**设备认证系统**

**一个很简单很简单的spring boot项目**

**目录结构说明**

它采用maven的多module构建,共分为七个版块

* business-center:是业务中心模块，此模块主要包含一个application启动模块，其它模块均为业务相关的模块
  + application:系统启动入口模块，因此需要组装哪些模块，只需要在此入口的pom中引入就可以
* common-component-center:是系统的一些复用较用的模块，也称公共业务模块。它包含了:
  + file-center:文件中心模块，包含文件的整体和分片上传,支持本地存储和FTP存储
  + param-center:基础参数配置中心模块，主要包含三块业务功能：
    - 1、参数类型功能，这个基础参数的入口，类型有：基础参数列表、基础参数树形。
    - 2、基础参数列表结构。
    - 3、基础参数树形结构
  + websocket-center:websocket中心模块，目前支持单台的内存模式和分布式的redis发布订阅模式
* inner-intergration:是一个工具类的核心包模块，主要是把系统中一些公共的代码抽离，封闭成功能单一的工具类包，它包含了:
  + basic-spring-boot-starter:系统是最核心的模块，定义了公共的config、枚举的处理、异常的定义是处理、一此工具类、标准的Web请求的输入输出
  + common-spring-boot-starter:在basic-spring-boot-starter的基础上对系统再次封装，这义了用户认证、异步线程的代理、系统中用户相关的枚举类、系统的访问拦截器、系统中的基础model及用户模块相关model、vo的定义
  + data-log-spring-boot-starter:系统中审计日志的模块
  + db-spring-boot-starter:连接数据的必要的模块，同时也支持多数据源的切换
  + log-spring-boot-starter:系统中请求的流水日志记录模块
  + mongo-spring-boot-starter:mongoDb的核心模块
  + mybatis-plus-spring-boot-starter:mybatis-plus的核心模块
  + redis-spring-boot-starter:redis的配置的核心模块，支持单节点和集群配置
  + rsa-spring-boot-starter:是一个非对称加解密的模块，主要用于和第三方接口通信，对数据安全处理
  + security-spring-boot-starter:是核心中安全处理的核心
  + swagger-spring-boot-starter:用户生成系统请求的api模块
* job:是一个开源的定时任务管理模块
* sql:这是一个数据脚本文件
* system-center:这是系统用户相关的功能模块
  + 权限:系统中的公共权限池
  + 菜单:系统中所拥有的菜单
  + 菜单权限:标记着每一个菜单页面项所有拥有的权限
  + 角色:用户所拥有的角色，它和用户的关系:M:1
  + 角色授权:每一个角色所拥有的菜单及菜单页面权限
  + 用户:主要分为管理和普通用户
* web-portal:后端功能对应的前端页面模块，
  + individual-soldier-auth-front:本系统采用了前后端分离模式，前端使用Vue+Element UI+Ts模式进行开发，使用Vuex进行信息存储
    - assets:存在前端系统中所使用的图标
    - common:前端封装的一些公共模块
      * directives:指令的封装
      * event-hub:事件的总线
      * exception:异常处理
      * filters:过滤器
      * form-validator:系统验证工具类
      * utils:工具集合
      * web-socket:web-socket的封装
    - components:系统组件的封装
      * cell:单元格的封装，主要是配合text组件一起使用
      * file-uploader:上传组件，支持分片上传
      * header:前端页面框架上的header的封装
      * left:前端页面框架的左边菜单封装
      * page-tabs:前端页面多页签的封装
      * select-tree:在下拉列表框中封装树型结构列表
      * table:前端分页查询的封装
      * text:单元格组件封装
    - plugins:主是把各种插件加入到Vue中
    - router:公共路由的处理
    - store:基于vuex的各种信息存储
      * index:对外导出
      * menu-collapse-store:菜单折叠状态的存储
      * menu-store:菜单的存储
      * permission-store:权限信息的存储
      * storage:使用widows存储
      * user-store:用户信息存储
      * web-socket-store:web-socket与后端交互信息的存储
    - views:前端的vue页面
      * index:登录后跳转的页面目录
        + business:业务页面
        + common:公共页面

审计日志

文件上传

基础参数管理

* + - * + home:默认主页
        + system:用户相关页面
      * login:登录页面
  + fetch:对Axios的封装
  + permission:对页面登录权限的处理
  + settings:一些公共的系统属性定义
  + shims-vue.d:vue全局属性定义
    - KWRequest:标准请求
    - KWResponse:标准输出
    - Model:公共model定义
    - KWRule:验证规则

**后端系统功能或模块的使用说明**

* 1、任何后端功能模块的使用，需先在pom中引用

pom.xml

<dependencies>

<!--公共模块 -->

<dependency>

<groupId>com.key.win</groupId>

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

</dependency>

...

</dependencies>

* 2、系统中所有的model都需要继承MybatisID实体类，标注着有统一的结构体，这样在保存时系统才能自动填充一些公共的属性，如操作人、创建时间等等

SysUser.java

@ApiModel("用户实体")

@Data

@TableName("sys\_user")

@EqualsAndHashCode(callSuper = true)

public class SysUser extends MybatisID {

...

}

* 3、系统的异步处理：目前系统异常处理主要包含：业务异常类:BizException、用户非法异常:UserIllegalException、权限异常:AccessDeniedException,如需要自定义异常，首页请定义自己异常 XxxException 类，并继承BizException异常类，其次在GlobalExceptionHandler类中完成自己的异常处理

UserIllegalException:

if (refreshTokenVo == null) {

logger.error("refreshToken is null !!");

throw new UserIllegalException("refreshToken过期");

}

AccessDeniedException:

if (!dbUser.getEnabled()) {

logger.error("用户{}帐号被禁用", dbUser.getUserName());

throw new AccountDisabledException("您的帐号被禁用！");

}

BizException:

if (!CollectionUtils.isEmpty(existUsers)) {

logger.error("{}用户已经存在!", sysUser.getUserName());

throw new BizException("用户已经存在！");

}

* 4、系统中枚举的使用：创建枚举之后并添加注解：@JsonSerialize(using = TextureEnumSerializerCode.class)，其目的是在对象输入是json序列化时，对其按时指定的方式序列化

@JsonSerialize(using = TextureEnumSerializerCode.class)

public enum SexEnum {

...

}

* 5、系统的标准输出

5.1、分页返回对象PageResult<T>

@PostMapping("/findSysUserByPaged")

@ApiOperation(value = "User分页")

@LogAnnotation(module = "system", recordRequestParam = false)

@PreAuthorize("hasAuthority('" + AUTHORITY\_PREFIX + "QUERY::PAGED')")

public PageResult<SysUser> findSysUserByPaged(@RequestBody PageRequest<SysUser> pageRequest) {

return sysUserService.findSysUserByPaged(pageRequest);

}

5.2、其它均返回Result<T>

@PostMapping("/updateSysUser")

@ApiOperation(value = "用户更新")

@LogAnnotation(module = "system", recordRequestParam = false)

@PreAuthorize("hasAuthority('" + AUTHORITY\_PREFIX + "MODIFY')")

public Result updateSysUser(@RequestBody SysUser sysUser) {

boolean b = sysUserService.updateSysUser(sysUser);

return Result.result(b);

}

* 6、系统登录用户的使用

6.1、用户登录后，通过token把用户数据存储redis中,可通过AuthenticationUtil工具类获取当前登录的用户信息

6.2、AuthenticationUtil.getAuthentication()获取当前登录的用户信息

6.3、AuthenticationUtil.setCurrentUser(Authentication sysUser)设计用户信息到ThreadLocal中

6.4、AuthenticationUtil.getUserId()获取当前登录用户的Id

6.5、AuthenticationUtil.getUserName()获取当前登录用户的用户名

6.6、AuthenticationUtil.getToken()获取当前登录用户的token

* 7、data-log-spring-boot-starter组件的使用,记录系统中增删改的数据变化日志记录

7.1、在pom中加入data-log-spring-boot-starter依赖

<dependency>

<groupId>com.key.win</groupId>

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

</dependency>

7.2、开启@EnableDataLog(不需要关注,因为在DataLogConfig类中已经开启)

7.3、具体使用方法

7.3.1、在方法前添加注解

@DataLog

@DeleteMapping("/delete/{id}")

@ApiOperation(value = "删除")

@LogAnnotation(module = "device-auth", recordRequestParam = false)

@PreAuthorize("hasAuthority('" + AUTHORITY\_PREFIX + "DELETE')")

public Result delete(@PathVariable Long id) {

boolean b = customerInfoService.removeById(id);

return Result.result(b);

}

7.3.2、手动调用

@Autowired

private SysDataLogService sysDataLogService;

@Override

public synchronized DeviceAuthResponseVo saveOrUpdateDeviceAuth(DeviceAuth deviceAuth) {

...业务处理...

sysDataLogService.saveDataLog("接收设备重复提交认证信息，进行覆盖操作", getDataLogFkId(deviceAuthByUniqueCode.getId()));

...业务处理...

}

7.4、审计结果如下：

7.4.1、新增：新插入数据：[DEVICE\_CUSTOMER\_INFO]{"sequence":"CNO.606160180316098562","authDeviceCode":"9999","expireDeviceDate":"2022-09-3000:00:00.0","authDeviceNum":"999","isVerify":"1","companyName":"999","companyAddress":"999","companyPhone":"999","leadName":"999","leadMobile":"13599999999","leadPhone":"","projectNo":"999","projectName":"999","createDate":"2022-09-2714:37:06.436","updateDate":"2022-09-2714:37:06.436","createUserId":"19","enableFlag":"1","createUserName":"admin","version":"0","id":"21"}

7.4.2、修改：修改表：[DEVICE\_CUSTOMER\_INFO]id：[18]把字段[authDeviceNum]从[10]改为[100]

7.4.3、删除：修改表：[DEVICE\_CUSTOMER\_INFO]id：[39]记录被删除！

* 8、db-spring-boot-starter组件的使用

8.1、在pom中加入db-spring-boot-starter依赖

<dependency>

<groupId>com.key.win</groupId>

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

</dependency>

8.2、非多数据的使用

spring:

datasource:

username: root

password: key-win123

url: jdbc:mysql://127.0.0.1:3307/individual-soldier-auth?useUnicode=true&characterEncoding=UTF-8&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=Asia/Shanghai

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

...

8.3、多数据源的使用：请注意spring.datasource.dynamic.enable为true开启

8.3.1、首先完成多数据源的yml配置(请参考application-ds.yml的配置)

spring:

datasource:

dynamic:

enable: true

primary: core #设置默认的数据源或者数据源组,默认值即为master

druid:

# JDBC 配置(驱动类自动从url的mysql识别,数据源类型自动识别)

core:

url: jdbc:mysql://127.0.0.1:3307/single-soldier-wireless?useUnicode=true&characterEncoding=UTF-8&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=Asia/Shanghai

username: root

password: key-win123

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

log:

url: jdbc:mysql://127.0.0.1:3307/log-center?useUnicode=true&characterEncoding=UTF-8&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=Asia/Shanghai

username: root

password: key-win123

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

...

8.3.2、在目标方法或类上完成@DataSource(name = "log")注解的配置，标明此方法或类使用数据log数据源

8.3.3、如果还想配置更多的数据源，请参考mybatis-plus中的DS组件

* 9、log-spring-boot-starter组件的使用，记录系统的输入输出参数

9.1、在pom中加入log-spring-boot-starter依赖

<dependency>

<groupId>com.key.win</groupId>

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

</dependency>

9.2、只需要在目标方法上加@LogAnnotation(module = "system", recordRequestParam = false)

@DeleteMapping("/delete/{id}")

@ApiOperation(value = "删除")

@LogAnnotation(module = "system", recordRequestParam = false)

@PreAuthorize("hasAuthority('" + AUTHORITY\_PREFIX + "DELETE')")

public Result delete(@PathVariable Long id) {

boolean b = sysUserService.deleteById(id);

return Result.result(b);

}

9.3、需要注意的是，如果recordRequestParam的值为true时，会将此流水记录保存至数据库的sys\_log表中

* 10、mongo-spring-boot-starter组件的使用

10.1、在pom中加入mongo-spring-boot-starter依赖

<dependency>

<groupId>com.key.win</groupId>

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

</dependency>

10.2、数据源配置

spring:

data:

mongodb:

uri: mongodb://127.0.0.1:27017/ssw

10.3、XxxService继承IMongoService<T>接口

public interface EventLogService extends IMongoService<EventLog> {

...

}

10.4、XxxServiceImp实现类继承MongoServiceImpl<T>抽像类,实现XxxService接口,实现可调用父类的方法进行增删改查操作

@Service

public class EventLogServiceImpl extends MongoServiceImpl<EventLog> implements EventLogService {

...

}

* 11、mybatis-plus-spring-boot-starter组件的使用

11.1、在pom中加入mybatis-plus-spring-boot-starter依赖

<dependency>

<groupId>com.key.win</groupId>

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

</dependency>

11.2、XxxDao继承BaseMapper<T>或KeyWinMapper<T>

@Mapper

public interface DeviceAuthVoDao extends KeyWinMapper<DeviceAuthVo> {

}

11.3、XxxService继承IService<T>接口

public interface DeviceAuthService extends IService<DeviceAuth> {

...

}

11.4、XxxServiceImp实现类继承ServiceImpl<XxxDao, T>抽像类,实现XxxService接口,实现可调用父类的方法进行增删改查操作

@Service

public class DeviceAuthServiceImpl extends ServiceImpl<DeviceAuthDao, DeviceAuth> implements DeviceAuthService {

...

