1.1 文章管理模块

控制层

@RestController
  
@RequestMapping("/article")
  
public class ArticleController{
  
 @Resource
  
 private ArticleService articleService;
  
  
 @GetMapping("/articleList")
  
 public R list(Long pageNum, Long pageSize, Long categoryId){
  
 PageVo pageVo = articleService.articleList(pageNum, pageSize, categoryId);
  
 return R.okResult(pageVo);
  
 }
  
  
 @GetMapping("/hotArticleList")
  
 public R hotArticleList(){
  
 List<HotArticleVo> hotArticleVos = articleService.hotArticleList();
  
 return R.okResult(hotArticleVos);
  
 }
  
  
 @GetMapping("/{id}")
  
 public R getArticleDetail(@PathVariable("id") Long id){
  
 ArticleDetailVo articleDetail = articleService.getArticleDetail(id);
  
 return R.okResult(articleDetail);
  
 }
  
  
 @PutMapping("/updateViewCount/{id}")
  
 public R updateViewCount(@PathVariable("id") Long id){
  
 articleService.updateViewCount(id);
  
 return R.okResult();
  
 }
  
}

服务层

@Service("articleService")  
public class ArticleServiceImpl extends ServiceImpl<ArticleDao, Article> implements ArticleService {  
 @Resource  
 private CategoryService categoryService;  
 @Resource  
 private RedisTemplate<String, Map<String, Long>> redisTemplate;  
  
 /\*\*  
 \* 查询热门文章  
 \* 1. 必须是正式文章  
 \* 2. 按照浏览量进行排序  
 \* 3. 只查询前十条  
 \* @return 精简过数据后的 Vo 对象  
 \*/  
 @Override  
 public List<HotArticleVo> hotArticleList() {  
 LambdaQueryWrapper<Article> queryWrapper = new LambdaQueryWrapper<>();  
 queryWrapper.eq(Article::getStatus, 0);  
 queryWrapper.orderByDesc(Article::getViewCount);  
 Page<Article> pageInfo = new Page<>(1, 10);  
 this.page(pageInfo, queryWrapper);  
  
 List<Article> articles = pageInfo.getRecords();  
  
 return BeanCopy.copyBeanList(articles, HotArticleVo.class);  
 }  
  
 /\*\*  
 \* 分页查询文章  
 \* 1. 必须是正式发布的文章  
 \* 2. 对isTop降序排列  
 \* 3. 整合分类名称  
 \* 4. 精简查询结果  
 \*/  
 @Override  
 public PageVo articleList(Long pageNum, Long pageSize, Long categoryId) {  
 // 筛选符合条件的文章  
 LambdaQueryWrapper<Article> articleWrapper = new LambdaQueryWrapper<>();  
 articleWrapper.eq(Objects.nonNull(categoryId) && categoryId > 0,  
 Article::getCategoryId, categoryId);  
 articleWrapper.eq(Article::getStatus, 0);  
 articleWrapper.orderByDesc(Article::getIsTop);  
  
 // 查询数据库获取文章  
 Page<Article> pageInfo = new Page<>(pageNum, pageSize);  
 this.page(pageInfo, articleWrapper);  
  
 // 封装到vo对象，精简查询结果  
 List<ArticleVo> articleVos = BeanCopy.copyBeanList(pageInfo.getRecords(), ArticleVo.class);  
  
 List<ArticleVo> articleVoList = articleVos.stream().peek(articleVo -> {  
 Category category = categoryService.getById(articleVo.getCategoryId());  
 String categoryName = category.getName();  
 articleVo.setCategoryName(categoryName);  
 }).collect(Collectors.toList());  
  
 // 查询分类名称，植入vo对象中  
 return new PageVo(articleVoList, pageInfo.getTotal());  
 }  
  
 @Override  
 public ArticleDetailVo getArticleDetail(Long id) {  
 Article article = getById(id);  
 ArticleDetailVo articleDetailVo = BeanCopy.copyBean(article, ArticleDetailVo.class);  
  
