真实执行业务逻辑

数据库

Redis

配置中心（Nacos,Apollp,Spring Cloud Config )

问题： 服务器太多了 ， IP不可控还是很麻烦，需要人工修改

分布式锁，只有抢到锁的服务器才能执行业务逻辑

坏处：增加成本，但不多

好处：不用手动配置，多少个服务器都一样

**单机就会存在单点故障**

**锁**

有限资源的情况下，控制同一时间段 只有某些线程 （ 用户 / 服务器 ） 能访问到资源

Java 实现锁： synchronized 关键字， 并发包类

问题： 只对单个JVM 有效

**分布式锁**

为什么需要分布式锁？

有限资源的情况下，控制同一时间端只有某些线程 （ 用户 / 服务器 ） 能访问到资源

单个锁只对单个jvm有

**分布式锁实现的关键**

**抢锁机制**

怎么保证同一时间只有1 台 服务器抢到 锁？

**核心思想**： 先来的人先把数据改成自己的标识（ 服务器 ip ） ，后来的人发现表示已存在，就抢锁失败，继续等待。等先来的人执行方法结束，把标识清空，其他的人继续抢锁

MySQL 数据库： select for update 行级锁 （最简单）

（乐观锁）

（主流）Redis 实现： 内存数据库 **读写速度快**。 支持 setnx ,lua 脚本， 比较方便我们实现分布式锁

setnx : set if not exits 如果不存在 则设置； 只有设置成功才会返回true , 否则返回false

**注意事项**

用完锁 要释放(腾出地方）

锁一定要加过期时间

如果方法执行时间过长，锁提前过去了？

问题：

连锁效应：释放掉别人的锁

这样会存在多个方法同时执行的情况

解决方案：

续期 （伪代码）

boolean end = false;

new Thread( ()->{

if(!end)}{

续期

}

)

end=true;

释放锁的时候，有可能先判断出是自己的锁，但这时锁过期了，最后还是释放了别人的锁

//原子操作

if(get lock == A ){

// set lock B

del lock

}

Redis + lua 脚本实现<https://blog.csdn.net/feiying0canglang/article/details/113258494>

Redis 如果是集群（而不是只有一个Redis),如果分布式锁的数据不同步怎么办？

红锁

**拒绝自己写！**

**Redisson 实现 分布式锁**

Java 客户端 数据网络

实现了很多Java里支持的接口和数据结构

Redisson 是 一个 Java 操作 Redis 的客户端，  **提供了大量的分布式数据集来简化对Redis 的操作和使用，可以让开发者像使用本地集合一样使用Redis, 完全感知不到Redis 的存在**

**2种引入方式**

spring boot starter （不推荐， 版本迭代太快，容易冲突） <https://github.com/redisson/redisson/tree/master/redisson-spring-boot-starter>

直接引入 <https://github.com/redisson/redisson#quick-start>

**示例代码**

//list 数据存在本地 JVM 内存种

List<String> list=new ArrayList<>();

list.add("jayin");

System.out.println("list:"+list.get(0));

list.remove(0);

//数据存在 redis 的内存种

RList<String> rList=redissonClient.getList("test-list");

// rList.add("jayin");

System.out.println("rList:"+rList.get(0));

rList.remove(0);

**定时任务+锁**

waitTime 设置为 0 ，只抢一次，抢不到就放弃

注意释放锁要写在 finally 中

**实现代码**

void testWatchDog() {

  RLock lock = redissonClient.getLock("yupao:precachejob:docache:lock");

  try {

      // 只有一个线程能获取到锁

      if (lock.tryLock(0, -1, TimeUnit.MILLISECONDS)) {

          // todo 实际要执行的方法

          doSomeThings();

          System.out.println("getLock: " + Thread.currentThread().getId());

      }

  } catch (InterruptedException e) {

      System.out.println(e.getMessage());

  } finally {

      // 只能释放自己的锁

      if (lock.isHeldByCurrentThread()) {

          System.out.println("unLock: " + Thread.currentThread().getId());

          lock.unlock();

      }

  }

}

**看门狗机制**

redisson 中提供的续期机制

开一个监听线程，如果方法还没执行完，就帮你重置redis 锁的过期时间

原理：

监听当前线程，默认过期时间是30秒，每10秒续期一次（补到30秒）

如果线程挂掉（注意debug模式也会被它当成服务器宕机），则不会续期

<https://blog.csdn.net/YXXXYX/article/details/126360055>

Zookeeper 实现（不推荐）

**redisson 实现分布式锁**

引入依赖

<dependency>

<groupId>org.redisson</groupId>

<artifactId>redisson</artifactId>

<version>3.17.5</version>

</dependency>

写redisson 配置

@Configuration

@ConfigurationProperties(prefix = "spring.redis")

@Data

public class RedissonConfig {

private String host;

private String port;

@Bean

public RedissonClient redissonClient(){

// 1. 创建配置

Config config = new Config();

String redisAddress= String.format("redis://%s:%s",host,port);

config.useSingleServer().setAddress(redisAddress).setDatabase(3);

// 2. 创建实例

RedissonClient redisson = Redisson.create(config);

return redisson;

}

}

编写测试类

@SpringBootTest

public class RedissonTest {

@Resource

private RedissonClient redissonClient;

@Test

void test(){

//list 数据存在本地 JVM 内存种

List<String> list=new ArrayList<>();

list.add("jayin");

System.out.println("list:"+list.get(0));

list.remove(0);

//数据存在 redis 的内存种

RList<String> rList=redissonClient.getList("test-list");

// rList.add("jayin");

System.out.println("rList:"+rList.get(0));

rList.remove(0);

// map

Map<String,Integer> map=new HashMap<>();

map.put("yupi",10);

map.get("yupi");

RMap<Object,Object> map1=redissonClient.getMap("test-map");

//map1.put("yupi",10);

System.out.println(map1.get("yupi"));

map1.remove("yupi");

}

}

**定时任务＋锁**

修改定时任务

/\*\*

\* 缓存预热任务

\*/

@Component

@Slf4j

public class PreCacheJob {

@Resource

private UserService userService;

@Resource

private RedisTemplate<String, Object> redisTemplate;

@Resource

private RedissonClient redissonClient;

//重点用户

private List<Long> mainUserList= Arrays.asList(1L);

@Scheduled(cron = "0 4 0 \* \* \*")

public void doCacheRecommendUser(){

RLock lock=redissonClient.getLock("yupao:precachejob:docache:lock");

try {

//只有一个线程能获取到锁 timeout:锁一定要加过期时间

if (lock.tryLock(0,30000L,TimeUnit.MILLISECONDS)){

System.out.println("getlock:"+Thread.currentThread().getId());

for (Long userId : mainUserList){

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

Page<User> userPage=userService.page(new Page<>(1,20),queryWrapper);

String redisKey = String.format("yupao:user:recommend:%s", userId);

ValueOperations<String,Object> valueOperations = redisTemplate.opsForValue();

//写缓存

try {

valueOperations.set(redisKey,userPage,30000, TimeUnit.MILLISECONDS);

}catch (Exception e){

log.error("redis set key error",e);

}

}

}

} catch (InterruptedException e) {

log.error("doCacheRecommendUser error",e);

}finally {

// 判断当前的锁是不是这个线程加的锁

if (lock.isHeldByCurrentThread()) {

System.out.println("unlock:"+Thread.currentThread().getId());

//用完锁要释放

lock.unlock();

}

}

}

}

打包项目，在终端打开两个，主程序启动（由于定时任务太过于麻烦，所以我们提取出来写一个测试）

    @Test

    void testWatchDog(){

        RLock lock = redissonClient.getLock("yupao:precachejob:docache:lock");

        try{

            //只有一个线程能获取到锁

            if (lock.tryLock(0,-1, TimeUnit.MILLISECONDS)){

                Thread.sleep(300000);

                System.out.println("getLock: "+Thread.currentThread().getId());

            }

        }catch(InterruptedException e){

            System.out.println(e.getMessage());

        }finally {

            //只能释放自己的锁

            if (lock.isHeldByCurrentThread()){

                System.out.println("unlock: "+Thread.currentThread().getId());

                lock.unlock();

            }

        }

    }

注意锁的存在时间要设置为-1（开启开门狗），默认锁的过期时间是30秒，通过sleep实现

运行，通过quickredis观察，可以发现 每 10 秒续期一次（补到 30 秒）

**踩坑处：不要用debug启动，会被认为是宕机**

**直播9**

**组队功能**

理解为 王者荣誉

**理想的应用场景**

我要跟别人一起参加竞赛或者做项目，可以发起组队或者加入别人的队伍

**需求分析**

用户可以 **创建** 一个队伍， 设置队伍的人数，队伍的名称（标题），描述，超时时间

队长，剩余的人数

聊天？

公开或private 或 加密

信息流中不展示已过期的队伍

根据标签或名称搜索队伍

展示队伍列表，根据名称搜索队伍 ，信息流中不展示已过去的队伍

修改队伍信息

用户可以加入队伍（其他人，未满，未过期），允许加入多个队伍，但是要有个上限

是否需要队长同意？筛选审批？

用户可以退出队伍 （ 如果队长退出，权限转移给第二早加入的用户 -- 先来后到）

队长可以解散队伍

分享队伍 ==》 邀请其他用户加入队伍

队伍满人后发送消息通知

**创建队伍 （接口）设计**

用户可以 **创建** 一个队伍， 设置队伍的人数，队伍的名称（标题），描述，超时时间

队长，剩余的人数

聊天？

公开或private 或 加密

信息流中不展示已过期的队伍

根据标签或名称搜索队伍

请求参数是否为空？

是否登录，未登录不允许创建

校验信息

队伍人数 > 1 且 <= 20

队伍标题 <= 20

描述 <= 512

status 是否公开（int）不传默认为 0（公开）

如果 status 是加密状态，一定要有密码，且密码 <= 32

超时时间 > 当前时间

校验用户最多创建 5 个队伍

插入队伍信息到队伍表

插入用户 => 队伍关系到关系表

**事务注解**

@Transactional(rollbackFor = Exception.class)

要么数据操作成功，要么失败

**数据库表设计**

队伍表team

字段：

id 主键 bigint （ 最简单，连续，放url 上比较简短，但缺点是爬虫

name 队伍名称

description 描述

maxNum 最大人数

expireTime 过期时间

userId 创建人 id

status 0 - 公开，1 - 私有，2 - 加密

password 密码

createTime 创建时间

updateTime 更新时间

isDelete 是否删除

create table team

(

  id           bigint auto\_increment comment 'id'

       primary key,

  name   varchar(256)                   not null comment '队伍名称',

  description varchar(1024)                      null comment '描述',

  maxNum    int      default 1                 not null comment '最大人数',

  expireTime    datetime  null comment '过期时间',

  userId            bigint comment '用户id',

   status    int      default 0                 not null comment '0 - 公开，1 - 私有，2 - 加密',

   password varchar(512)                       null comment '密码',

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

  updateTime   datetime default CURRENT\_TIMESTAMP null on update CURRENT\_TIMESTAMP,

  isDelete     tinyint  default 0                 not null comment '是否删除'

)

   comment '队伍';

用户 - 队伍表 user-team

字段：

id 主键

userId 用户 id

teamId 队伍 id

joinTime 加入时间

createTime 创建时间

updateTime 更新时间

isDelete 是否删除

create table user\_team

(

  id           bigint auto\_increment comment 'id'

       primary key,

  userId            bigint comment '用户id',

  teamId            bigint comment '队伍id',

  joinTime datetime  null comment '加入时间',

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

  updateTime   datetime default CURRENT\_TIMESTAMP null on update CURRENT\_TIMESTAMP,

  isDelete     tinyint  default 0                 not null comment '是否删除'

)

   comment '用户队伍关系';

两个关系：

用户加了哪些队伍？

队伍有哪些用户？

方式：

建立用户 - 队伍关系表 teamId userId (便于修改，查询性能高一点，可以选择这个，不用全表遍历）

用户表补充已加入的队伍字段，队伍表补充已加入的用户字段 （ 便于查询，不用写多对多的代码，可以直接根据队伍查用户，根据用户查队伍）

**为什么需要请求参数包装类？**

请求参数名称 / 类型 和实体类不一样

有一些参数用不到，如果要自动生成接口文档，会增加理解成本

多个字段类映射到同一个对象

**为什么需要包装类**

可能有些字段需要隐藏，不能返回给前端

或者有些字段某些方法是不关心的

**建表**

-- 队伍表

create table team

(

id bigint auto\_increment comment 'id'

primary key,

name varchar(256) not null comment '队伍名称',

description varchar(1024) null comment '描述',

maxNum int default 1 not null comment '最大人数',

expireTime datetime null comment '过期时间',

userId bigint comment '用户id',

status int default 0 not null comment '0 - 公开，1 - 私有，2 - 加密',

password varchar(512) null comment '密码',

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

updateTime datetime default CURRENT\_TIMESTAMP null on update CURRENT\_TIMESTAMP,

isDelete tinyint default 0 not null comment '是否删除'

)

comment '队伍';

-- 用户 -队伍关系表

create table user\_team

(

id bigint auto\_increment comment 'id'

primary key,

userId bigint comment '用户id',

teamId bigint comment '队伍id',

joinTime datetime null comment '加入时间',

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

updateTime datetime default CURRENT\_TIMESTAMP null on update CURRENT\_TIMESTAMP,

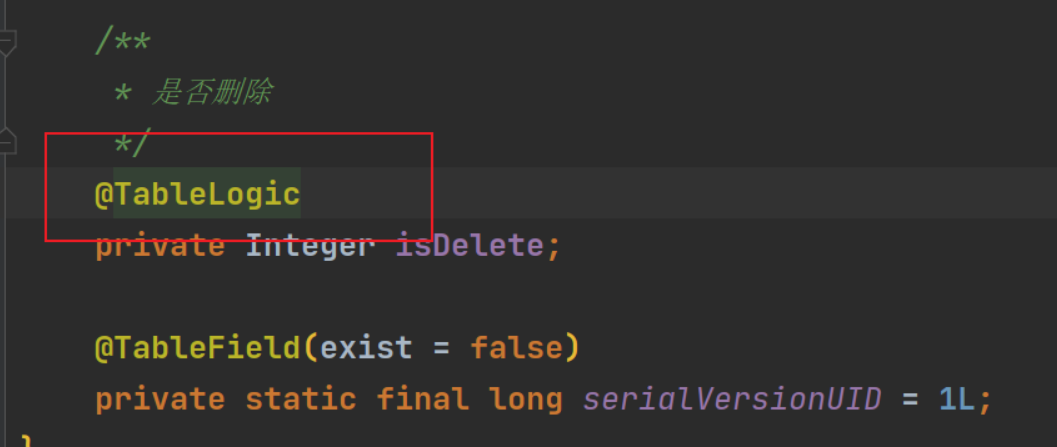
isDelete tinyint default 0 not null comment '是否删除'

)

comment '用户队伍关系';

使用MybatisX-Generator生成domain，service和mapper文件，然后把生成的文件都移到对应的目录里面，别忘了把mapper.xml里的路径改成自己对应的。

PS：别忘了在team和user\_team类中的is\_delete字段添加@TableLogic注解，实现逻辑删除



**初始化**

**TeamContrller**

/\*\*

\* 队伍接口

\*/

@RestController

@RequestMapping("/team")

@CrossOrigin(origins = { "http://127.0.0.1:5173"},allowCredentials = "true")

@Slf4j

public class TeamController {

@Resource

private UserService userService;

@Resource

private TeamService teamService;

@PostMapping("/add")

public BaseResponse<Long> addTeam(@RequestBody TeamAddRequest teamAddRequest, H[ttpServletRequest req](http://127.0.0.1:5173)uest) {

if (teamAddRequest == null) {

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

User loginUser = userService.getLoginUser(request);

Team team = new Team();

BeanUtils.copyProperties(teamAddRequest, team);

long teamId = teamService.addTeam(team, loginUser);

return ResultUtils.success(teamId);

}

@PostMapping("/delete")

public BaseResponse<Boolean> deleteTeam(@RequestBody long id){

if (id<=0){

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

boolean save = teamService.removeById(id);

if(!save){

throw new BusinessException(ErrorCode.SYSTEM\_ERROR,"删除失败");

}

return ResultUtils.success(true);