}

11.5、分页查询的实现，传入参数的类型是PageRequest<T>,返回参数的类型PageResult<T>,只需要创建一MybatisPageServiceTemplate<T,RT>对象，转入baseMapper,重写constructWrapper方法就可以

@Override

public PageResult<DeviceAuthVo> findDeviceAuthByPaged(PageRequest<DeviceAuthVo> t) {

MybatisPageServiceTemplate<DeviceAuthVo, DeviceAuthVo> query = new MybatisPageServiceTemplate<DeviceAuthVo, DeviceAuthVo>(deviceAuthVoDao) {

@Override

protected AbstractWrapper constructWrapper(DeviceAuthVo deviceAuth) {

return buildDeviceAuthLambdaQueryWrapper(deviceAuth);

}

//select da.\*,dci.company\_name,dci.project\_no,dci.enabled\_verification from device\_auth da INNER JOIN device\_customer\_info dci on da.auth\_code = dci.auth\_device\_code

protected String constructNativeSql() {

return "select \* from (select da.\*,dci.company\_name,dci.project\_no,dci.is\_verify,dci.project\_name,dci.sequence from device\_auth da INNER JOIN device\_customer\_info dci on da.auth\_code = dci.auth\_device\_code where da.enable\_flag = 1 and dci.enable\_flag = 1 ) tmp";

}

};

return query.doPagingQuery(t);

}

11.6、请参考https://editor.csdn.net/md/?articleId=122747931

* 12、redis-spring-boot-starter组件的使用

12.1、在pom中加入redis-spring-boot-starter依赖

<dependency>

<groupId>com.key.win</groupId>

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

</dependency>

12.2、单节点的配置

spring:

redis:

################### redis 单机版 start ##########################

host: 127.0.0.1

port: 6379

timeout: 6000

database: 8

lettuce:

pool:

max-active: 10 # 连接池最大连接数（使用负值表示没有限制）,如果赋值为-1，则表示不限制；如果pool已经分配了maxActive个jedis实例，则此时pool的状态为exhausted(耗尽)

max-idle: 8 # 连接池中的最大空闲连接 ，默认值也是8

max-wait: 100 # # 等待可用连接的最大时间，单位毫秒，默认值为-1，表示永不超时。如果超过等待时间，则直接抛出JedisConnectionException

min-idle: 2 # 连接池中的最小空闲连接 ，默认值也是0

shutdown-timeout: 100ms

12.3、集群节点的配置

spring:

redis:

cluster:

nodes: 130.75.131.237:7000,130.75.131.238:7000,130.75.131.239:7000,130.75.131.237:7001,130.75.131.238:7001,130.75.131.239:7001

#130.75.131.237:7000,130.75.131.238:7000,130.75.131.239:7000,130.75.131.237:7001,130.75.131.238:7001,130.75.131.239:7001

#192.168.3.157:7000,192.168.3.158:7000,192.168.3.159:7000,192.168.3.157:7001,192.168.3.158:7001,192.168.3.159:7001

timeout: 1000 # 连接超时时间（毫秒）

lettuce:

pool:

max-active: 10 # 连接池最大连接数（使用负值表示没有限制）,如果赋值为-1，则表示不限制；如果pool已经分配了maxActive个jedis实例，则此时pool的状态为exhausted(耗尽)

max-idle: 8 # 连接池中的最大空闲连接 ，默认值也是8

max-wait: 100 # # 等待可用连接的最大时间，单位毫秒，默认值为-1，表示永不超时。如果超过等待时间，则直接抛出JedisConnectionException

min-idle: 2 # 连接池中的最小空闲连接 ，默认值也是0

shutdown-timeout: 100ms

12.4、使用

@Autowired

private static RedisTemplate<String, Object> redisTemplate;

* 13、rsa-spring-boot-starter组件的使用

13.1、在pom中加入rsa-spring-boot-starter依赖

<dependency>

<groupId>com.key.win</groupId>

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

</dependency>

13.2、生成非对称加密的证书，请参考rsa-spring-boot-starter模块中的rsa-encryptor文件中的README.md文件

13.3、输出加密：只需要返回对象是EncryptResponse，就可以自动加密

@PostMapping("/auto")

@ApiOperation(value = "新增/更新")

@LogAnnotation(module = "device-auth", recordRequestParam = false)

public EncryptResponse saveOrUpdate(@RequestBody DeviceAuthRequestVo deviceAuth) {

DeviceAuthResponseVo deviceAuthResponseVo = deviceAuthService.saveOrUpdateDeviceAuth((DeviceAuth) deviceAuth);

return EncryptResponse.succeed(deviceAuthResponseVo);

}

13.4、请求解密，只需要在对就的实体类中继承IEncryptor，就会自动解密转换成实体对象

@Data

@ApiModel("设备端提交设备信息Vo")

public class DeviceAuthRequestVo extends DeviceAuth implements IEncryptor {

}

* 14、security-spring-boot-starter组件的使用，主是系统权鉴功能，类似于spring security的@PreAuthorize注解的功能

14.1、在pom中加入security-spring-boot-starter依赖

<dependency>

<groupId>com.key.win</groupId>

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

</dependency>

14.2、通过spring.global.method.security.enable为true来开启鉴权功能

14.3、鉴权使用，@PreAuthorize("hasAuthority('具体的权限code')")

@PostMapping("/updateExpireDeviceDateAndSendAuthInfo")

@ApiOperation(value = "所有认证设备信息")

@LogAnnotation(module = "device-auth", recordRequestParam = false)

@PreAuthorize("hasAuthority('" + AUTHORITY\_PREFIX + "MODIFY')")

public Result updateExpireDeviceDateAndSendAuthInfo(@RequestBody DeviceAuth deviceAuth) {

return Result.succeed(deviceAuthService.updateExpireDeviceDate(deviceAuth));

}

* 15、swagger-spring-boot-starter组件的使用

15.1、在pom中加入swagger-spring-boot-starter依赖

<dependency>

<groupId>com.key.win</groupId>

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

</dependency>

15.2、在Controller上的使用,@Api("XxxApi")

@RestController

@RequestMapping("/deviceAuth/\*")

@Api("客户相关的api")

public class DeviceAuthCtrl {

...

}

15.3、在方法上的使用, @ApiOperation(value = "xxx")

@PostMapping("/findDeviceAuthByPaged")

@ApiOperation(value = "客户信息分页")

@LogAnnotation(module = "device-auth", recordRequestParam = false)

@PreAuthorize("hasAuthority('" + AUTHORITY\_PREFIX + "QUERY::PAGED')")

public PageResult<DeviceAuthVo> findDeviceAuthByPaged(@RequestBody PageRequest<DeviceAuthVo> t) throws Exception {

PageResult<DeviceAuthVo> deviceAuthByPaged = deviceAuthService.findDeviceAuthByPaged(t);

List<DeviceAuthVo> data = deviceAuthByPaged.getData();

if (!CollectionUtils.isEmpty(data)) {

for (DeviceAuthVo datum : data) {

datum.setIsOnLine(DeviceAuthUtils.isOnLineByUniqueCode(datum.getUniqueCode()));

}

}

return deviceAuthByPaged;

}

15.4、在实体上的使用，@ApiModel("xxx")

@Data

@EqualsAndHashCode(callSuper = true)

@ApiModel("客户实体Vo")

public class DeviceAuthVo extends DeviceAuth {

...

}

15.5、请注意swagger的开启，只在dev、uat、ds环境在会生成swagger-api

@Configuration

@EnableSwagger2

@Profile({"dev", "uat", "ds"})

public class Swagger2Config implements WebMvcConfigurer {

...

}

* 16、系统的登录流程梳理

16.1、页面组装用户名和密码为json格式，并提交至SysUserCtrl的login方法

16.2、login方法会对提交提交进行初步的校验

@PostMapping("/login")

@ApiOperation(value = "登录")

@LogAnnotation(module = "system", recordRequestParam = false)

public Result login(@RequestBody SysUser sysUser, HttpServletRequest request) {

String userAgent = request.getHeader("user-agent");

if (sysUser == null) {

logger.error("user is null !!!");

throw new BizException("用户或密码为空！");

}

if (StringUtils.isBlank(sysUser.getUserName())) {

logger.error("username is null !!!");

throw new BizException("用户或密码为空！");

}

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

logger.error("password is null !!!");

throw new BizException("用户或密码为空！");

}

Map<String, Object> token = sysUserService.login(sysUser, userAgent);

return Result.succeed(token, "登录成功！");

}

16.3、调用sysUserService.login方法进行登录验证：

16.3.1、根据用户名在数据库查找当前用户信息

16.3.2、判断当前用户是否存在

16.3.3、当前当前用户是否出现多个

16.3.4、校验密码

16.3.5、检查帐号状态

List<SysUser> list = this.findSysUserByUserName(sysUser.getUserName());

if (list == null || list.size() == 0) {

logger.error("{}用户不存在！", sysUser.getUserName());

throw new AccountNotFoundException("用户名或密码有误！");

}

if (list.size() > 1) {

logger.error("{}用户存在{}个", sysUser.getUserName(), list.size());

throw new AccountException("帐号不唯一,请联系管理员！");

}

SysUser dbUser = list.get(0);

String encode = myPasswordEncoder.encode(sysUser.getPassword());

if (!encode.equals(dbUser.getPassword())) {

logger.error("用户{}的密码有误", dbUser.getUserName());

throw new BadCredentialsException("用户名或密码有误！");

}

checkIsEnabled(dbUser);

16.4、创建登录用户Authentication

Authentication loginUser = new Authentication();

16.5、根据用户类型设置用户鉴权信息，角色、菜单、权限

private void setUserExtInfo(String userAgent, SysUser dbUser, Authentication loginUser) {

BeanUtils.copyProperties(dbUser, loginUser);

List<SysGroup> groupByUserId = sysUserGroupDao.findGroupByUserId(dbUser.getId());

List<SysRole> rolesByUserId = sysUserRoleDao.findRolesByUserId(dbUser.getId());

loginUser.setSysGroups(groupByUserId);

loginUser.setSysRoles(rolesByUserId);

if (dbUser.getType() == UserTypeEnum.ADMIN) {

//List<SysMenuPermission> permissionDaoByRoleIds = sysMenuPermissionService.list();

List<SysMenu> menus = sysMenuService.list();

List<SysPermission> sysPermissions = sysPermissionService.list();

loginUser.setPermissions(getMenuPermissions(menus, sysPermissions));

loginUser.setMenus(menus);

} else if (!CollectionUtils.isEmpty(rolesByUserId)) {

Set<Long> roleIds = rolesByUserId.stream().map(SysRole::getId).collect(Collectors.toSet());

List<SysRoleMenuPermission> grantMenus = sysRoleMenuPermissionService.findGrantMenus(roleIds);

Set<Long> menuIds = grantMenus.stream().map(SysRoleMenuPermission::getMenuId).collect(Collectors.toSet());

List<SysMenu> menus = sysMenuService.findSysMenuByMenuIds(menuIds);

List<SysRoleMenuPermission> grantMenuPermissions = sysRoleMenuPermissionService.findGrantMenuPermissions(roleIds);

Set<Long> menuPermissionIds = grantMenuPermissions.stream().map(SysRoleMenuPermission::getMenuPermissionId).collect(Collectors.toSet());

List<SysMenuPermission> sysMenuPermissionByIds = sysMenuPermissionService.findSysMenuPermissionByIds(menuPermissionIds);

loginUser.setPermissions(sysMenuPermissionByIds);

loginUser.setMenus(menus);

}

Collections.sort(loginUser.getMenus(), new Comparator<SysMenu>() {

@Override

public int compare(SysMenu o1, SysMenu o2) {

return o1.getSort() - o2.getSort();

}

});

setLoginType(loginUser, userAgent);

}

16.6、创建用户认证信息保存redis并返回

String guid = UUIDUtils.getGUID();

loginUser.setToken(guid);

loginUser.setOnLine(true);

loginUser.setRefreshToken(refreshToken);

// loginUser.setDataPermissionVo(this.getCurrentUserDataPermissions(loginUser));

//缓存用户

AuthenticationUtil.setAuthenticationToRedis(loginUser);

AuthenticationUtil.setRefreshTokenToRedis(refreshToken, loginUser);

return this.getUserToken(guid, refreshToken);

16.7、前端获取用户信息的api

访问地址:http://x.x.x.x:9902/user/current

@GetMapping("/current")

@ApiOperation(value = "获取当前登录用户")

@LogAnnotation(module = "system", recordRequestParam = false)

public Result getLoginAppUser() {

return Result.succeed(AuthenticationUtil.getAuthentication());

}

16.8、token失效刷新api

访问地址:http://x.x.x.x:9902/user/refresh/{token}

@GetMapping("/refresh/{token}")

@ApiOperation(value = "refresh")

@LogAnnotation(module = "system", recordRequestParam = false)

public Result refreshToken(@PathVariable String token, HttpServletRequest request) {

String userAgent = request.getHeader("user-agent");

try {

Map<String, Object> res = sysUserService.refreshToken(token, userAgent);

return Result.succeed(res, "刷新成功！");

} catch (Exception e) {

logger.error("refreshToken error:{}", e.getMessage(), e);

return Result.failed(401, e.getMessage());

}

}

16.9、获取当前用户的访问菜单Api

访问地址:http://x.x.x.x:9902/menu/current

@GetMapping("/current")

@ApiOperation(value = "获取当前用户授权的Menus")

@LogAnnotation(module = "system", recordRequestParam = false)

public Result getCurrentMenus() {

Authentication authentication = AuthenticationUtil.getAuthentication();

return Result.succeed(MenuUtils.treeBuilder(authentication.getMenus()));