 // 数据库中目前存放的文章浏览量不够准确，需要从redis中获取  
 /\* 为什么返回的不是Long类型？  
 \* 注意踩坑，我们往redis中存放的访问量这个变量的数据类型为Long  
 \* 但是redis在返回数据却是Integer类型  
 \* 实际在redis存的是具体的数字，那么当读取的时候，代码并不知道缓存中的数字到底是Integer类型还是Long类型  
 \* 于是乎会使用实际数字的值进行自动转换——也就是说返回Integer类型，而强转会报异常因此我们先用Number来承接返回值  
 \* 随后使用.longValue()方法转换成Long类型。  
 \*/  
 Number viewCount = (Number) redisTemplate.opsForHash().get("article:viewCount", id.toString());  
 assert viewCount != null;  
 articleDetailVo.setViewCount(viewCount.longValue());  
  
 // 从数据库获取文章分类名称  
 Long categoryId = articleDetailVo.getCategoryId();  
 Category category = categoryService.getById(categoryId);  
 if(category != null){  
 articleDetailVo.setCategoryName(category.getName());  
 }  
 return articleDetailVo;  
 }  
  
 @Override  
 public void updateViewCount(Long id) {  
 redisTemplate.boundHashOps("article:viewCount").increment(id.toString(), 1);  
 }  
}

1.2 评论管理模块

控制层

@RestController  
@RequestMapping("/comment")  
public class CommentController {  
 @Resource  
 private CommentService commentService;  
  
 @GetMapping("/commentList")  
 public R commentList(Long articleId, Long pageNum, Long pageSize){  
 PageVo pageVo = commentService.listWithTree(articleId, pageNum, pageSize);  
 return R.okResult(pageVo);  
 }  
  
 @PostMapping  
 public R addComment(@RequestBody Comment comment){  
 comment.setCreateBy(SecurityUtils.getUserId());  
 commentService.save(comment);  
 return R.okResult();  
 }  
}

服务层

@Service("commentService")
  
public class CommentServiceImpl extends ServiceImpl<CommentDao, Comment> implements CommentService {
  
 @Resource
  
 private UserService userService;
  
  
 /\*\*
  
 \* 分页查询评论以及所有评论下的评论
  
 \* 1. 查询根评论，按时间排序
  
 \* 2. 查询每一个评论用户的用户名
  
 \* 3. 整合成列表返回
  
 \* @param articleId 文章id
  
 \* @return 返回查询好的评论分页对象
  
 \*/
  
 @Override
  
 public PageVo listWithTree(Long articleId, Long pageNum, Long pageSize) {
  
 LambdaQueryWrapper<Comment> queryWrapper = new LambdaQueryWrapper<>();
  
 queryWrapper.eq(Comment::getArticleId, articleId);
  
 queryWrapper.eq(Comment::getRootId, -1);
  
 queryWrapper.orderByDesc(Comment::getCreateTime);
  
  
 Page<Comment> pageInfo = new Page<>(pageNum, pageSize);
  
 page(pageInfo, queryWrapper);
  
  
 List<CommentVo> commentVoList = toCommentVoList(pageInfo.getRecords());
  
 List<CommentVo> commentVos = commentVoList.stream().peek(commentVo -> {
  
 List<CommentVo> children = getChildren(commentVo.getId());
  
 commentVo.setChildren(children);
  
 }).collect(Collectors.toList());
  
  
 return new PageVo(commentVos, pageInfo.getTotal());
  
 }
  
  
 /\*\*
  
 \* 根据根评论的id查询所对应的子评论的集合
  
 \* @param id 根评论的id
  
 \*/
  
 private List<CommentVo> getChildren(Long id){
  
 LambdaQueryWrapper<Comment> queryWrapper = new LambdaQueryWrapper<>();
  
 queryWrapper.eq(Comment::getRootId, id);
  
 queryWrapper.orderByAsc(Comment::getCreateTime);
  