}

@PostMapping("/update")

public BaseResponse<Boolean> deleteTeam(@RequestBody Team team){

if (team==null){

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

boolean save = teamService.updateById(team);

if(!save){

throw new BusinessException(ErrorCode.SYSTEM\_ERROR,"更新失败");

}

return ResultUtils.success(true);

}

@GetMapping("/get")

public BaseResponse<Team> getTeamById(long id){

if(id<=0){

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

Team team = teamService.getById(id);

if (team==null){

throw new BusinessException(ErrorCode.NULL\_ERROR);

}

return ResultUtils.success(team);

}

@GetMapping("/list")

public BaseResponse<List<Team>> listTeams(TeamQuery teamQuery){

if (teamQuery==null){

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

Team team=new Team();

BeanUtils.copyProperties(teamQuery,team);

QueryWrapper<Team> queryWrapper=new QueryWrapper<>(team);

List<Team> teamList = teamService.list(queryWrapper);

return ResultUtils.success(teamList);

}

@GetMapping("/list/page")

public BaseResponse<Page<Team>> listTeamsByPage(TeamQuery teamQuery){

if (teamQuery==null){

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

Team team=new Team();

BeanUtils.copyProperties(teamQuery,team);

Page<Team> page = new Page<>(teamQuery.getPageNum(), teamQuery.getPageSize([));](http://127.0.0.1:5173)

QueryWrapper<Team> queryWrapper=new QueryWrapper<>(team);

Page<Team> resultPage=teamService.page(page,queryWrapper);

return ResultUtils.success(resultPage);

}

}

**dto包，包装类TeamQuery**

/\*\*

\* 队伍查询封装类

\*/

@Data

public class TeamQuery extends PageRequest {

/\*\*

\* id

\*/

@TableId(type = IdType.AUTO)

private Long id;

/\*\*

\* 队伍名称

\*/

private String name;

/\*\*

\* 描述

\*/

private String description;

/\*\*

\* 最大人数

\*/

private Integer maxNum;

/\*\*

\* 用户id

\*/

private Long userId;

/\*\*

\* 0 - 公开，1 - 私有，2 - 加密

\*/

private Integer status;

}

**common包-分页请求参数包装类**

/\*\*

\* 通用分页请求参数

\*/

@Data

public class PageRequest implements Serializable {

private static final long serialVersionUID = 1L;

/\*\*

\* 页面大小

\*/

protected int pageSize;

/\*\*

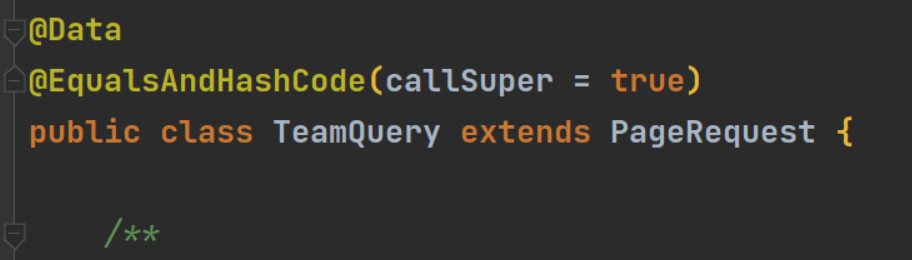
\* 当前是第几页

\*/

protected int pageNum;

}

同时在Team Query 上继承这个分页请求，后面接口会用上



**测试**

**细化接口（根据具体的需求）**

这边我们会运用到队伍的状态，即 公开，私有等，所以我们提前写一个队伍状态枚举类

/\*\*

\* 队伍状态枚举

\*/

public enum TeamStatusEnum {

PUBLIC(0,"公开"),

PRIVATE(1,"私有"),

SECRET(2,"加密");

private int value;

private String text;

public static TeamStatusEnum getEnumByValue(Integer value) {

if (value == null){

return null;

}

TeamStatusEnum[] values = TeamStatusEnum.values();

for (TeamStatusEnum teamStatusEnum : values){

if (teamStatusEnum.value == value){

return teamStatusEnum;

}

}

return null;

}

TeamStatusEnum(int value, String text) {

this.value = value;

this.text = text;

}

public int getValue() {

return value;

}

public void setValue(int value) {

this.value = value;

}

public String getText() {

return text;

}

public void setText(String text) {

this.text = text;

}

}

**正式开始细化接口**

service

public interface TeamService extends IService<Team> {

/\*\*

\* 创建队伍

\* @param team

\* @param loginUser

\* @return

\*/

long addTeam(Team team, User loginUser);

}

Impl

@Service

public class TeamServiceImpl extends ServiceImpl<TeamMapper, Team> implements TeamService {

@Resource

private UserTeamService userTeamService;

@Override

@Transactional(rollbackFor = Exception.class)

public long addTeam(Team team, User loginUser) {

// 1. 请求参数是否为空？

if (team == null) {

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

// 2. 是否登录，未登录不允许创建

if (loginUser == null) {

throw new BusinessException(ErrorCode.NOT\_LOGIN);

}

final long userId = loginUser.getId();

// 3. 校验信息

// 1. 队伍人数 > 1 且 <= 20

int maxNum = Optional.ofNullable(team.getMaxNum()).orElse(0);

if (maxNum < 1 || maxNum > 20) {

throw new BusinessException(ErrorCode.PARAMS\_ERROR, "队伍人数不满足要求");

}

// 2. 队伍标题 <= 20

String name = team.getName();

if (StringUtils.isBlank(name) || name.length() > 20) {

throw new BusinessException(ErrorCode.PARAMS\_ERROR, "队伍标题不满足要求");

}

// 3. 描述 <= 512

String description = team.getDescription();

if (StringUtils.isNotBlank(description) && description.length() > 512) {

throw new BusinessException(ErrorCode.PARAMS\_ERROR, "队伍描述过长");

}

// 4. status 是否公开（int）不传默认为 0（公开）

int status = Optional.ofNullable(team.getStatus()).orElse(0);

TeamStatusEnum statusEnum = TeamStatusEnum.getEnumByValue(status);

if (statusEnum == null) {

throw new BusinessException(ErrorCode.PARAMS\_ERROR, "队伍状态不满足要求");

}

// 5. 如果 status 是加密状态，一定要有密码，且密码 <= 32

String password = team.getPassword();

if (TeamStatusEnum.SECRET.equals(statusEnum)) {

if (StringUtils.isBlank(password) || password.length() > 32) {

throw new BusinessException(ErrorCode.PARAMS\_ERROR, "密码设置不正确");

}

}

// 6. 超时时间 > 当前时间

Date expireTime = team.getExpireTime();

if (new Date().after(expireTime)) {

throw new BusinessException(ErrorCode.PARAMS\_ERROR, "超时时间 > 当前时间");

}

// 7. 校验用户最多创建 5 个队伍

// todo 有 bug，可能同时创建 100 个队伍

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

queryWrapper.eq("userId", userId);

long hasTeamNum = this.count(queryWrapper);

if (hasTeamNum >= 5) {

throw new BusinessException(ErrorCode.PARAMS\_ERROR, "用户最多创建 5 个队伍");

}

// 8. 插入队伍信息到队伍表

team.setId(null);

team.setUserId(userId);

boolean result = this.save(team);

Long teamId = team.getId();

if (!result || teamId == null) {

throw new BusinessException(ErrorCode.PARAMS\_ERROR, "创建队伍失败");

}

// 9. 插入用户 => 队伍关系到关系表

UserTeam userTeam = new UserTeam();

userTeam.setUserId(userId);

userTeam.setTeamId(teamId);

userTeam.setJoinTime(new Date());

result = userTeamService.save(userTeam);

if (!result) {

throw new BusinessException(ErrorCode.PARAMS\_ERROR, "创建队伍失败");

}

return teamId;

}

}

**优化完成控制层**

这里因为完善了业务层，所以在controller 层 可以简化一下代码，我们需要新建一个队伍添加请求封装类，（便于前端知道该输入 哪些参数） 新得请求封装类位于model 包里的request 包

@Data

public class TeamAddRequest implements Serializable {

private static final long serialVersionUID =12312312451231L;

/\*\*

\* 队伍名称

\*/

private String name;

/\*\*

\* 描述

\*/

private String description;

/\*\*

\* 最大人数

\*/

private Integer maxNum;

/\*\*

\* 过期时间

\*/

@JsonFormat(locale="zh", timezone="GMT+8", pattern="yyyy-MM-dd HH:mm:ss")

private Date expireTime;

/\*\*

\* 用户id

\*/

private Long userId;

/\*\*

\* 0 - 公开，1 - 私有，2 - 加密

\*/

private Integer status;

/\*\*

\* 密码

\*/

private String password;

}

**修改addTeam接口如下：**

@PostMapping("/add")

public BaseResponse<Long> addTeam(@RequestBody TeamAddRequest teamAddRequest, HttpServletRequest request) {

if (teamAddRequest == null) {

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

User loginUser = userService.getLoginUser(request);

Team team = new Team();

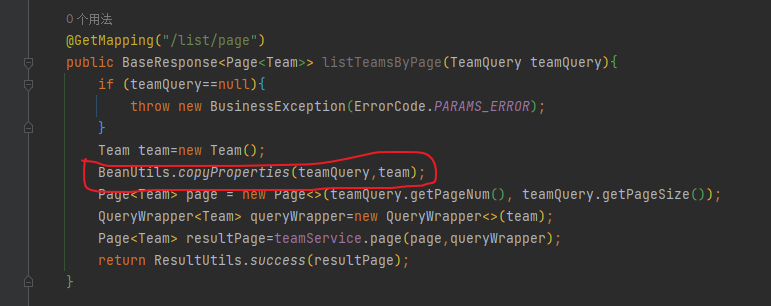
BeanUtils.copyProperties(teamAddRequest,team);

long teamId = teamService.addTeam(team, loginUser);

return ResultUtils.success(teamId);

}

**注意 ：BeanUtils.copyProperties(teamAddRequest,team); 两个参数的位置原来写错了，现在需要调换**



**测试**

**bug**

[关于日期格式你必须知道的坑( yyyy-MM-dd HH:mm:ss SSS)\_yyyy-mm-dd hh:mm:ss.sss\_37358143的博客-CSDN博客](https://blog.csdn.net/qq_37358143/article/details/103868657)

**直播10**

续上直播9

**查询队伍列表**

分页展示队伍列表破，根据名称，最大人数 ，搜索队伍 ，信息流中 不展示已过期的队伍

从请求参数中取出队伍名称，如果存在则作为查询条件

不展示已过期的队伍（根据 过期时间 筛选 ）

可以通过某个**关键字**同时对名称和描述查询

**只有管理员才能查看加密还有非公开的房间**

关联查询已加入队伍的创建人信息

关联查询已加入队伍的用户信息（可能会很耗费性能，建议大家自己写SQL 的方式实现）

**实现方式**

1）自己写 SQL

// 1. 自己写 SQL

// 查询队伍和创建人的信息

// select \* from team t left join user u on t.userId = u.id

// 查询队伍和已加入队伍成员的信息

// select \*

// from team t

//         left join user\_team ut on t.id = ut.teamId

//         left join user u on ut.userId = u.id;

**修改队伍信息**

判断请求参数是否为空

查询队伍是否存在

只有管理员或者队伍创建人可以修改

如果用户传入的新值和老值一致，就不用update了（可自行实现，降低数据库使用次数）

**如果队伍状态改为加密，必须要有密码**

更新成功

**用户可以加入队伍**

其他人，未满，未过期，允许加入多个队伍，但是要有个上限

用户最多加入5个队伍

队伍必须存在，只能加入未满，未过期的队伍

不能重复加入已加入的队伍（幂等性）

禁止加入私有队伍

如果加入的队伍是加密，必须密码匹配

新增队伍， 用户关联信息

**注意，一定要加上事务注解**

注意，并发请求时可能出现问题

**完善查询，更新，加入队伍接口**

**为了保护数据不被暴露，所以我们要新建封装类**

在model包新建vo包，并创建TeamUserVO（队伍和用户信息封装类），UserVO类（用户封装类），这两个类是返回给前端看

**TeamUserVo**

/\*\*

\* 队伍和用户信息封装类（脱敏）

\*/

@Data

public class TeamUserVo implements Serializable {

private static final long serialVersionUID = 2606242941233427192L;

/\*\*

\* id

\*/

@TableId(type = IdType.AUTO)

private Long id;

/\*\*

\* 队伍名称

\*/

private String name;

/\*\*

\* 描述

\*/

private String description;

/\*\*

\* 最大人数

\*/

private Integer maxNum;

/\*\*

\* 过期时间

\*/

private Date expireTime;

/\*\*

\* 用户id

\*/

private Long userId;

/\*\*

\* 0 - 公开，1 - 私有，2 - 加密

\*/

private Integer status;

/\*\*

\* 创建时间

\*/

private Date createTime;

/\*\*

\*更新时间

\*/

private Date updateTime;

/\*\*

\* 创建人用户列表

\*/

UserVo createUser;

}

**UserVo**

/\*\*

用户包装类（脱敏）

\* @author 13436

\*/

@Data

public class UserVo implements Serializable {

private static final long serialVersionUID = 1L;

/\*\*

\* id

\*/

@TableId(type = IdType.AUTO)

private Long id;

/\*\*

\* 用户昵称

\*/

private String username;

/\*\*

\* 登录账号

\*/

private String userAccount;

/\*\*

\* 用户头像

\*/

private String avatarUrl;

/\*\*

\* 性别

\*/

private Integer gender;

/\*\*

\* 电话

\*/

private String phone;

/\*\*

\* 邮箱

\*/

private String email;

/\*\*

\* 用户状态 0 正常

\*/

private Integer userStatus;

/\*\*

\* 创建时间

\*/

private Date createTime;

/\*\*

\* 更新时间

\*/

private Date updateTime;

/\*\*

\* 标签列表json

\*/

private String tags;

/\*\*

\* 是否删除

\*/

@TableLogic

private Integer isDelete;

/\*\*

\* 用户角色 0 - 普通用户 1 - 管理员

\*/

private Integer userRole;

/\*\*

\* 星球编号

\*/

private String planetCode;

}

**查询队伍**

在编写的过程中发现TeamQuery中缺少一个字段，现在补上

/\*\*

\* 搜索关键字（同时对队伍名称和描述搜索）

\*/

private String searchText;

**Controller**

@GetMapping("/list")

public BaseResponse<List<TeamUserVo>> listTeams(TeamQuery teamQuery,HttpServletRequest request){

if (teamQuery==null){

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

boolean isAdmin = userService.isAdmin(request);

List<TeamUserVo> teamList = teamService.listTeams(teamQuery,isAdmin);

return ResultUtils.success(teamList);

}

**Service**

/\*\*

\* 搜索队伍

\* @param teamQuery

\* @param isAdmin

\* @return

\*/

List<TeamUserVo> listTeams(TeamQuery teamQuery,boolean isAdmin);

**Impl**

@Override

public List<TeamUserVo> listTeams(TeamQuery teamQuery,boolean isAdmin) {

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

//组合查询条件

if (teamQuery!=null){

//根据id查

Long id=teamQuery.getId();

if (id!=null&& id >0){

queryWrapper.eq("id",id);

}

//可以通过某个关键字同时对名称和描述查询

String searchText=teamQuery.getSearchText();

if (StringUtils.isNotBlank(searchText)){

queryWrapper.and(qw->qw.like("name",searchText).or().like("description",searchText));

}

//从请求参数中取出队伍名称，如果存在则作为查询条件

String name=teamQuery.getName();

if (StringUtils.isNotBlank(name)){

queryWrapper.like("name",name);

}

//根据 “描述” 的类似信息 查询

String description = teamQuery.getDescription();

if (StringUtils.isNotBlank(description)){

queryWrapper.like("description",description);

}

Integer maxNum=teamQuery.getMaxNum();

//查询最大人数相等的

if (maxNum != null && maxNum > 0){

queryWrapper.eq("maxNum",maxNum);

}

Long userId=teamQuery.getUserId();

//根据创建人来查询

if (userId != null && userId > 0){

queryWrapper.eq("userId",userId);

}

//根据状态来查询

Integer status = teamQuery.getStatus();

TeamStatusEnum statusEnum=TeamStatusEnum.getEnumByValue(status);

if (statusEnum == null){

statusEnum=TeamStatusEnum.PUBLIC;