}

* 17、系统的登出流程梳理

访问地址:http://x.x.x.x:9902/user/logout

@GetMapping("/logout")

@ApiOperation(value = "登出当前登录用户")

@LogAnnotation(module = "system", recordRequestParam = false)

public Result logout() {

Authentication authentication = AuthenticationUtil.getAuthentication();

if (authentication != null) {

authentication.setOnLine(false);

}

return AuthenticationUtil.logout() ? Result.succeed("操作成功") : Result.succeed("操作失败");

}

* 18、系统拦截请求验证用户，并设置当前登录用户信息本地线程上

18.1、系统默认会拦截所有请求，如有要旅行的url,请在yml中进行配置

spring:

web:

request:

white: /user/login,/user/refresh/\*\*,/user/register,/swagger-ui.html/\*\*,/swagger-resources/\*\*,/v2/\*\*,/webjars/\*\*,/csrf,/favicon.ico,/error,/,/individual-soldier-auth/\*\*,/api/auth/\*\*

18.2、拦截验证用户

public class LoginInterceptor implements HandlerInterceptor {

private final Logger logger = LoggerFactory.getLogger(this.getClass());

@Override

public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {

String token = this.extractToken(request);

if (StringUtils.isBlank(token)) {

throw new UserIllegalException("token缺失！");

}

logger.info("获取用户token：{}", token);

Authentication user = AuthenticationUtil.getAuthenticationToRedis(token);

if (user == null) {

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

}

AuthenticationUtil.setCurrentUser(user);

AuthenticationUtil.setAuthenticationTokenExpires(token);

return true;

}

protected String extractToken(HttpServletRequest request) {

String token = this.extractHeaderToken(request);

if (token == null) {

logger.warn("Token not found in headers. Trying request parameters.");

token = request.getParameter(IndividualSoldierAuthConstantUtils.REQUEST\_TOKEN\_KEY);

if (token == null) {

logger.error("Token not found in request parameters. illegal request.");

}

}

return token;

}

protected String extractHeaderToken(HttpServletRequest request) {

Enumeration headers = request.getHeaders(IndividualSoldierAuthConstantUtils.REQUEST\_HEADER\_AUTHORIZATION);

String value;

do {

if (!headers.hasMoreElements()) {

return null;

}

value = (String) headers.nextElement();

} while (!value.toLowerCase().startsWith(IndividualSoldierAuthConstantUtils.TOKEN\_BEARER\_VAUE.toLowerCase()));

String authHeaderValue = value.substring(IndividualSoldierAuthConstantUtils.TOKEN\_BEARER\_VAUE.length()).trim();

// int commaIndex = authHeaderValue.indexOf(44);

// if (commaIndex > 0) {

// authHeaderValue = authHeaderValue.substring(0, commaIndex);

// }

return authHeaderValue;

}

}

* 19、CorsConfig跨域处理

@Configuration

public class CorsConfig {

/\*\*

\* 跨域支持

\*

\* @return

\*/

@Bean

public CorsFilter corsFilter() {

final UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();

final CorsConfiguration config = new CorsConfiguration();

config.setAllowCredentials(true); // 允许cookies跨域

config.addAllowedOrigin("\*");// #允许向该服务器提交请求的URI，\*表示全部允许

config.addAllowedHeader("\*");// #允许访问的头信息,\*表示全部

config.setMaxAge(18000L);// 预检请求的缓存时间（秒），即在这个时间段里，对于相同的跨域请求不会再预检了

config.addAllowedMethod("\*");// 允许提交请求的方法，\*表示全部允许

source.registerCorsConfiguration("/\*\*", config);

return new CorsFilter(source);

}

}

* 20、事务处理，采用全局事务处理，无需要在每个方法添加注解

20.1、yml中的配置,主要是设置是否启用，切面及超时间

spring:

tx:

manager:

enabled: true

service:

pointcut:

expression: execution (\* com.key.win..service.impl.\*Impl.\*(..))

method:

timeout: -1

20.2、配置类

/\*\*

\* AOP事务处理

\* <p>

\* spring事务处理优先使用事务配置顺序

\* 方法级别@Transactional -> 类级别@Transactional -> AOP配置

\*/

@Aspect

@Configuration

@ConditionalOnProperty(name = "spring.tx.manager.enabled", matchIfMissing = false, havingValue = "true")

public class TxAdviceConfig {

private final Logger logger = LoggerFactory.getLogger(this.getClass());

/\*\*

\* service事务aop

\*/

// module.tx.service.enabled

public static String TX\_SERVICE\_ENABLED = "true";

public static String TX\_SERVICE\_POINTCUT\_EXPRESSION = "execution (\* com.key.win..service.impl.\*Impl.\*(..))";

public static int TX\_SERVICE\_METHOD\_TIMEOUT = 5;

@Value("${spring.tx.service.pointcut.expression}")

public void setTX\_SERVICE\_POINTCUT\_EXPRESSION(String TX\_SERVICE\_POINTCUT\_EXPRESSION) {

TxAdviceConfig.TX\_SERVICE\_POINTCUT\_EXPRESSION = TX\_SERVICE\_POINTCUT\_EXPRESSION;

}

@Value("${spring.tx.service.method.timeout:5}")

public void setTX\_SERVICE\_METHOD\_TIMEOUT(int TX\_SERVICE\_METHOD\_TIMEOUT) {

TxAdviceConfig.TX\_SERVICE\_METHOD\_TIMEOUT = TX\_SERVICE\_METHOD\_TIMEOUT;

}

/\*\*

\* 配置事务管理器使用springboot默认的

\*

\* 关于事务管理器，不管是JPA还是JDBC等都实现自接口 PlatformTransactionManager

\* 如果你添加的是 spring-boot-starter-jdbc 依赖，框架会默认注入 DataSourceTransactionManager 实例。

\* 如果你添加的是 spring-boot-starter-data-jpa 依赖，框架会默认注入 JpaTransactionManager 实例。

\*

\* 手工注解@Bean 将被优先加载，框架不会重新实例化其他的 PlatformTransactionManager 实现类。

\* @param dataSource

\* @return

\* @throws Exception

\*/

/\*@Bean

public DataSourceTransactionManager transactionManager(DataSource dataSource) throws Exception {

return new DataSourceTransactionManager(dataSource);

}\*/

/\*\*

\* 事务的实现Advice

\*

\* @return

\* @see org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration$TransactionTemplateConfiguration

\*/

@Bean

public TransactionInterceptor txAdvice(/\*@Qualifier("txManager") \*/PlatformTransactionManager transactionManager) {

logger.warn("#[Tx Config:TX\_SERVICE\_METHOD\_TIMEOUT]" + TX\_SERVICE\_METHOD\_TIMEOUT);

/\* 只读事务，不做更新操作， 不超时 \*/

RuleBasedTransactionAttribute readOnlyTx = new RuleBasedTransactionAttribute();

readOnlyTx.setName("PTA-READONLY-TX");

readOnlyTx.setDescriptor("PTA READONLY TX");

readOnlyTx.setReadOnly(true);

//readOnlyTx.setPropagationBehavior(TransactionDefinition.PROPAGATION\_NOT\_SUPPORTED);

readOnlyTx.setPropagationBehavior(TransactionDefinition.PROPAGATION\_SUPPORTS);

/\* 当前存在事务就使用当前事务，当前不存在事务就创建一个新的事务 \*/

RuleBasedTransactionAttribute requiredTx = new RuleBasedTransactionAttribute();

requiredTx.setName("PTA-REQUIRED-TX");

requiredTx.setDescriptor("PTA REQUIRED TX");

requiredTx.setRollbackRules(Collections.singletonList(new RollbackRuleAttribute(Exception.class)));

requiredTx.setPropagationBehavior(TransactionDefinition.PROPAGATION\_REQUIRED);

requiredTx.setTimeout(TX\_SERVICE\_METHOD\_TIMEOUT);

Map<String, TransactionAttribute> txMap = new HashMap<>();

txMap.put("get\*", readOnlyTx);

txMap.put("query\*", readOnlyTx);

txMap.put("find\*", readOnlyTx);

txMap.put("select\*", readOnlyTx);

txMap.put("qry\*", readOnlyTx);

txMap.put("add\*", requiredTx);

txMap.put("save\*", requiredTx);

txMap.put("insert\*", requiredTx);

txMap.put("update\*", requiredTx);

txMap.put("delete\*", requiredTx);

txMap.put("\*", requiredTx);

NameMatchTransactionAttributeSource source = new NameMatchTransactionAttributeSource();

source.setNameMap(txMap);

TransactionInterceptor txAdvice = new TransactionInterceptor(transactionManager, source);

return txAdvice;

}

/\*\*

\* 切面的定义 pointcut及advice

\*

\* @param txAdvice

\* @return

\*/

@Bean

public Advisor txAdviceAdvisor(/\*@Qualifier("txAdvice") \*/TransactionInterceptor txAdvice) {

logger.warn("#[Tx Config:TX\_SERVICE\_POINTCUT\_EXPRESSION]" + TX\_SERVICE\_POINTCUT\_EXPRESSION);

AspectJExpressionPointcut pointcut = new AspectJExpressionPointcut();

pointcut.setExpression(TX\_SERVICE\_POINTCUT\_EXPRESSION);

return new DefaultPointcutAdvisor(pointcut, txAdvice);

}

}

* 21、websocket的使用

21.1、在pom中加入websocket-spring-boot-starter依赖

<dependency>

<groupId>com.key.win</groupId>

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

</dependency>

21.2、yml中的配置,spring.web.socket.exporter.enable值为ture是启用websocket,而spring.web.socket.exporter.cluster的值为true时是webscoke集群方式，采用redis的发布订阅方式

spring:

web:

socket:

exporter:

enable: true

cluster: true

path: /ws/{token}

21.3、websocket的使用,请求地址为: ws://x.x.x.x:9902/ws/{token}

21.3.1、websocket client端一般都是通过http发起请请求，地址http://x.x.x.x:9902/ws/\*\*,入口类：Controller:WebSocketCtrl

21.3.2、websocket server端会直接将消息推送至websocket的client端，MessageSendUtil.java和WebSocketUtil是两个推送的工具类

* 22、druid的使用

22.1、在pom中加入websocket-spring-boot-starter依赖

<dependency>

<groupId>com.alibaba</groupId>

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

<version>${druid.version}</version>

</dependency>

22.2、yml中的配置,使用方式一

datasource:

username: root

password: key-win123

url: jdbc:mysql://127.0.0.1:3307/individual-soldier-auth?useUnicode=true&characterEncoding=UTF-8&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=Asia/Shanghai

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

druid:

#连接池配置(通常来说，只需要修改initialSize、minIdle、maxActive

initial-size: 1

max-active: 20

min-idle: 1

# 配置获取连接等待超时的时间

max-wait: 60000

#打开PSCache，并且指定每个连接上PSCache的大小

pool-prepared-statements: true

max-pool-prepared-statement-per-connection-size: 20

validation-query: SELECT 'x'

test-on-borrow: false

test-on-return: false

test-while-idle: true

#配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒

time-between-eviction-runs-millis: 60000

#配置一个连接在池中最小生存的时间，单位是毫秒

min-evictable-idle-time-millis: 300000

filters: stat,wall

# StatViewServlet监控器。开启后，访问http://域名/druid/index.html

stat-view-servlet:

enabled: true # 开启 StatViewServlet，即开启监控功能

login-username: daniel # 访问监控页面时登录的账号

login-password: 1234 # 密码

url-pattern: /druid/\* # Servlet的映射地址，不填写默认为"/druid/\*"。如填写其它地址，访问监控页面时，要使用相应的地址

reset-enable: false # 是否允许重置数据（在页面的重置按钮）。（停用后，依然会有重置按钮，但重置后不会真的重置数据）

# allow: 192.168.1.2,192.168.1.1 # 监控页面访问白名单。默认为127.0.0.1。与黑名单一样，支持子网掩码，如128.242.127.1/24。多个ip用英文逗号分隔

# deny: 18.2.1.3 # 监控页面访问黑名单

# 配置 WebStatFilter（StatFilter监控器中的Web模板）

web-stat-filter:

enabled: true # 开启 WebStatFilter，即开启监控功能中的 Web 监控功能

url-pattern: /\* # 映射地址，即统计指定地址的web请求

exclusions: '\*.js,\*.gif,\*.jpg,\*.png,\*.css,\*.ico,/druid/\*' # 不统计的web请求，如下是不统计静态资源及druid监控页面本身的请求

session-stat-enable: true # 是否启用session统计

session-stat-max-count: 1 # session统计的最大个数，默认是1000。当统计超过这个数，只统计最新的

principal-session-name: userName # 所存用户信息的serssion参数名。Druid会依照此参数名读取相应session对应的用户名记录下来（在监控页面可看到）。如果指定参数不是基础数据类型，将会自动调用相应参数对象的toString方法来取值

principal-cookie-name: userName # 与上类似，但这是通过Cookie名取到用户信息

profile-enable: true # 监控单个url调用的sql列表（试了没生效，以后需要用再研究）

filter:

wall:

enabled: false # 开启SQL防火墙功能

config:

select-allow: true # 允许执行Select查询操作

delete-allow: false # 不允许执行delete操作

create-table-allow: false # 不允许创建表

# 更多用法，参考官方文档：https://github.com/alibaba/druid/wiki/%E9%85%8D%E7%BD%AE-wallfilter

22.3、yml中的配置,使用方式二

datasource:

...

druid:

# JDBC 配置(驱动类自动从url的mysql识别,数据源类型自动识别)

core:

url: jdbc:mysql://127.0.0.1:3307/single-soldier-wireless?useUnicode=true&characterEncoding=UTF-8&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=Asia/Shanghai

username: root

password: key-win123

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

log:

url: jdbc:mysql://127.0.0.1:3307/log-center?useUnicode=true&characterEncoding=UTF-8&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=Asia/Shanghai

username: root

password: key-win123

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

#连接池配置(通常来说，只需要修改initialSize、minIdle、maxActive

initial-size: 1

max-active: 20

min-idle: 1

# 配置获取连接等待超时的时间

max-wait: 60000

#打开PSCache，并且指定每个连接上PSCache的大小

pool-prepared-statements: true

max-pool-prepared-statement-per-connection-size: 20

validation-query: SELECT 'x'

test-on-borrow: false

test-on-return: false

test-while-idle: true

#配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒

time-between-eviction-runs-millis: 60000

#配置一个连接在池中最小生存的时间，单位是毫秒

min-evictable-idle-time-millis: 300000

filters: stat,wall

# StatViewServlet监控器。开启后，访问http://域名/druid/index.html

stat-view-servlet:

enabled: true # 开启 StatViewServlet，即开启监控功能

login-username: daniel # 访问监控页面时登录的账号

login-password: 1234 # 密码

url-pattern: /druid/\* # Servlet的映射地址，不填写默认为"/druid/\*"。如填写其它地址，访问监控页面时，要使用相应的地址

reset-enable: false # 是否允许重置数据（在页面的重置按钮）。（停用后，依然会有重置按钮，但重置后不会真的重置数据）

# allow: 192.168.1.2,192.168.1.1 # 监控页面访问白名单。默认为127.0.0.1。与黑名单一样，支持子网掩码，如128.242.127.1/24。多个ip用英文逗号分隔

# deny: 18.2.1.3 # 监控页面访问黑名单

# 配置 WebStatFilter（StatFilter监控器中的Web模板）

web-stat-filter:

enabled: true # 开启 WebStatFilter，即开启监控功能中的 Web 监控功能

url-pattern: /\* # 映射地址，即统计指定地址的web请求

exclusions: '\*.js,\*.gif,\*.jpg,\*.png,\*.css,\*.ico,/druid/\*' # 不统计的web请求，如下是不统计静态资源及druid监控页面本身的请求

session-stat-enable: true # 是否启用session统计

session-stat-max-count: 1 # session统计的最大个数，默认是1000。当统计超过这个数，只统计最新的

principal-session-name: userName # 所存用户信息的serssion参数名。Druid会依照此参数名读取相应session对应的用户名记录下来（在监控页面可看到）。如果指定参数不是基础数据类型，将会自动调用相应参数对象的toString方法来取值

principal-cookie-name: userName # 与上类似，但这是通过Cookie名取到用户信息

profile-enable: true # 监控单个url调用的sql列表（试了没生效，以后需要用再研究）

filter:

wall:

enabled: false # # 开启SQL防火墙功能 WallFilter得check未通过，delete删除不被允许 当前项目不允许数据库删除，删除为逻辑删除，修改状态值等

config:

select-allow: true # 允许执行Select查询操作

delete-allow: false # 不允许执行delete操作

create-table-allow: false # 不允许创建表

# 更多用法，参考官方文档：https://github.com/alibaba/druid/wiki/%E9%85%8D%E7%BD%AE-wallfilter

21.4、通过 http://127.0.0.1:9902/druid/index.html 访问查看，daniel/1234

**前端系统功能或模块的使用说明**

* 1、前端的路径在项目中的web-portal文件夹下的individual-soldier-auth-front下的view文件中，目前有三个环境变量，既dev、uat、prod。主要配置了build文件之后的输入目录信息，环境变量、及访问的api，在dev环境中采用的代理模式，主要解决在dev环境中，因为没有登录而造成的跨域问题。修改后端的请求路径，请在这三个文件中修改

1.1、.env.development

//生成地址

outputDir = "production"

VUE\_APP\_MODE = 'production'

NODE\_ENV = 'prod'

VUE\_APP\_HTTP\_BASE\_URL = '/api'

VUE\_APP\_WEBSOCKET\_BASE\_WS\_URL='ws://127.0.0.1:9902/ws/'

VUE\_APP\_TEXT = '生产环境'

1.1.1、代理配置:vue.config.js

module.exports = {

outputDir: 'build/' + (process.env.outputDir ? process.env.outputDir : 'dist'),

devServer: {

proxy: {

'/api': {

// target: 'https://192.168.1.147:8443',

target: 'http://127.0.0.1:9902',

ws: true,

changeOrigin: true,

pathRewrite: {

'^/api': ''

}

}

}

}

}

1.2、.env.uat

//生成地址

outputDir = "uat"

VUE\_APP\_MODE = 'uat'

NODE\_ENV = 'uat'

VUE\_APP\_HTTP\_BASE\_URL = 'http://127.0.0.1:9002'

VUE\_APP\_WEBSOCKET\_BASE\_WS\_URL='ws://127.0.0.1:9002/ws/'

VUE\_APP\_TEXT = 'uat环境'

1.3、.env.production

//生成地址

outputDir = "production"

VUE\_APP\_MODE = 'production'

NODE\_ENV = 'prod'

VUE\_APP\_HTTP\_BASE\_URL = 'http://127.0.0.1:9002'

VUE\_APP\_WEBSOCKET\_BASE\_WS\_URL='ws://127.0.0.1:9002/ws/'

VUE\_APP\_TEXT = '生产环境'

* 2、fetch.ts文件是所有访问后端的http请求的拦截器，主要定义的请求的头的处理，对token的处理，正常返回值和异常情况的处理

2.1、创建Axios实例，获取访问后端的api及超时时间

const instance: AxiosInstance = axios.create({

baseURL: getHttpDomain(),

timeout: 3000

})

2.1.1、在getHttpDomain方法中对访问环境变量的api进行获取

get-env.ts

export function getHttpDomain(): string {

const baseURL = process.env.VUE\_APP\_HTTP\_BASE\_URL

return processBaseURL(baseURL)

}

2.2、请求头的处理

instance.interceptors.request.use(

(config: AxiosRequestConfig): AxiosRequestConfig => {

config.headers = config.headers || {}

config.headers['x-requested-with'] = 'XMLHttpRequest'

if (!config.headers['Content-Type']) {

config.headers['Content-Type'] = 'application/json;charset=UTF-8'

}

const token: string | null = localStorage.getItem('access\_token')

if (token) {

config.headers.Authorization = 'Bearer ' + token

}

return config

},

(err: AxiosError) => {

return errorHandle(err)

}

)

2.3、返回值的处理，这主要分为正常的返回值处理和异常情况下的处理:正常情况的下的处理直接转为KWResponse.Result | KWResponse.PageResult | void，而异常情况处理就要分很多情况

instance.interceptors.response.use(

(result: AxiosResponse): KWResponse.Result | KWResponse.PageResult | void => {

return result.data as KWResponse.Result | KWResponse.PageResult | void

},

(err: AxiosError) => {

return errorHandle(err)

}

)

2.4、异常情况的处理

function errorHandle(err: AxiosError): Promise<unknown> {

if (err.code === 'ECONNABORTED' && err.message.indexOf('timeout') !== -1) {

return onTimeoutError(err)

} else if (err.response === undefined) {

return onNetWorkError(err)

}

let message = ''

switch (err.response?.status) {

case 400:

message = '请求错误'

break

case 401:

return onAuthenticationError(err)

case 403:

message = '拒绝访问'

break

case 404:

message = `请求地址出错: ${err.response?.config.url}`

break

case 408:

message = '请求超时'

break

case 500:

message = '服务器内部错误'

break

case 501:

message = '服务未实现'

break

case 502:

message = '网关错误'

break

case 503:

message = '服务不可用'

break

case 504:

message = '网关超时'

break

case 505:

message = 'HTTP版本不受支持'

break

default:

message = '其他情况'

}

Message.error((err.response.data as string) || message)

return Promise.reject(err)

}

2.5、token失效后根据refreshToken来刷新token,当token失效后会返回http状态码为401，这个时候就执行了onAuthenticationError(err)方法进行token刷新动作，如成功刷新后，再次请求目标api，如刷新失败后，清空相关token，跳转至登录页面

async function onAuthenticationError(error: AxiosError) {

local.clear(settings.activePath)

const { code, data }: KWResponse.Result<LoginResponse> = await getNewToken()

if (code === 200) {

local.save(settings.accessToken, data.access\_token)

local.save(settings.refreshToken, data.refresh\_token)

return await instance.request(error.config)

} else {

Message.error('用户失效，请重新登录！')

local.clear(settings.accessToken)

local.clear(settings.refreshToken)

router.replace({

path: '/login'

})

return error

}

}

async function getNewToken(): Promise<KWResponse.Result<LoginResponse>> {

const refreshToken = local.getStr(settings.refreshToken)

return await instance.get('user/refresh/' + refreshToken, {})

}

* 3、shims-vue.d.ts文件说明，此文件中定义系统的全局公共模型，主要分：KWResponse、Model、KWRequest、KWRule几部分。

3.1、系统中的标准model基类，在Model命令空间下，需要说明的是id在后端是long类型，但由于后端long类型长度过大，前端接后会导致精度丢失的问题，因为前端将使用string来接收这个字段返回的值(返回这也需要后端的配合，在该字段的输出json时需以字符串输出)

namespace Model {

interface Id {

id: string

}

interface Version extends Id {

version: number

}

interface BaseField extends Version {

createDate: number

updateDate: number | null

createUserId: string

updateUserId: string | null

enableFlag: boolean

createUserName: string

updateUserName: string | null

}

interface Name {

name: string

}

interface CodeField extends Name {

code: string

}

interface ParentId {

parentId: string

}

interface EnumEntity {

code: string

stringValue: string

text: string

}

}

3.2、KWRequest标准的分页请求需要模型，T代表具体的实体模型

namespace KWRequest {

type OrderDir = 'ASC' | 'DESC'

type MethodType = 'POST' | 'GET'

interface PageRequest<T = undefined> {

pageNo: number

pageSize: number

sortName?: string

sortDir?: OrderDir

t?: T

}

}

3.3、KWResponse 标准的返回值模型，非分页统一返回Result<T>，分页统一返回PageResult<T>，T代表具体的实体模型

namespace KWResponse {

interface BaseResult {

code: number

msg: string

}

interface Result<T = undefined> extends BaseResult {

data: T

}

interface PageResult<T = undefined> extends BaseResult {

pageNo: number

pageSize: number

count: number

data: T[]

totalPage: number

}

}

3.4、KWRule Form验证模型，定义了最常见的验证规则

namespace KWRule {

interface CallbackFunction {

(error?: Error): void

}

interface TriggerRule {

trigger: string | Array<string>

}

interface MessageRule extends TriggerRule {

message: string

}

interface Rule extends MessageRule {

required: boolean

}

interface DateRule extends MessageRule {

type: string

}

interface ArrayRule extends MessageRule {

type: string

}

interface MixinRule extends MessageRule {

min: number

max: number

}

interface ValidatorFunction {

// eslint-disable-next-line no-use-before-define

(rule: ValidatorRule, value: string, cb: CallbackFunction): void

}

interface ValidatorRule extends TriggerRule {

validator: ValidatorFunction

}

interface TransformFunction {

(value: string): number

}

interface NumberRule extends MessageRule {

type: string

transform: TransformFunction

}

}

* 4、setting.ts 系统设置，主要定义一些利用比较高的变更及系统的中的一些开关

interface ISettings {

title: string // Overrides the default title

activePath: string

accessToken: string

refreshToken: string

menuTypeItem: string

menuTypeDirectory: string

defaultAvatar: string

isEnablePermission: boolean

isEnableWebSocket: boolean

}

// You can customize below settings :)

const settings: ISettings = {

title: 'key-win后台管理',

activePath: 'activePath',

accessToken: 'access\_token',

refreshToken: 'refresh\_token',

menuTypeItem: '菜单',

menuTypeDirectory: '目录',

defaultAvatar: require('./assets/head.png'),

isEnablePermission: true,

isEnableWebSocket: true

}

export default settings

* 5、permission.ts 前端的请求拦截器，用来判断当前用户是否登录，如果没有就直接跳转至登录，在每个请求开始对请求做登录校验，结束时设置html的title

router.beforeEach(async (to: Route, from: Route, next: NavigationGuardNext): Promise<void> => {

NProgress.start()

// to 将访问哪一个路径

// from 代表从哪个路径跳转而来

// next 是一个函数,表示放行

// next() 放行 next('/login') 强制跳转

if (to.path === '/login') {

return next()

}

// 获取token

const refreshToken = local.getAny(settings.refreshToken)

if (!refreshToken) {

return next('/login')

} else {

const dynamicRoutes: Array<RouteConfig> = PermissionModule.getDynamicRoutes

// 如果是第一次登录成功中转过来，dynamicRoutes的长度必为0，这个时候就需要获取用户信息、当前登录用户的菜单及操作权限

if (dynamicRoutes.length === 0) {

getUserInfo(to, from, next)

} else {

next()

}

}

})

router.afterEach((to: Route) => {

NProgress.done()

// set page title

document.title = getPageTitle((to.meta as RouteMeta).title)

})

