摘要

随着信息技术的发展，现在已经由工业时代进入了信息时代。从Web1.0、Web2.0直到现在的Web3.0时代，从最开始的只有PC机能够浏览信息直到现在的手机、平板电脑、专用终端都能够浏览信息，进行数据交互。Web网站在近几年也进入了高速发展期，各种开发语言、框架、模式较之以前有了很大的变化。

此次毕业设计采用Spring-boot、Vue.js、Ajax技术，实现监控公司组织结构管理系统，具有一个RPC服务风格的服务器，以及MVVM风格的SPA网页。

**关键词：**前后分离，Spring-boot，SPA，Ajax

ABSTRACT

With the development of information technology, the industrial age has now entered the age of information. From Web1.0 and Web2.0 to the present Web3.0 era, from the very beginning when only PC could browse information to the present age when smart devices, tablet computers and dedicated terminals can browse information and conduct data interaction. Websites in recent years have also entered a period of rapid development. A variety of development languages, frameworks, and patterns has changed a lot.

This graduation project uses Spring-Boot, Vue.js, and Ajax technology to achieve the monitoring company organizational structure management system, with an RPC service style server, as well as MVVM style SPA web page.

**Key Words:** Separation Before and After, Spring-Boot, SPA, Ajax.

**目 录**

[第1章 课题整体框架 1](#_Toc7187655)

[1.1. 课题任务 1](#_Toc7187656)

[1.2. 课题要求 1](#_Toc7187657)

[1.3. 研究意义 1](#_Toc7187658)

[第2章 设计方案 4](#_Toc7187659)

[2.1. 开发环境 4](#_Toc7187661)

[2.2. 软件结构图 4](#_Toc7187662)

[2.3. 功能调用时序图 4](#_Toc7187663)

[第3章 功能实现 6](#_Toc7187664)

[3.1. 功能实现思路 6](#_Toc7187666)

[3.2. 项目生成 6](#_Toc7187667)

[3.2.1. 后端开发流程 6](#_Toc7187668)

[3.2.2. 前端开发流程 9](#_Toc7187669)

[3.3. 后端项目详细介绍 11](#_Toc7187670)

[3.3.1. 工程目录结构 11](#_Toc7187671)

[3.3.2. 模块简介 12](#_Toc7187672)

[3.3.3. 模块依赖 13](#_Toc7187673)

[3.3.4. 日志模块 14](#_Toc7187674)

[3.3.5. 缓存模块 17](#_Toc7187675)

[3.3.6. 跨域处理模块 19](#_Toc7187676)

[3.3.7. 验证码模块 21](#_Toc7187677)

[3.3.8. 认证授权模块 22](#_Toc7187678)

[3.3.9. 控制器模块 29](#_Toc7187679)

[3.3.10. 数据库设计 29](#_Toc7187680)

[3.3.11. DTO模块 34](#_Toc7187681)

[3.3.12. 异常模块 35](#_Toc7187682)

[3.3.13. 数据库操作模块 36](#_Toc7187683)

[3.3.14. Service模块 39](#_Toc7187684)

[3.3.15. 启动类 49](#_Toc7187685)

[3.4. 前端项目详细介绍 50](#_Toc7187686)

[3.4.1. SPA介绍 50](#_Toc7187687)

[3.4.2. 模块化 50](#_Toc7187688)

[3.4.3. Vue 50](#_Toc7187689)

[3.4.4. 路由控制 59](#_Toc7187690)

[3.4.5. 数据交互 63](#_Toc7187691)

[第4章 调试与实现 70](#_Toc7187692)

[4.1. 调试中遇到的重点与难点 70](#_Toc7187694)

[4.1.1. 调试重点 70](#_Toc7187695)

[4.1.2. 调试难点 70](#_Toc7187696)

[4.2. 解决方案 70](#_Toc7187697)

[4.3. 实现展示 71](#_Toc7187698)

[4.3.1. 登录页面 71](#_Toc7187699)

[4.3.2. 组织结构管理页面 71](#_Toc7187700)

[4.3.3. 设备管理页面 72](#_Toc7187701)

[4.3.4. 人脸信息管理页面 72](#_Toc7187702)

[4.3.5. 权限管理页面 73](#_Toc7187703)

[4.3.6. 日志管理 73](#_Toc7187704)

[4.3.7. 资源结构管理页面 74](#_Toc7187705)

[第5章 总结 75](#_Toc7187706)

[参考文献 77](#_Toc7187707)

[致谢 78](#_Toc7187708)

第1章 课题整体框架

* 1. 课题任务

1. 公司组织结构数据展示：根据监控公司组织结构设计出一套合理的数据结构，并把数据结构实现且将它在浏览器上合理展示出来。
2. 权限控制：针对不同职位的员工具有不同的操作权限。
3. 公司业务数据结构数据展示：对公司主营监控业务有查看业务数据的功能。
   1. 课题要求

操作逻辑人性化，数据结构正确，程序功能完善，程序健壮性好。

* 1. 研究意义
     1. 研究背景

近几年Web前端技术发展迅速，2010年MV\*框架出现，2014年HTML5正式发布，2015年ECMAScript6发布，2016年Chrome浏览器占据市场半壁江山。可以说现在的前端技术已经形成了一套规范，网页已经由最初的Web Site演变成了Web App。

后端服务Java语言语法一直以厚重繁杂，官方标准EJB实现起来复杂难懂，后来出现的Spring框架缓解了EJB的复杂，但又出现了繁杂的XML配置。Java程序员们才从EJB坑中跳出来，又进了一个新坑。直到现在Spring-boot的出现，现在大大的缓解了Java开发服务器应用繁杂厚重的困难。

* + 1. HTML5概述

定义：万维网的核心语言、标准通用标记语言下的一个应用超文本标记语言（HTML）的第五次重大修改。HTML5的设计目的是为了在移动设备上支持多媒体。新的语法特征被引进以支持这一点，如video、audio和canvas 标记。

HTML5还引进了新的功能，可以真正改变用户与文档的交互方式，包括：新的解析规则增强了灵活性、新属性、淘汰过时的或冗余的属性、一个HTML5文档到另一个文档间的拖放功能、离线编辑、信息传递的增强、详细的解析规则、多用途互联网邮件扩展（MIME）和协议处理程序注册、在SQL数据库中存储数据的通用标准（Web SQL）。

HTML5未来会更侧重于移动优先，因为现在很大一部分的人已经到了离开手机就没法生活的地步了，移动优先已经成了一种趋势，不管开发什么，都以移动为主。

* + 1. JavaScript概述

JavaScript一种直译式脚本语言，是一种动态类型、弱类型、基于原型的语言，内置支持类型。它的解释器被称为JavaScript引擎，为浏览器的一部分，广泛用于客户端的脚本语言，最早是在HTML（标准通用标记语言下的一个应用）网页上使用，用来给HTML网页增加动态功能。JavaScript包括ECMAScript、BOM、DOM。BOM是浏览器内置对象，DOM是文档树对象。

ECMAScript 6（简称ES6）是于2015年6月正式发布的JavaScript语言的标准，正式名为ECMAScript 2015（ES2015）。它的目标是使得JavaScript语言可以用来编写复杂的大型应用程序，成为企业级开发语言。

* + 1. Web App概述

此处Web App指的是单页应用(Single Page Application)。早期web应用的前后端交互模式是这样的，每个html作为一个功能元件，通过刷新、超链接、表单提交等方式，将页面组织起来后给用户提供交互。浏览器只作为展示层，将MVC全置于后端，加重了服务端的体量，开发中主要以后端为主。页面展示依赖于不同的功能元件，所以必须依靠刷新页面，或者跳转路由来实现功能块的切换，这种方式严重耗费资源，同时用户体验很差。

而单页应用，没有了令人诟病的页面频繁刷新，同时节约浏览器资源，路由响应比较及时，提升了用户的体验。前端组件化是将独立完整的功能模块封装到一个组件中，代码结构更加规范，便于代码维护，同时模块化后的组件可以在不同的场景中进行复用，极大地加快了迭代开发的速度。

* + 1. Spring boot概述

Spring boot是一些库的集合，它能够被任意项目的构建系统所使用，Spring Boot致力于在蓬勃发展的快速应用开发领域(rapid application development)成为领导者。具有以下特点：创建独立的Spring应用程序；嵌入的Tomcat，无需部署WAR文件；简化Maven配置；自动配置Spring；提供生产就绪型功能，如指标，健康检查和外部配置；绝对没有代码生成并且对XML也没有配置要求。

* + 1. Ajax概述

Ajax 即“Asynchronous Javascript And XML”（异步 JavaScript 和 XML），是指一种创建交互式网页应用的网页开发技术。使用 JavaScript 向服务器提出请求并处理响应而不阻塞用户核心对象XMLHttpRequest。通过这个对象，您的 JavaScript 可在不重载页面的情况与 Web 服务器交换数据，即在不需要刷新页面的情况下，就可以产生局部刷新的效果。这个特点奠定了SPA的基础。