}

if (!isAdmin && ! statusEnum.equals(TeamStatusEnum.PUBLIC)){

throw new BusinessException(ErrorCode.NO\_AUTH);

}

queryWrapper.eq("status",statusEnum.getValue());

}

//不展示已过期的队伍

// expireTime is null or expireTime > now()

queryWrapper.and(qw->qw.gt("expireTime",new Date()).or().isNull("expireTime"));

List<Team> teamList=this.list(queryWrapper);

if (CollectionUtils.isEmpty(teamList)){

return new ArrayList<>();

}

List<TeamUserVo> teamUserVoList=new ArrayList<>();

//关联查询已加入队伍的创建人信息

for (Team team: teamList){

Long userId=team.getUserId();

if (userId==null){

continue;

}

User user=userService.getById(userId);

TeamUserVo teamUserVo=new TeamUserVo();

BeanUtils.copyProperties(team,teamUserVo);

// 脱敏用户信息

if (user != null){

UserVo userVo=new UserVo();

BeanUtils.copyProperties(user,userVo);

teamUserVo.setCreateUser(userVo);

}

teamUserVoList.add(teamUserVo);

}

return teamUserVoList;

}

**修改队伍信息**

**用户登录请求体**

@Data

public class TeamUpdateRequest implements Serializable {

private static final long serialVersionUID = 3191241716373120793L;

/\*\*

\* id

\*/

private Long id;

/\*\*

\* 队伍名称

\*/

private String name;

/\*\*

\* 描述

\*/

private String description;

/\*\*

\* 过期时间

\*/

@JsonFormat(locale="zh", timezone="GMT+8", pattern="yyyy-MM-dd HH:mm:ss")

private Date expireTime;

/\*\*

\* 0 - 公开，1 - 私有，2 - 加密

\*/

private Integer status;

/\*\*

\* 密码

\*/

private String password;

}

**Controller**

@PostMapping("/update")

public BaseResponse<Boolean> updateTeam(@RequestBody TeamUpdateRequest teamUpdateRequest,HttpServletRequest request){

if (teamUpdateRequest==null){

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

User loginUser = userService.getLoginUser(request);

boolean save = teamService.updateTeam(teamUpdateRequest,loginUser);

if(!save){

throw new BusinessException(ErrorCode.SYSTEM\_ERROR,"更新失败");

}

return ResultUtils.success(true);

}

**Service**

/\*\*

\* 更新队伍

\* @param teamUpdateRequest

\* @param loginUser

\* @return

\*/

boolean updateTeam(TeamUpdateRequest teamUpdateRequest,User loginUser);

**Impl**

@Override

public boolean updateTeam(TeamUpdateRequest teamUpdateRequest,User loginUser) {

// 1.判断请求参数是否为空

if (teamUpdateRequest == null){

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

Long id = teamUpdateRequest.getId();

if (id==null&& id<=0){

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

// 2.查询队伍是否存在

Team oldTeam=this.getById(id);

if (oldTeam ==null){

throw new BusinessException(ErrorCode.NULL\_ERROR,"请求队伍为空");

}

// 3.只有管理员或者队伍创建人可以修改

if(!(oldTeam.getUserId().equals(loginUser.getId())) && !userService.isAdmin(loginUser)){

throw new BusinessException(ErrorCode.NO\_AUTH);

}

// 4.如果队伍状态改为加密，必须要有密码

TeamStatusEnum statusEnum=TeamStatusEnum.getEnumByValue(teamUpdateRequest.getStatus());

if(statusEnum.equals(TeamStatusEnum.SECRET)){

if(StringUtils.isBlank(teamUpdateRequest.getPassword())){

throw new BusinessException(ErrorCode.PARAMS\_ERROR, "加密房间必须要设置密码");

}

}

Team updateTeam=new Team();

BeanUtils.copyProperties(teamUpdateRequest,updateTeam);

// 5.如果用户传入的新值和老值一致，就不用update了（可自行实现，降低数据库使用次数）

// 6.更新成功

return this.updateById(updateTeam);

}

**用户加入队伍**

**加入队伍请求体**

/\*\*

\* 队伍

\* @TableName team

\*/

@Data

public class TeamJoinRequest implements Serializable {

/\*\*

\* id

\*/

private Long teamId;

/\*\*

\* 密码

\*/

private String password;

private static final long serialVersionUID = 1L;

}

**Controller**

@PostMapping("/join")

public BaseResponse<Boolean> joinTeam(@RequestBody TeamJoinRequest teamJoinRequest,HttpServletRequest request){

if (teamJoinRequest==null){

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

User loginUser = userService.getLoginUser(request);

boolean result=teamService.joinTeam(teamJoinRequest,loginUser);

return ResultUtils.success(result);

}

**Service**

/\*\*

\* 加u人队伍

\* @param teamJoinRequest

\* @return

\*/

boolean joinTeam(TeamJoinRequest teamJoinRequest,User loginUser);

**Impl**

@Override

public boolean joinTeam(TeamJoinRequest teamJoinRequest,User loginUser) {

if (teamJoinRequest==null){

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

// 队伍是否存在

Long teamId=teamJoinRequest.getTeamId();

if (teamId==null || teamId <= 0){

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

Team team =this.getById(teamId);

if(team==null){

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"队伍不存在");

}

//只能加入未过期的队伍

Date expireTime= team.getExpireTime();

if (expireTime != null && expireTime.before(new Date())){

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"队伍已过期");

}

//禁止加入私有队伍

Integer status= team.getStatus();

TeamStatusEnum teamStatusEnum = TeamStatusEnum.getEnumByValue(status);

if (TeamStatusEnum.PRIVATE.equals(teamStatusEnum)){

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"禁止加入私有队伍");

}

//加入队伍是加密，必须密码匹配

String password=teamJoinRequest.getPassword();

if (TeamStatusEnum.SECRET.equals(teamStatusEnum)){

if(StringUtils.isBlank(password) || !password.equals(team.getPassword())){

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"密码错误");

}

}

//该用户已加入的队伍数量

long userId= loginUser.getId();

QueryWrapper<UserTeam> userTeamQueryWrapper=new QueryWrapper<>();

userTeamQueryWrapper.eq("userId",userId);

long hasJoinNum = userTeamService.count(userTeamQueryWrapper);

// 用户最多加入5个队伍

if (hasJoinNum > 5){

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"最多创建和加入5 个 队伍");

}

//不能重复加入已加入的队伍

userTeamQueryWrapper =new QueryWrapper<>();

userTeamQueryWrapper.eq("userId",userId);

userTeamQueryWrapper.eq("teamId",teamId);

long teamHasJoinNumTeam=userTeamService.count(userTeamQueryWrapper);

if (teamHasJoinNumTeam>0){

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"用户已加入该队伍");

}

//已加入队伍的人数

userTeamQueryWrapper =new QueryWrapper<>();

userTeamQueryWrapper.eq("teamId",teamId);

long teamHasJoinNum=userTeamService.count(userTeamQueryWrapper);

if (teamHasJoinNum >= team.getMaxNum()){

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"队伍已满");

}

//更新队伍信息

UserTeam userTeam=new UserTeam();

userTeam.setUserId(userId);

userTeam.setTeamId(teamId);

userTeam.setJoinTime(new Date());

return userTeamService.save(userTeam);

}

**直播11**

**用户可以退出队伍**

请求参数： 队伍id

校验请求参数

校验队伍是否存在

校验我是否已加入队伍

如果队伍

只剩一人，队伍解散

还有其他人

如果是队长退出队伍，权限转移给第二早加入的用户 -- 先来后到

只用取id最小的2条数据

非队长，自己退出队伍

**队伍可以解散队伍**

请求参数： 队伍id

业务流程：

校验请求参数

校验队伍是否存在

校验你是不是队伍的队长

移除所有加入队伍的关联信息

删除队伍

**获取当前用户已加入的队伍**

**获取当前用户创建的队伍**

复用 listTeam 方法，只新增查询条件，不做修改（开闭原则）

**分享队伍**

邀请其他用户加入队伍

业务流程：

生成分享链接 （ 二维码 ）

用户访问链接，可以点击加入

**代码实现**

**退出队伍**

**新建退出请求体**

/\*\*

\* 用户退出队伍请求体

\*

\* @author 13436

\*/

@Data

public class TeamQuitRequest implements Serializable {

/\*\*

\* id

\*/

private Long teamId;

private static final long serialVersionUID = 1L;

}

**Controller**

@PostMapping("/quit")

public BaseResponse<Boolean> quitTeam(@RequestBody TeamQuitRequest teamQuitRequest,HttpServletRequest request){

if (teamQuitRequest==null){

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

User loginUser=userService.getLoginUser(request);

boolean result=teamService.quitTeam(teamQuitRequest,loginUser);

return ResultUtils.success(result);

}

**Service**

/\*\*

\* 退出队伍

\* @param teamQuitRequest

\* @param loginUser

\* @return

\*/

boolean quitTeam(TeamQuitRequest teamQuitRequest, User loginUser);

**Impl**

@Override

@Transactional(rollbackFor = Exception.class)

public boolean quitTeam(TeamQuitRequest teamQuitRequest, User loginUser) {

//校验请求参数

if (teamQuitRequest== null){

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

Long teamId= teamQuitRequest.getTeamId();

Team team=getTeamById(teamId);

// 校验我是否已加入队伍

long userId =loginUser.getId();

UserTeam queryUserTeam =new UserTeam();

queryUserTeam.setTeamId(teamId);

queryUserTeam.setUserId(userId);

QueryWrapper<UserTeam> queryWrapper=new QueryWrapper<>(queryUserTeam);

long count =userTeamService.count(queryWrapper);

if(count==0){

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"未加入队伍");

}

long teamHasJoinNum=this.countTeamUserByTeamId(teamId);

//队伍只剩一人，队伍解散

if (teamHasJoinNum==1){

//删除队伍

this.removeById(teamId);

}else {

//还有其他人

//是否为队长

//队伍还剩至少两个

if (team.getUserId() == userId){

//把队伍转移给最早加入的用户

//1. 查询已加入队伍的所有用户和加入时间

QueryWrapper<UserTeam> userTeamQueryWrapper=new QueryWrapper<>();

userTeamQueryWrapper.eq("teamId",teamId);

userTeamQueryWrapper.last("order by id asc limit 2");

List<UserTeam> userTeamList = userTeamService.list(userTeamQueryWrapper);

if (CollectionUtils.isEmpty(userTeamList) || userTeamList.size() <= 1){

throw new BusinessException(ErrorCode.SYSTEM\_ERROR);

}

UserTeam nextUserTeam=userTeamList.get(1);

Long nextTeamLeaderId=nextUserTeam.getUserId();

//更新当前队伍的队长

Team updateTeam = new Team();

updateTeam.setId(teamId);

updateTeam.setUserId(nextTeamLeaderId);

boolean result=this.updateById(updateTeam);

if (!result){

throw new BusinessException(ErrorCode.SYSTEM\_ERROR,"更新队伍队长失败");

}

}

}

//移除关系

return userTeamService.remove(queryWrapper);

}

**这里我们由于多次需要获得队伍当前人数，所以封装了countTeamUserByTeamId方法**

/\*\*

\* 获取队伍当前人数

\* @param teamId

\* @return

\*/

private long countTeamUserByTeamId(long teamId){

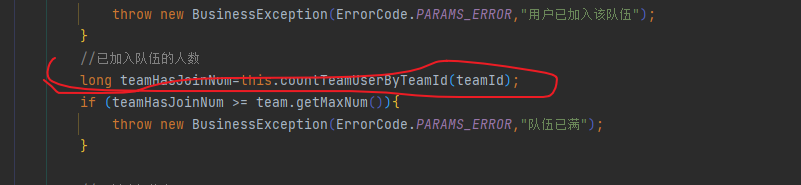
QueryWrapper<UserTeam> userTeamQueryWrapper=new QueryWrapper<>();

userTeamQueryWrapper.eq("teamId",teamId);

return userTeamService.count(userTeamQueryWrapper);

}

同时在joinTeam方法里修改代码



**解散队伍**

**Controller**

@PostMapping("/delete")

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

if (id<=0){

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

User loginUser= userService.getLoginUser(request);

boolean save = teamService.deleteTeam(id,loginUser);

if(!save){

throw new BusinessException(ErrorCode.SYSTEM\_ERROR,"删除失败");

}

return ResultUtils.success(true);

}

**Service**

/\*\*

\* 删除（解散）队伍

\* @param id

\* @return

\*/

boolean deleteTeam(long id,User loginUser);

**Imp**

@Override

@Transactional(rollbackFor = Exception.class)

public boolean deleteTeam(long id,User loginUser) {

// 校验队伍是否存在

Team team=getTeamById(id);

long teamId=team.getId();

// 校验你是不是队伍的队长

if (!team.getUserId() .equals(loginUser.getId()) ){

throw new BusinessException(ErrorCode.NO\_AUTH,"无访问权限");

}

// 移除所有加入队伍的关联信息

QueryWrapper<UserTeam> userTeamQueryWrapper=new QueryWrapper<>();

userTeamQueryWrapper.eq("teamId",teamId);

boolean result=userTeamService.remove(userTeamQueryWrapper);

if (!result){

throw new BusinessException(ErrorCode.SYSTEM\_ERROR,"删除队伍关联信息失败");

}

// 删除队伍

return this.removeById(teamId);

}

**我们需要根据id获取队伍信息，这个代码我们重复的写，所以提取出来**

/\*\*

\* 根据id 获取队伍信息

\* @param teamId

\* @return

\*/

private Team getTeamById(Long teamId) {

if (teamId==null || teamId <= 0){

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

Team team =this.getById(teamId);

if(team==null){

throw new BusinessException(ErrorCode.PARAMS\_ERROR,"队伍不存在");

}

return team;