* 6、登录流程解析

6.1、用户登录Login.vue,前端用户名密码不为空后，封装json后，调用user/login进行登录，登录成功后返回token信息对象，跳转至index.vue

login(): void {

this.loginFormRef.validate(async valid => {

if (!valid) {

return false

}

console.log(valid)

const { code, data, msg }: KWResponse.Result<LoginResponse> = await LoginApi(this.loginForm)

if (code === 200) {

// 登录成功

local.save(settings.accessToken, data.access\_token)

local.save(settings.refreshToken, data.refresh\_token)

local.clear(settings.activePath)

this.$router.push('/index')

} else {

this.$message.error(msg)

}

})

}

6.2、根据路由跳转至index.vue过程中，进入请求拦截器中(上面第5部分)

6.2.1、通过user/getLoginApp的api获取用户信息

export const getUserInfo = async (to: Route, from: Route, next: NavigationGuardNext): Promise<void> => {

const { code, data, msg }: KWResponse.Result<LoginSuccessUserInfo> = await UserInfoApi()

console.log(data)

if (code === 200) {

UserModule.changeUser(data)

getMenus(to, from, next)

} else {

Message.error(msg || '获取用户失败！')

}

}

6.2.2、保存登录用户信息到UserModule的Vuex中

6.2.3、通过menu/current的api获取菜单信息

export const getMenus = async (to: Route, from: Route, next: NavigationGuardNext): Promise<void> => {

const { code, data, msg }: KWResponse.Result<Array<MenuResponse>> = await CurrentMenuApi()

if (code === 200) {

console.log(data)

const menus: Array<MenuResponse> = data // data.filter(item => item.name.indexOf('vue') > -1) // 暂时先这么处理

// this.menus = data

MenuModule.changeMenu(menus)

PermissionModule.generateRoutes()

// router.addRoutes(PermissionModule.getDynamicRoutes)

if (settings.isEnableWebSocket) {

SocketModule.initSocket()

}

next({ ...to, replace: true } as RawLocation)

} else {

Message.error(msg || '获取当前用户菜单失败！')

}

}

6.2.4、把菜单信息存储到中MenuModule的vuex中

6.2.5、PermissionModule的generateRoutes方法通过进行动态路由解析组装，并把路由信息动态加载到router中，进行实际路由

export interface IPermissionState {

routes: Array<RouteConfig>

dynamicRoutes: Array<RouteConfig>

}

@Module({ dynamic: true, store, name: 'permission' })

class PermissionStore extends VuexModule implements IPermissionState {

public routes: Array<RouteConfig> = []

public dynamicRoutes: Array<RouteConfig> = []

public indexRoute: RouteConfig = {

path: '/index',

name: 'Index',

redirect: '/home',

component: () => import(/\* webpackChunkName: "index" \*/ '@/views/index/Index.vue'),

children: [

// 这个是空白页面，重新加载当前页面会用到

{

name: 'blank',

path: '/blank'

},

{

path: '/home',

name: 'home',

component: () => import(/\* webpackChunkName: "home" \*/ '@/views/index/home/Home.vue'),

meta: {

title: '首页'

}

}

]

}

public defaultRout: RouteConfig = {

path: '/',

redirect: '/index'

}

public route404: RouteConfig = {

path: '\*',

name: '/404',

component: () => import(/\* webpackChunkName: "test" \*/ '@/components/404.vue')

}

get getDynamicRoutes() {

return this.dynamicRoutes

}

@Mutation

public SET\_ROUTES(childrenRoutes: Array<RouteConfig>): void {

this.routes = []

this.dynamicRoutes = []

if (childrenRoutes.length > 0) {

childrenRoutes.forEach(item => {

const children = this.indexRoute.children as Array<RouteConfig>

children.push(item)

})

}

// (this.indexRoute.children as Array<RouteConfig>).push(this.route404)

this.routes = constantRoutes.concat(this.indexRoute)

this.routes.push(this.defaultRout)

this.routes.push(this.route404)

this.dynamicRoutes = [this.indexRoute, this.defaultRout, this.route404]

// router.addRoutes(this.dynamicRoutes)

router.addRoute(this.indexRoute)

router.addRoute(this.defaultRout)

router.addRoute(this.route404)

}

@Mutation

public CLEAR\_ROUTES(): void {

this.routes = []

this.dynamicRoutes = []

router.replace({ path: '/login' })

}

@Action({ commit: 'CLEAR\_ROUTES' })

public clearRoutes(): void {

console.log('clearRoutes')

}

@Action

public generateRoutes(): void {

const menus: Array<MenuResponse> = MenuModule.getMenus

const routes: Array<RouteConfig> = []

asyncRouter(menus, routes)

this.context.commit('SET\_ROUTES', routes)

// this.SET\_ROUTES(routes)

}

}

export const asyncRouter = (menus: Array<MenuResponse>, routes: Array<RouteConfig>): void => {

if (menus != null && menus.length > 0) {

menus.forEach(menu => {

if (menu.subMenus != null && menu.subMenus.length > 0) {

asyncRouter(menu.subMenus, routes)

} else {

/\*\*

\* Note: sub-menu only appear when route children.length >= 1

\* Detail see: https://panjiachen.github.io/vue-element-admin-site/guide/essentials/router-and-nav.html

\*

\* hidden: true if set true, item will not show in the sidebar(default is false)

\* alwaysShow: true if set true, will always show the root menu

\* if not set alwaysShow, when item has more than one children route,

\* it will becomes nested mode, otherwise not show the root menu

\* redirect: noRedirect if set noRedirect will no redirect in the breadcrumb

\* name:'router-name' the name is used by <keep-alive> (must set!!!)

\* meta : {

roles: ['admin','editor'] control the page roles (you can set multiple roles)

title: 'title' the name show in sidebar and breadcrumb (recommend set)

icon: 'svg-name'/'el-icon-x' the icon show in the sidebar

noCache: true if set true, the page will no be cached(default is false)

affix: true if set true, the tag will affix in the tags-view

breadcrumb: false if set false, the item will hidden in breadcrumb(default is true)

activeMenu: '/example/list' if set path, the sidebar will highlight the path you set

}

\*/

const route: RouteConfig = {

path: menu.url,

name: menu.url.substr(1),

component: loadViewsd(menu.path),

meta: {

title: menu.name

}

}

routes.push(route)

}

})

}

}

export const loadViewsd = (view: string) => {

// eslint-disable-next-line @typescript-eslint/no-explicit-any

return (resolve: (...modules: any[]) => void): void => require([`@/views/index/${view}.vue`], resolve)

}

export const PermissionModule = getModule(PermissionStore)

6.3、通过上面的路由信息看到，路由到index之后，其实跳转到home路由，进入到Home.vue页面

index.vue

<template>

<div class="index">

<el-container class="index-container">

<LeftMenu :menusList="menus"></LeftMenu>

<el-container>

<el-header>

<HeaderNav></HeaderNav>

</el-header>

<PageTabs :keep-alive-component-instance="keepAliveComponentInstance" />

<el-main>

<div ref="keepAliveContainer" style="padding-top:20px;background-color: #fff;">

<keep-alive>

<router-view :key="$route.fullPath" />

</keep-alive>

</div>

</el-main>

<el-footer>

<div class="footer">

Copyright © {{ new Date().getFullYear() }} key-win All rights reserved.

<span class="pull-right">Version 2.0</span>

</div>

</el-footer>

</el-container>

</el-container>

<KWUploader></KWUploader>

</div>

</template>

6.3.1、LeftMenu组件，进行菜单渲染

6.3.2、HeaderNav组件，进行主页面头部渲染

6.3.3、PageTabs组件，进行页面多页签渲染

6.3.4、KWUploader组件，进行上传页面渲染

6.4、Home.vue页面渲染

* 7、添加一个的业务功能
  + 7.1、在views/index/business下业务文件
  + 7.2、在业务文件下新建interface，此文件夹一般存在对应的后端返回的模型的interfaces
  + 7.3、xxx-api.ts文件，些文件对应访问后端的api
  + 7.4、Xxx.vue文件，对应页面文件
* 8、权限的使用:系统的权限是细粒度权限，已经设计到button或link上

8.1、系统的权限存储在UserModule中的loginUser对象中的permissions属性中，是一个数组集合

const permissions = (UserModule.loginUser as LoginSuccessUserInfo).permissions

8.2、权限工具类，提供基于权限code的验证或基于角色code的验证和一些公共的验证权限方法

hasPermission: (code: string): boolean => {...}

hasRole: (code: string): boolean => {...}

hasAnyRole(roles: Array<string>): boolean {...}

hasRoles(roles: Array<string>): boolean {...}

hasAnyPermission(codes: Array<string>): boolean {...}

hasPermissions(codes: Array<string>): boolean {...}

hasPermissionForDelete: (permissionPrefix: string): boolean => {...}

hasPermissionForUpdate: (permissionPrefix: string): boolean => {...}

hasPermissionForAdd: (permissionPrefix: string): boolean => {...}

hasPermissionForEnabled: (permissionPrefix: string): boolean => {...}

hasPermissionForGetId: (permissionPrefix: string): boolean => {...}

hasPermissionForQueryList: (permissionPrefix: string): boolean => {...}

hasPermissionForQueryPaged: (permissionPrefix: string): boolean => {...}

hasPermissionForExport: (permissionPrefix: string): boolean => {...}

hasPermissionForImport: (permissionPrefix: string): boolean => {...}

hasPermissionForDownload: (permissionPrefix: string): boolean => {...}

hasPermissionForUpload: (permissionPrefix: string): boolean => {...}

8.3、为方便权限验证封装了一些指标

const hasPermissionAdd: DirectiveOptions = {...}

const hasPermissionDelete: DirectiveOptions = {...}

const hasPermission: DirectiveOptions = {...}

const hasPermissionDownload: DirectiveOptions = {...}

const hasPermissionEnabled: DirectiveOptions = {...}

const hasPermissionExport: DirectiveOptions = {...}

const hasPermissionGetId: DirectiveOptions = {...}

const hasPermissionImport: DirectiveOptions = {...}

const hasPermissionQueryList: DirectiveOptions = {...}

const hasPermissionQueryPage: DirectiveOptions = {...}

const hasPermissionUpdate: DirectiveOptions = {...}

const hasPermissionUpload: DirectiveOptions = {...}

8.4、v-xxx指令的权限验证

8.4.1、查询权限、添加用户权限

<el-input placeholder="请输入内容" v-model="t.nickName" v-hasPermissionQueryPage="userPermission">

<el-button slot="append" class="search-primary" icon="el-icon-search" @click="searchUser"></el-button>

</el-input>

<el-button type="primary" @click="addUser" v-hasPermissionAdd="userPermission">添加用户</el-button>

8.4.2、userPermission权限基于是一个工具类，把所有业务对应的权限前缀定义到PermissionPrefixUtils中，方便后续应用

userPermission = PermissionPrefixUtils.user

const PermissionPrefixUtils = {

user: 'system::user::SysUser::',

role: 'system::sys-role::SysRole::',

roleMenuPermission: 'system::role-menu-permission::SysRoleMenuPermission::',

permission: 'system::permission::SysPermission::',

menuPermission: 'system::menu-permission::SysMenuPermission::',

menu: 'system::menu::SysMenu::',

dictType: 'common::param::dict-type::DictType::',

dictTree: 'common::param::dict-tree::DictTree::',

dictData: 'common::param::dict-data::DictData::',

fileInfo: 'common::file::FileInfo::',

dataLog: 'common::data-log::DataLog::',

customerInfo: 'business::customer::CustomerInfo::',

deviceAuth: 'business::device::DeviceAuth::'

}

export default PermissionPrefixUtils

8.5、调用方法手动验证

8.5.1、状态权限验证html片段

<el-tooltip effect="dark" content="字典数据管理" v-if="hasPermission(scope.row)" placement="top" :enterable="false">

...