第2章 设计方案

1. 1. 开发环境

操作系统：Windows10 x64、JDK1.8、Node.js 10。

开发工具：IntelliJ IDEA、Vue CLI。

服务器版本：Tomcat9。

数据库：MySQL5.7、Redis3.2.1。

浏览器：Chrome。

* 1. 软件结构图

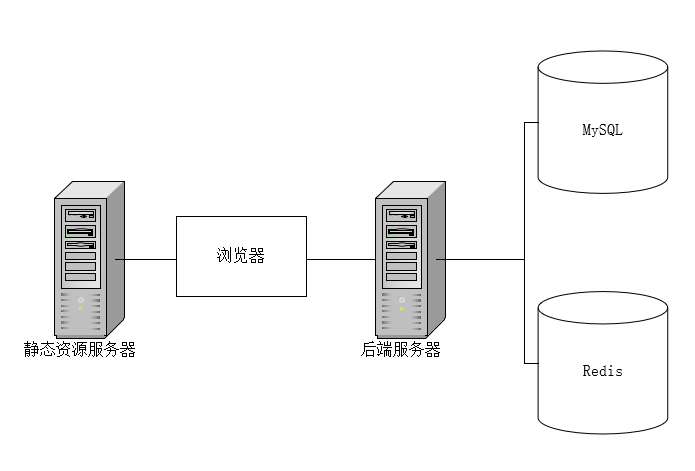


图2-1软件结构图

* 1. 功能调用时序图

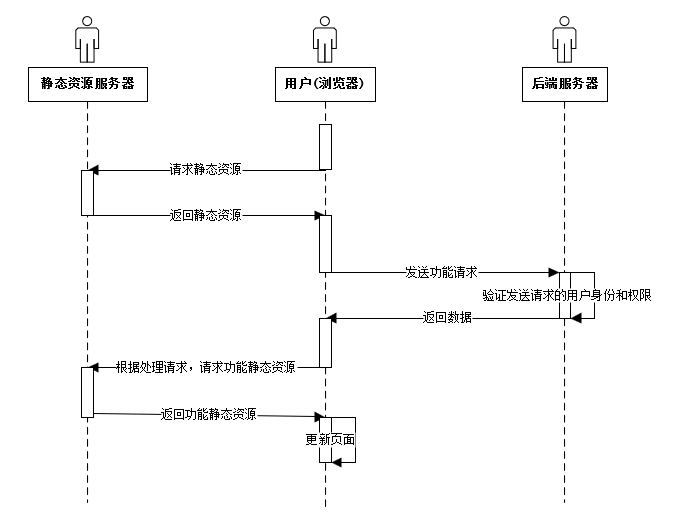


图2-2功能调用时序图

第3章 功能实现

1. 1. 功能实现思路

本项目使用前后分离概念，后端服务器采用RPC风格服务器，前端服务为HTML5、CSS3、JavaScript实现的单页应用，通过HTTP协议与后端服务进行数据交互。前端使用的网络库为Axios。流程如下：

1. 用户通过浏览器访问页面。
2. 浏览器访问静态资源服务器，服务器返回整个单页应用，第一次加载时由于数据量较大且浏览器未做缓存，所以第一次加载会较慢，只要不清楚缓存，后面的页面加载和渲染将非常快。
3. 用户通过网页登录、获得身份Token。
4. 用户每次访问后端API时，需要携带Token，后端程序通过Token判断用户身份与权限进而返回相应的JSON数据和HTTP状态码。
5. 单页应用根据后端返回的JSON数据、HTTP状态码做出相应判断展现出数据或引导至错误页面。