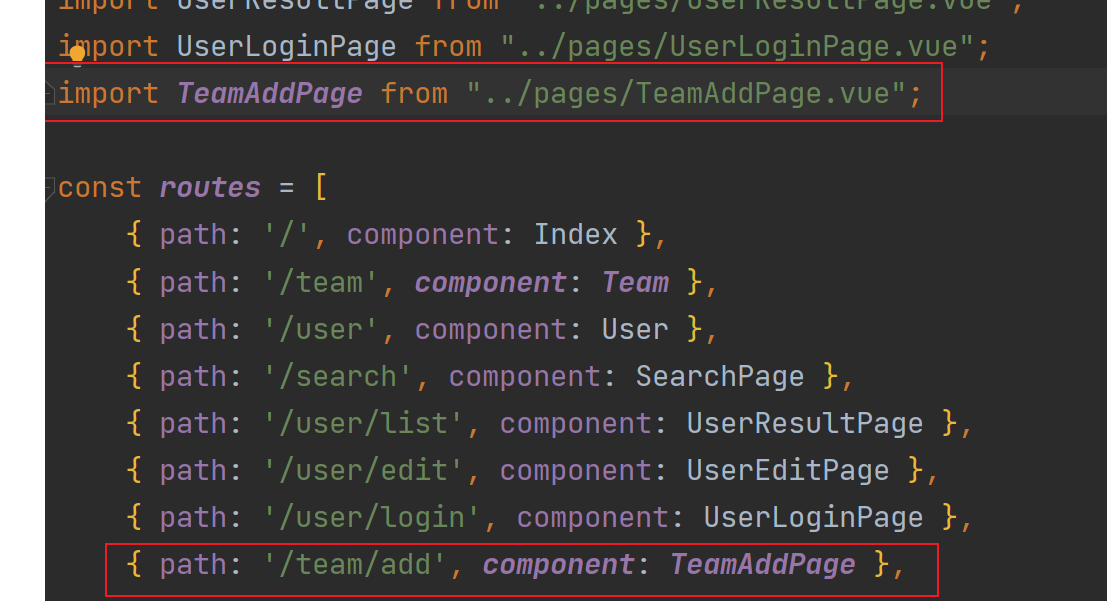
}

**前端设计**

**TeamAddPage**

**准备**

添加路由



在TeamPage 里写一个添砖到TeamAddPage



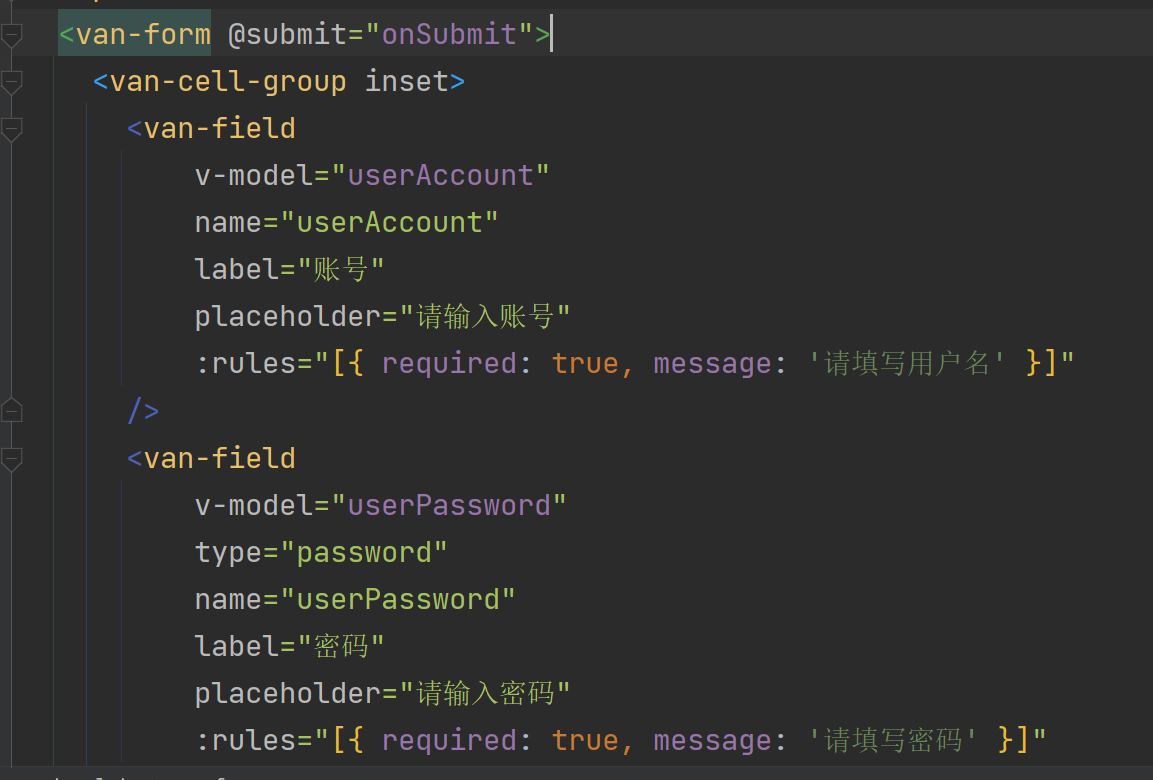
**设计**

主要是在vant组件库里选择合适的组件粘贴

**队伍名和描述名**

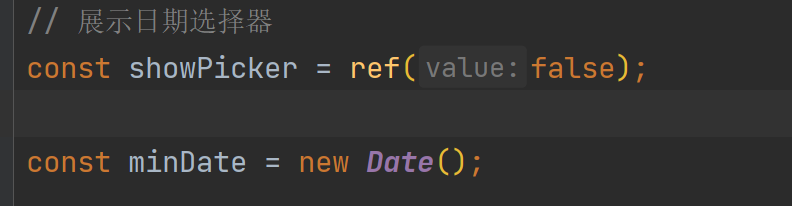
我们可以发现队伍名和描述名类似于用户登录页面的表单组件，所以拿来即用（修改下参数）这个主要是运用了表单，单元格，输入框这三个组件，其中描述使用了高度自适应**参数我们可以从后台获得（knife4j接口文档）**

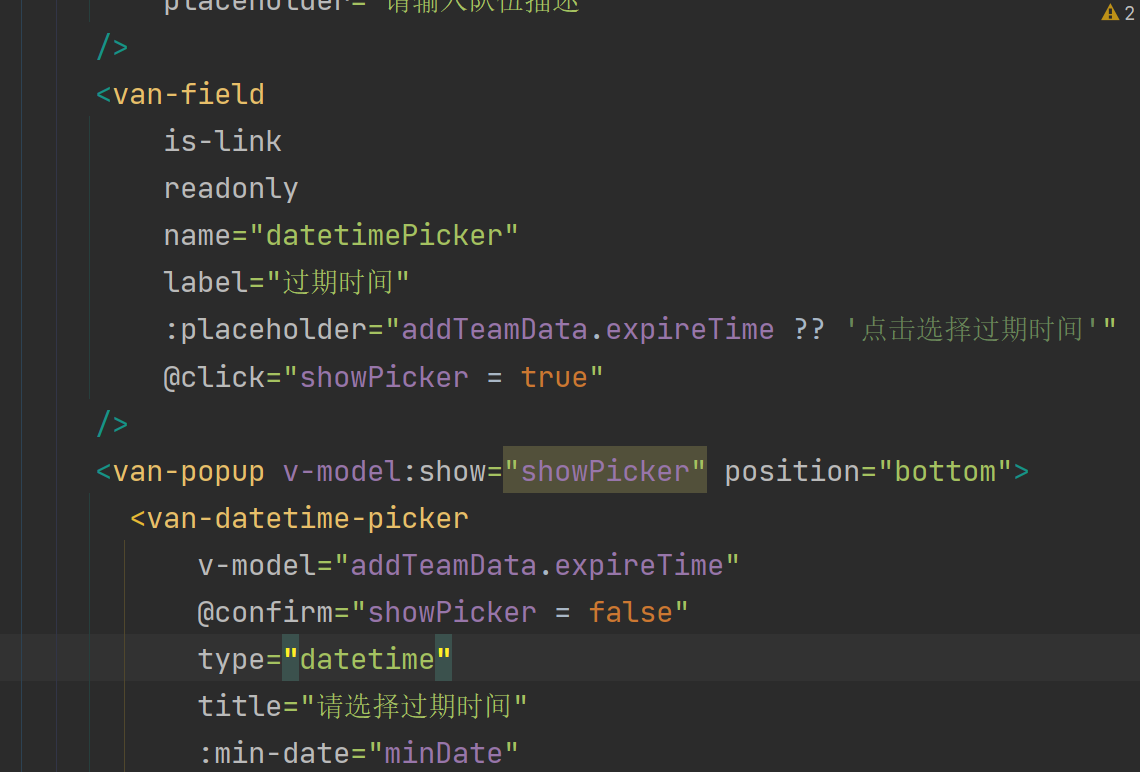




**过期时间**

我们选择vant里的DatetimePicker选择完整时间这里的min-date 我们不能直接new Date(),因为这会导致页面一直渲染，从而页面加载不出来，我能得在建一个常量min-date，同时这个日期默认不显示，我们要在JS里展示日期选择器





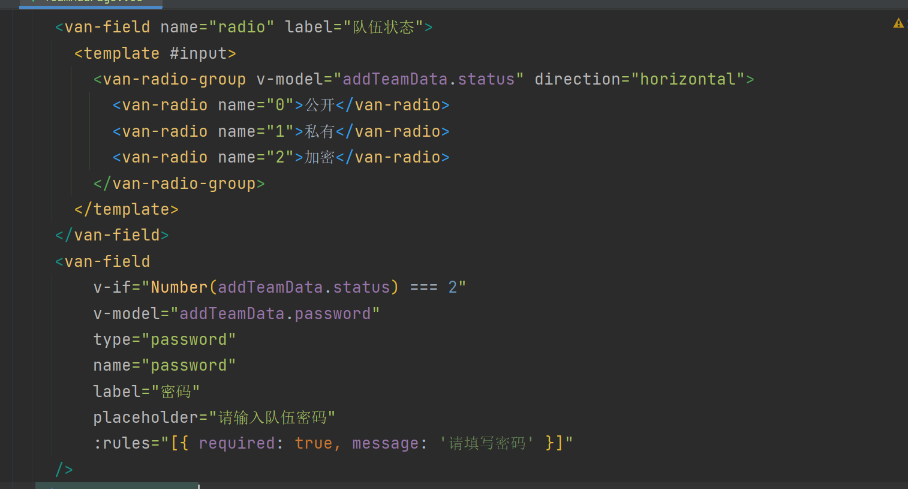
**最大人数**

这里我们选择Stepper不进器里的限制输入范围



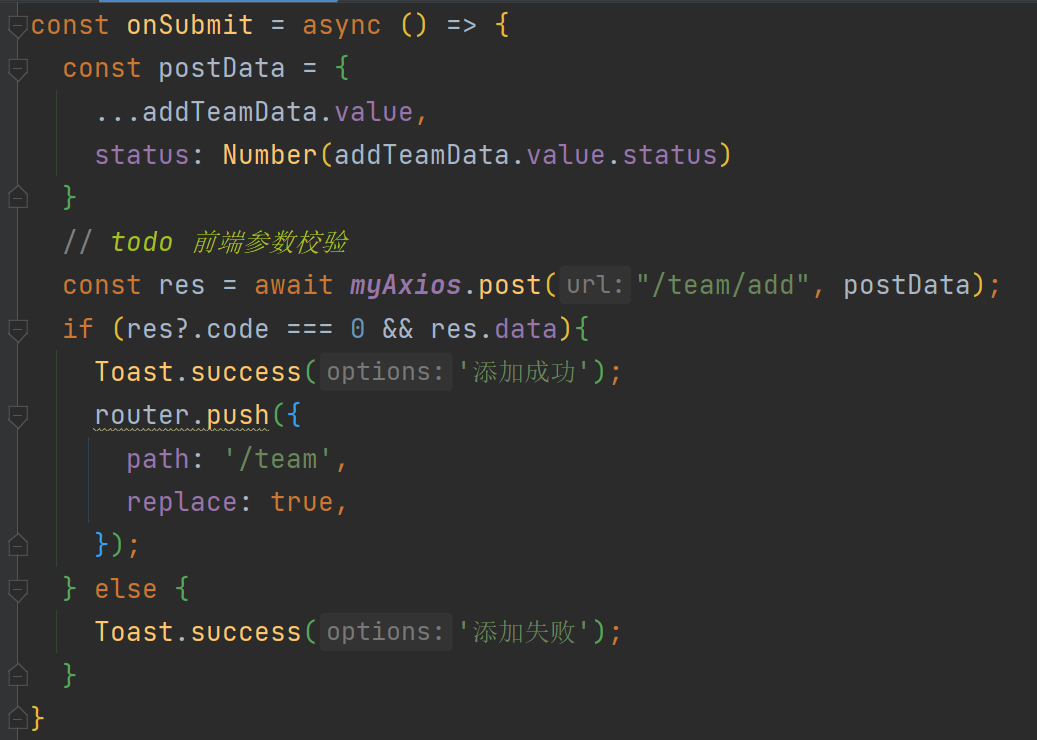
**队伍状态（当只有选择加密队伍时，才会跳出密码框）**

这里我们选择表单类型里的单选框，和field输入框。**注意一定要在判断状态时，把类型转为Number，因为通过打印可得，状态是字符串类型的。**



**提交按钮**

native-type="submit"属性， 点击自动获取van-field name中的值组成的对象。关键是提交所传的的状态也要转换成Number，同时创建成功后跳转到队伍页面AddTeamPage页面完整代码如下：





**完整代码**

<template>

<div id="teamAddPage">

<van-form @submit="onSubmit">

<van-cell-group inset>

<van-field

v-model="addTeamData.name"

name="name"

label="队伍名"

placeholder="请输入队伍名"

:rules="[{ required: true, message: '请输入队伍名' }]"

/>

<van-field

v-model="addTeamData.description"

rows="4"

autosize

label="队伍描述"

type="textarea"

placeholder="请输入队伍描述"

/>

<van-field

is-link

readonly

name="datetimePicker"

label="过期时间"

:placeholder="addTeamData.expireTime ?? '点击选择过期时间'"

@click="showPicker = true"

/>

<van-popup v-model:show="showPicker" position="bottom">

<van-datetime-picker

v-model="addTeamData.expireTime"

@confirm="showPicker = false"

type="datetime"

title="请选择过期时间"

:min-date="minDate"

/>

</van-popup>

<van-field name="stepper" label="最大人数">

<template #input>

<van-stepper v-model="addTeamData.maxNum" max="10" min="3"/>

</template>

</van-field>

<van-field name="radio" label="队伍状态">

<template #input>

<van-radio-group v-model="addTeamData.status" direction="horizontal">

<van-radio name="0">公开</van-radio>

<van-radio name="1">私有</van-radio>

<van-radio name="2">加密</van-radio>

</van-radio-group>

</template>

</van-field>

<van-field

v-if="Number(addTeamData.status) === 2"

v-model="addTeamData.password"

type="password"

name="password"

label="密码"

placeholder="请输入队伍密码"

:rules="[{ required: true, message: '请填写密码' }]"

/>

</van-cell-group>

<div style="margin: 16px;">

<van-button round block type="primary" native-type="submit">

提交

</van-button>

</div>

</van-form>

</div>

</template>

<script setup lang="ts">

import {useRouter} from "vue-router";

import {ref} from "vue";

import myAxios from "../plugins/myAxios";

import {Toast} from "vant";

const router = useRouter();

// 展示日期选择器

const showPicker = ref(false);

const minDate = new Date();

const initFormData = {

"name": "",

"description": "",

"expireTime": null,

"maxNum": 3,

"password": "",

"status": 0,

}

// 需要用户填写的表单数据

const addTeamData = ref({...initFormData})

// 提交

const onSubmit = async () => {

const postData = {

...addTeamData.value,

status: Number(addTeamData.value.status)

}

// todo 前端参数校验

const res = await myAxios.post("/team/add", postData);

if (res?.code === 0 && res.data){

Toast.success('添加成功');

router.push({

path: '/team',

replace: true,

});

} else {

Toast.success('添加失败');

}

}

</script>

<style scoped>

#teamPage {

}

</style>

**设计队伍列表**

**定义队伍类型**

import {UserType} from "./user";

/\*\*

\* 用户类别

\*/

export type TeamType = {

id: number;

name: string;

description: string;

expireTime?:Date;

maxNum:number;

password:string;

// todo 定义枚举值更规范

status: number;

createTime: Date;

updateTime: Date;

createUser?: UserType;

};

**创建队伍卡片列表组件**

<template>

<van-card

v-for="user in props.teamList"

:desc="user.profile"

:title="`${user.username} (${user.planetCode})`"

:thumb="user.avatarUrl"

>

<template #tags>

<van-tag plain type="danger" v-for="tag in user.tags" style="margin-right: 8px; margin-top: 8px" >

{{ tag }}

</van-tag>

</template>

<template #footer>

<van-button size="mini">联系我</van-button>

</template>

</van-card>

</template>

<script setup lang="ts">

import {TeamType} from "../models/team";

interface TeamCardListProps{

teamList: TeamType[];

}

const props= withDefaults(defineProps<TeamCardListProps>(),{

//@ts-ignore

teamList: [] as TeamType[]

});