</el-tooltip>

8.5.2、hasPermission验证

hasPermission(data: SysDictType): boolean {

if ((data.type as Model.EnumEntity).stringValue === Type.列表) {

return PermissionUtil.hasPermission(PermissionCodeUtils.dictTypeGrantDictTypeGotoDictData)

} else {

return PermissionUtil.hasPermission(PermissionCodeUtils.dictTypeGrantDictTypeGotoDictTree)

}

}

hasPermissionEnabled(): boolean {

return PermissionUtil.hasPermissionForEnabled(this.dicTypePermissionPrefix)

}

* 9、v-viewer指令的使用，在img的父元素中添加v-viewer

<div v-viewer>

<img src="@/assets/404.gif" title="点击预览" width="300px" height="200px">&nbsp;&nbsp;&nbsp;&nbsp;

<img src="@/assets/head.png" title="点击预览" width="200px" height="200px">&nbsp;&nbsp;&nbsp;&nbsp;

<img src="@/assets/keywin.png" width="300px" title="点击预览" height="200px">&nbsp;&nbsp;&nbsp;&nbsp;

<img src="@/assets/logo.png" title="点击预览" width="200px" height="200px">

</div>

* 10、table组件的使用

10.1、导入分页组件，定义KWTable<T, RT>，注意泛型，T为输出对象，RT为输出对象

import KWTable from '@/components/table/Table.vue'

@Ref('kwTableRef')

readonly kwTableRef!: KWTable<UserSearchRequest, UserInfo>

10.2、编写table模板，需要注意的是请在table模板中指定url

<KWTable url="user/findSysUserByPaged" style="width: 100%" ref="kwTableRef">

<el-table-column type="index" width="80" label="序号"></el-table-column>

<el-table-column prop="userName" sortable="custom" label="帐号"> </el-table-column>

<el-table-column prop="nickName" sortable="custom" label="昵称"> </el-table-column>

<el-table-column prop="phone" sortable="custom" label="手机"> </el-table-column>

<el-table-column

prop="sex"

label="性别"

sortable="custom"

:formatter="

row => {

return row.sex.text

}

"

>

</el-table-column>

<el-table-column prop="createDate" label="创建时间" sortable="custom">

<template slot-scope="scope">{{ scope.row.createDate | dateTimeFormat }}</template>

</el-table-column>

<el-table-column prop="isEnabled" label="状态" sortable="custom">

<template v-slot="scope">

<el-switch v-model="scope.row.enabled" active-color="#13ce66" inactive-color="#ff4949" @change="userStatuChanged(scope.row, scope.row.enabled)"> </el-switch>

</template>

</el-table-column>

<el-table-column label="操作">

<template v-slot="scope">

<el-button type="primary" icon="el-icon-edit" size="mini" @click="showEditDialog(scope.row.id)"></el-button>

<el-tooltip effect="dark" content="重置密码" placement="top" :enterable="false">

<el-button type="warning" icon="el-icon-setting" size="mini" @click="passwordReset(scope.row.id)"></el-button>

</el-tooltip>

</template>

</el-table-column>

</KWTable>

10.3、如果有查询条件，请定义分页输入对象，然后调用查询方法进行数据加载

t: UserSearchRequest = { nickName: '' }

this.kwTableRef.loadByCondition(this.t)

10.4、最后此组件主要是借鉴：https://gitee.com/virens/vue-demo/blob/master/src/components/table/VirTable.vue

* 11、file-uploader组件的使用

全局上传插件，两种调用方式

11.1. 作为全局页面的组件，使用event bus

调用方法：EventHub.$emit('openUploader', {params: {}, options: {}})

params: 发送给服务器的额外参数；

options：上传选项，目前支持 target、testChunks、mergeFn、accept

监听函数：EventHub.$on('fileAdded', fn); 文件选择后的回调

EventHub.$on('fileSuccess', fn); 文件上传成功的回调，监听后记得释放

11.1.1、使用在index.vue中引入

import KWUploader from '@/components/file-uploader/GlobalUploader.vue'

@Component({

components: {

...,

KWUploader

}

})

<KWUploader></KWUploader>

11.1.2、在Home.vue调用

<el-button type="primary" size="medium" @click="upload">上传</el-button>

upload(): void {

// 打开文件选择框

Bus.$emit('openUploader', {

// 给服务端的额外参数

params: {

bizType: 'default'

}

})

}

2. 作为普通组件在单个页面中调用，使用props

* 12、Tip组件的使用

12.1、导入组件

import KWCell from '@/components/cell/Cell.vue'

import KWText from '@/components/text/Text.vue'

@Component({

components: {

KWCell,

KWText

}

})

12.2、组件使用

<KWCell :gap="15" label="滕王阁序" style="width: 300px">

<KWText value="豫章故郡，洪都新府。星分翼轸，地接衡庐。襟三江而带五湖，控蛮荆而引瓯越。物华天宝，龙光射牛斗之墟；人杰地灵，徐孺下陈蕃之榻。雄州雾列，俊采星驰。" :row="1" />

</KWCell>

* 13、src-directive指令的使用

<img v-src="headImgUrl" class="user-avatar" />

get headImgUrl(): string | null {

return (UserModule.loginUser as LoginSuccessUserInfo).user.headImgUrl as string

}

* 14、form的使用及验证

14.1、html片段

<el-form :model="customerInfoForm" :inline="true" :rules="customerInfoFormRules" ref="customerInfoFormRef" label-width="100px">

<el-form-item label="客户编号" prop="sequence">

<el-input v-model="customerInfoForm.sequence" style="max-width: 220px;"

:disabled="customerInfoSequenceDisabled">

</el-input>

</el-form-item>

<el-form-item label="客户名称" prop="companyName">

<el-input v-model="customerInfoForm.companyName" style="max-width: 220px;"></el-input>

</el-form-item>

<el-form-item label="客户地址" prop="companyAddress">

<el-input v-model="customerInfoForm.companyAddress" style="max-width: 220px;"></el-input>

</el-form-item>

<el-form-item label="客户电话" prop="companyPhone">

<el-input v-model="customerInfoForm.companyPhone" style="max-width: 220px;"></el-input>

</el-form-item>

<el-form-item label="联系人姓名" prop="leadName">

<el-input v-model="customerInfoForm.leadName" style="max-width: 220px;"></el-input>

</el-form-item>

<el-form-item label="联系人手机" prop="leadMobile">

<el-input v-model="customerInfoForm.leadMobile" style="max-width: 220px;"></el-input>

</el-form-item>

<el-form-item label="项目号" prop="projectNo">

<el-input v-model="customerInfoForm.projectNo" style="max-width: 220px;"></el-input>

</el-form-item>

<el-form-item label="项目名称" prop="projectName">

<el-input v-model="customerInfoForm.projectName" style="max-width: 220px;"></el-input>

</el-form-item>

<el-form-item label="授权码" prop="authDeviceCode">

<el-input v-model="customerInfoForm.authDeviceCode" :disabled="customerInfoAuthDeviceCodeDisabled"

style="max-width: 220px;"></el-input>

</el-form-item>

<el-form-item label="授权设备数" prop="authDeviceNum">

<el-input v-model="customerInfoForm.authDeviceNum" type="number" style="max-width: 220px;"></el-input>

</el-form-item>

<el-form-item label="是否校验日期" prop="isVerify">

<el-radio-group @change="isVerifChange" v-model="customerInfoForm.isVerify" style="width: 202px;">

<el-radio label="是"></el-radio>

<el-radio label="否"></el-radio>

</el-radio-group>

</el-form-item>

<el-form-item label="授权到期日期" prop="expireDeviceDate">

<el-date-picker v-model="expireDeviceDate" @input="onDatePickerChange" type="date" placeholder="授权到期日期"

style="max-width: 220px;">

</el-date-picker>

</el-form-item>

</el-form>

14.2、form验证

readonly customerInfoFormRules: { companyName: Array<KWRule.Rule>; companyAddress: Array<KWRule.Rule>; companyPhone: Array<KWRule.Rule>; leadName: Array<KWRule.Rule>; leadMobile: Array<KWRule.Rule | KWRule.ValidatorRule>; projectNo: Array<KWRule.Rule>; authDeviceCode: Array<KWRule.Rule | KWRule.MixinRule>; authDeviceNum: Array<KWRule.NumberRule>; expireDeviceDate: Array<KWRule.ValidatorRule> } = {

companyName: [FormValidatorRule.requiredRule('请输入客户名称')],

companyAddress: [FormValidatorRule.requiredRule('请输入客户地址')],

companyPhone: [FormValidatorRule.requiredRule('请输入客户电话')],

leadName: [FormValidatorRule.requiredRule('请输入联系人姓名')],

leadMobile: [FormValidatorRule.requiredRule('请输入联系人手机'), { validator: FormValidatorRule.checkMobeli, trigger: 'blur' }],

projectNo: [FormValidatorRule.requiredRule('请输入项目号')],

authDeviceCode: [FormValidatorRule.requiredRule('请输入授权码'), FormValidatorRule.mixinRul(4, 10, '授权码值的长度4~10个字符之间')],

authDeviceNum: [FormValidatorRule.numberRule('请输入授权设备台数')],

expireDeviceDate: [{ validator: this.checkExpireDeviceDate, trigger: 'blur' }]

}

// 验证设备的授权到期日期

checkExpireDeviceDate(rule: KWRule.ValidatorRule, value: string, cb: KWRule.CallbackFunction): void {

if (this.customerInfoForm.isVerify === '是') {

if (!this.expireDeviceDate) {

cb(new Error('请选择设备到期日期'))

}

if (new Date().getTime() >= (this.expireDeviceDate as Date).getTime()) {

cb(new Error('设备到期日期必须大于当前日期'))

}

}

return cb()

}

* 15、web-socket的使用

15.1、在对应的环境变量中配置VUE\_APP\_WEBSOCKET\_BASE\_WS\_URL的api的地址：VUE\_APP\_WEBSOCKET\_BASE\_WS\_URL='ws://127.0.0.1:9902/ws/'

15.2、在settings中开启isEnableWebSocket:true

15.2、启动流程:在登录成功后并加载用户信息和权限数据后，开启websocket连接

permission.ts

if (settings.isEnableWebSocket) {

SocketModule.initSocket()

}

15.4、weboscket-client端中所有发出的请求都走http请求，发出请求的入口Controller:WebSocketCtrl,请求地址为:http://x.x.x.x:9902/ws/\*\*,主要有如下的方法：

15.4.1、请求地址:http://x.x.x.x:9902/ws/sendByUserName/{userName}/{message}

@ApiOperation(value = "给单用户发送信息")

@GetMapping("/sendByUserName/{userName}/{message}")

@LogAnnotation(module = "websocket-center", recordRequestParam = false)

public Result sendByUserName(@PathVariable String userName, @PathVariable String message) {...}

15.4.2、请求地址:http://x.x.x.x:9902/ws/sendByToken/{toToken}/{message}

@ApiOperation(value = "给单用户发送信息")

@GetMapping("/sendByToken/{toToken}/{message}")

@LogAnnotation(module = "websocket-center", recordRequestParam = false)

public Result sendByToken(@PathVariable String toToken, @PathVariable String message) {...}

15.4.3、请求地址:http://x.x.x.x:9902/ws/sendByUserName/msg

@ApiOperation(value = "给单用户发送信息")

@PostMapping("/sendByUserName/msg")

@LogAnnotation(module = "websocket-center", recordRequestParam = false)

public Result sendMsgByUserName(@RequestBody WebsocketUserMessage websocketUserMessage) {...}

15.4.4、请求地址:http://x.x.x.x:9902/ws/sendByToken/msg

@ApiOperation(value = "给单用户发送信息")

@PostMapping("/sendByToken/msg")

@LogAnnotation(module = "websocket-center", recordRequestParam = false)

public Result sendMsgByToken(@RequestBody WebsocketTokenMessage websocketTokenMessage) {...}

15.4.5、请求地址:http://x.x.x.x:9902/ws/sendByTokens/msg

@ApiOperation(value = "给多用户发送信息")

@PostMapping("/sendByTokens/msg")

@LogAnnotation(module = "websocket-center", recordRequestParam = false)

public Result sendMsgByTokens(@RequestBody WebsocketTokensMessage websocketTokensMessage) {...}

15.4.6、请求地址:http://x.x.x.x:9902/ws/send/group/msg

@ApiOperation(value = "给用户列表发送信息")

@PostMapping("/send/group/msg")

@LogAnnotation(module = "websocket-center", recordRequestParam = false)

public Result sendGroupMsg(@RequestBody WebsocketGroupMessage websocketGroupMessage) {...}

15.4、weboscket只接收来至服务端推送的信息,接收推送消息的入口在:web-socket-store.ts中的initSocket()方法。获取WebSocketActionProcess实例对象，并把返回的json信息传入processAction方法进行处理

15.5、websocket的接收到消息的处理流程

15.5.1、IWebSocketBaseMessage是websocket推消息的基类接口:

export interface IWebSocketBaseMessage {

token: string

fromUserName: string

message: string

action: string

}

15.5.1.1、其它接口均以继承IWebSocketBaseMessage来替换业务的差异:

export interface WebSocketMessage extends IWebSocketBaseMessage {

toUserName: string

}

export interface WebSocketGroupMessage extends IWebSocketBaseMessage {

toUserNames: Array<string>

}

export interface WebSocketGroupIdMessage extends IWebSocketBaseMessage {

groupId: string

}

15.5.2、IAction是所有消息的接口，此接口只定义了两个方法:

export interface IAction {

processMessage(): void

getFullJson(): string

}

15.5.2.1、消息的处理方法：processMessage(): void

15.5.2.2、getFullJson(): string 获取完整的json

15.5.2.3、WebSocketBaseAction一个抽象类，实现了IAction是所有消息的方法，它定义的websocket接收的实体对象。

export abstract class WebSocketBaseAction implements IAction {

webSocketMessage!: WebSocketMessage

abstract processMessage(): void

getFullJson(): string {

return JSON.stringify(this.webSocketMessage)

}

}

15.5.2.4、其它消息处理均继承WebSocketBaseAction，实现抽象方法processMessage处理具体的业务，它定义的websocket接收的实体对象

export class MessageNotifyAction extends WebSocketBaseAction {

processMessage(): void {

Notification.success(this.webSocketMessage.message)

}

}

15.5.3、WebSocketActionProcess是一个单例的类，其主要目的是根据不同的action调用不同的processMessage方法进行业务处理，需要注意是多一个Action类，就需要在initAction方法中把自己注入进去

class WebSocketActionProcess {

private static \_instance: WebSocketActionProcess | null = null

private static \_items: { [key: string]: IAction } = {}

private WebSocketActionProcess() {

console.log('private constructor')

}

// 添加具体的Action到\_items中来

private static initAction() {

WebSocketActionProcess.getInstance().set('webSocketAction', new WebSocketAction())

WebSocketActionProcess.getInstance().set('messageNotifyAction', new MessageNotifyAction())

WebSocketActionProcess.getInstance().set('deviceOnLineNotifyAction', new DeviceOnLineNotifyAction())

WebSocketActionProcess.getInstance().set('deviceOffLineNotifyAction', new DeviceOffLineNotifyAction())

}

// 获得实例对象

public static getInstance(): WebSocketActionProcess {

if (!this.\_instance) {

this.\_instance = new WebSocketActionProcess()

this.initAction()

}

return this.\_instance

}

set(key: string, value: IAction): void {

WebSocketActionProcess.\_items[key] = value

console.log(`set cache with key: '${key}', value: '${value}'`)

}

get(key: string): IAction {

const value = WebSocketActionProcess.\_items[key]

console.log(`get cache value: '${value}' with key: '${key}'`)

return value

}

// 解析调用具体的processMessage方法

processAction(jsonStr: string): void {

const josn = JSON.parse(jsonStr)

const action = this.get(josn.action)

const webSocketBaseAction = action as WebSocketBaseAction

webSocketBaseAction.webSocketMessage = josn

action.processMessage()

}

}

* 16、Element的组件使用
  + 16.1 src/plugins/element.js
  + 16.2 在import中导入对应的组件
  + 16.3 添加Vue.use(组件)
* 17、LeftMenu组件说明，此组件是页面框架的左边菜单区域

17.1、在index.vue中从MenuModule中获取菜单，并把menus做为参数传为LeftMenu组件中

import LeftMenu from '@/components/left/LeftMenu.vue'

@Component({

components: {

...