整个过程中前后分离，仅通过HTTP协议交互数据，降低前后端耦合，使整个项目可以并行开发，大大提高开发速度与项目质量。

* 1. 项目生成
     1. 后端开发流程

打开Spring官网的Spring boot 的初始化页面 https://start.spring.io/

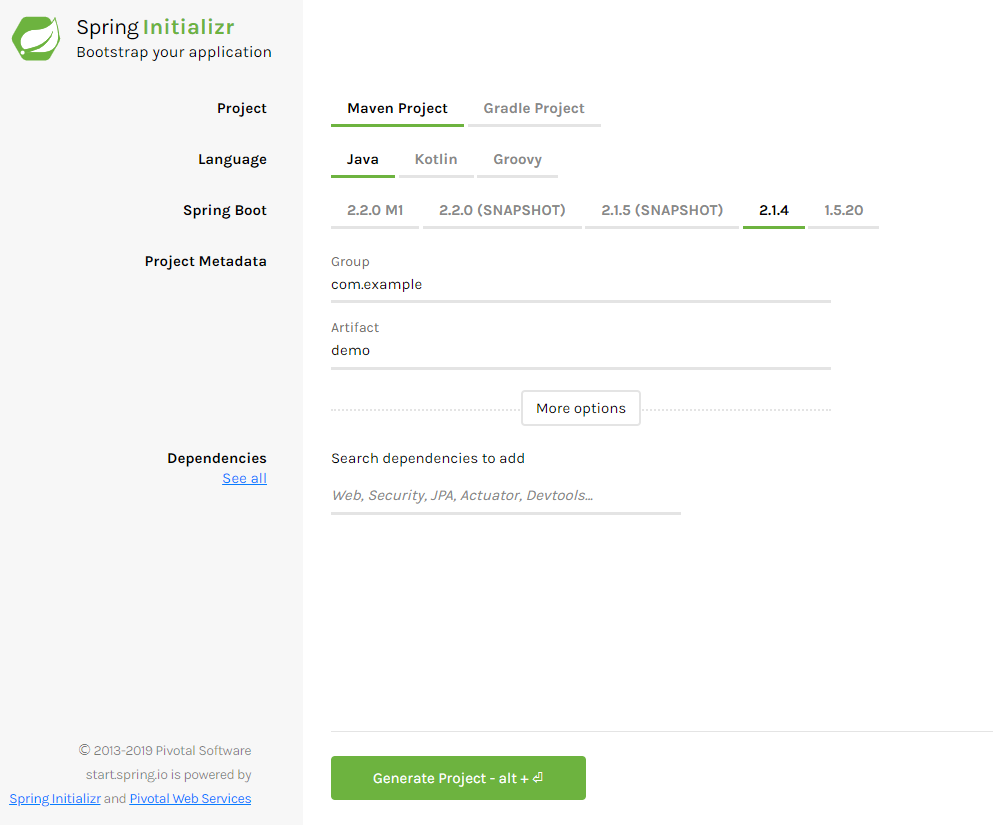


图3-1 Spring boot 初始化界面

填写Spring boot版本、构建工具、项目信息，以及开发所需要的功能模块

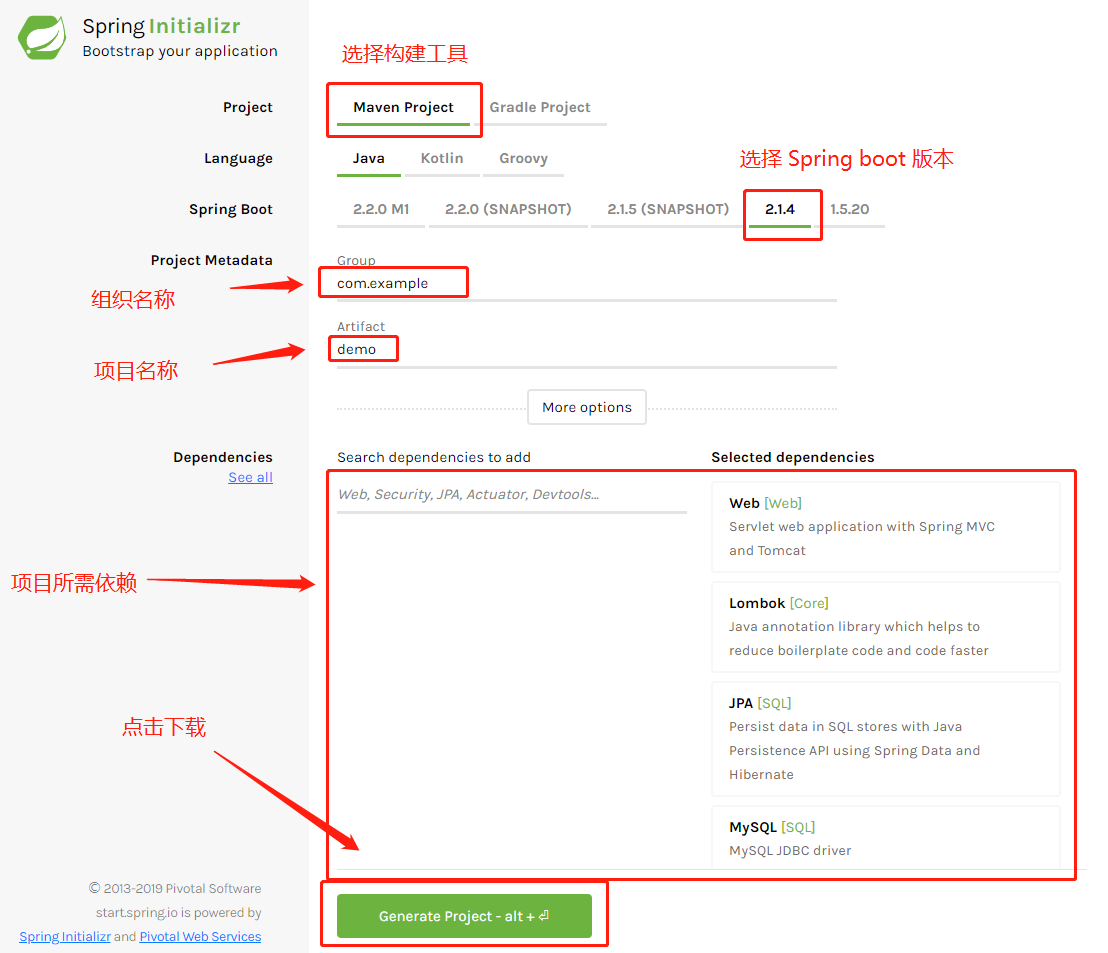


图3-2 Spirng boot 填写信息界面

生成的Spring boot项目工程

src：Java代码文件和测试Java代码文件以及资源。

.mvn：Maven工程管理工具Jar包。

.gitignore：存放版本控制工具排除跟踪文件的正则表达式。

Pom.xml：Maven工程依赖管理文件。

其他文件：工具命令。

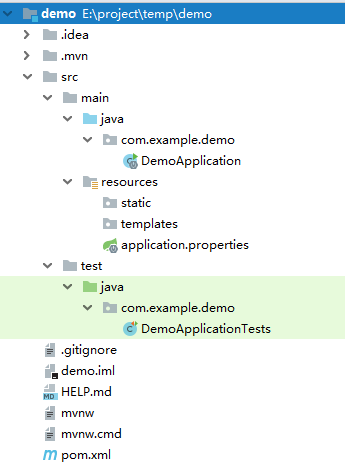


图3-3 工程结构图

* + 1. 前端开发流程

打开命令行，使用Vue CLI生成项目。

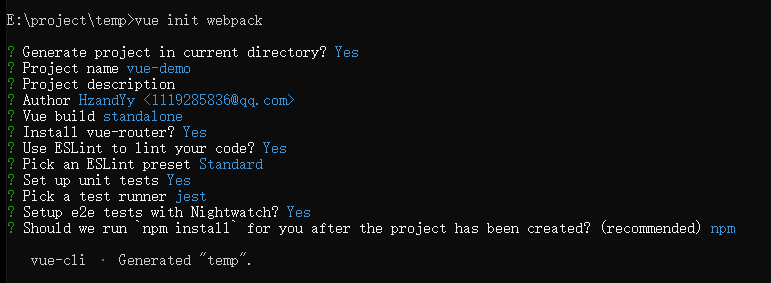


图3-4 Vue初始化图

Vue.CLI生成工程目录。

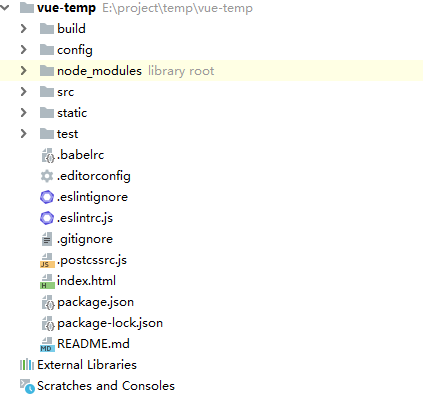


图3-5 Vue工程结构图

测试是否正确。

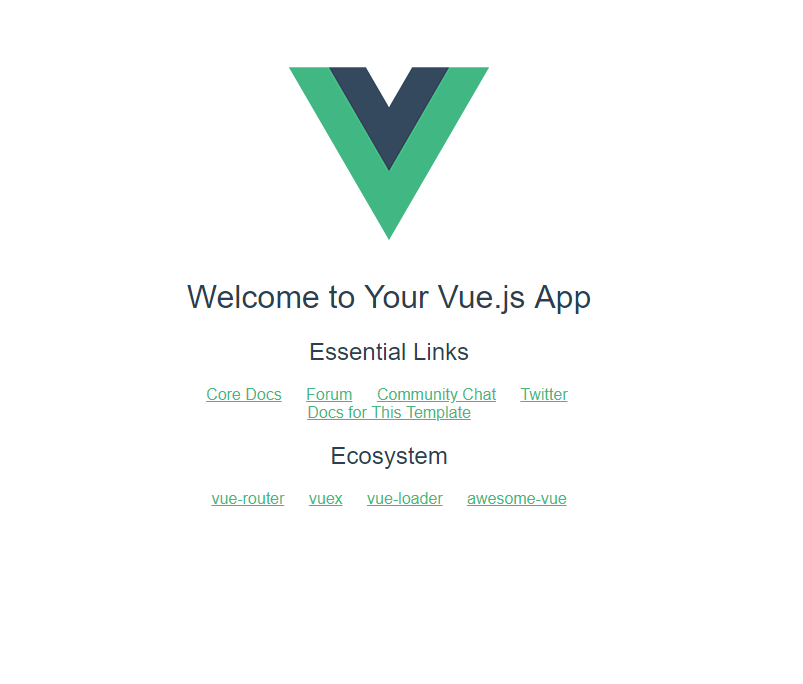


图3-6 Vue运行图

* 1. 后端项目详细介绍
     1. 工程目录结构

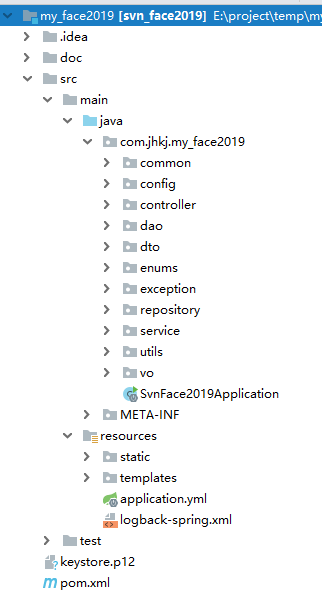


图3-7 实际项目工程结构图

* + 1. 模块简介

common包：存放AOP程序。

config包：存放用户缓存、Token过滤器、默认Token、图片验证码生成器、权限框架认证授权配置。

controller包：存放功能控制器类。

dao包：存放数据库实体、数据库数据信息类。

dto包：存放API请求参数类与控制器与服务层之间的数据交互类。

enums包：存放各种状态位枚举。

exception包：存放自定义异常类。

repository包：存放数据库操作接口。

service包：存放功能服务接口和功能服务接口实现类。

utils包：存放全局功能类。

vo包：存放HTTP响应数据类。

application.yml：全局配置类。

keystore.p12：HTTPS加密密钥。

pom.xml：依赖管理文件。

* + 1. 模块依赖

依赖项目如下

// Spring boot 核心包

spring-boot-starter-aop

spring-boot-starter-tomcat

spring-boot-starter-data-jpa

spring-boot-starter-data-redis

spring-boot-devtools

spring-boot-starter-test

springfox-swagger2

springfox-swagger-ui

pagehelper

// shiro 认证授权包

shiro-all

shiro-redis

// 数据持久化包

jedis

mysql-connector-java

//工具包

zip4j

poi

poi-ooxml

guava

kaptcha

java-jwt

fastjson

commons-io

commons-lang3

commons-fileupload

encrypt-body-spring-boot-starter

lombok

* + 1. 日志模块

能通过对日志进行统计、分析、综合，就能有效地掌握网站运行状况，发现和排除错误原因，了解客户访问分布等，更好的加强系统的维护和管理。所以我们在本项目中使用Spring AOP的环绕增强为每个控制器添加日志功能。

以下是代码实现。

// 定义切点

@Pointcut("execution(\* com.jhkj.my\_face2019.controller.system.\*.\*(..))")

public void webLog() {

}

// 定义切面和结合切点以及增强逻辑实现

@Around(value = "webLog()")

public Object doAroundAdvice(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {

HttpServletRequest request = ((ServletRequestAttributes) RequestContextHolder.getRequestAttributes()).getRequest();

Signature signature = proceedingJoinPoint.getSignature();

LogEntity logEntity = new LogEntity();

//获取ip

String ip = Tools.getClientIp();

//获取url

String url = request.getRequestURL().toString();

log.info("请求的URL：" + url);

log.info(signature.getDeclaringTypeName() + "的" + signature.getName() + "方法执行开始");

long startTime = System.currentTimeMillis();

try {

//获取执行方法的返回值

Object object = proceedingJoinPoint.proceed();

long endTime = System.currentTimeMillis();

log.info("程序运行时间：" + (endTime - startTime) + "ms");

//如果是执行的页面跳转方法，返回值为String不为JsonData

if (object instanceof String) {

String path = (String) object;

return path;

}

if (object instanceof HashMap) {

HashMap path = (HashMap) object;

return path;

}

ResponseResult result = (ResponseResult) object;

log.info(signature.getDeclaringTypeName() + "的" + signature.getName() + "方法执行结束");

//存储日志信息

logEntity.setRequestURL(url);

logEntity.setOperatorIp(ip);

logEntity.setValueCode(result != null ? LogUtil.setValueCode(result.getStatus()) : "");

// 以功能为划分进行日志存储

setLogInfo(logEntity, signature);

return result;

} finally {

//只有当日志类型不为空时，才进行记录

if (!StringUtils.isEmpty(logEntity.getLogType())) {

logRepository.save(logEntity);

}

}

}

* + 1. 缓存模块

持久化的数据如果每次都从数据库上去读，那样造成的IO负担是相当大的，极大的影响程序性能，故使用缓存机制。本项目使用Redis数据库作为缓存。

以下是代码实现。

// 让Spring boot 读取这个配置

@Configuration

public class RedisCacheConfig extends CachingConfigurerSupport {

// 声明为一个Spring 的Bean 且读取application.yml的spring.redis 配置文件

// 得到一个与Redis的连接

@Bean

@ConfigurationProperties("spring.redis")

public JedisConnectionFactory jedisConnectionFactory(){

return new JedisConnectionFactory();

}

// 配置Redis Key的生成策略

@Bean

@Override

public KeyGenerator keyGenerator(){

return (target, method, params) -> {

StringBuilder sb = new StringBuilder();

sb.append(target.getClass().getName());

sb.append(method.getName());

for (Object obj : params) {

sb.append(obj.toString());

}

return sb.toString();

};

}

// 管理缓存

@Bean

public CacheManager cacheManager(RedisConnectionFactory connectionFactory){

//此处可以设置缓存过期时间

RedisCacheManager rcm = RedisCacheManager.create(connectionFactory);

return rcm;

}

// 配置Redis 模板 进行redis操作 与JDBC Template类似

@Bean

public RedisTemplate<Object, Object> redisTemplate(JedisConnectionFactory jedisConnectionFactory) {

RedisTemplate<Object, Object> template = new RedisTemplate<>();

// 设置数据库连接

template.setConnectionFactory(jedisConnectionFactory);

// 设置key 序列化方案

template.setKeySerializer(new StringRedisSerializer());

// 设置 value 序列化方案

template.setValueSerializer(new GenericJackson2JsonRedisSerializer());

// 初始化操作

template.afterPropertiesSet();

return template;

}

}

* + 1. 跨域处理模块

跨域处理模块：由于浏览器对安全有考虑，浏览器对JavaScript施加了跨域限制。所以在服务端为响应添加了跨域支持。默认是没有的，需要自行添加，故添加处理跨域的过滤器。

以下是代码实现。

@Component

public class CorsFilter implements Filter {

@Override

public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) throws IOException, ServletException {

HttpServletResponse response = (HttpServletResponse) servletResponse;

HttpServletRequest request = (HttpServletRequest) servletRequest;

String origin = request.getHeader("Origin");

response.setHeader("Access-Control-Allow-Origin", origin);

response.setHeader("Access-Control-Allow-Methods", "POST, GET, OPTIONS, DELETE");

response.setHeader("Access-Control-Max-Age", "3600");

response.setHeader("Access-Control-Allow-Headers", "Content-Type, Access-Control-Allow-Headers, Authorization, X-Requested-With");

response.setHeader("Access-Control-Allow-Credentials", "true");

