

v1.4.17 ▾

...

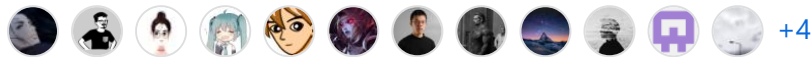
halo / src / main / java / run / halo / app / service / impl / PostServiceImpl.java / <> Jump to ▾



fuzui Fix number of comment in detail of post and sheet and add it into det... ... ✓

History

16 contributors



1055 lines (859 sloc) | 39.3 KB

...

```

1 package run.halo.app.service.impl;
2
3 import static org.springframework.data.domain.Sort.Direction.DESC;
4 import static run.halo.app.model.support.HaloConst.URL_SEPARATOR;
5
6 import java.util.ArrayList;
7 import java.util.Calendar;
8 import java.util.Collection;
9 import java.util.Collections;
10 import java.util.HashMap;
11 import java.util.HashSet;
12 import java.util.LinkedList;
13 import java.util.List;
14 import java.util.Map;
15 import java.util.Objects;
16 import java.util.Optional;
17 import java.util.Set;
18 import java.util.stream.Collectors;
19 import javax.persistence.criteria.Predicate;
20 import javax.persistence.criteria.Root;
21 import javax.persistence.criteria.Subquery;
22 import javax.validation.constraints.NotNull;
23 import lombok.extern.slf4j.Slf4j;
24 import org.apache.commons.lang3.StringUtils;
25 import org.springframework.context.ApplicationEventPublisher;
26 import org.springframework.data.domain.Page;
27
28 import org.springframework.data.domain.Pageable;
29 import org.springframework.data.domain.Sort;