LeftMenu,

...

}

})

this.menus = MenuModule.getMenus

<LeftMenu :menusList="menus"></LeftMenu>

17.2、在leftMenu组件进行循环打印渲染

<el-menu background-color="#333744" text-color="#fff" active-text-color="#409EFF" unique-opened :collapse="collapseMenuState" :collapse-transition="false" router :default-active="activePath">

<!-- 一级菜单 -->

<template v-for="item in menusList">

<el-submenu v-if="item.isHidden === false && item.isMenu === 1" :index="item.id+''" :key="item.id">

<!-- 一级菜单模板区 -->

<template slot="title">

<i :class="item.css"></i>

<span>{{ item.name }}</span>

</template>

<!--二级菜单-->

<template v-for="subItem in item.subMenus">

<el-menu-item v-if="subItem.isHidden === false && subItem.isMenu === 1" :index="subItem.url" :key="subItem.id" @click="saveActivePath(subItem.url)">

<!-- 二级菜单模板区 -->

<template slot="title">

<i :class="subItem.css"></i>

<span>{{ subItem.name }}</span>

</template>

</el-menu-item>

</template>

</el-submenu>

</template>

</el-menu>

// 接收父组件传过来的参数，默认值一个空数组

export default class LeftMenu extends Vue {

...

@Prop({ default: [] })

readonly menusList!: Array<MenuResponse> | []

...

}

* 18、HeaderNav组件说明，此组件是页面框架的顶部区域，主要功能是获取用户信息，加载头像及修改用户信息等内容

18.1、在index.vue中从MenuModule中获取菜单，并把menus做为参数传为LeftMenu组件中

import HeaderNav from '@/components/header/HeaderNav.vue'

@Component({

components: {

...

HeaderNav,

...

}

})

<HeaderNav></HeaderNav>

18.2、在HeaderNav组件中获取用户信息并使用

get nickName(): string {

const user = (UserModule.loginUser as LoginSuccessUserInfo).user

console.log(user && user.nickName)

if (user !== null) {

return user.nickName

}

return ''

}

get headImgUrl(): string | null {

return (UserModule.loginUser as LoginSuccessUserInfo).user.headImgUrl as string

}

<div class="avatar-wrapper">

<img v-src="headImgUrl" class="user-avatar" />

<div>

<span>{{ nickName }}</span>

<i class="el-icon-caret-bottom" />

</div>

</div>

* 19、PageTabs组件的说明，此组件是系统中的多页签的应用，应该中已经应用，不需要二使用

19.1、在index.vue中引入PageTabs组件

<PageTabs :keep-alive-component-instance="keepAliveComponentInstance" />

<el-main>

<div ref="keepAliveContainer" style="padding-top:20px;background-color: #fff;">

<keep-alive>

<router-view :key="$route.fullPath" />

</keep-alive>

</div>

</el-main>

import PageTabs from '@/components/page-tabs/PageTabs.vue'

@Component({

components: {

...

PageTabs,

...

}

})

mounted(): void {

if (this.$refs.keepAliveContainer) {

// eslint-disable-next-line @typescript-eslint/no-explicit-any

this.keepAliveComponentInstance = (this.$refs.keepAliveContainer as any).childNodes[0].\_\_vue\_\_ // 获取keep-alive的控件实例对象

}

}

19.2、PageTabs组件

<div class="\_\_common-layout-pageTabs">

<el-scrollbar>

<div class="\_\_tabs">

<div class="\_\_tab-item" v-for="item in openedPageRouters" :class="{

'\_\_is-active': item.fullPath == $route.fullPath

}" :key="item.fullPath" @click="onClick(item)" @contextmenu.prevent="showContextMenu($event, item)">

{{ item.meta.title }}

<span class="el-icon-close" @click.stop="onClose(item)" @contextmenu.prevent.stop=""

:style="openedPageRouters.length <= 1 ? 'width:0;' : ''"></span>

</div>

</div>

</el-scrollbar>

<div v-show="contextMenuVisible">

<ul :style="{ left: contextMenuLeft + 'px', top: contextMenuTop + 'px' }" class="\_\_contextmenu">

<li>

<el-button type="text" @click="reload()" size="mini">

重新加载

</el-button>

</li>

<li>

<el-button type="text" @click="closeOtherLeft" :disabled="false" size="mini">关闭左边</el-button>

</li>

<li>

<el-button type="text" @click="closeOtherRight" :disabled="false" size="mini">关闭右边</el-button>

</li>

<li>

<el-button type="text" @click="closeOther" size="mini">关闭其他</el-button>

</li>

</ul>

</div>

</div>

</template>

<script lang="ts">

import { Component, Prop, Vue, Watch } from 'vue-property-decorator'

import { Route, RouteMeta } from 'vue-router'

@Component

export default class PageTabs extends Vue {

@Prop({ default: {} })

// eslint-disable-next-line @typescript-eslint/no-explicit-any

keepAliveComponentInstance: any

@Prop({ default: 'blank' })

blankRouteName!: string // 空白路由的name值

contextMenuVisible = false // 右键菜单是否显示

contextMenuLeft = 0 // 右键菜单显示位置

contextMenuTop = 0 // 右键菜单显示位置

contextMenuTargetPageRoute: Route | null = null // 右键所指向的菜单路由

openedPageRouters: Array<Route> = [] // 已打开的路由页面

@Watch('$route', { immediate: true })

routechange(to: Route, from: Route): void {

console.log(to, from)

this.openPage(to)

}

mounted(): void {

// 添加点击关闭右键菜单

window.addEventListener('click', this.closeContextMenu)

}

destroyed(): void {

window.removeEventListener('click', this.closeContextMenu)

}

// 隐藏右键菜单

closeContextMenu(): void {

this.contextMenuVisible = false

this.contextMenuTargetPageRoute = null

}

openPage(route: Route): void {

if (route.name === this.blankRouteName) {

return

}

const isExist = this.openedPageRouters.some(item => item.fullPath === route.fullPath)

if (!isExist) {

const openedPageRoute = this.openedPageRouters.find(item => item.path === route.path)

// 判断页面是否支持不同参数多开页面功能，如果不支持且已存在path值一样的页面路由，那就替换它

if (!(route.meta as RouteMeta).canMultipleOpen && openedPageRoute != null) {

this.delRouteCache(openedPageRoute.fullPath)

this.openedPageRouters.splice(this.openedPageRouters.indexOf(openedPageRoute), 1, route)

} else {

this.openedPageRouters.push(route)

}

}

}

// 点击页面标签卡时

onClick(route: Route): void {

if (route.fullPath !== this.$route.fullPath) {

this.$router.push(route.fullPath)

}

}

// 关闭页面标签时

onClose(route: Route): void {

let index: number = this.openedPageRouters.indexOf(route)

this.delPageRoute(route)

if (route.fullPath === this.$route.fullPath) {

// 删除页面后，跳转到上一页面

index = index === 0 ? 0 : index - 1

const r: Route = this.openedPageRouters[index]

this.$router.replace({ name: r.name as string })

}

}

// 右键显示菜单

// eslint-disable-next-line @typescript-eslint/no-explicit-any, @typescript-eslint/explicit-module-boundary-types

showContextMenu(e: any, route: Route): void {

this.contextMenuTargetPageRoute = route

this.contextMenuLeft = e.layerX

this.contextMenuTop = e.layerY

this.contextMenuVisible = true

}

// 重载页面

reload(): void {

const contextMenuTargetPageRoute = this.contextMenuTargetPageRoute as Route

this.delRouteCache(contextMenuTargetPageRoute.fullPath)

if (contextMenuTargetPageRoute.fullPath === this.$route.fullPath) {

this.$router.replace({ name: this.blankRouteName }).then(() => {

this.$router.replace({ name: contextMenuTargetPageRoute.name as string })

})

}

}

// 关闭其他页面

closeOther(): void {

const contextMenuTargetPageRoute = this.contextMenuTargetPageRoute as Route

for (let i = 0; i < this.openedPageRouters.length; i++) {

const r = this.openedPageRouters[i]

if (r !== this.contextMenuTargetPageRoute) {

this.delPageRoute(r)

i--

}

}

if (contextMenuTargetPageRoute.fullPath !== this.$route.fullPath) {

this.$router.replace({ name: contextMenuTargetPageRoute.name as string })

}

}

// 根据路径获取索引

getPageRouteIndex(fullPath: string): number {

for (let i = 0; i < this.openedPageRouters.length; i++) {

if (this.openedPageRouters[i].fullPath === fullPath) {

return i

}

}

return -1

}

// 关闭左边页面

closeOtherLeft(): void {

const contextMenuTargetPageRoute = this.contextMenuTargetPageRoute as Route

let index = this.openedPageRouters.indexOf(this.contextMenuTargetPageRoute as Route)

const currentIndex = this.getPageRouteIndex(this.$route.fullPath)

if (index > currentIndex) {

this.$router.replace({ name: contextMenuTargetPageRoute.name as string })

}

for (let i = 0; i < index; i++) {

const r = this.openedPageRouters[i]

this.delPageRoute(r)

i--

index--

}

}

// 关闭右边页面

closeOtherRight(): void {

const contextMenuTargetPageRoute = this.contextMenuTargetPageRoute as Route

const index = this.openedPageRouters.indexOf(this.contextMenuTargetPageRoute as Route)

const currentIndex = this.getPageRouteIndex(this.$route.fullPath)

for (let i = index + 1; i < this.openedPageRouters.length; i++) {

const r = this.openedPageRouters[i]

this.delPageRoute(r)

i--

}

if (index < currentIndex) {

this.$router.replace({ name: contextMenuTargetPageRoute.name as string })

}

}

// 删除页面

delPageRoute(route: Route): void {

const routeIndex = this.openedPageRouters.indexOf(route)

if (routeIndex >= 0) {

this.openedPageRouters.splice(routeIndex, 1)

}

this.delRouteCache(route.fullPath)

}

// 删除页面缓存

delRouteCache(key: string): void {

const cache = this.keepAliveComponentInstance.cache

const keys = this.keepAliveComponentInstance.keys

for (let i = 0; i < keys.length; i++) {

if (keys[i] === key) {

keys.splice(i, 1)

if (cache[key] != null) {

delete cache[key]

}

break

}

}

}

}

* 20、drag-directive指令的使用,直接在目标上添加指令v-drag

<div class="v-im" id="v-im" v-drag>

</div>

* 21、time-directive指令的使用

<span v-time="item.timestamp" />

* 22、filter的使用

22.1、定义filter

Vue.filter('dateTimeFormat', function(dateTime: string | number | Date): string {

return dateFormat(dateTime, DateFormatType.DateTime)

})

22.2、使用filter

{{ scope.row.createDate | dateTimeFormat }}

**docker化的部署使用说明**

* 1、docker的部署有两种方式：

1.1、通过docker或docker-compose部署

1.2、通过rancher+k8s容器化部署

* 2、打包应用生成docker镜像,上传至docker私服仓库,方式有两种：

2.1、通过jenkins打包生成docker镜像上传至私服仓库

2.2、通过docker-maven-plugin插件上传docker镜像至私服仓库

* 3、docker的使用公共配置(docker-maven-plugin方式配置),在根pom.xml中配置ad

3.1、docker-maven-plugin的配置，主要是配置serverId（主要是登录docker仓库中，这个对应的是setting.xml中server配置的id）、imageName、dockerDirectory、dockerHost等

<build>

<pluginManagement>

<plugins>

<plugin>

<groupId>com.spotify</groupId>

<artifactId>docker-maven-plugin</artifactId>

<version>1.2.2</version>

<configuration>

<serverId>docker-hub</serverId> <!--mvn setting.xml中server配置的那个id-->

<imageName>${docker.registry.url}/${docker.image.prefix}/${project.artifactId}</imageName>

<!-- 镜像tag-->

<imageTags>

<imageTag>latest</imageTag>

</imageTags>

<!--覆盖相同标签镜像-->

<forceTags>true</forceTags>

<!--Dockerfile文件的配置-->

<dockerDirectory>src/main/docker</dockerDirectory>

<!-- docker远程服务器地址 -->

<dockerHost>${docker.registry.host}</dockerHost>

<!-- 上传镜像-->

<!--<pushImage>true</pushImage>-->

<pushImageTag>true</pushImageTag>

<!-- 重试次数-->

<retryPushCount>3</retryPushCount>

<resources>

<resource>

<targetPath>/</targetPath>

<directory>${project.build.directory}</directory>

<include>${project.build.finalName}.jar</include>

<include>jdk-8u261-linux-x64.rpm</include>

</resource>

</resources>

</configuration>

</plugin>

</plugins>

</pluginManagement>

</build>

3.2、settings.xml的配置，主要是配置serverId（主要和上边的serverId对应,提供docker登录时的用户名和密码)