String method = request.getMethod();

if (method.equalsIgnoreCase("OPTIONS")) {

servletResponse.getOutputStream().write("Success".getBytes("utf-8"));

} else {

filterChain.doFilter(servletRequest, servletResponse);

}

}

}

* + 1. 验证码模块

图片验证码生成器模块：如果没有验证码，如果不怀好意的使用者恶意登录或获取人员信息，故添加验证码，不法分子还要恶意登录难度就会高很多了。本项目使用的是Google下的kaptcha生成的验证码。

以下是代码实现。

@Component

public class KaptchaConfig {

@Bean

public DefaultKaptcha getDefaultKaptcha() {

com.google.code.kaptcha.impl.DefaultKaptcha defaultKaptcha = new com.google.code.kaptcha.impl.DefaultKaptcha();

Properties properties = new Properties();

// 图片边框

properties.setProperty("kaptcha.border", "yes");

// 设置颜色

properties.setProperty("kaptcha.border.color", "7,84,74");

// 字体颜色

properties.setProperty("kaptcha.textproducer.font.color", "red");

// 图片宽

properties.setProperty("kaptcha.image.width", "180");

// 图片高

properties.setProperty("kaptcha.image.height", "50");

// 字体大小

properties.setProperty("kaptcha.textproducer.font.size", "30");

// 验证码长度

properties.setProperty("kaptcha.textproducer.char.length", "4");

// 字体

properties.setProperty("kaptcha.textproducer.font.names", "宋体,楷体,微软雅黑");

Config config = new Config(properties);

defaultKaptcha.setConfig(config);

return defaultKaptcha;

}

}

* + 1. 认证授权模块

认证授权模块：一套系统是否安全，在认证授权是非常重要的。特别是像现在的RPC风格服务器、RESTFul风格服务器，都是基于Web API的服务器。如果没有认证授权，那么服务器资源将非常的危险。本项目使用的是Apache下的Shiro框架，认证是基于的用户名密码和Token两种认证模式，授权是以RBAC（基于角色的权限访问控制）的授权方式。

以下是代码实现。

// 下面是一段验证逻辑

// 封装成Shiro能够验证的Token

JwtToken token = new JwtToken(null, dto.getUsername(), dto.getPassword());

// 得到用户主体

Subject subject = SecurityUtils.getSubject();

try {

// 登录,即身份验证

// Subject.login(token) 会根据配置的属性 验证用户身份以及授权给用户 并返回主体认证、授权信息

subject.login(token);

// 根据认证信息 确定是否是真正的用户

if (!subject.isAuthenticated()) {

return ResponseResult.e(ResponseCode.SIGN\_IN\_INPUT\_FAIL);

}

} catch (DisabledAccountException e) {

return ResponseResult.e(ResponseCode.SIGN\_IN\_INPUT\_FAIL);

} catch (Exception e) {

return ResponseResult.e(ResponseCode.SIGN\_IN\_FAIL);

}

以下是Token配置

public class JwtToken implements AuthenticationToken {

private String token;

private String username;

private String password;

private String uid;

public JwtToken(String token, String username, String password) {

this.token = token;

this.username = username;

this.password = password;

}

@Override

public Object getPrincipal() {

return token;

}

@Override

public Object getCredentials() {

return token;

}

}

以下是认证流程和授权流程

public class MyRealm extends AuthorizingRealm {

@Autowired

private UserService userService;

@Autowired

private CacheManager cacheManager;

/\*\*

\* 验证此Token 是否为一个有效的Token

\* 有效Token 都会实现一个 AuthenticationToken 接口

\*/

@Override

public boolean supports(AuthenticationToken token) {

return token instanceof JwtToken;

}

/\*\*

\* 授权认证

\*/

@Override

protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken authenticationToken) throws AuthenticationException {

JwtToken token = (JwtToken) authenticationToken;

UserEntity user = new UserEntity();

//UsernamePasswordToken对象用来存放提交的登录信息

String username = token.getUsername() != null ? token.getUsername() : JwtUtil.getUsername(token.getToken());

try {

user = userService.findUserByUsername(username);

} catch (RequestException e) {

throw new DisabledAccountException(e.getMsg());

}

if (user == null) {

throw new DisabledAccountException("用户不存在");

}

if (user.getStatus() != 1) {

throw new DisabledAccountException("用户账户已经锁定,暂时无法登陆");

}

if (token.getUsername() == null) {

token.setUsername(user.getUsername());

}

String sign = JwtUtil.sign(user.getId(), user.getUsername(), user.getPassword());

if (token.getToken() == null) {

token.setToken(sign);

}

token.setUid(user.getId());

return new SimpleAuthenticationInfo(token, user.getPassword(), user.getId());

}

/\*\*

\* 获取用户权限信息

\*/

@Override

protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principalCollection) {

log.info("Shiro权限验证执行");

JwtToken jwtToken = new JwtToken();

BeanUtils.copyProperties(principalCollection.getPrimaryPrincipal(), jwtToken);

if (jwtToken.getUsername() != null) {

//权限信息对象info,用来存放查出的用户的所有的角色（role）及权限（permission）,此消息放到

//Redis数据库中进行存储

SimpleAuthorizationInfo info = new SimpleAuthorizationInfo();

// Shiro有时间间隔机制，2分钟内不会重复执行该方法

UserEntity findUser = userService.findByUserName(jwtToken.getUsername(), true);

if (findUser != null) {

if (findUser.getRoles() != null) {

findUser.getRoles().forEach(roleEntity -> {

//用户的角色集合

info.addRole(roleEntity.getName());

if (roleEntity.getResources() != null) {

roleEntity.getResources().forEach(v -> {

if (!"".equals(v.getPermission().trim())) {

//用户的角色对应的所有权限

info.addStringPermission(v.getPermission());

}

});

}

});

}

return info;

}

}

throw new DisabledAccountException("用户信息异常，请重新登录！");

}

public void clearAuthByUserId(String uid, Boolean author, Boolean out) {

//获取所有session

Cache<Object, Object> cache = cacheManager

.getCache(MyRealm.class.getName() + ".authorizationCache");

cache.remove(uid);

}

public void clearAuthByUserIdCollection(List<String> userList, Boolean author, Boolean out) {

Cache<Object, Object> cache = cacheManager

.getCache(MyRealm.class.getName() + ".authorizationCache");

userList.forEach(cache::remove);

}

}

* + 1. 控制器模块

HTTP协议中，主机后的路径就代表资源的具体路径，为了区分不同的资源所以将不同的功能资源放到了不同的URI上面。本项目实际的控制器采用的是Spring MVC 项目的DispatchServlect，将请求分发到不同的URI上面。

以下是代码实现。

// 让Spring 知道这是一个控制器

@Controller

// 此控制器的前缀

@RequestMapping("/demo")

public class DemoController {

// 这个方法会处理 http://host:port/demo/test 的请求

@RequestMapping("/test")

public String test() {

return "test";

}

}

* + 1. 数据库设计

|  |  |  |  |
| --- | --- | --- | --- |
| 设备表 | t\_equipment |  |  |
| 字段 | 类型 | 特殊 | 备注 |
| id | bingint | 自增 | 主键 |
| equipment\_number | varchar(255) | NOT NULL | 设备编号 |
| secretKey | varchar(255) | NOT NULL | 设备密钥 |
| status | int | NOT NULL | 状态 |
| model\_number | varchar(255) | NOT NULL | 设备型号 |
| software\_version | varchar(255) | NOT NULL | 软件版本 |
| equipment\_port | varchar(255) | NOT NULL | 设备端口号 |
| account | varchar(255) | NOT NULL | 登录账号 |
| password | varchar(255) | NOT NULL | 登录密码 |
| camera\_type | varchar(255) | NOT NULL | 摄像头类型 |
| coordinate | varchar(255) | NOT NULL | 坐标 |
| organization\_id | bigint | NOT NULL | 组织Id |
| gmt\_create | datetime | NOT NULL | 创建时间 |
| gmt\_modified | datetime | NOT NULL | 上次修改时间 |

|  |  |  |  |
| --- | --- | --- | --- |
| 人脸图库表 | t\_face\_user |  |  |
| 字段 | 类型 | 特殊 | 备注 |
| id | bingint | 自增 | 主键 |
| image\_status | int | NOT NULL | 人脸图片状态 |
| register\_status | int | NOT NULL | 图片注册状态 |
| thumbnail\_path | varchar(255) | NOT NULL | 微缩图路径 |
| real\_image\_path | varchar(255) | NOT NULL | 实际图路径 |
| characteristic\_file\_path | varchar(255) | NOT NULL | 特征文件路径 |
| map\_depot\_version | varchar(255) | NOT NULL | 人脸图库版本 |
| user\_id | bingint | NOT NULL | 员工Id |
| gmt\_create | datetime | NOT NULL | 创建时间 |
| gmt\_modified | datetime | NOT NULL | 上次修改时间 |

|  |  |  |  |
| --- | --- | --- | --- |
| 日志表 | t\_log |  |  |
| 字段 | 类型 | 特殊 | 备注 |
| id | bingint | 自增 | 主键 |
| log\_type | int | NOT NULL | 日志类型 |
| log\_title | int | NOT NULL | 日志标题 |
| value\_code | varchar(255) | NOT NULL | 操作过后的信息 |
| request\_url | varchar(255) | NOT NULL | 请求的URL |
| operator | varchar(255) | NOT NULL | 操作人员账号 |
| operator\_ip | bigint | NOT NULL | 操作人员IP |
| gmt\_create | datetime | NOT NULL | 创建时间 |
| gmt\_modified | datetime | NOT NULL | 上次修改时间 |