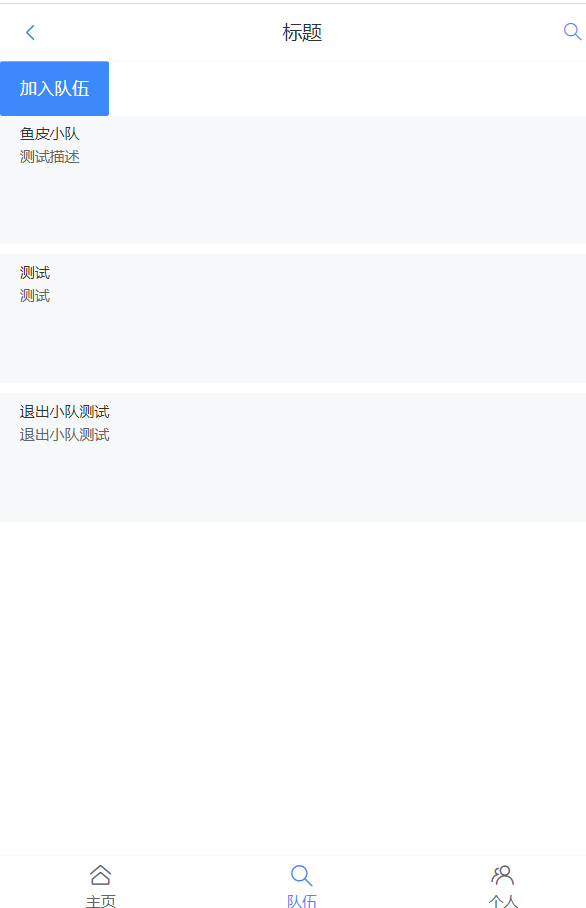
</script>

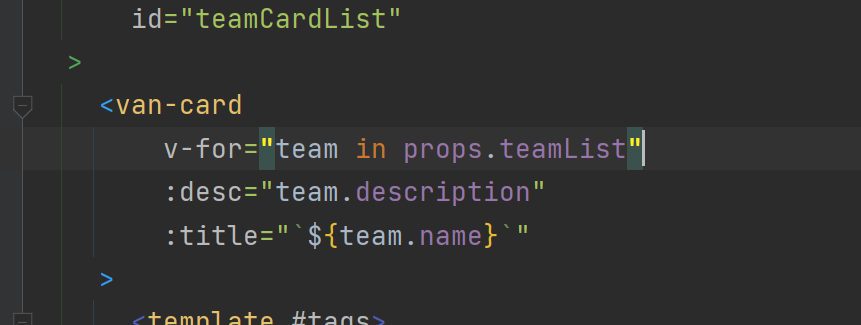
<style scoped>

</style>

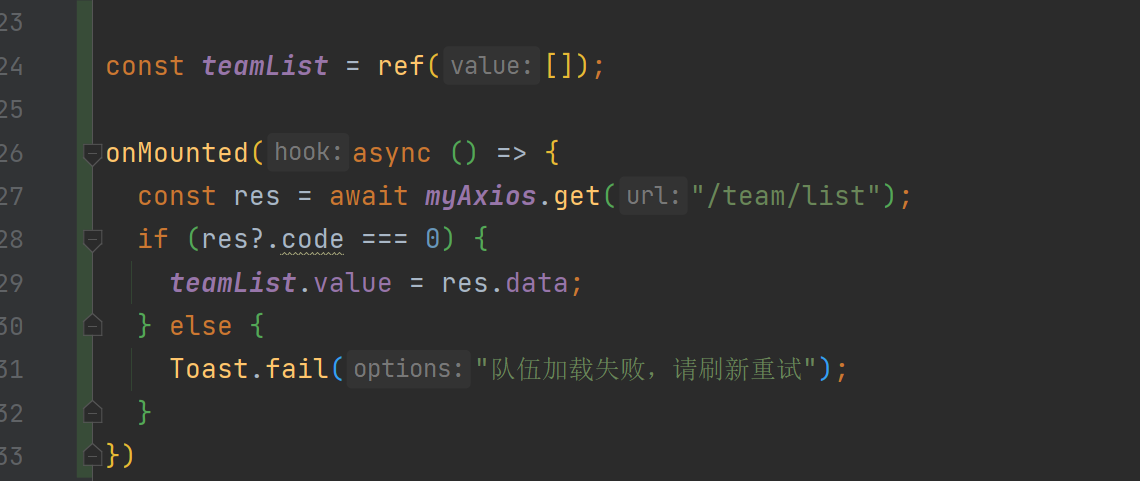
**将组件挂载在TeamPage**

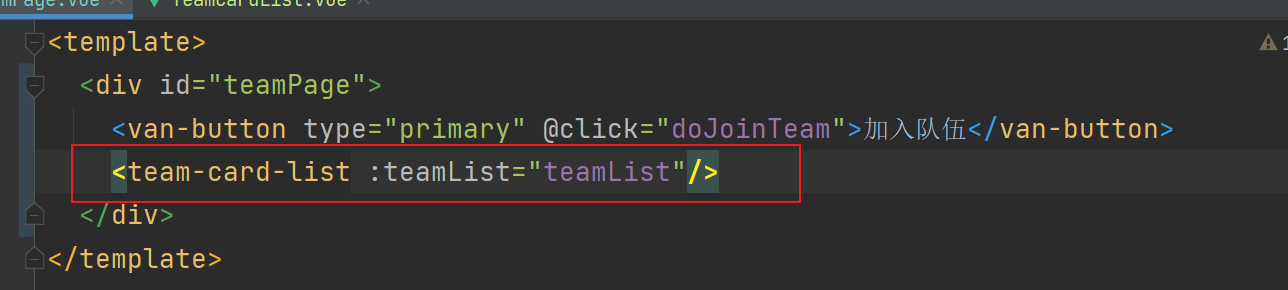
注意：引入team-card-list时，编译器可能不会帮你把引入的类型自动带上，需自己添加将team-card-list里的原来的用户参数换成队伍的，测试一下刷新页面，成功加载出组件（就是很丑，展示不齐全）





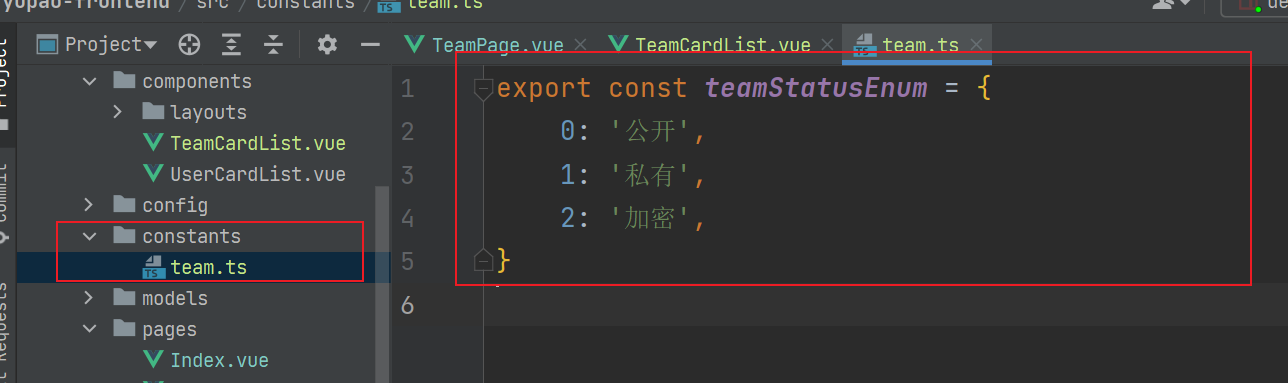






**完善teamcardlist组件**

添加队伍状态，最大人数等以及实现加入队伍功能我们下方要涉及到队伍的状态，我们先创建队伍状态常量 team.ts完整代码如下：



 <template>

   <div>

     <van-card

         v-for="team in props.teamList"

         :thumb="mouse"

         :desc="team.description"

         :title="`${team.name}`"

     >

       <template #tags>

         <van-tag plain type="danger" style="margin-right: 8px; margin-top: 8px">

          {{

            teamStatusEnum[team.status]

          }}

         </van-tag>

       </template>

       <template #bottom>

         <div>

           {{ '最大人数: ' + team.maxNum }}

         </div>

         <div v-if="team.expireTime">

           {{ '过期时间: ' + team.expireTime }}

         </div>

         <div>

           {{ '创建时间: ' + team.createTime }}

         </div>

       </template>

       <template #footer>

         <van-button size="small" type="primary"  plain @click="doJoinTeam(team.id)">加入队伍</van-button>

       </template>

     </van-card>

   </div>

 ​

 </template>

 ​

 <script setup lang="ts">

 import {TeamType} from "../models/team";

 import {teamStatusEnum} from "../constants/team";

 import mouse from '../assets/mouse.jpg';

 import myAxios from "../plugins/myAxios";

 import {Toast} from "vant";

 ​

 import {useRouter} from "vue-router";

 ​

 interface TeamCardListProps {

   teamList: TeamType[];

 }

 ​

 const props = withDefaults(defineProps<TeamCardListProps>(), {

   // @ts-ignore

   teamList: [] as TeamType[],

 });

 ​

 const router = useRouter();

 ​

 /\*\*

  \* 加入队伍

  \*/

 const doJoinTeam = async (id:number) => {

   const res = await myAxios.post('/team/join', {

     teamId: id,

  });

   if (res?.code === 0) {

     Toast.success('加入成功');

  } else {

     Toast.fail('加入失败' + (res.description ? `，${res.description}` : ''));

  }

 }

 ​

 </script>

 ​

 <style scoped>

 #teamCardList :deep(.van-image\_\_img) {

   height: 128px;

   object-fit: unset;

 }

 </style>

 ​

注意：1.这里thumb我们引入了图片（显示更美观），可以把自己心仪的图片放入assets里并引入 2. 加入队伍里面失败，写的形式是模板字符串，可自行了解 3. 引入样式，原来的图片过于宽，要指定高度关闭自适应，这里我们使用样式穿透，不然不起作用，**如果我们给部分组件引入的样式不起作用，都可以使用样式穿透！**

**测试**

刷新页面，页面显示如下，点击加入队伍，成功显示插入失败，已加入队伍（这里我是房主，所以加入失败，也可以登入其他用户测试）



**直播12**

**笔记**

开发前端功能

搜索队伍

更新队伍（仅创始人可见）

查看个人已加入队伍

查看个人已创建队伍

解散队伍

退出队伍

查看个人已加入队伍

随机匹配

完成之前遗留的问题 TODO

**前端不同页面怎么传递数据？**

url querystring（xxx?id=1）比较适用于页面跳转

url（/team/:id，xxx/1）

hash (/team#1)

localStorage

context（全局变量，同页面或整个项目要访问公共变量）

**随机匹配**

为了帮大家更快地发现和自己兴趣相同的朋友

匹配 1 个还是匹配多个？

答：匹配多个，并且按照匹配的相似度从高到低排序

怎么匹配？（根据什么匹配）

答：标签 tags还可以根据 user\_team 匹配加入相同队伍的用户

本质：找到有相似标签的用户

举例：

用户 A：[Java, 大一, 男]

用户 B：[Java, 大二, 男]

用户 C：[Python, 大二, 女]

用户 D：[Java, 大一, 女]

**1. 怎么匹配**

找到有共同标签最多的用户（TopN）

共同标签越多，分数越高，越排在前面

如果没有匹配的用户，随机推荐几个（降级方案）

编辑距离算法：<https://blog.csdn.net/DBC_121/article/details/104198838>

最小编辑距离：字符串 1 通过最少多少次增删改字符的操作可以变成字符串 2

余弦相似度算法：<https://blog.csdn.net/m0_55613022/article/details/125683937>

（如果需要带权重计算，比如学什么方向最重要，性别相对次要）

**2. 怎么对所有用户匹配，取 TOP**

直接取出所有用户，依次和当前用户计算分数，取 TOP N（54 秒）

优化方法：

切忌不要在数据量大的时候循环输出日志（取消掉日志后 20 秒）

Map 存了所有的分数信息，占用内存解决：(自己实现）

维护一个固定长度的有序集合（sortedSet），只保留分数最高的几个用户（时间换空间）

e.g.【3, 4, 5, 6, 7】取 TOP 5，id 为 1 的用户就不用放进去了

细节：剔除自己

尽量只查需要的数据：

过滤掉标签为空的用户

根据部分标签取用户（前提是能区分出来哪个标签比较重要）

只查需要的数据（比如 id 和 tags） 11s ---> 2s

提前查？（定时任务）

提前把所有用户给缓存（不适用于经常更新的数据）

提前运算出来结果，缓存（针对一些重点用户，提前缓存）

大数据推荐，比如说有几亿个商品，难道要查出来所有的商品？

难道要对所有的数据计算一遍相似度？

检索 => 召回 => 粗排 => 精排 => 重排序等等

检索：尽可能多地查符合要求的数据（比如按记录查）

召回：查询可能要用到的数据（不做运算）

粗排：粗略排序，简单地运算（运算相对轻量）

精排：精细排序，确定固定排位

**分表学习建议**

mycat、sharding sphere 框架

一致性 hash

**队伍操作权限控制**

加入队伍： 仅非队伍创建人可见，且未加入队伍的人可见

更新队伍： 仅创建人可见

解散队伍： 仅创建人可见

退出队伍： 创建人不可见，仅已加入队伍的人可见

**前端页面开发**

**搜索框**

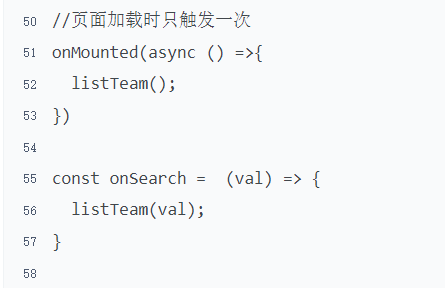
TeamPage



因为，一次性挂载本质性也是搜索队伍，所以把代码提取出来



挂载和搜索框修改为下图所示：



测试

代码：

<template>

<div id="teamPage">

<van-search v-model="searchText" placeholder="搜索队伍" @search="onSearch"/>

<van-button type="primary" @click="doJoinTeam">加入队伍</van-button>

<team-card-list :teamList="teamList"/>

<van-empty v-if="teamList?.length < 1" description="数据为空" />

</div>

</template>

<script setup>

import {useRouter} from "vue-router";

import TeamCardList from "../components/TeamCardList.vue";

import {onMounted, ref} from "vue";

import myAxios from "../plugins/myAxios.ts";

import {Toast} from "vant";

const router=useRouter();

const searchText = ref('');

// 跳转到加入队伍页

const doJoinTeam = () =>{

router.push({

path: "/team/add"

})

}

const teamList = ref([]);

/\*\*

\* 搜索队伍

\* @param val

\* @returns {Promise<void>}

\*/

const listTeam = async (val='') => {

const res= await myAxios.get("/team/list",{

params:{

searchText: val,

pageNum: 1,

}

});

if (res?.code === 0){

teamList.value = res.data;

}else {

Toast.fail('加载队伍失败，请刷新重试')

}

}

//页面加载时只触发一次

onMounted(async () =>{

listTeam();

})

const onSearch = (val) => {

listTeam(val);

}

</script>

<style scoped>

</style>

**更新页面**

分析：我们的更新页面和新建队伍页面类似，所以我们直接复制TeamAddPage,创建TeamUpdateTeam页面

**完善TeamCardList**

首先在队伍页面，创建一个按钮来跳转到更新页面，但是只有当前用户是队伍创建者才可以看到次按钮，我们可以直接写在TeamCardList组件里

按钮添加

<template #footer>

<van-button size="small" type="primary" plain @click="doJoinTeam(team.id)">加入队伍</van-button>

<van-button v-if="team.userId === currentUser?.id" size="small" plain

@click="doUpdateTeam(team.id)">更新队伍

</van-button>

由于需要判断当前用户是否为队伍创建者，我们要获取当前用户（调用以前写的方法）

const currentUser= ref();

onMounted( async ()=>{

currentUser.value= await getCurrentUser();

})

写跳转按钮的逻辑

/\*\*

\* 跳转至更新队伍页面

\* @param id

\*/

const doUpdateTeam = (id: number) =>{

router.push({

path: '/team/update',

query:{

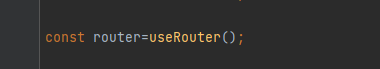
id,

}

})

}

PS：别忘了引入useRouter



完整代码如下：

<template>

<div id="teamCardList">

<van-card

v-for="team in props.teamList"

:thumb="IU"

:desc="team.description"

:title="`${team.name}`"

>

<template #tags>

<van-tag plain type="danger" style="margin-right: 8px; margin-top: 8px" >

{{

teamStatusEnum[team.status]

}}

</van-tag>

</template>

<template #bottom>

<div>

{{'最大人数：'+team.maxNum}}

</div>

<div v-if="team.expireTime">

{{'过期时间：'+team.expireTime}}

</div>

<div v-if="team.createTime">

{{'创建时间：'+team.createTime}}

</div>

</template>

<template #footer>

<van-button size="small" type="primary" plain @click="doJoinTeam(team.id)">加入队伍</van-button>

<van-button v-if="team.userId === currentUser?.id" size="small" plain

@click="doUpdateTeam(team.id)">更新队伍

</van-button>

<!-- todo 仅加入队伍可见-->

<van-button size="small" plain

@click="doQuitTeam(team.id)">退出队伍

</van-button>

<van-button v-if="team.userId === currentUser?.id" size="small" plain

@click="doDeleteTeam(team.id)">解散队伍

</van-button>

</template>

</van-card>

</div>

</template>

<script setup lang="ts">

import {TeamType} from "../models/team";

import {teamStatusEnum} from "../constants/team";

import IU from "../assets/IU.png";

import myAxios from "../plugins/myAxios";

import {Toast} from "vant";

import {getCurrentUser} from "../services/user";

import {onMounted, ref} from "vue";

import {useRouter} from "vue-router";

interface TeamCardListProps{

teamList: TeamType[];