<servers>

<server>

<id>docker-hub</id>

<username>admin</username>

<password>admin123</password>

</server>

</servers>

3.4、docker-maven-plugin其它的变量引用

<properties>

<docker.image.prefix>key-win</docker.image.prefix>

<!-- docker私有仓库地址 -->

<docker.registry.url>192.168.1.11:8083</docker.registry.url>

<docker.registry.host>http://192.168.1.11:2375</docker.registry.host>

</properties>

3.5、jdk-8u261-linux-x64.rpm 为应用的jdk的版本

3.6、build-base.xml 是一个ant插件的应用，主要是方便开发人员以双击形式操作调用maven命令打包发布docker镜像，主要用以下ant脚本：

<target name="mvn-clean" description="除目标目录中的生成结果" depends="os-mvn-init">

<mvn>

<!--<arg value="-X" />-->

<arg value="clean" />

</mvn>

<mkdir dir="${basedir}/target" />

</target>

<target name="mvn-package-local" depends="mvn-clean">

<mvn>

<arg value="-P${env-profile-name}" />

<arg value="-Dmaven.test.skip=true" />

<!-- 依据项目生成 jar 文件 -->

<arg value="package" />

</mvn>

</target>

<target name="mvn-build-docker" depends="mvn-package-local">

<copy todir="${basedir}/target" overwrite="true" file="${docker.java.jdk.filePath}" />

<mvn>

<arg value="-X" />

<!--<arg value="-U" />-->

<arg value="docker:build" />

</mvn>

</target>

<target name="mvn-package-docker-docker" >

<antcall target="mvn-build-docker">

<param name="env-profile-name" value="docker" />

</antcall>

</target>

3.7、build.properties ant打包配置文件，主要是配置jdk和maven的路径信息

ANT\_MAVEN\_HOME=D:/dev-env/apache-maven-3.3.9

ANT\_JAVA\_HOME=D:/dev-env/Java/jdk1.8.0\_102

* 4、后端的docker的使用

4.1、pom.xml配置，主要是配置spring-boot-maven-plugin，定义jar的名称

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

<executions>

<execution>

<goals>

<goal>repackage</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

<finalName>${project.artifactId}</finalName>

</build>

4.2、build-base.xml

<project name="xxx" default="mvn-clean">

<loadproperties srcFile="../../build.properties" encoding="UTF-8"/><!--注意build.properties的相对路径-->

<import file="../../build-base.xml"/><!--注意build-base.xml的相对路径-->

<property name="project.jar.filename" value="${ant.project.name}.jar"/>

<property name="project.filePath" value="${ant.project.name}" />

<property name="docker.java.jdk.filePath" value="../../jdk-8u261-linux-x64.rpm" /><!--注意jdk-8u261-linux-x64.rpm的相对路径-->

</project>

4.3、Dockerfile文件的配置，位置于当前项目中的src/main/docker目录下

FROM centos:centos7.1.1503

#拷贝jdk

ADD jdk-8u261-linux-x64.rpm jdk-8u261-linux-x64.rpm

RUN rpm -ivh jdk-8u261-linux-x64.rpm

RUN rm jdk-8u261-linux-x64.rpm

#设置环境变量

ENV JAVA\_HOME=/usr/java/jdk1.8.0\_261-amd64

ENV JRE\_HOME=$JAVA\_HOME/jre

ENV CLASSPATH=.:$JAVA\_HOME/lib/dt.jar:$JAVA\_HOME/lib/tools.jar:$JRE\_HOME/lib:$CLASSPATH

ENV PATH=/sbin:$JAVA\_HOME/bin:$PATH

#查看环境变量

RUN java -version

ENV LC\_ALL=zh\_CN.utf8

ENV LANG=zh\_CN.utf8

ENV LANGUAGE=zh\_CN.utf8

RUN localedef -c -f UTF-8 -i zh\_CN zh\_CN.utf8

#RUN sed -i 's/dl-cdn.alpinelinux.org/mirrors.ustc.edu.cn/g' /etc/apk/repositories

VOLUME /tmp

#注意xxxx要改写成自己的打jar后的jar的名称

ADD xxxx.jar app.jar

#RUN apk --no-cache add tzdata && \

# ln -sf /usr/share/zoneinfo/Asia/Shanghai /etc/localtime && \

# echo "Asia/Shanghai" > /etc/timezone

#设置时间

RUN /bin/cp /usr/share/zoneinfo/Asia/Shanghai /etc/localtime && echo 'Asia/Shanghai' >/etc/timezone

RUN date +'%x %X.%N'

RUN sh -c 'touch /app.jar'

ENV JAVA\_OPTS=""

ENTRYPOINT [ "sh", "-c", "java $JAVA\_OPTS -Djava.security.egd=file:/dev/./urandom -jar /app.jar" ]

4.4、application-docker.yml的配置（一定要有一个application-docker.yml文件，因为应用ant打包发布时传进去的就是docker变量），这里主要是数据库配置和redis配置，数据库建议配置:192.168.1.29，redis建议配置:192.168.1.13

4.4.1、数据库配置:

spring:

datasource:

username: root

password: root

url: jdbc:mysql://192.168.1.29:3306/individual-soldier-auth?useUnicode=true&characterEncoding=UTF-8&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=Asia/Shanghai

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

4.4.2、redis配置:

spring:

redis:

################### redis 单机版 start ##########################

host: 192.168.1.13

port: 6379

timeout: 6000

database: 1

lettuce:

pool:

max-active: 10 # 连接池最大连接数（使用负值表示没有限制）,如果赋值为-1，则表示不限制；如果pool已经分配了maxActive个jedis实例，则此时pool的状态为exhausted(耗尽)

max-idle: 8 # 连接池中的最大空闲连接 ，默认值也是8

max-wait: 100 # # 等待可用连接的最大时间，单位毫秒，默认值为-1，表示永不超时。如果超过等待时间，则直接抛出JedisConnectionException

min-idle: 2 # 连接池中的最小空闲连接 ，默认值也是0

shutdown-timeout: 100ms

################### redis 单机版 end ##########################

# cluster:

# nodes: 130.75.131.237:7000,130.75.131.238:7000,130.75.131.239:7000,130.75.131.237:7001,130.75.131.238:7001,130.75.131.239:7001

# #130.75.131.237:7000,130.75.131.238:7000,130.75.131.239:7000,130.75.131.237:7001,130.75.131.238:7001,130.75.131.239:7001

# #192.168.3.157:7000,192.168.3.158:7000,192.168.3.159:7000,192.168.3.157:7001,192.168.3.158:7001,192.168.3.159:7001

# timeout: 1000 # 连接超时时间（毫秒）

# lettuce:

# pool:

# max-active: 10 # 连接池最大连接数（使用负值表示没有限制）,如果赋值为-1，则表示不限制；如果pool已经分配了maxActive个jedis实例，则此时pool的状态为exhausted(耗尽)

# max-idle: 8 # 连接池中的最大空闲连接 ，默认值也是8

# max-wait: 100 # # 等待可用连接的最大时间，单位毫秒，默认值为-1，表示永不超时。如果超过等待时间，则直接抛出JedisConnectionException

# min-idle: 2 # 连接池中的最小空闲连接 ，默认值也是0

# shutdown-timeout: 100ms

4.5、将build.xml拖拽至idea中的ant面板中，打到其中的bulid.base.mv-package-docker-docker指令，双击执行既可(如果ant有问题，在ant面板中右键属性，配置ant版本及jdk版本)

* 5、前端的docker的使用

5.1、pom.xml配置，主要是配置docker-maven-plugin(添加pom.xml主是让maven能识别这一个maven项目，好用docker-maven-plugin插件命令)

<build>

<pluginManagement>

<plugins>

<plugin>

<groupId>com.spotify</groupId>

<artifactId>docker-maven-plugin</artifactId>

<version>1.2.2</version>

<configuration>

<serverId>docker-hub</serverId> <!--mvn setting.xml配置的那个id-->

<imageName>${docker.registry.url}/${docker.image.prefix}/${project.artifactId}</imageName>

<!-- 镜像tag-->

<imageTags>

<imageTag>latest</imageTag>

</imageTags>

<!--覆盖相同标签镜像-->

<forceTags>true</forceTags>

<dockerDirectory>docker</dockerDirectory>

<!-- docker远程服务器地址 -->

<dockerHost>${docker.registry.host}</dockerHost>

<!-- 上传镜像-->

<!--<pushImage>false</pushImage>-->

<pushImageTag>true</pushImageTag>

<!-- 重试次数-->

<retryPushCount>3</retryPushCount>

<resources>

<resource>

<targetPath>/</targetPath>

<directory>${project.build.directory}</directory>

</resource>

</resources>

</configuration>

</plugin>

</plugins>

</pluginManagement>

</build>

5.2、.env.docker文件，这个文件必须存在，是下边ant打包中命令中要用到的

//生成地址

outputDir = "dist"

VUE\_APP\_MODE = 'docker'

NODE\_ENV = 'docker'

VUE\_APP\_HTTP\_BASE\_URL = 'http://192.168.1.14:9902'

VUE\_APP\_WEBSOCKET\_BASE\_WS\_URL='ws://192.168.1.14:9902/ws/'

VUE\_APP\_TEXT = 'docker环境'

5.3、package.json文件，在scripts中添加如下配置信息，是下边ant打包中命令中要用到的

"docker": "vue-cli-service serve --mode docker",

"build:docker": "vue-cli-service build --mode docker"

5.4、build-base.xml

<project name="xx-xxx" default="mvn-clean"><!--xx-xx改为项目打包的名称-->

<loadproperties srcFile="../../../build.properties" encoding="UTF-8"/><!--注意build.properties的相对路径-->

<import file="../../../build-base.xml"/><!--注意build-base.xml的相对路径-->

<target name="deploy-auth-front-docker" depends="os-mvn-init">

<delete dir="${basedir}/build"/>

<delete dir="${basedir}/docker/dist"/>

<delete dir="${basedir}/target"/>

<mkdir dir="${basedir}/target"/>

<exec executable="cmd.exe">

<arg line="/c npm run build:docker"/>

</exec>

<copy todir="${basedir}/docker">

<fileset dir="${basedir}/build"/>

</copy>

<mvn>

<arg value="-X"/>

<!--<arg value="-U" />-->

<arg value="docker:build"/>

</mvn>

</target>

</project>

5.5、Dockerfile文件的配置，位置于当前项目中的docker目录下（主要目的是把通过npm run build:docker打成的dist包copy到nginx的指定目录，然后通过nginx.conf文件对外发布出来，供外界访问）

FROM nginx:1.19.2

# 将dist文件中的内容复制到 /usr/share/nginx/html/ 这个目录下面

COPY dist/ /usr/share/nginx/html/auth\_front

COPY nginx.conf /etc/nginx/nginx.conf

RUN sh -c 'chmod -R 777 /usr/share/nginx/html/auth\_front'

5.6、nginx.conf配置,这是上面dockerfile所依赖的文件，其主要目的是对外发布前端应用

worker\_processes auto;

#error\_log logs/error.log;

#error\_log logs/error.log notice;

#error\_log logs/error.log info;

#pid logs/nginx.pid;

events {

worker\_connections 1024;

}

http {

include mime.types;

default\_type application/octet-stream;

#log\_format main '$remote\_addr - $remote\_user [$time\_local] "$request" '

# '$status $body\_bytes\_sent "$http\_referer" '

# '"$http\_user\_agent" "$http\_x\_forwarded\_for"';

#access\_log logs/access.log main;

sendfile on;

#tcp\_nopush on;

#keepalive\_timeout 0;

keepalive\_timeout 65;

#gzip on;

client\_max\_body\_size 20m;

server {

listen 8882;

server\_name 127.0.0.1;

#charset koi8-r;

#access\_log logs/host.access.log main;

location / {

root /usr/share/nginx/html/auth\_front;

index index.html index.htm;

#try\_files $uri $uri/ /index.html;

}

#error\_page 404 /404.html;

# redirect server error pages to the static page /50x.html

#

error\_page 500 502 503 504 /50x.html;

location = /50x.html {

root html;

}

}

}

5.5、将build.xml拖拽至idea中的ant面板中，打到其中的deploy-auth-front-docker指令，双击执行既可(如果ant有问题，在ant面板中右键属性，配置ant版本及jdk版本)

**git上传大文件**

* 1、安装lfs

https://git-lfs.github.com/

* 2、开启lfs功能

git lfs install

* 3、选择文件类型（）这里我选择的是rpm类型

git lfs track "jdk-8u261-linux-x64.rpm"

* 4、然后配置远程仓库

执行完上面的命令后，会生成一个.gitattributes文件，要将其上传到远程gitee仓库。这里我把.gitattributes和大文件分开上传。

git add .gitattributes

git commit -m '上传.gitattributes'

git push origin master（如果提交不了，后面可以加一个-f）

* 5、上传大文件

git add jdk-8u261-linux-x64.rpm

git commit -m "上传大文件"

git push origin master

* 6、解决报错

报错信息：

warning: Authentication error: Authentication required: LFS only supported repository in paid enterprise.

batch response: LFS only supported repository in paid enterprise.

Uploading LFS objects: 0% (0/1), 0 B | 0 B/s, done.

error: failed to push some refs to 'https://xxx.xx/xxxx/xxxx.git'

解决方案:

6.1、删除./git/hooks/pre-push文件

6.2、再次 git push origin master