|  |  |  |  |
| --- | --- | --- | --- |
| 组织结构表 | t\_organization |  |  |
| 字段 | 类型 | 特殊 | 备注 |
| id | bingint | 自增 | 主键 |
| organization\_number | varchar(255) | NOT NULL | 组织编号 |
| name | varchar(255) | NOT NULL | 组织名称 |
| zip\_code | varchar(255) | NOT NULL | 组织邮编 |
| location | varchar(255) | NOT NULL | 组织地址 |
| description | varchar(255) | NOT NULL | 组织介绍 |
| sort\_weight | varchar(255) | NOT NULL | 排序权重 |
| admin\_name | varchar(255) | NOT NULL | 负责人姓名 |
| admin\_tel | varchar(255) | NOT NULL | 负责人电话 |
| status | int | NOT NULL | 组织状态 |
| type | int | NOT NULL | 组织类型 |
| level | int | NOT NULL | 组织层级 |
| path | varchar(255) | NOT NULL | 组织层级路径 |
| parent\_id | varchar(255) | NOT NULL | 父级组织ID |
| gmt\_create | datetime | NOT NULL | 创建时间 |
| gmt\_modified | datetime | NOT NULL | 上次修改时间 |

|  |  |  |  |
| --- | --- | --- | --- |
| 用户表 | t\_user |  |  |
| 字段 | 类型 | 特殊 | 备注 |
| id | bigint | 主键 | 自增 |
| organization\_path | varchar(255) | NOT NULL | 所属组织 |
| username | varchar(255) | NOT NULL | 登录用户名 |
| name | varchar(255) | NOT NULL | 用户姓名 |
| password | varchar(255) | NOT NULL | 登陆密码 |
| tel | varchar(255) | NOT NULL | 电话 |
| positioin | varchar(255) | NOT NULL | 职位 |
| positioin\_title | varchar(255) | NOT NULL | 岗位名称 |
| status | int | NOT NULL | 状态 |
| weight\_sort | int | NOT NULL | 排序权重 |
| face\_status | int | NOT NULL | 人脸信息注册状态 |
| gmt\_create | datetime | NOT NULL | 创建时间 |
| gmt\_modified | datetime | NOT NULL | 上次修改时间 |

|  |  |  |  |
| --- | --- | --- | --- |
| 资源表 | t\_resource |  |  |
| 字段 | 类型 | 特殊 | 备注 |
| id | bigint | 主键 | 主键 |
| model\_name | varchar(255) | NOT NULL | 资源名称 |
| sort\_weight | int | NOT NULL | 排序权重 |
| url | varchar(255) | NOT NULL | 请求URL |
| permission | varchar(255) | NOT NULL | shiro验证权限 |
| parent\_id | bigint | NOT NULL | 父级id |
| type | int | NOT NULL | 资源类型 |
| verification | varchar(255) | NOT NULL | 是否开启权限验证 |
| remark | varchar(255) | NOT NULL | 备注 |
| gmt\_create | datetime | NOT NULL | 创建时间 |
| gmt\_modified | datetime | NOT NULL | 上次修改时间 |

|  |  |  |  |
| --- | --- | --- | --- |
| 角色表 | t\_role |  |  |
| 字段 | 类型 | 特殊 | 备注 |
| id | bigint | 主键 | 主键 |
| name | varchar(255) | NOT NULL | 权限名称 |
| remark | varchar(255) | NOT NULL | 权限备注 |
| gmt\_create | datetime | NOT NULL | 创建时间 |
| gmt\_modified | datetime | NOT NULL | 上次修改时间 |

|  |  |  |  |
| --- | --- | --- | --- |
| 资源权限表 | t\_resource\_role |  |  |
| 字段 | 类型 | 特殊 | 备注 |
| id | bigint | 主键 | 自增主键 |
| role\_id | bigint | 外键 | 角色表外键 |
| resource\_id | bigint | 外键 | 资源表外键 |

* + 1. DTO模块

数据传输对象（Data Transfer Object），提供粗粒度的数据实体。给各个控制器，返回类型提供一个粗粒度的数据实体，方便开发人员使用，更体现出面向对象的思想。以下是一个例子。

以下是代码实现。

@Data

public class SignInDTO {

@NotBlank(message = "用户名不可以为空！")

private String username;

@NotBlank(message = "密码不可以为空！")

@Pattern(regexp = "^(\\w){6,18}$",message = "密码应为[A-Za-z0-9\_]组成的6-18位字符！")

private String password;

@NotBlank(message = "验证码不可以为空!")

private String kaptcha;

}

* + 1. 异常模块

Exception模块：程序时常会发生一些不被期望的事情，会阻止程序按照预期正常运行。当在单线程应用中，发生了异常，程序就会退出终止。Java中提供了优秀的解决方法，当程序异常发生时，可以按照程序预先设定好的异常处理逻辑，针对性地处理异常，让程序尽最大可能恢复正常并继续执行，且保持代码的清晰。

以下是代码实现。

// 定义一个异常

public class EntityNotFoundException extends Exception {

private String message;

public EntityNotFoundException(String message) {

super(message);

}

}

异常处理机制

try(InputStream inputStream = new FileInputStream("/filepath/filename")) {

// 当读取文件时，可能会发生文件不存在异常

// 还可能会发生读入输出异常

// 因此 Java 可以在程序中预先设定异常处理逻辑

} catch (FileNotFoundException e) {

// 发生异常时 打印方法栈

e.printStackTrace();

} catch (IOException e) {

// 发生异常时 打印方法栈

e.printStackTrace();

} finally {

// 此处方法无论如何都会执行

}

* + 1. 数据库操作模块

Repository接口是Sprng-data项目下面的一个主要接口，Spring-data的目的是简化数据库访问,并支持云服务的开源框架。本项目主要用于处理关系型数据库中的数据库操作。

以下是代码实现。

@Repository

public interface UserRepository extends JpaRepository<UserEntity,String>, JpaSpecificationExecutor<UserEntity> {

/\*\*

\* 根据员工姓名查询员工信息

\*

\* @param username

\* @return 查询到的员工信息

\*/

Optional<UserEntity> findByUsername(String username);

/\*\*

\* 通过用户名和手机号来查找用户

\* @param username

\* @param tel

\* @return

\*/

Optional<UserEntity> findByUsernameAndTel(String username,String tel);

/\*\*

\* 批量更新人员的状态

\* @param status

\* @param id

\*/

@Modifying

@Transactional

@Query("update UserEntity c set c.status = ?1 where c.id in ?2")

void statusChange(Integer status, List<String> id);

/\*\*

\* 单个改变人员状态

\* @param status

\* @param id

\*/

@Modifying

@Transactional

@Query("update UserEntity c set c.status = ?1 where c.id = ?2")

void statusChangeSig(Integer status,String id);

/\*\*

\* 批量删除人员信息

\* @param ids

\*/

@Modifying

@Transactional

@Query("delete from UserEntity s where s.id in (?1)")

void deleteBatch(String[] ids);

/\*\*

\* 根据人员id更新部门地址

\* @param path

\* @param id

\*/

@Transactional

@Modifying

@Query("update UserEntity c set c.organizationPath = ?1 where c.id = ?2")

void updatePathById(String path, String id);

/\*\*

\* 查询所有人员状态为0的用户

\* @return

\*/

@Query("select e from UserEntity e where e.userType = '0' order by e.weightSort asc ")

List<UserEntity> findNormalUser();

}

* + 1. Service模块

为了体现面向对象的思想，以及提高程序的扩展性以及封装性，为此，提供了service模块，将所有的实际功能逻辑都提供出一个接口方法，然后自己实现此接口。依靠Spring的依赖注入，对外程序只提供接口，实现类由Spring boot注入。这样的好处是如果修改了实现逻辑，只需要将注入的实现类修改即可，不用修改非service层的代码。体现了优质代码的高内聚、低耦合的特点。

以下是代码实现。

// 声明一个接口 管理组织结构

public interface OrganizationStructureService {

/\* 初始化树. \*/

OrganizationVO initStructureTree();

/\* 创建子节点. \*/

OrganizationVO createStructureChildNode(OrganizationDTO dto) throws NodeException;

/\* 删除子节点. \*/

OrganizationVO deleteStructureNode(OrganizationDTO dto) throws NodeException;

/\* 查询节点. \*/

OrganizationVO queryStructureNode(OrganizationDTO dto) throws NodeException;

/\* 更新节点. \*/

OrganizationVO updateStructureNode(OrganizationDTO dto) throws NodeException;

/\*\*

\* 查找一个单位

\* @param id

\* @return

\*/

OrganizationEntity findStructureById(Long id);

}

以下是接口实现

// 声明为一个Spring 的Bean

// Service 表示它是一个服务

@Service

public class OrganizationServiceImpl implements OrganizationService {

private static final String NODE\_NOT\_FOUND = "未找到该节点";

private static final String NODE\_NOT\_FOUND\_PARENT = "未找到该节点的父节点";

private static final String NODE\_DELTED = "该节点已被删除";

@Autowired

private OrganizationRepository organizationRepository;

@Override

public OrganizationVO initStructureTree() {

Sort sort = new Sort(Sort.Direction.ASC, "type");

List<OrganizationEntity> queryData = organizationRepository.findNormalEntities(sort);

OrganizationVO root = new OrganizationVO();

root.setName("root");

root.setParentId(0L);

root.setId(0L);

root.setChild(buildTree(queryData));

return root;

}

@Override

public OrganizationVO createStructureChildNode(OrganizationDTO dto) throws NodeException {

long parentId = dto.getParentId();

String parentPath = "";

OrganizationEntity entity = new OrganizationEntity();

BeanUtils.copyProperties(dto, entity);

// 当是顶级时 父ID设置为0

if (parentId == 0) {

} else {

OrganizationEntity parentNode;

Optional<OrganizationEntity> optional = isPresent(parentId, NODE\_NOT\_FOUND\_PARENT);

parentNode = optional.get();

parentNode.getChildOrganizations().add(entity);

entity.setParentOrganization(parentNode);

parentPath = parentNode.getPath();

}

organizationRepository.save(entity);

if (parentId == 0) {

entity.setPath(entity.getName());

} else {

entity.setPath(parentPath + ">" + entity.getName());

}

organizationRepository.save(entity);