```

```
29 import org.springframework.data.jpa.domain.Specification;
30 import org.springframework.lang.NonNull;
31 import org.springframework.lang.Nullable;
32 import org.springframework.stereotype.Service;
33 import org.springframework.transaction.annotation.Transactional;
34 import org.springframework.util.Assert;
35 import org.springframework.util.CollectionUtils;
36 import run.halo.app.event.logger.LogEvent;
37 import run.halo.app.event.post.PostVisitEvent;
38 import run.halo.app.exception.NotFoundException;
39 import run.halo.app.model.dto.post.BasePostMinimalDTO;
40 import run.halo.app.model.dto.post.BasePostSimpleDTO;
41 import run.halo.app.model.entity.Category;
42 import run.halo.app.model.entity.Post;
43 import run.halo.app.model.entity.PostCategory;
44 import run.halo.app.model.entity.PostComment;
45 import run.halo.app.model.entity.PostMeta;
46 import run.halo.app.model.entity.PostTag;
47 import run.halo.app.model.entity.Tag;
48 import run.halo.app.model.enums.CommentStatus;
49 import run.halo.app.model.enums.LogType;
50 import run.halo.app.model.enums.PostPermalinkType;
51 import run.halo.app.model.enums.PostStatus;
52 import run.halo.app.model.params.PostParam;
53 import run.halo.app.model.params.PostQuery;
54 import run.halo.app.model.properties.PostProperties;
55 import run.halo.app.model.vo.ArchiveMonthVO;
56 import run.halo.app.model.vo.ArchiveYearVO;
57 import run.halo.app.model.vo.PostDetailVO;
58 import run.halo.app.model.vo.PostListVO;
59 import run.halo.app.model.vo.PostMarkdownVO;
60 import run.halo.app.repository.PostRepository;
61 import run.halo.app.repository.base.BasePostRepository;
62 import run.halo.app.service.AuthorizationService;
63 import run.halo.app.service.CategoryService;
64 import run.halo.app.service.OptionService;
65 import run.halo.app.service.PostCategoryService;
66 import run.halo.app.service.PostCommentService;
67 import run.halo.app.service.PostMetaService;
68 import run.halo.app.service.PostService;
69 import run.halo.app.service.PostTagService;
70 import run.halo.app.service.TagService;
71 import run.halo.app.utils.DateUtils;
72 import run.halo.app.utils.HaloUtils;
73 import run.halo.app.utils.MarkdownUtils;
74 import run.halo.app.utils.ServiceUtils;
75 import run.halo.app.utils.SlugUtils;
76
77 /**
```

```
78  * Post service implementation.
79  *
80  * @author johnniang
81  * @author ryanwang
82  * @author guqing
83  * @author evanwang
84  * @author coor.top
85  * @author Raremaa
86  * @date 2019-03-14
87  */
88  @Slf4j
89  @Service
90  public class PostServiceImpl extends BasePostServiceImpl<Post> implements PostService {
91
92      private final PostRepository postRepository;
93
94      private final TagService tagService;
95
96      private final CategoryService categoryService;
97
98      private final PostTagService postTagService;
99
100     private final PostCategoryService postCategoryService;
101
102     private final PostCommentService postCommentService;
103
104     private final ApplicationEventPublisher eventPublisher;
105
106     private final PostMetaService postMetaService;
107
108     private final OptionService optionService;
109
110     private final AuthorizationService authorizationService;
111
112     public PostServiceImpl(BasePostRepository<Post> basePostRepository,
113         OptionService optionService,
114         PostRepository postRepository,
115         TagService tagService,
116         CategoryService categoryService,
117         PostTagService postTagService,
118         PostCategoryService postCategoryService,
119         PostCommentService postCommentService,
120         ApplicationEventPublisher eventPublisher,
121         PostMetaService postMetaService,
122         AuthorizationService authorizationService) {
123         super(basePostRepository, optionService);
124         this.postRepository = postRepository;
125         this.tagService = tagService;
126         this.categoryService = categoryService;
```

```

127         this.postTagService = postTagService;
128         this.postCategoryService = postCategoryService;
129         this.postCommentService = postCommentService;
130         this.eventPublisher = eventPublisher;
131         this.postMetaService = postMetaService;
132         this.optionService = optionService;
133         this.authorizationService = authorizationService;
134     }
135
136     @Override
137     public Page<Post> pageBy(PostQuery postQuery, Pageable pageable) {
138         Assert.notNull(postQuery, "Post query must not be null");
139         Assert.notNull(pageable, "Page info must not be null");
140
141         // Build specification and find all
142         return postRepository.findAll(buildSpecByQuery(postQuery), pageable);
143     }
144
145     @Override
146     public Page<Post> pageBy(String keyword, Pageable pageable) {
147         Assert.notNull(keyword, "keyword must not be null");
148         Assert.notNull(pageable, "Page info must not be null");
149
150         PostQuery postQuery = new PostQuery();
151         postQuery.setKeyword(keyword);
152         postQuery.setStatus(PostStatus.PUBLISHED);
153
154         // Build specification and find all
155         return postRepository.findAll(buildSpecByQuery(postQuery), pageable);
156     }
157
158     @Override
159     @Transactional
160     public PostDetailVO createBy(Post postToCreate, Set<Integer> tagIds, Set<Integer> categoryIds,
161         Set<PostMeta> metas, boolean autoSave) {
162         PostDetailVO createdPost = createOrUpdate(postToCreate, tagIds, categoryIds, metas);
163         if (!autoSave) {
164             // Log the creation
165             LogEvent logEvent = new LogEvent(this, createdPost.getId().toString(),
166                 LogType.POST_PUBLISHED, createdPost.getTitle());
167             eventPublisher.publishEvent(logEvent);
168         }
169         return createdPost;
170     }
171
172     @Override
173     public PostDetailVO createBy(Post postToCreate, Set<Integer> tagIds, Set<Integer> categoryIds,
174         boolean autoSave) {
175         PostDetailVO createdPost = createOrUpdate(postToCreate, tagIds, categoryIds, null);