}

//withDefaults给属性附默认值

const props= withDefaults(defineProps<TeamCardListProps>(),{

// @ts-ignore

teamList:[] as TeamType[],

});

const currentUser= ref();

const router=useRouter();

onMounted( async ()=>{

currentUser.value= await getCurrentUser();

})

/\*\*

\* 加人队伍

\* @param id

\*/

const doJoinTeam= async (id : number) =>{

const res= await myAxios.post('/team/join',{

teamId: id

});

if (res?.code === 0){

Toast.success('加入成功');

}else {

Toast.fail('加入失败'+ (res.description ? `,${res.description}` : ''));

}

}

/\*\*

\* 跳转至更新队伍页面

\* @param id

\*/

const doUpdateTeam = (id: number) =>{

router.push({

path: '/team/update',

query:{

id,

}

})

}

/\*\*

\* 退出队伍

\* @param id

\*/

const doQuitTeam = async (id: number) => {

const res = await myAxios.post('/team/quit', {

teamId: id

});

if (res?.code === 0) {

Toast.success('操作成功');

} else {

Toast.fail('操作失败' + (res.description ? `,${res.description}` : ''));

}

}

/\*\*

\* 解散队伍

\* @param id

\*/

const doDeleteTeam = async (id: number) => {

const res = await myAxios.post('/team/delete', {

id,

});

if (res?.code === 0) {

Toast.success('操作成功');

} else {

Toast.fail('操作失败' + (res.description ? `,${res.description}` : ''));

}

}

</script>

<style scoped>

#teamCardList :deep(.van-image\_\_img){

width: 100%;

height: 130%;

object-fit: unset;

}

</style>

**修改TeamUpdatePage**

删除不能修改的组件（最大人数）和固定显示的参数（initFormData），修改提交逻辑（由于是复制得来的，千万别忘了，不然就是增加队伍了）

关键是获取之前队伍的信息。引入Route，来获取上个页面传来的参数

定义变量id

截图.png

挂载获取之前队伍的信息



完整代码：

<template>

<div id="teamAddPage">

<van-form @submit="onSubmit">

<van-cell-group inset>

<van-field

v-model="addTeamData.name"

name="name"

label="队伍名"

placeholder="请输入队伍名"

:rules="[{ required: true, message: '请填写队伍名' }]"

/>

<van-field

v-model="addTeamData.description"

rows="4"

autosize

label="队伍描述"

type="textarea"

placeholder="请输入队伍描述"

/>

<van-field

is-link

readonly

name="datetimePicker"

label="时间选择"

:placeholder="addTeamData.expireTime ?? '点击选择过期时间'"

@click="showPicker = true"

/>

<van-popup v-model:show="showPicker" position="bottom">

<van-datetime-picker

v-model="addTeamData.expireTime"

@confirm="showPicker=false"

type="datetime"

title="选择完整时间"

:min-date="minDate"

/>

</van-popup>

<van-field name="radio" label="队伍状态">

<template #input>

<van-radio-group v-model="addTeamData.status" direction="horizontal">

<van-radio name="0">公开</van-radio>

<van-radio name="1">私有</van-radio>

<van-radio name="2">加密</van-radio>

</van-radio-group>

</template>

</van-field>

<van-field

v-if="Number(addTeamData.status) === 2"

v-model="addTeamData.password"

type="password"

name="password"

label="密码"

placeholder="请输入队伍密码"

:rules="[{ required: true, message: '请填写密码' }]"

/>

</van-cell-group>

<div style="margin: 16px;">

<van-button round block type="primary" native-type="submit">

提交

</van-button>

</div>

</van-form>

</div>

</template>

<script setup>

import {useRoute, useRouter} from "vue-router";

import {onMounted, ref} from "vue";

import myAxios from "../plugins/myAxios.ts";

import {Toast} from "vant";

const router =useRouter();

const route = useRoute();

//展示日期选择器

const showPicker = ref(false);

const minDate=new Date();

const id = route.query.id;

//需要用户填写的表单数据

const addTeamData=ref({})

//获取之前的队伍信息

onMounted(async () => {

if (id <= 0){

Toast.fail('加载队伍失败');

return;

}

const res= await myAxios.get("/team/get",{

params:{

id,

}

});

if (res?.code === 0){

addTeamData.value = res.data;

}else {

Toast.fail('加载队伍失败，请刷新重试')

}

})

//提交

const onSubmit = async () =>{

const postData = {

...addTeamData.value,

status: Number(addTeamData.value.status)

}

//todo 前端参数校验

const res = await myAxios.post("/team/update",postData);

if (res?.code === 0 && res.data){

Toast.success('更新成功');

router.push({

path: '/team',

replace: true,

});

}else {

Toast.success('更新失败')

}

}

</script>

<style scoped>

</style>

测试

**查看个人已加入的队伍**

**后端接口**

复用 listTeam 方法，只新增查询条件，不做修改（开闭原则）

获取当前用户创建的队伍

/\*\*

\* 获取我创建的队伍

\* @param teamQuery

\* @param request

\* @return

\*/

@GetMapping("/list/my/create")

public BaseResponse<List<TeamUserVo>> listMyCreateTeams(TeamQuery teamQuery,HttpServletRequest request){

if (teamQuery==null){

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

User loginUser=userService.getLoginUser(request);

teamQuery.setUserId(loginUser.getId());

List<TeamUserVo> teamList = teamService.listTeams(teamQuery,true);

return ResultUtils.success(teamList);

}

我们查询加入的队伍需要用到id的列表，所以在Teamquery里增加idList字段



获取我加入的队伍

/\*\*

\* 获取我加入的队伍

\* @param teamQuery

\* @param request

\* @return

\*/

@GetMapping("/list/my/join")

public BaseResponse<List<TeamUserVo>> listMyJoinTeams(TeamQuery teamQuery,HttpServletRequest request){

if (teamQuery==null){

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

User loginUser=userService.getLoginUser(request);

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

queryWrapper.eq("userId",loginUser.getId());

List<UserTeam> userTeamList=userTeamService.list(queryWrapper);

//取出不重复的队伍 id

// teamId userId

// 1 , 2

// 1 , 3

// 2 , 3

// result

// 1 ==> 2 , 3

// 2 ==> 3

Map<Long, List<UserTeam>> listMap = userTeamList.stream()

.collect(Collectors.groupingBy(UserTeam::getTeamId));

List<Long> idList=new ArrayList<>(listMap.keySet());

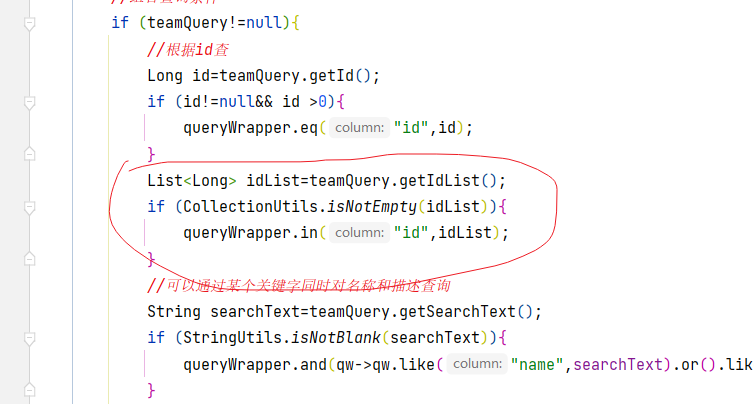
teamQuery.setIdList(idList);

List<TeamUserVo> teamList = teamService.listTeams(teamQuery,true);

return ResultUtils.success(teamList);

}

修改下listTeam方法，加一层校验



测试

**前端页面**

我们复制一份UserPage，命名为UserUpdatePage，修改UserPage（我们只需要当前用户，修改信息，我创建的队伍，我加入的队伍）

修改UserPage如下：

<template>

<template v-if="user">

<van-cell title="当前用户" :value="user?.username"/>

<van-cell title="修改信息" is-link to="/user/update" />

<van-cell title="我创建的队伍" is-link to="/user/team/create" />

<van-cell title="我加入的队伍" is-link to="/user/team/join" />

</template>

</template>

<script setup lang="ts">

import {onMounted, ref} from "vue";

import {useRouter} from "vue-router";

import {getCurrentUser} from "../services/user";

const user=ref();

const router = useRouter();

onMounted( async ()=>{

user.value= await getCurrentUser();

})

const toEdit = (editKey: string, currentValue: string, editName: string) => {

router.push({

path: '/user/edit',

query: {

editKey,

currentValue,

editName

}

})

}

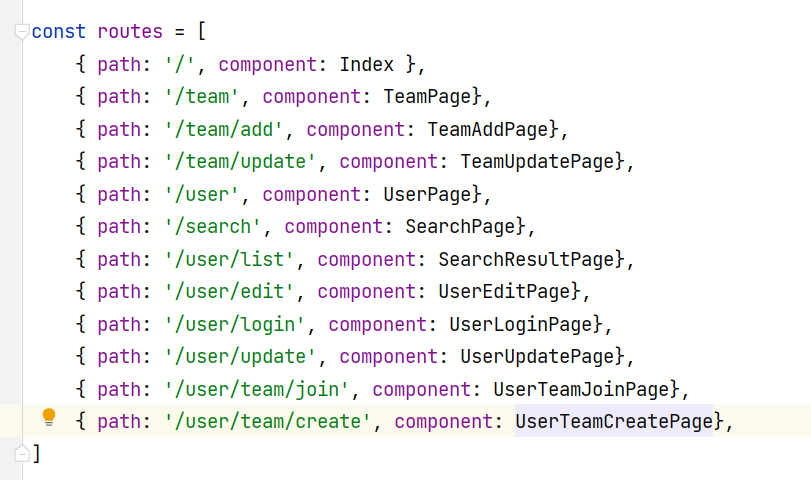
</script>

<style scoped>

</style>

创建页面：查询加入队伍页面和查询创建队伍页面（复制TeamPage页面，形式相同） PS:别忘了在路由里面添加这两个页面

因为我们把原来的用户页面改为用户更新页面，路由里也要修改



查询加入队伍页面

<template>

<div id="teamPage">

<van-search v-model="searchText" placeholder="搜索队伍" @search="onSearch"/>

<van-button type="primary" @click="doJoinTeam">加入队伍</van-button>

<team-card-list :teamList="teamList"/>

<van-empty v-if="teamList?.length < 1" description="数据为空" />

</div>

</template>

<script setup>

import {useRouter} from "vue-router";

import TeamCardList from "../components/TeamCardList.vue";

import {onMounted, ref} from "vue";

import myAxios from "../plugins/myAxios.ts";

import {Toast} from "vant";

const router=useRouter();

const searchText = ref('');

// 跳转到加入队伍页

const doJoinTeam = () =>{

router.push({

path: "/team/add"

})

}

const teamList = ref([]);

/\*\*

\* 搜索队伍

\* @param val

\* @returns {Promise<void>}

\*/

const listTeam = async (val='') => {

const res= await myAxios.get("/team/list/my/join",{

params:{

searchText: val,

pageNum: 1,

}

});

if (res?.code === 0){

teamList.value = res.data;

}else {

Toast.fail('加载队伍失败，请刷新重试')

}

}

//页面加载时只触发一次

onMounted(async () =>{

listTeam();

})

const onSearch = (val) => {

listTeam(val);

}

</script>

<style scoped>

</style>

查询创建队伍页面

<template>

<div id="teamPage">

<van-search v-model="searchText" placeholder="搜索队伍" @search="onSearch"/>

<van-button type="primary" @click="doJoinTeam">创建队伍</van-button>

<team-card-list :teamList="teamList"/>

<van-empty v-if="teamList?.length < 1" description="数据为空" />

</div>

</template>

<script setup>

import {useRouter} from "vue-router";

import TeamCardList from "../components/TeamCardList.vue";

import {onMounted, ref} from "vue";

import myAxios from "../plugins/myAxios.ts";

import {Toast} from "vant";

const router=useRouter();

const searchText = ref('');

// 跳转到加入队伍页

const doJoinTeam = () =>{

router.push({

path: "/team/add"

})

}

const teamList = ref([]);

/\*\*

\* 搜索队伍

\* @param val

\* @returns {Promise<void>}

\*/

const listTeam = async (val='') => {

const res= await myAxios.get("/team/list/my/create",{

params:{

searchText: val,

pageNum: 1,

}

});

if (res?.code === 0){

teamList.value = res.data;

}else {

Toast.fail('加载队伍失败，请刷新重试')

}

}

//页面加载时只触发一次

onMounted(async () =>{

listTeam();

})

const onSearch = (val) => {

listTeam(val);

}

</script>

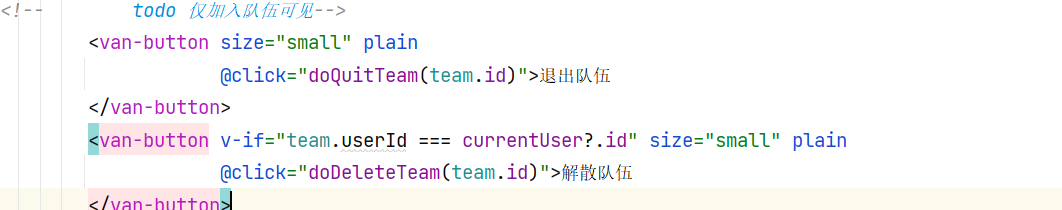
<style scoped>

</style>

测试

**退出和解散队伍**

**TeamCartList添加按钮**



**写入方法**

/\*\*

\* 退出队伍

\* @param id

\*/

const doQuitTeam = async (id: number) => {

const res = await myAxios.post('/team/quit', {

teamId: id

});

if (res?.code === 0) {

Toast.success('操作成功');

} else {

Toast.fail('操作失败' + (res.description ? `,${res.description}` : ''));

}

}

/\*\*

\* 解散队伍

\* @param id

\*/

const doDeleteTeam = async (id: number) => {

const res = await myAxios.post('/team/delete', {

id,

});

if (res?.code === 0) {

Toast.success('操作成功');

} else {

Toast.fail('操作失败' + (res.description ? `,${res.description}` : ''));

}

}

测试

出现bug



报错，这是因为我们后端接口没有封装对象（偷懒)，所以我们封装一个删除请求DeleteRequest



/\*\*

\* 通用删除请求参数

\*/

@Data

public class DeleteRequest implements Serializable {

private static final long serialVersionUID = 1L;

private long id;

}

修改“删除接口”

@PostMapping("/delete")

public BaseResponse<Boolean> deleteTeam(@RequestBody DeleteRequest deleteRequest, HttpServletRequest request){

if (deleteRequest == null || deleteRequest.getId() <= 0){

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

long id = deleteRequest.getId();

User loginUser= userService.getLoginUser(request);

boolean save = teamService.deleteTeam(id,loginUser);

if(!save){

throw new BusinessException(ErrorCode.SYSTEM\_ERROR,"删除失败");

}

return ResultUtils.success(true);

}

**随机匹配**

为了帮助大家更快地发现和自己兴趣相同的朋友

编辑距离算法：<https://blog.csdn.net/DBC_121/article/details/104198838>

最小编辑距离：字符串 1 通过最少多少次增删改字符的操作可以变成字符串 2

余弦相似度算法：<https://blog.csdn.net/m0_55613022/article/details/125683937>（如果需要带权重计算，比如学什么方向最重要，性别相对次要）

**后端编写**

这里我们使用了编辑距离算法

把这个方法放在工具类（新建一个utils包）里面，并写一个测试类测试

我们推荐是通过标签类（所以我们传的参数应该是字符型的列表），修改整理为：

Utils:

/\*\*

\* 算法工具类测试

\*/

public class AlgorithmUtils {

/\*\*

\* 编辑距离算法（用于计算最相似的两组标签）

\* 原理：https://blog.csdn.net/DBC\_121/article/details/104198838

\*

\* @param tagList1

\* @param tagList2

\* @return

\*/

public static int minDistance(List<String> tagList1, List<String> tagList2) {

int n = tagList1.size();

int m = tagList2.size();

if (n \* m == 0) {

return n + m;