OrganizationVO result = new OrganizationVO();

BeanUtils.copyProperties(entity, result);

return result;

}

@Override

public OrganizationVO deleteStructureNode(OrganizationDTO dto) throws NodeException {

Optional<OrganizationEntity> optional = isPresent(dto.getId(), NODE\_NOT\_FOUND);

OrganizationEntity entity = optional.get();

entity.setStatus(OrganizationStatusEnum.DELETE);

organizationRepository.save(entity);

OrganizationVO vo = new OrganizationVO();

BeanUtils.copyProperties(entity, vo);

return vo;

}

@Override

public OrganizationVO queryStructureNode(OrganizationDTO dto) throws NodeException {

Optional<OrganizationEntity> optional = isPresent(dto.getId(), NODE\_NOT\_FOUND);

OrganizationVO vo = new OrganizationVO();

OrganizationEntity entity = optional.get();

vo.setType(entity.getType().name());

vo.setStatus(entity.getStatus().name());

WebUtil.copyNonNullProperties(entity, vo);

return vo;

}

@Override

public OrganizationVO updatStructureeNode(OrganizationDTO dto) throws NodeException {

Optional<OrganizationEntity> optional = isPresent(dto.getId(), NODE\_NOT\_FOUND);

OrganizationEntity entity = optional.get();

WebUtil.copyNonNullProperties(dto, entity);

organizationRepository.save(entity);

OrganizationVO result = new OrganizationVO();

BeanUtils.copyProperties(entity, result);

return result;

}

@Override

public OrganizationEntity findStructureById(Long id) {

return organizationRepository.findById(id).get();

}

private Optional<OrganizationEntity> isPresent(long id, String message) throws NodeException {

Optional<OrganizationEntity> optional = organizationRepository.findById(id);

if (!optional.isPresent())

throw new NodeException(message);

OrganizationEntity entity = optional.get();

if (OrganizationStatusEnum.DELETE.equals(entity.getStatus()))

throw new NodeException(NODE\_DELTED);

return optional;

}

private List<OrganizationVO> buildTree(List<OrganizationEntity> organizationEntities) {

List<OrganizationVO> treeNodes = new ArrayList<>();

List<OrganizationVO> rootNodes = getRootNodes(organizationEntities);

for (OrganizationVO vo : rootNodes) {

buildChildNodes(vo, organizationEntities);

treeNodes.add(vo);

}

return treeNodes;

}

/\*\*

\* 递归子节点

\*

\* @param vo

\*/

private void buildChildNodes(OrganizationVO vo, List<OrganizationEntity> organizationEntities) {

List<OrganizationVO> child = getChildNodes(vo, organizationEntities);

if (!child.isEmpty()) {

for (OrganizationVO item : child) {

buildChildNodes(item, organizationEntities);

}

vo.setChild(child);

}

}

/\*\*

\* 获取父节点下所有的子节点

\*

\* @param parentVo

\* @return

\*/

private List<OrganizationVO> getChildNodes(OrganizationVO parentVo, List<OrganizationEntity> organizationEntities) {

long parentId = parentVo.getId();

List<OrganizationVO> childNodes = new ArrayList<>();

for (OrganizationEntity entity : organizationEntities) {

if (parentId == entity.getParentOrganization().getId()) {

OrganizationVO vo = new OrganizationVO();

vo.setParentId(parentId);

vo.setId(entity.getId());

vo.setName(entity.getName());

vo.setStatus(entity.getStatus().name());

vo.setType(entity.getType().name());

vo.setAdminName(entity.getAdminName());

vo.setAdminTel(entity.getAdminTel());

childNodes.add(vo);

}

}

return childNodes;

}

/\*\*

\* 判断是否为根节点

\*

\* @param entity

\* @return

\*/

public boolean rootNode(OrganizationEntity entity) {

boolean isRoot = false;

if (OrganizationTypeEnum.COMPANY.equals(entity.getType())) {

isRoot = true;

}

return isRoot;

}

/\*\*

\* 构建根节点

\*

\* @param organizationEntities

\* @return

\*/

public List<OrganizationVO> getRootNodes(List<OrganizationEntity> organizationEntities) {

List<OrganizationVO> rootNodes = new ArrayList<>();

Iterator<OrganizationEntity> iterator = organizationEntities.iterator();

while (iterator.hasNext()) {

OrganizationEntity entity = iterator.next();

if (rootNode(entity)) {

OrganizationVO vo = new OrganizationVO();

vo.setId(entity.getId());

vo.setName(entity.getName());

vo.setParentId(0L);

rootNodes.add(vo);

iterator.remove();

}

}

return rootNodes;

}

}

* + 1. 启动类

由于Spring boot内嵌Tomcat，所以就不需要打包成war包再部署到Tomcat里。所以需要一个启动类。同时它也是Spring boot依赖注入的核心，默认扫描他目录及子目录下的Spring 注解。

以下是代码实现。

// 以下为实现类

@SpringBootApplication

public class SvnFace2019Application extends SpringBootServletInitializer {

public static void main(String[] args) {

SpringApplication.run(SvnFace2019Application.class, args);

}

}

* 1. 前端项目详细介绍
     1. SPA介绍

本项目采用的SPA，单页应用， 就是只有一张HTML页面的应用。此应用页面不会发生刷新，它的速度和流畅会和原生应用相差不多，所有的数据交互、页面更新全部都由Ajax完成。所有的逻辑都由JavaScript来控制，因此程序的复杂性相较于以前那种Web模式就不是一个级别的了。

* + 1. 模块化

随着web2.0时代的到来，Ajax技术得到广泛应用，jQuery等前端库层出不穷，前端代码日益膨胀，此时在JS方面就会考虑使用模块化规范去管理。所以我们可以将复杂的程序依据一定的规则(规范)封装成几个块(文件), 并进行组合在一起，块的内部数据与实现是私有的, 只是向外部暴露一些接口(方法)与外部其它模块通信。这个和面向对象的编程思想有点异曲同工之妙。

但由于JavaScript的语法特性，最初的模块化命名空间污染严重，复用性、维护性差，直到有了CommonJS模块规范，每个文件就是一个模块，有自己的作用域。在一个文件里面定义的变量、函数、类，都是私有的，对其他文件不可见。这样就不会发生命名污染，且提高了复用性和维护性，而且在运行时，仅会第一次加载，之后会缓存，再次运行时就会直接读取缓存结果，想让模块在此运行则必须清除缓存。这样也就又一次提高了SPA的运行效率和流畅感。

* + 1. Vue

Vue是一套用于构建用户界面的渐进式框架。与其它大型框架不同的是，Vue 被设计为可以自底向上逐层应用。Vue 的核心库只关注视图层，不仅易于上手，还便于与第三方库或既有项目整合。另一方面，当与现代化的工具链以及各种支持类库结合使用时，Vue 也完全能够为复杂的单页应用提供驱动。

组件：如果说CommonJs是JavaScript的一套模块化规范，那么组件就是SPA的一个模块化规范，只不过粒度是由开发人员所决定，可以是整个功能，也可能只是一个按钮。本次项目的粒度只达到了功能模块。

以下是代码实现。

<template>

<div class="login\_wrapper">

<div class="login">

<el-form :model="formLogin">

<el-form-item>

<h2 class="title">后台管理系统</h2>

</el-form-item>

<el-form-item>

<el-input v-model="formLogin.loginName" placeholder="账号"></el-input>

</el-form-item>

<el-form-item>

<el-input type="password" v-model="formLogin.password" placeholder="密码"></el-input>

</el-form-item>

<el-form-item>

<el-input style="width: 50%;display: inline-block" v-model="formLogin.verification\_code"

class="form-control" id="focusedInput" type="text" placeholder="请输入验证码"></el-input>

<img style="display: inline;width: 130px;float: right" v-bind:src="imgUrl" class="img-code"

id="kaptchaImage" @click="change()" title="点击换一张"/>

</el-form-item>

<el-form-item>

<el-button type="primary" @click="login">登录</el-button>

<span v-show="this.errorInfo.isShowError" class='error'>

{{this.errorInfo.text}}

</span>

</el-form-item>

</el-form>

</div>

</div>

</template>

<style lang="scss">

$input\_width: 300px;

.login\_wrapper {

height: 100%;

display: flex;

align-items: center;

justify-content: center;

.login {

width: 460px;

height: 480px;

margin-top: -150px;

border: 1px solid #eaeaea;

box-shadow: 0 0 25px #cac6c6;

.title {

text-align: center;

color: #505458;

}

.el-form-item\_\_content {

width: $input\_width;

}

.el-button {

width: $input\_width;

}

.el-form {

margin: 30px 80px auto 80px;

.error {

display: block;

text-align: center;

color: red;

}

}

}

}

.recover {

position: absolute;

bottom: 0px;

cursor: pointer;

color: #E6A23C;

display: none;

}

.bei {

position: absolute;

bottom: 20px;

cursor: pointer;

color: #505458;

}

</style>

<script>

import apis from '../apis/apis';

export default {

name: 'login',

data() {

return {

formLogin: { //表单对象

loginName: '',

password: '',

verification\_code: ''

},

errorInfo: {

text: '登陆失败,请重试',

isShowError: false //显示错误提示

},

checked: "",

imgUrl: "/api/account/defaultKaptcha?d=" + new Date \* 1

}

},

mounted() {

document.onkeydown = (event) => {

let e = event || window.event || arguments.callee.caller.arguments[0];

if (e && e.keyCode == 130) { // enter 键

this.login();

}

};

},

methods: {

change() {

this.imgUrl = ("/api/account/defaultKaptcha?d=" + new Date \* 1)

},

login() {

//调用后端登陆接口

apis.shiroApi.loginIn(this.formLogin)

.then((data) => {

console.log('success:', data);

if (data && data.data) {

let json = data.data;

if (json.status == '2') {

console.log('登录成功');

sessionStorage.setItem('token', json.data)

this.$router.push({path: "/index"});

return;

} else if (json.message) {

this.errorInfo.text = json.message;

} else if (json.status = '-7') {

this.errorInfo.isShowError = true;

this.errorInfo.text = json.msg;