```

```

176         if (!autoSave) {
177             // Log the creation
178             LogEvent logEvent = new LogEvent(this, createdPost.getId().toString(),
179                 LogType.POST_PUBLISHED, createdPost.getTitle());
180             eventPublisher.publishEvent(logEvent);
181         }
182         return createdPost;
183     }
184
185     @Override
186     @Transactional
187     public PostDetailVO updateBy(Post postToUpdate, Set<Integer> tagIds, Set<Integer> categoryIds,
188         Set<PostMeta> metas, boolean autoSave) {
189         // Set edit time
190         postToUpdate.setEditTime(DateUtils.now());
191         PostDetailVO updatedPost = createOrUpdate(postToUpdate, tagIds, categoryIds, metas);
192         if (!autoSave) {
193             // Log the creation
194             LogEvent logEvent = new LogEvent(this, updatedPost.getId().toString(),
195                 LogType.POST_EDITED, updatedPost.getTitle());
196             eventPublisher.publishEvent(logEvent);
197         }
198         return updatedPost;
199     }
200
201     @Override
202     public Post getBy(PostStatus status, String slug) {
203         return super.getBy(status, slug);
204     }
205
206     @Override
207     public Post getBy(Integer year, Integer month, String slug) {
208         Assert.notNull(year, "Post create year must not be null");
209         Assert.notNull(month, "Post create month must not be null");
210         Assert.notNull(slug, "Post slug must not be null");
211
212         Optional<Post> postOptional = postRepository.findBy(year, month, slug);
213
214         return postOptional
215             .orElseThrow(() -> new NotFoundException("查询不到该文章的信息").setErrorData(slug));
216     }
217
218     @NonNull
219     @Override
220     public Post getBy(@NonNull Integer year, @NonNull String slug) {
221         Assert.notNull(year, "Post create year must not be null");
222         Assert.notNull(slug, "Post slug must not be null");
223
224         Optional<Post> postOptional = postRepository.findBy(year, slug);

```

```
225
226     return postOptional
227         .orElseThrow(() -> new NotFoundException("查询不到该文章的信息").setErrorData(slug));
228 }
229
230 @Override
231 public Post getBy(Integer year, Integer month, String slug, PostStatus status) {
232     Assert.notNull(year, "Post create year must not be null");
233     Assert.notNull(month, "Post create month must not be null");
234     Assert.notNull(slug, "Post slug must not be null");
235     Assert.notNull(status, "Post status must not be null");
236
237     Optional<Post> postOptional = postRepository.findBy(year, month, slug, status);
238
239     return postOptional
240         .orElseThrow(() -> new NotFoundException("查询不到该文章的信息").setErrorData(slug));
241 }
242
243 @Override
244 public Post getBy(Integer year, Integer month, Integer day, String slug) {
245     Assert.notNull(year, "Post create year must not be null");
246     Assert.notNull(month, "Post create month must not be null");
247     Assert.notNull(day, "Post create day must not be null");
248     Assert.notNull(slug, "Post slug must not be null");
249
250     Optional<Post> postOptional = postRepository.findBy(year, month, day, slug);
251
252     return postOptional
253         .orElseThrow(() -> new NotFoundException("查询不到该文章的信息").setErrorData(slug));
254 }
255
256 @Override
257 public Post getBy(Integer year, Integer month, Integer day, String slug, PostStatus status) {
258     Assert.notNull(year, "Post create year must not be null");
259     Assert.notNull(month, "Post create month must not be null");
260     Assert.notNull(day, "Post create day must not be null");
261     Assert.notNull(slug, "Post slug must not be null");
262     Assert.notNull(status, "Post status must not be null");
263
264     Optional<Post> postOptional = postRepository.findBy(year, month, day, slug, status);
265
266     return postOptional
267         .orElseThrow(() -> new NotFoundException("查询不到该文章的信息").setErrorData(slug));
268 }
269
270 @Override
271 public List<Post> removeByIds(Collection<Integer> ids) {
272     if (CollectionUtils.isEmpty(ids)) {
273         return Collections.emptyList();
274     }
275 }
```

```

274     }
275     return ids.stream().map(this::removeById).collect(Collectors.toList());
276 }
277
278 @Override
279 public Post getBySlug(String slug) {
280     return super.getBySlug(slug);
281 }
282
283 @Override
284 public List<ArchiveYearVO> listYearArchives() {
285     // Get all posts
286     List<Post> posts = postRepository
287         .findAllByStatus(PostStatus.PUBLISHED, Sort.by(DESC, "createTime"));
288
289     return convertToYearArchives(posts);
290 }
291
292 @Override
293 public List<ArchiveMonthVO> listMonthArchives() {
294     // Get all posts
295     List<Post> posts = postRepository
296         .findAllByStatus(PostStatus.PUBLISHED, Sort.by(DESC, "createTime"));
297
298     return convertToMonthArchives(posts);
299 }
300
301 @Override
302 public List<ArchiveYearVO> convertToYearArchives(List<Post> posts) {
303     Map<Integer, List<Post>> yearPostMap = new HashMap<>(8);
304
305     posts.forEach(post -> {
306         Calendar calendar = DateUtils.convertTo(post.getCreateTime());
307         yearPostMap.computeIfAbsent(calendar.get(Calendar.YEAR), year -> new LinkedList<>())
308             .add(post);
309     });
310
311     List<ArchiveYearVO> archives = new LinkedList<>();
312
313     yearPostMap.forEach((year, postList) -> {
314         // Build archive
315         ArchiveYearVO archive = new ArchiveYearVO();
316         archive.setYear(year);
317         archive.setPosts(convertToListVo(postList));
318
319         // Add archive
320         archives.add(archive);
321     });
322