}

int[][] d = new int[n + 1][m + 1];

for (int i = 0; i < n + 1; i++) {

d[i][0] = i;

}

for (int j = 0; j < m + 1; j++) {

d[0][j] = j;

}

for (int i = 1; i < n + 1; i++) {

for (int j = 1; j < m + 1; j++) {

int left = d[i - 1][j] + 1;

int down = d[i][j - 1] + 1;

int left\_down = d[i - 1][j - 1];

if (!Objects.equals(tagList1.get(i - 1), tagList2.get(j - 1))) {

left\_down += 1;

}

d[i][j] = Math.min(left, Math.min(down, left\_down));

}

}

return d[n][m];

}

/\*\*

\* 编辑距离算法（用于计算最相似的两个字符串）

\* 原理：https://blog.csdn.net/DBC\_121/article/details/104198838

\*

\* @param word1

\* @param word2

\* @return

\*/

public static int minDistance(String word1, String word2) {

int n = word1.length();

int m = word2.length();

if (n \* m == 0) {

return n + m;

}

int[][] d = new int[n + 1][m + 1];

for (int i = 0; i < n + 1; i++) {

d[i][0] = i;

}

for (int j = 0; j < m + 1; j++) {

d[0][j] = j;

}

for (int i = 1; i < n + 1; i++) {

for (int j = 1; j < m + 1; j++) {

int left = d[i - 1][j] + 1;

int down = d[i][j - 1] + 1;

int left\_down = d[i - 1][j - 1];

if (word1.charAt(i - 1) != word2.charAt(j - 1)) {

left\_down += 1;

}

d[i][j] = Math.min(left, Math.min(down, left\_down));

}

}

return d[n][m];

}

}

Test

/\*\*

\* 算法工具类测试

\*/

public class AlgorithumUtilsTest {

@Test

void test() {

String str1 = "鱼皮是狗";

String str2 = "鱼皮不是狗";

String str3 = "鱼皮是鱼不是狗";

// String str4 = "鱼皮是猫";

// 1

int score1 = AlgorithmUtils.minDistance(str1, str2);

// 3

int score2 = AlgorithmUtils.minDistance(str1, str3);

System.out.println(score1);

System.out.println(score2);

}

@Test

void testCompareTags() {

List<String> tagList1 = Arrays.asList("Java", "大一", "男");

List<String> tagList2 = Arrays.asList("Java", "大一", "女");

List<String> tagList3 = Arrays.asList("Python", "大二", "女");

// 1

int score1 = AlgorithmUtils.minDistance(tagList1, tagList2);

// 3

int score2 = AlgorithmUtils.minDistance(tagList1, tagList3);

System.out.println(score1);

System.out.println(score2);

}

}

在UserController里写入获取最匹配的用户的接口

/\*\*

\* 获取最匹配的用户

\* @param num

\* @param request

\* @return

\*/

@GetMapping("match")public BaseResponse<List<User>> matchUser(long num, HttpServletRequest request){

if (num <= 0 || num > 20){

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

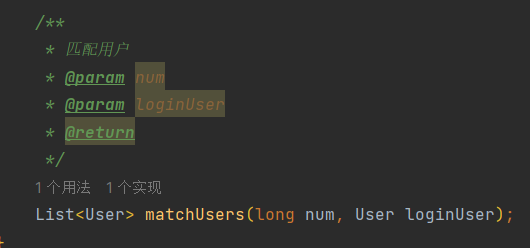
}

User user = userService.getLoginUser(request);

return ResultUtils.success(userService.matchUsers(num,user));

}

在USerService里写入matchUsers方法并实现



具体的实现方法本期直播并未完美的完成（遗留bug），所以结合13期的内容，修复了排序的问题

下面就是具体的代码: (这里由于鱼皮踩坑过多，同时自己也没有完全理解，过程就省略)

@Override

public List<User> matchUsers(long num, User loginUser) {

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

queryWrapper.select("id", "tags");

queryWrapper.isNotNull("tags");

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

String tags = loginUser.getTags();

Gson gson = new Gson();

List<String> tagList = gson.fromJson(tags, new TypeToken<List<String>>() {

}.getType());

// 用户列表的下标 => 相似度

List<Pair<User, Long>> list = new ArrayList<>();

// 依次计算所有用户和当前用户的相似度

for (int i = 0; i < userList.size(); i++) {

User user = userList.get(i);

String userTags = user.getTags();

// 无标签或者为当前用户自己

if (StringUtils.isBlank(userTags) || user.getId() == loginUser.getId()) {

continue;

}

List<String> userTagList = gson.fromJson(userTags, new TypeToken<List<String>>() {

}.getType());

// 计算分数

long distance = AlgorithmUtils.minDistance(tagList, userTagList);

list.add(new Pair<>(user, distance));

}

// 按编辑距离由小到大排序

List<Pair<User, Long>> topUserPairList = list.stream()

.sorted((a, b) -> (int) (a.getValue() - b.getValue()))

.limit(num)

.collect(Collectors.toList());

// 原本顺序的 userId 列表

List<Long> userIdList = topUserPairList.stream().map(pair -> pair.getKey().getId()).collect(Collectors.toList());

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

userQueryWrapper.in("id", userIdList);

// 1, 3, 2

// User1、User2、User3

// 1 => User1, 2 => User2, 3 => User3

Map<Long, List<User>> userIdUserListMap = this.list(userQueryWrapper)

.stream()

.map(user -> getSafetyUser(user))

.collect(Collectors.groupingBy(User::getId));

List<User> finalUserList = new ArrayList<>();

for (Long userId : userIdList) {

finalUserList.add(userIdUserListMap.get(userId).get(0));

}

return finalUserList;

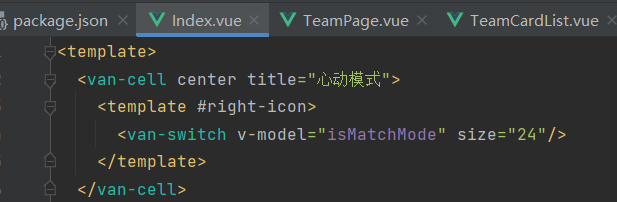
}

测试

**直播13**

**主页切换功能**

在vant组件库里寻找switch开关，复制到主页index里面



定义一个开关切换常量，默认为关闭

截图.png

现在不需要一次性挂载，写一个加载的方法，并且写一个监听器（当开关切换时，“更换页面”）

/\*\* \* 加载数据 \*/const loadData = async () => {

let userListData ; loading.value=true; //心动模式，根据标签匹配用户 if (isMatchMode.value){

const num =10; userListData= await myAxios.get('/user/match', {

params: {

num, }, })

.then(function (response){

console.log('/user/match success',response); // Toast.success('请求成功'); return response?.data; //返回数据， 可选链操作符，避免数据为 null 或 undefined 时 报错 })

.catch(function (error){

console.error('/user/match error',error); Toast.fail('请求失败'); })

}else {

//普通模式，直接分页查询用户 userListData= await myAxios.get('/user/recommend', {

params: {

pageSize: 8, pageNum: 1, }, })

.then(function (response){

console.log('/user/recommend success',response); // Toast.success('请求成功'); return response?.data?.records; //返回数据， 可选链操作符，避免数据为 null 或 undefined 时 报错 })

.catch(function (error){

console.error('/user/recommend error',error); Toast.fail('请求失败'); })

}

//数据序列化 if (userListData){

userListData.forEach((user : UserType) => {

if (user.tags) {

user.tags = JSON.parse(user.tags); }

})

userList.value = userListData; }

loading.value=false;}

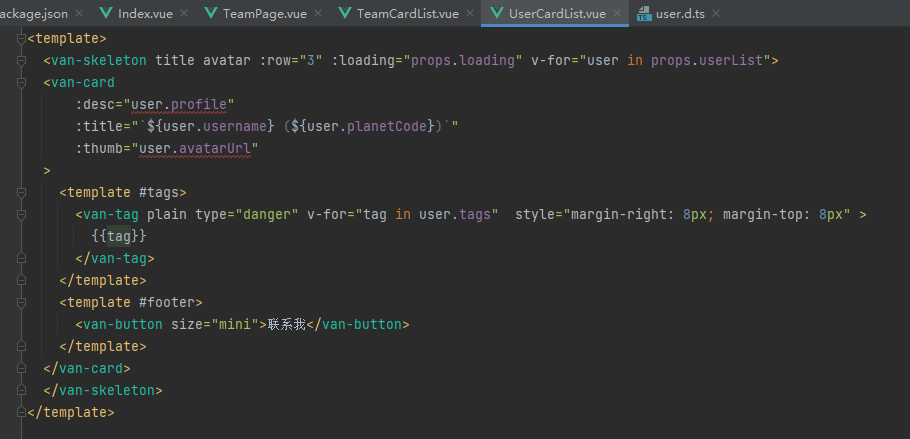
watchEffect( ()=>{

loadData();})

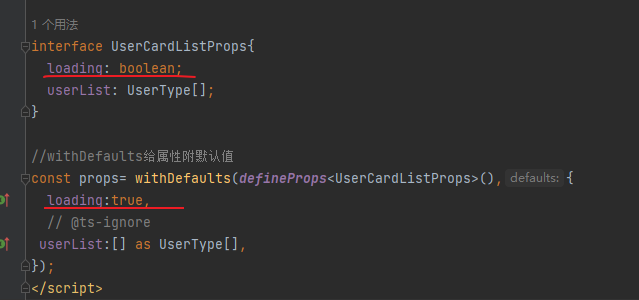
**TODO**

**加载骨架屏特效**

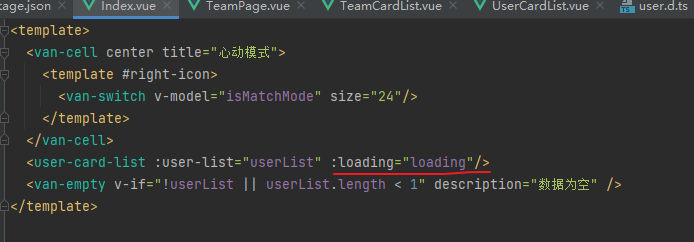
解决： van-skeleton组件



在js里添加这两个参数



到index的user-card-list里引用



同时别忘了引入loading常量，并在loadData方法里，在开始和结尾处分别使loading设置为true和false

**仅加入队伍和创建队伍的人能看到队伍操作按钮( listTeam 接口要能获取我加入的队伍状态）**

**队伍操作权限控制**

加入队伍： 仅非队伍创建人、且未加入队伍的人可见

更新队伍：仅创建人可见

解散队伍：仅创建人可见

退出队伍：创建人不可见，仅已加入队伍的人可见

方案1：前端查询我加入了哪些队伍列表，然后判断每个队伍id是否在列表中（前端要多发一次请求）

方案2： 在后端去做上述的事 （ 推荐 ）

**在TeamUserVo 添加 “是否已加入队伍的字段”**



**修改ListTeam的接口，加入是否已经加入队伍的判断**

@GetMapping("/list")

public BaseResponse<List<TeamUserVO>> listTeams(TeamQuery teamQuery, HttpServletRequest request) {

if (teamQuery == null) {

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

boolean isAdmin = userService.isAdmin(request);

// 1、查询队伍列表

List<TeamUserVO> teamList = teamService.listTeams(teamQuery, isAdmin);

final List<Long> teamIdList = teamList.stream().map(TeamUserVO::getId).collect(Collectors.toList());

// 2、判断当前用户是否已加入队伍

QueryWrapper<UserTeam> userTeamQueryWrapper = new QueryWrapper<>();

try {

User loginUser = userService.getLoginUser(request);

userTeamQueryWrapper.eq("userId", loginUser.getId());

userTeamQueryWrapper.in("teamId", teamIdList);

List<UserTeam> userTeamList = userTeamService.list(userTeamQueryWrapper);

// 已加入的队伍 id 集合

Set<Long> hasJoinTeamIdSet = userTeamList.stream().map(UserTeam::getTeamId).collect(Collectors.toSet());

teamList.forEach(team -> {

boolean hasJoin = hasJoinTeamIdSet.contains(team.getId());

team.setHasJoin(hasJoin);

});

} catch (Exception e) {}

return ResultUtils.success(teamList);

}

**修改前端页面的user-card-list 里面 4 个按钮出现的判断条件**

<template #footer>

<!-- 仅非队伍创建人可见，且未加入队伍的人可见-->

<van-button size="small" type="primary" v-if="team.userId !== currentUser?.id && !team.hasJoin" plain

@click="doJoinTeam(team.id)">加入队伍

</van-button>

<!-- 仅创建人可见-->

<van-button v-if="team.userId === currentUser?.id" size="small" plain

@click="doUpdateTeam(team.id)">更新队伍

</van-button>

<!--仅创建人可见-->

<van-button v-if="team.id !== currentUser?.id && team.hasJoin" size="small" plain

@click="doQuitTeam(team.id)">退出队伍

</van-button>

<!-- 创建人不可见，仅已加入队伍的人可见-->

<van-button v-if="team.userId === currentUser?.id" size="small" type="danger" plain

@click="doDeleteTeam(team.id)">解散队伍

</van-button>

</template>

测试

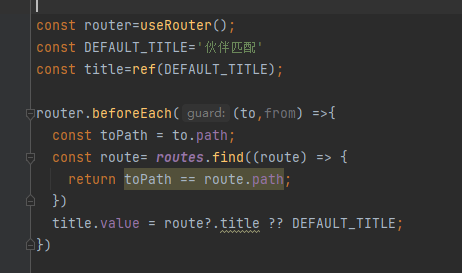
**前端死标题【】更改**

解决：使用 router.beforEach 根据要跳转页面的url，路径匹配 config / routes 配置的title 字段

1.配置路由里的title字段

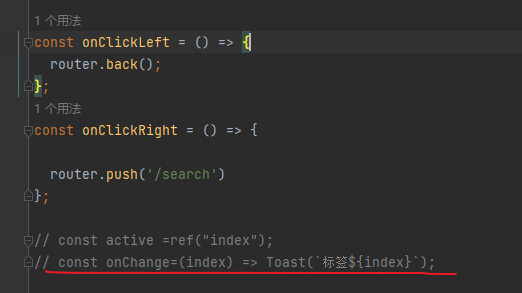


在BasicLayout里增加根据路由切换标题



同时把原来用于测试的Toast响应（请求成功）删除，全局搜索删除

别忘了，把这句也删除



刷新，切换到不同页面，测试标签栏是否更换，以及请求成功是否不再出现

**直播14**

**笔记**

**优化、上线**

1、强制登录，自动跳转到登录页

解决：axios 全局配置响应拦截、并且添加重定向

2、区分公开和加密房间；加入有密码的房间，要指定密码3、展示已加入队伍人数

4、重复加入队伍的问题（加锁、分布式锁）并发请求时可能出现问题

**分布式锁**

**上线**

先区分多环境：前端区分开发和线上接口，后端 prod 改为用线上公网可访问的数据库

前端：Vercel（免费）

<https://vercel.com/>

后端：微信云托管（部署容器的平台，付费）

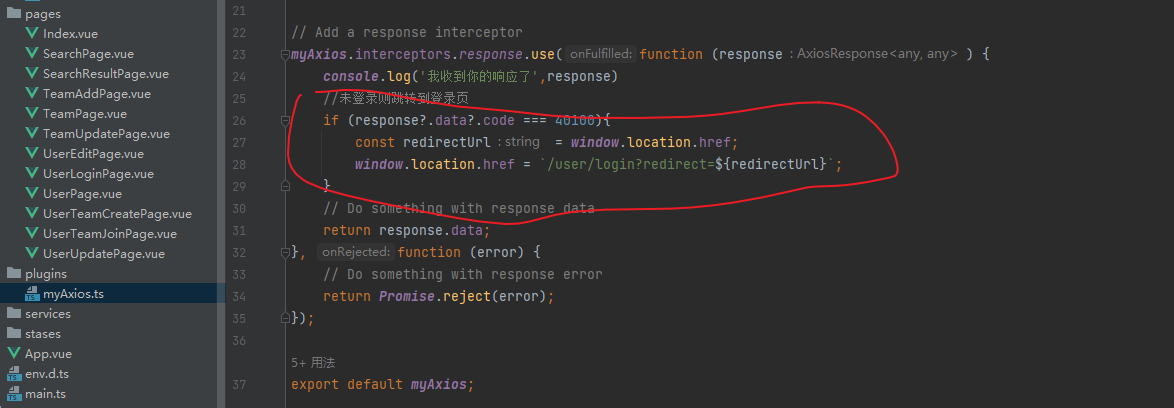
<https://cloud.weixin.qq.com/cloudrun/service>