} else if (json.status = '-4') {

this.errorInfo.isShowError = true;

this.errorInfo.text = '账号或密码错误';

}

}

this.errorInfo.isShowError = true;

this.$store.dispatch("loginLog", loginLog);

}).catch((err) => {

let result = err.response.data.errors[0]

console.log(result.defaultMessage)

this.errorInfo.isShowError = true;

this.errorInfo.text = result.defaultMessage;

});

},

rollBackTables() {

let text = '数据还原';

apis.shiroApi.rollBackTables()

.then(data => {

let alertText = '';

if (data.data.status == 'SUCCESS') {

text += '成功';

alertText = text + ',请重新登陆';

} else {

text += '失败';

alertText = text + ',请重试';

}

this.$alert(alertText, '提示', {

confirmButtonText: '确定',

});

log(text);

})

.catch(e => {

this.$alert('数据还原异常,请重试', '提示', {

confirmButtonText: '确定',

});

text += '失败';

log(text);

});

console.log(text);

function log(text) {

let loginLog = {

ip: returnCitySN["cip"],

city: returnCitySN["cname"] + '-' + text

};

apis.shiroApi.loginLog(loginLog);

}

}

}

}

</script>

* + 1. 路由控制

以前那种网站，浏览器所指URL是真实的后端资源路径，到了SPA里，路由起一个页面导航，数据传参的效果。实际访问的URL全部由JavaScript来决定。由于本项目做了权限访问控制，会根据用户的权限分发给不同的界面，变成了动态路由。

以下是代码实现。

/\*\*

\*

\* 懒加载动态路由

\*/

function lazy(code) {

//return resolve => require(["@/components/" + name + ".vue"], resolve)

return getComponetByPath('/' + code);

}

function getComponetByPath(path) {

for (var i = 0; i < dynamicRouter.length; i++) {

var route = dynamicRouter[i];

if (route.path == path) {

return route.component;

}

}

}

/\*\*

\* 路由定义

\* path跟配置里的资源标示对应,同时跟url地址对应

\*/

const dynamicRouter = [

{

name: 'index',

path: '/index',

component: resolve => require(['@/components/index.vue'], resolve),

meta: {

title: '系统首页'

}

},

{

name: 'staffAdmin',

path: '/staffAdmin',

component: resolve => require(['@/components/tablepage/staffAdmin.vue'], resolve),

meta: {

title: '人员管理'

}

},

{

name: 'faceAdmin',

path: '/faceAdmin',

component: resolve => require(['@/components/tablepage/faceAdmin.vue'], resolve),

meta: {

title: '人脸管理'

}

},

{

name: 'equipmentAdmin',

path: '/equipmentAdmin',

component: resolve => require(['@/components/tablepage/equipmentAdmin.vue'], resolve),

meta: {

title: '设备管理'

}

},

{

name: 'orgAdmin',

path: '/orgAdmin',

component: resolve => require(['@/components/tablepage/orgAdmin.vue'], resolve),

meta: {

title: '组织管理'

}

},

{

name: 'premissionAdmin',

path: '/premissionAdmin',

component: resolve => require(['@/components/tablepage/permissionAdmin.vue'], resolve),

meta: {

title: '权限管理'

}

},

{

name: 'earth',

path: '/earth',

component: resolve => require(['@/components/mapAdmin/mapAdmin.vue'], resolve),

meta: {

title: '离线地图',

}

}

,

{

name: 'log',

path: '/log',

component: resolve => require(['@/components/tablepage/log.vue'], resolve),

meta: {

title: '日志'

}

},

{

name: 'resadmin',

path: '/resadmin',

component: resolve => require(['@/components/tablepage/resadmin.vue'], resolve),

meta: {

title: '树组件页'

}

}

]

export { lazy }

* + 1. 数据交互

原生的JavaScript使用的是XMLHttpRequest对象，但此对象的方法封装不够到位，因此出现了许多网络库，如：JQuery、RxJs、Axios。本项目使用的是Axios。

以下是代码实现。

// 以下是一个功能模块的网路请求。

import Axios from "axios";

/\*\*

\* 人脸信息API

\*/

const token = sessionStorage.getItem("token");

const faceUserURL = "/api/FaceUser";

const header = {

"content-type": "application/json;charset-utf-8",

"Access-Control-Allow-Headers": "Content-Type",

Authorization: token,

};

export default {

/\*\*

\* 通过ID 删除某一个人脸信息

\* @param id

\* @returns {AxiosPromise<any>}

\*/

removeFaceUserItem: function (id) {

const requestURL = faceUserURL + "/remove";

const param = {

id: id

};

const config = {

headers: header

};

return Axios.post(requestURL, param, config);

},

/\*\*

\* 通过ID 查找人脸信息

\* @param id

\* @returns {AxiosPromise<any>}

\*/

findFaceUserInfoById: function (id) {

const requestURL = faceUserURL + "/find";

const param = {

id: id

};

const config = {

headers: header

};

return Axios.post(requestURL, param, config);

},

/\*\*

\* 通过ID 更新实体注册状态

\* @param id

\* @param registerStatus 只能是 YES|NO

\* @returns {AxiosPromise<any>}

\*/

updateFaceUserRegisterStatus: function (id, registerStatus) {

console.log(id + "\t" + registerStatus);

if (registerStatus !== "YES" && registerStatus !== "NO") {

return;

}

const requestURL = faceUserURL + "/register";

const param = {

id: id,

registerStatus: registerStatus

};

const config = {

headers: header

};

return Axios.post(requestURL, param, config);

},

/\*\*

\* 通过version查询人脸信息 可能会有分页

\* @param version

\* @param pageNumber

\* @returns {AxiosPromise<any>}

\*/

getFaceUserInfoListByVersion: function (version, pageNumber,pageSize){

if (pageNumber === undefined || pageNumber === null) {

pageNumber = 0;

}

const requestURL = faceUserURL + "/find/mapVersion";

const param = {

mapDepotVersion: version,

pageNo:pageNumber,

pageSize:pageSize,

};

const config = {

headers: header,

};

return Axios.post(requestURL, param, config);

},

/\*\*

\* 分页获取人脸信息

\* @param pageNumber

\* @returns {AxiosPromise<any>}

\*/

getFaceUserInfoList: function (pageNumber,pageSize) {

if (pageNumber === undefined || pageNumber === null) {

pageNumber = 0;

}

const requestURL = faceUserURL + "/list";

const param = {pageNo: pageNumber,pageSize:pageSize};

const config = {

};

return Axios.post(requestURL, param, config);

}

}

调用网络请求，并更新页面

initPage() {

faceUserApi

.getFaceUserInfoList(this.pageInfo.currentPage,this.pageInfo.pageSize)

.then(res => {

let result = res.data;

this.tableData = result.data.pageData;

this.pageInfo.pageTotal=result.data.total;

});

}

第4章 调试与实现

1. 1. 调试中遇到的重点与难点
      1. 调试重点
2. 功能调试。
   1. 功能是否正确实现。
3. 健壮性调试。
   1. 格式错误的数据是否存在提示。
   2. 是否存在注入攻击。
   3. 通过API访问是否有认证。
4. 用户体验感。
   1. 对于用户的错误操作，是否有能够接受的提示。
   2. 用户操作逻辑是否有问题。
      1. 调试难点
5. 对HTTP协议还不够熟悉，时常把携带数据放错位置。
6. 时常会出现Web页面、服务器日志、数据库数据出现乱码。
7. 服务打不开，端口被占用。
   1. 解决方案
8. 仔细的看了一遍HTTP资料，针对每个API写好了测试文件。
9. 为服务器添加字符集处理过滤器、前端的请求字符集统一设定请求头。
10. 查看端口被什么程序占用，杀死该程序。
    1. 实现展示
       1. 登录页面