  
 List<Comment> commentList = list(queryWrapper);
  
 return toCommentVoList(commentList);
  
 }
  
  
 private List<CommentVo> toCommentVoList(List<Comment> list){
  
 List<CommentVo> commentVos = BeanCopy.copyBeanList(list, CommentVo.class);
  
 return commentVos.stream().peek(commentVo -> {
  
 // 通过用户id获取用户名
  
 Long userId = commentVo.getCreateBy();
  
 User user = userService.getById(userId);
  
 if(Objects.nonNull(user.getUserName()))
  
 commentVo.setUsername(user.getNickName());
  
 if(Objects.nonNull(user.getAvatar()))
  
 commentVo.setAvatar(user.getAvatar());
  
  
 // 如果有该评论是回复 别人的评论，那么通过自身携带的别人id查询别人的用户名
  
 if (commentVo.getToCommentId() != -1) {
  
 User toUser = userService.getById(commentVo.getToCommentUserId());
  
 if(Objects.nonNull(toUser.getNickName()))
  
 commentVo.setToCommentUserName(toUser.getNickName());
  
 if(Objects.nonNull(user.getAvatar()))
  
 commentVo.setAvatar(user.getAvatar());
  
 }
  
 }).collect(Collectors.toList());
  
 }
  
}

1.3 文件管理模块

控制层

@RestController  
@Slf4j  
public class CommonController {  
 @Value("${pei-blog.filePath}")  
 private String filePath;  
  
 @PostMapping("/upload")  
 public R upload(MultipartFile file) throws IOException{  
 // 生成新的文件名，获取uuid，获取旧文件名后缀。  
 String uuid = UUID.randomUUID().toString();  
 String originalFilename = file.getOriginalFilename();  
 assert originalFilename != null;  
 String suffix = originalFilename.substring(originalFilename.lastIndexOf("."));  
 String newFileName = uuid + suffix;  
  
 // 判断是否存在当前存放文件的目录，如果不存在则创建一个  
 File dir = new File(filePath);  
 if(!dir.exists()){  
 boolean isMk = dir.mkdirs();  
 log.info("{}创建目录{}", isMk ? "成功" : "失败" , filePath);  
 }  
  
 // 存储文件到指定目录  
 file.transferTo(new File(filePath+newFileName));  
  
 return R.okResult(newFileName);  
 }  
  
 @RequestMapping("/download")  
 public void download(String name, HttpServletResponse response) throws IOException{  
 FileInputStream inputStream = new FileInputStream(filePath+name);  
  
 ServletOutputStream outputStream = response.getOutputStream();  
 response.setContentType("image/jpeg");  
  
 int len = 0;  
 byte[] bytes = new byte[1024];  
 while ((len = inputStream.read(bytes)) != -1){  
 outputStream.write(bytes, 0, len);  
 outputStream.flush();  
 }  
  
 outputStream.close();  
 inputStream.close();  
 }  
}

1.4 用户管理模块

@RestController
  
@RequestMapping("/user")
  
public class UserController {
  
 @Resource
  
 private UserService userService;
  
  
 @GetMapping("/userInfo")
  
 public R getUserInfo(){
  
 UserInfoVo userInfoVo = userService.userInfo();
  
 return R.okResult(userInfoVo);
  
 }
  
  
 @PutMapping("/userInfo")
  
 public R updateUserInfo(@RequestBody User user){
  
 Long userId = SecurityUtils.getUserId();
  
 user.setId(userId);
  
  
 userService.updateById(user);
  
 return R.okResult(HttpCodeEnum.SUCCESS);
  
 }
  
  
 @PostMapping("/register")
  
 public R register(@RequestBody User user){
  
 userService.register(user);
  
 return R.okResult(HttpCodeEnum.SUCCESS);
  
 }
  
}
  
  
@RestController
  
public class BlogLoginController {
  
 @Resource
  
 private BlogLoginService blogLoginService;
  