**（免备案！！！）**

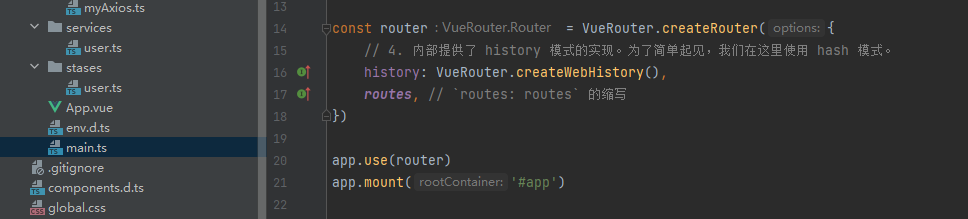
**TODO**

**强制登录，前端全局响应拦截，自动跳转到登录页**

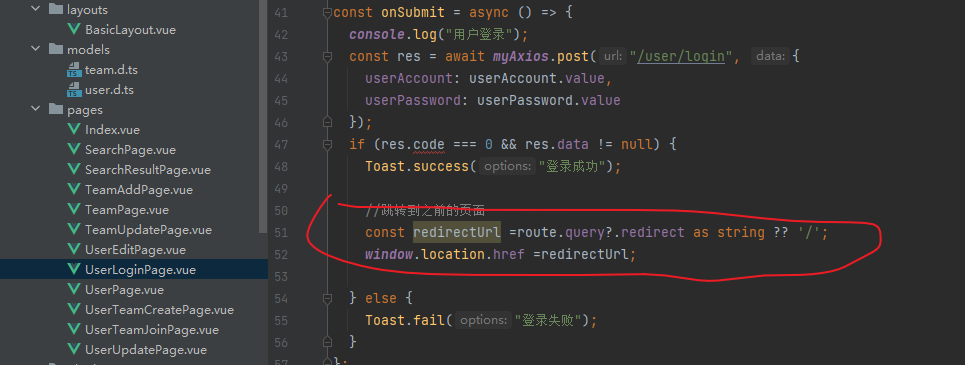
解决： axios 全局配置响应拦截，



这里我们要改变history 模式的实现，在main.ts里修改



当登录成功后，重定向到个人用户页面 PS：别忘了引入route

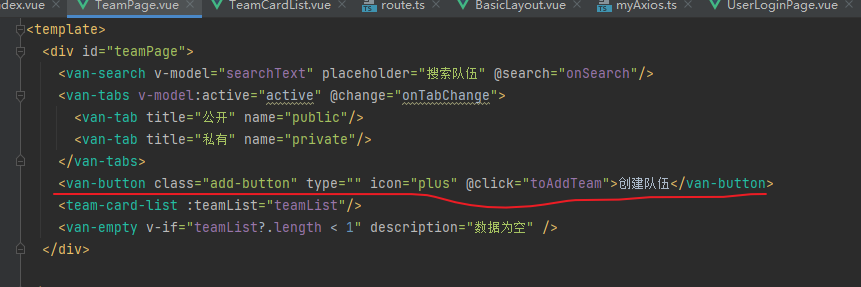


**修改队伍页面的加入队伍按钮为创建队伍**

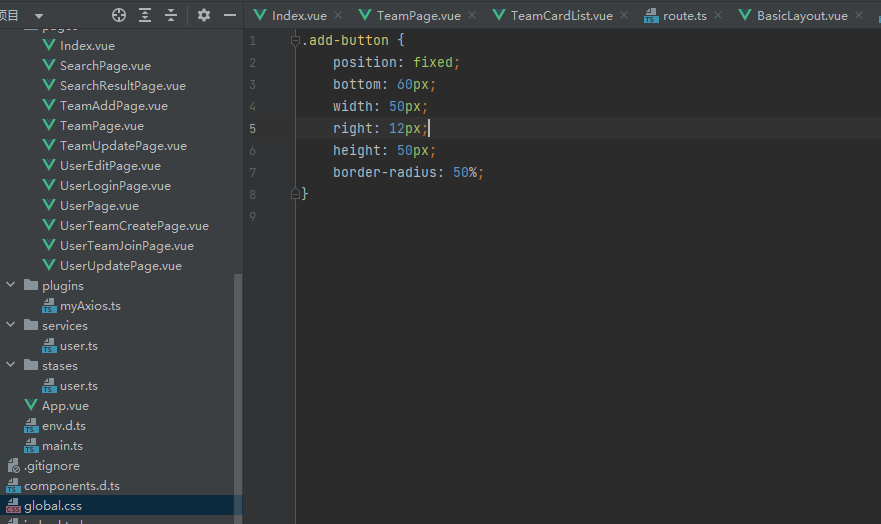
在TeamPage页面，修改加入队伍为创建队伍（按钮部分）

把doJoinTeam全局修改为toAddTeam

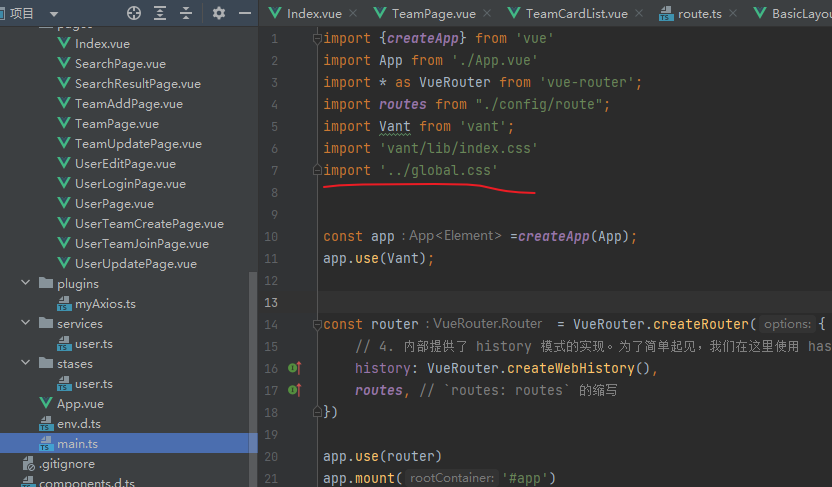
这个按钮太丑了，我们更换它的样式，变成圆形放在右下角



写一个全局样式

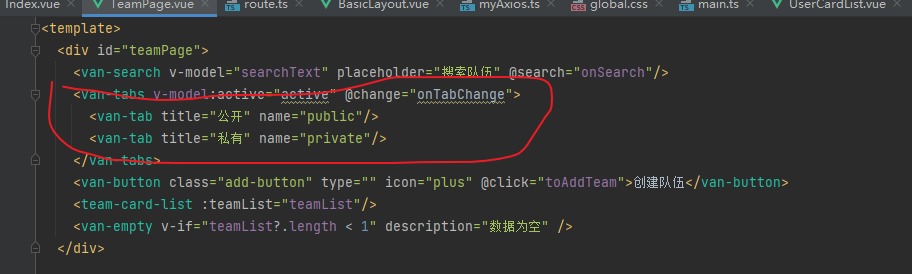


在main.ts中引入



**区分公开和加密房间，加入有密码的房间，要指定密码**

在TeamPage页面加入tabs标签，来区分公开还是加密



后端我们以前根据状态查询只查询公开，现在修改为当不是管理员和私人才会报权限错误



回到前端，我们在TeamPage页面实现onTabChange方法



上面定义的active是为了页面默认显示公开队伍

修改搜索队伍，传入状态



现在点击公开和加密可以切换查看不同类型的队伍

加密队伍需要输入密码才可以加入，我们这使用Dialog 弹出框组件，把它放入team-card-list里（最下面的位置）

<van-dialog v-model:show="showPasswordDialog" title="请输入密码" show-cancel-button @confirm="doJoinTeam" @cancel="doJoinCancel">

<van-field v-model="password" placeholder="请输入密码"/>

</van-dialog>

在js添加

// 弹窗

const VanDialog = Dialog.Component;

在里面修改加入doJoinTeam方法，实现doJoinCancel方法和判断是不是加密房间preJoinTeam方法

/\*\*

\* 加入队伍

\*/

const doJoinTeam = async () => {

if (!joinTeamId.value){

return;

}

const res = await myAxios.post('/team/join', {

teamId: joinTeamId.value,

password: password.value

});

if (res?.code === 0) {

Toast.success('加入成功');

doJoinCancel();

} else {

Toast.fail('加入失败' + (res.description ? `，${res.description}` : ''));

}

}

const showPasswordDialog = ref(false);

const password = ref('');

const joinTeamId = ref(0);

/\*\*

\* 判断是不是加密房间，是的话显示密码框

\* @param team

\*/

const preJoinTeam = (team: TeamType) => {

joinTeamId.value = team.id;

if (team.status === 0) {

doJoinTeam()

} else {

showPasswordDialog.value = true;

}

}

const doJoinCancel = () => {

joinTeamId.value = 0;

password.value = '';

}

测试

**展示队伍已加入人数**

后端还未实现，所以在获取队伍列表接口，获取这个参数

首先在封装类里添加字段（TeamUserVO）



修改listTeams接口：

@GetMapping("/list")

public BaseResponse<List<TeamUserVO>> listTeams(TeamQuery teamQuery, HttpServletRequest request) {

if (teamQuery == null) {

throw new BusinessException(ErrorCode.PARAMS\_ERROR);

}

boolean isAdmin = userService.isAdmin(request);

// 1、查询队伍列表

List<TeamUserVO> teamList = teamService.listTeams(teamQuery, isAdmin);

final List<Long> teamIdList = teamList.stream().map(TeamUserVO::getId).collect(Collectors.toList());

// 2、判断当前用户是否已加入队伍

QueryWrapper<UserTeam> userTeamQueryWrapper = new QueryWrapper<>();

try {

User loginUser = userService.getLoginUser(request);

userTeamQueryWrapper.eq("userId", loginUser.getId());

userTeamQueryWrapper.in("teamId", teamIdList);

List<UserTeam> userTeamList = userTeamService.list(userTeamQueryWrapper);

// 已加入的队伍 id 集合

Set<Long> hasJoinTeamIdSet = userTeamList.stream().map(UserTeam::getTeamId).collect(Collectors.toSet());

teamList.forEach(team -> {

boolean hasJoin = hasJoinTeamIdSet.contains(team.getId());

team.setHasJoin(hasJoin);

});

} catch (Exception e) {}

// 3、查询已加入队伍的人数

QueryWrapper<UserTeam> userTeamJoinQueryWrapper = new QueryWrapper<>();

userTeamJoinQueryWrapper.in("teamId", teamIdList);

List<UserTeam> userTeamList = userTeamService.list(userTeamJoinQueryWrapper);

// 队伍 id => 加入这个队伍的用户列表

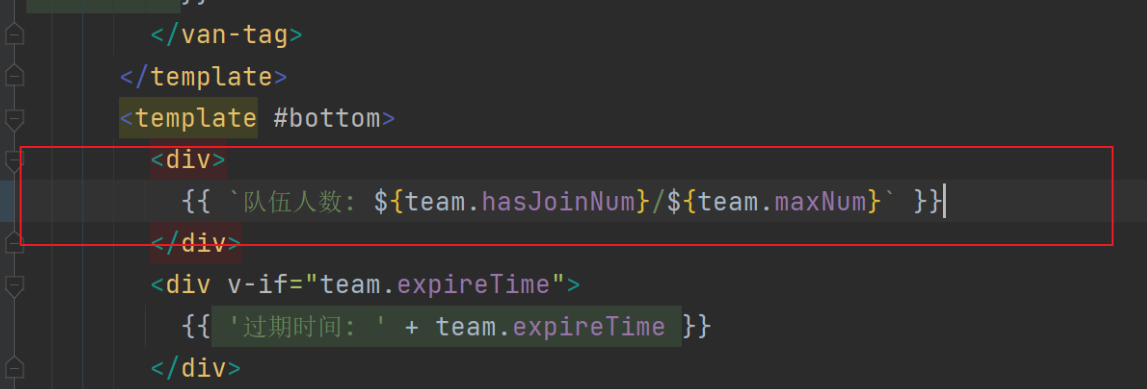
Map<Long, List<UserTeam>> teamIdUserTeamList = userTeamList.stream().collect(Collectors.groupingBy(UserTeam::getTeamId));

teamList.forEach(team -> team.setHasJoinNum(teamIdUserTeamList.getOrDefault(team.getId(), new ArrayList<>()).size()));

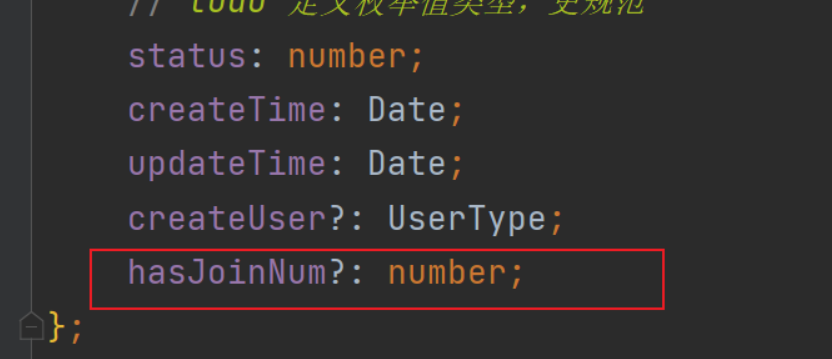
return ResultUtils.success(teamList);

}

在前端的TeamCardList里修改原来的最大人数为已加入人数



如果爆红的在队伍规范类型里添加字段



分布式锁 导致的 其他服务器数据不同意的问题

或者多个Redis 里 的数据不一致

重复加入队伍的问题（加锁，分布式锁）

用户登录页跳转

标签

前端拦截器统一输出日志

**简历写法**

**项目介绍**

基于Vue 3 + SpringBoot 2 的移动端网站，实现了用户管理，按标签检索用户，推荐相似用户，组队等功能。

**主要工作：**

用户登录： 使用Redis实现分布式 Session ， 解决 集群间登录态同步问题，并使用 Hash 代替 String 来存储用户信息，节约了 xx% 的内存并便于单字段的修改。 （需要自己实际测试对比数据，节省内存的原因是 不用保存序列化对象信息或者 JSON 的一些额外字符串）

对于项目中复杂的集合处理（ 比如 为 队伍列表 关联已加入队伍的用户） 使用 Java 8 Stream API 和 Lambda 表达式来简化编码

使用 Easy Excel 读取收集来的基础用户信息，并通过自定义线程池 + CompletableFuture 并发编程提高批量导入数据库的性能。 实测导入100万行的时间从 xx 秒 缩短至 xx 秒 （需要自己实际测试对比数据）

使用Redis缓存首页高频访问的用户信息列表，将接口相应时长从xx秒 缩至 xx 秒， 且 通过自定义Redis序列化器来解决数据乱码，空间浪费的问题

为解决首次访问系统的用户主页加载过慢的问题，使用Spring Scheduler 定时任务来实现缓存预热，并通过分布式锁保证多机部署时定时任务不会重复执行，

为解决同一用户重复加入队伍，入伍人数超限的问题，使用Redisson 分布式锁来实现操作互斥，保证了接口幂等性

使用编辑距离算法实现根据标签匹配最相似用户的功能，并通过优先队列来减少 TOP N 运算过程中的内存占用

自主编写 Dockerfile , 并通过第三方容器托管平台实现自动化镜像构建及容器部署，提高部署上线效率

使用 Knife4j + Swagger 自动生成后端接口文档，并通过编写 ApiOperation 等注解补充接口注释，避免人工编写维护文档的麻烦

前端使用 Vant UI 组件库，并封装了全局通用的Layout 组件，使 主页，搜索页，组队页 布局一致，减少重复代码

基于 Vue Router 全局路由守卫实现了根据不同页面来动态切换导航栏标题，通过在全局路由配置文件扩展 title 字段来减少无意义的 if else 代码