图4-1 登录页面

* + 1. 组织结构管理页面



图4-2 组织结构管理页面

* + 1. 设备管理页面

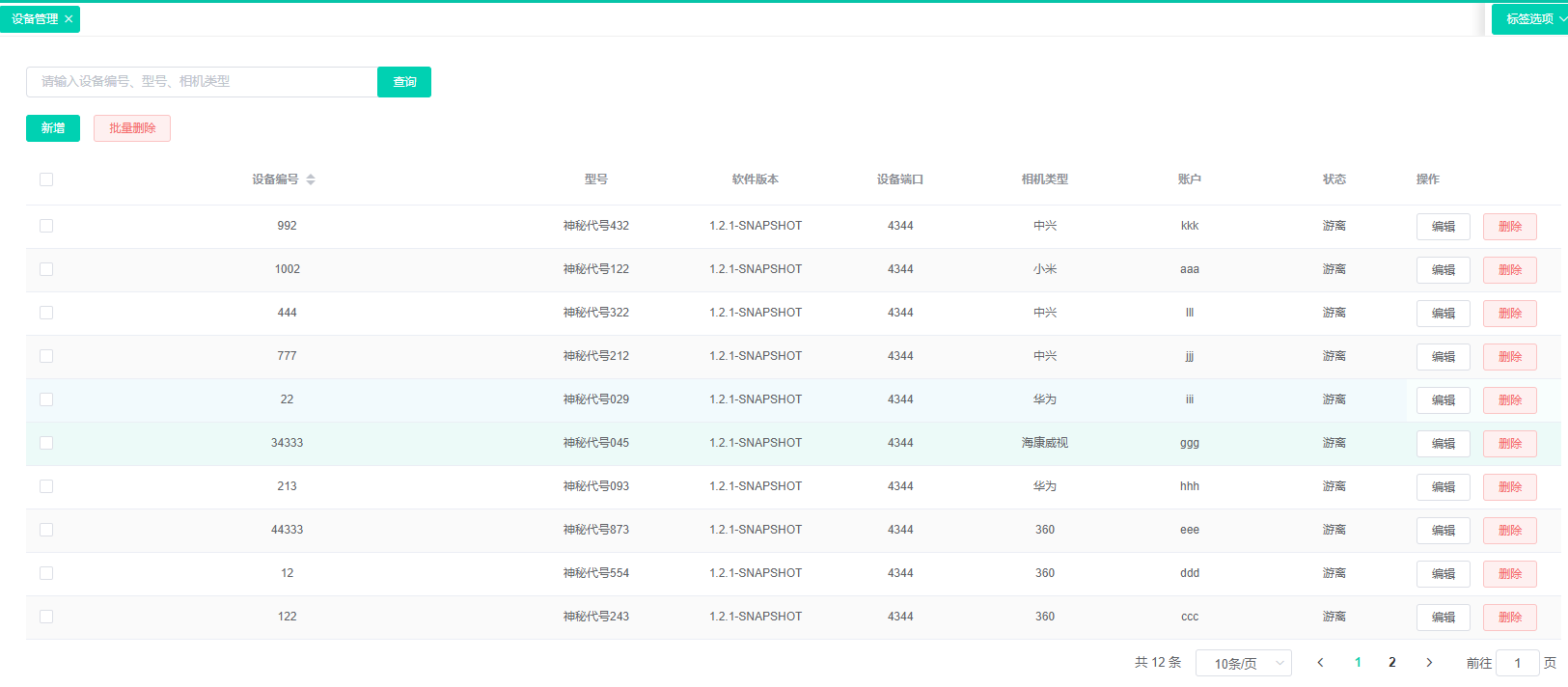


图4-3 设备管理页面

* + 1. 人脸信息管理页面



图4-4 人脸信息管理页面

* + 1. 权限管理页面



图4-5 权限管理页面

* + 1. 日志管理

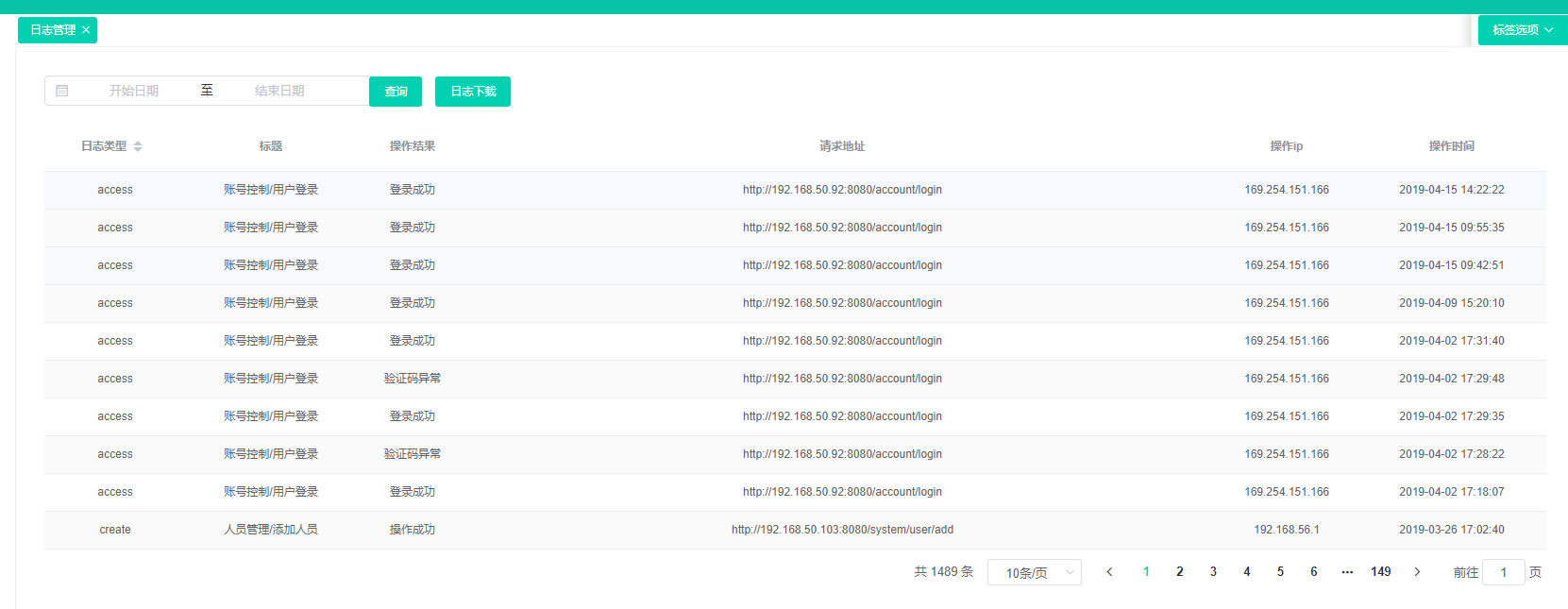


图4-6 日志管理页面

* + 1. 资源结构管理页面



图4-7 资源结构管理页面

第5章 总结

本项目选用技术有Spring boot、Vue.js、Axios，这些框架技术都基础技术在我们看来都对底层技术做了很好的封装。但只是使用这些技术是非常简单的，所以我们应该看到更深层次的东西，这些框架究竟是怎么实现的。

Spring boot，本质上面来讲就是一套Spring框架的全家桶，本项目有使用Spring MVC、Spring AOP、Spring Data JPA。Spring MVC，本质上面是对Java Web那一套体系的封装，使用一个DispatchServlet的类，分发请求得到的请求，个人认为这样做的好处是可以将除了请求这一个需要遵守Java Web的规范外，其他的实现全部都可以由Spring MVC来定义，这样做对修改关闭、对扩展开放就基本上能满足这个思想了。Spring AOP，面向切面编程，使用代理的方法针对实际方法逻辑做了很好的分层，让每个方法只做自己的事情，做到了高内聚、低耦合，极大的提高了程序代码的质量。Spring Data JPA，底层默认使用的Hibernate框架实现，众所周知，Hibernate学习门槛较高，理念繁多、优化困难，但Spring Data JPA大大的简化了这些，内部就把Hibernate的复杂性封装并实现了，大大的降低了使用者的门槛，让使用者仅仅需要很基础的Java知识和SQL知识就能实现自己想要的功能。

Vue.js，非常的简单易用，（本人主攻Angular2+），如果只需要使用它的一些简单的功能，只需要在<script>标签里面引用它的JS类库即可。做单页应用，只需使用脚手架搭建出项目，创建.vue文件即可，相较于本人之前使用的Angular2+，它没了那么多条条款款、规规矩矩。但Vue和Angular2+相比较后，个人发现了一些问题，可能是因为我是做后端的开发人员，深知一个好的工程项目是必须有那么一些规范的，不然也不会每个公死都有自己的那一套代码规范，前端的技术逐渐由直接引用的CSS、JavaScript，慢慢的变成了现在要使用的打包工具打包，逐渐的工程化了起来，而且现在的Web前端技术它的模式逐渐的向桌面应用靠齐，只不过一个是直接运行在操作系统上的，一个是运行在浏览器上的，如果我们把浏览器想象成一个虚拟机，那么现在的前端技术就和桌面应用一样了。所以前端技术需要工程化的问题，迫在眉睫。

Axios类库，本质上一个HTTP请求类库，要想使用好这个类库，不能只知道GET、POST、DELETE、PUT、怎么传输数据，更该学习一下HTTP事务、代理、网关、响应头、认证方法、不同的HTTP版本之间有什么区别。、

在功能设计上面，我们不能为了实现功能而去做功能。我们更需要的是去更好的了解需求，因为本质上面，我们做应用是为客户做应用，用户用着好用、程序能够正确运行才是用户最需要的。所以我们的每一条提示语句、每一个弹出框要让使用者觉得实际有效，这样才能提高用户体验感。当有了用户需求，我们实现功能的时候，最好能够画一画UML图，因为功能是非常好实现的，但功能实现的好与不好才是关键，画UML图可以让我们知道整体程序的流程走向，功能点覆盖的是否完全，反馈是否合理。

在本项目中，遇到的问题还是挺多的，有技术使用上的、操作系统和程序环境版本的、有字符编码的。这些问题给了我们很多解决问题的思路，如果是一个相对大众的问题，去论坛博客查找解决方案，如果是一个不那么大众的问题，就去官方文档查找解决方案。这样做以后大大的提高了我们解决问题的效率。

本次毕业设计，使我学习到了很多，加深了对技术的了解认识，此次毕业设计是我受益颇多。

参考文献

[1] “Spring-boot reference guide”，https://docs.spring.io/springboot/docs/2.1.4.RELEASE/reference/htmlsingle/

[2] “Vue reference guide”, https://cn.vuejs.org/v2/guide/

[3] “axios reference guide”，http://www.axios-js.com/zh-cn/docs/

[4] (德)Christian Bauer著. Hibernate实战(第二版).蒲成译. 清华大学出版社, 2016.9

[5] (美) David Gourley 著. HTTP权威指南. 陈涓. 赵振平译. 人民邮电出版社,2012.9

[6] (美) 尼古拉斯·泽卡斯著. JavaScript高级程序设计(第3版). 李松峰, 曹力译, 人民邮电出版社, 2012.3

[7] Baron Schwartz, Peter Zaitsev, Vadim Tkachenko著. 高性能MySQL(第3版). 宁海元, 周振兴, 彭立勋等译. 电子工业出版社, 2013.5

致谢

四年的大学学习生活在即将划上一个句号，而于我的人生来说却仅仅只是一个逗号，我将面对新的征程的开始。本研究及论文是在我的导师徐艳的亲切关怀和耐心的指导下完成的。伟人、名人固然为我所崇拜，可是我更迫切地想要把我的敬意献给给一位平凡的人，我的导师徐艳老师。也许我不是您最出色的学生，但您却是我所最尊敬的老师。您是如此的治学严谨，学识渊博，视野广阔，思想深刻，您用心为我营造一种良好的学术氛围，让我的论文更加的严谨。至此论文付梓之际，我的心情无法保持平静，从开始选择课题到论文的顺利答辩，有无数可敬的师长、朋友给了我很多的帮助，在这里请您接受我诚挚的谢意! 最后，再次对那些在论文完成过程中，关心、帮助我的同学和朋友们表示衷心地感谢！