  
 @PostMapping("/login")
  
 public R login(@RequestBody User user){
  
 return R.okResult(blogLoginService.login(user));
  
 }
  
  
 @PostMapping("/logout")
  
 public R logout(){
  
 blogLoginService.logout();
  
 return R.okResult(HttpCodeEnum.SUCCESS);
  
 }
  
}

服务层

@Service("userService")
  
public class UserServiceImpl extends ServiceImpl<UserDao, User> implements UserService {
  
 @Resource
  
 private PasswordEncoder passwordEncoder;
  
  
 @Override
  
 public UserInfoVo userInfo() {
  
 User user = getById(SecurityUtils.getUserId());
  
 return BeanCopy.copyBean(user, UserInfoVo.class);
  
 }
  
  
 @Override
  
 public User register(User user) {
  
 // TODO 对数据进行校验
  
 String encodePassword = passwordEncoder.encode(user.getPassword());
  
 user.setPassword(encodePassword);
  
 save(user);
  
 return user;
  
 }
  
}
  
  
@Service
  
public class BlogLoginServiceImpl implements BlogLoginService {
  
 @Resource
  
 private AuthenticationManager authenticationManager;
  
 @Resource
  
 private RedisTemplate<String, String> redisTemplate;
  
  
 /\*\*
  
 \* 博客系统认证登录的逻辑实现
  
 \* 1. 创建一个用于spring security认证的token传入用户名密码
  
 \* 2. 调用authenticationManager的认证方法进行认证
  
 \* 3. 如果没有认证信息，说明不是本系统的用户，返回用户名密码错误
  
 \* 4. 如果是本系统成员，我们可以从认证信息中获取用户信息
  
 \* 这里来的用户信息类型是spring security提供的UserDetails接口，我们在项目中需要实现它。
  
 \* 5. 从UserDetails中获取我们自己的数据库实体对象，获取用户id生成jwt token
  
 \* jwt token将连带着部分用户信息返回给前端
  
 \* 6. 把用户信息存放到redis中，下次认证授权会先根据前端传来的jwt token查询redis中是否有信息
  
 \* 如果没有说明用户要么没登陆，要么不是本系统用户
  
 \*
  
 \* @param user 前端传来的用户名和密码，封装为数据库的user对象
  
 \* @return 返回给前端的用户信息和token
  
 \*/
  
 @Override
  
 public BlogUserLoginVo login(User user) {
  
 UsernamePasswordAuthenticationToken token = new UsernamePasswordAuthenticationToken(
  
 user.getUserName(), user.getPassword());
  
 Authentication authenticate = authenticationManager.authenticate(token);
  
  
 if(Objects.isNull(authenticate)){
  
 throw new CustomException("用户名密码错误");
  
 }
  
 LoginUser loginUser = (LoginUser) authenticate.getPrincipal();
  
 String userId = loginUser.getUser().getId().toString();
  
 String jwt = JwtUtil.createJWT(userId);
  
  
 redisTemplate.opsForValue().set("blogLogin:"+userId, JSONObject.toJSONString(loginUser));
  
 UserInfoVo userInfoVo = BeanCopy.copyBean(loginUser.getUser(), UserInfoVo.class);
  
 return new BlogUserLoginVo(jwt, userInfoVo);
  
 }
  
  
 /\*\*
  
 \* 用户下线功能，首先从SecurityContextHolder获取对应用户的认证授权信息
  
 \* 然后获取到用户信息——用户id，根据用户id找到redis中对应值删除
  
 \*/
  
 @Override
  
 public void logout() {
  
 UsernamePasswordAuthenticationToken authenticationToken = (UsernamePasswordAuthenticationToken)
  
 SecurityContextHolder.getContext().getAuthentication();
  
 LoginUser loginUser = (LoginUser) authenticationToken.getPrincipal();
  
 redisTemplate.delete("blogLogin:"+loginUser.getUser().getId());
  
 }
  
}