```

```

323         // Sort this list
324         archives.sort(new ArchiveYearVO.ArchiveComparator());
325
326         return archives;
327     }
328
329     @Override
330     public List<ArchiveMonthVO> convertToMonthArchives(List<Post> posts) {
331
332         Map<Integer, Map<Integer, List<Post>>> yearMonthPostMap = new HashMap<>(8);
333
334         posts.forEach(post -> {
335             Calendar calendar = DateUtils.convertTo(post.getCreateTime());
336
337             yearMonthPostMap.computeIfAbsent(calendar.get(Calendar.YEAR), year -> new HashMap<>())
338                 .computeIfAbsent(calendar.get(Calendar.MONTH) + 1,
339                     month -> new LinkedList<>())
340                 .add(post);
341         });
342
343         List<ArchiveMonthVO> archives = new LinkedList<>();
344
345         yearMonthPostMap.forEach((year, monthPostMap) ->
346             monthPostMap.forEach((month, postList) -> {
347                 ArchiveMonthVO archive = new ArchiveMonthVO();
348                 archive.setYear(year);
349                 archive.setMonth(month);
350                 archive.setPosts(convertToListVo(postList));
351
352                 archives.add(archive);
353             }));
354
355         // Sort this list
356         archives.sort(new ArchiveMonthVO.ArchiveComparator());
357
358         return archives;
359     }
360
361     @Override
362     public PostDetailVO importMarkdown(String markdown, String filename) {
363         Assert.notNull(markdown, "Markdown document must not be null");
364
365         // Gets frontMatter
366         Map<String, List<String>> frontMatter = MarkdownUtils.getFrontMatter(markdown);
367         // remove frontMatter
368         markdown = MarkdownUtils.removeFrontMatter(markdown);
369
370         PostParam post = new PostParam();
371         post.setStatus(null);

```



```
372
373     List<String> elementValue;
374
375     Set<Integer> tagIds = new HashSet<>();
376
377     Set<Integer> categoryIds = new HashSet<>();
378
379     if (frontMatter.size() > 0) {
380         for (String key : frontMatter.keySet()) {
381             elementValue = frontMatter.get(key);
382             for (String ele : elementValue) {
383                 ele = HaloUtils.strip(ele, "[", "]");
384                 ele = StringUtils.strip(ele, "\\");
385                 ele = StringUtils.strip(ele, "\\");
386                 if ("".equals(ele)) {
387                     continue;
388                 }
389                 switch (key) {
390                     case "title":
391                         post.setTitle(ele);
392                         break;
393                     case "date":
394                         post.setCreateTime(DateUtils.parseDate(ele));
395                         break;
396                     case "permalink":
397                         post.setSlug(ele);
398                         break;
399                     case "thumbnail":
400                         post.setThumbnail(ele);
401                         break;
402                     case "status":
403                         post.setStatus(PostStatus.valueOf(ele));
404                         break;
405                     case "comments":
406                         post.setDisallowComment(Boolean.parseBoolean(ele));
407                         break;
408                     case "tags":
409                         Tag tag;
410                         for (String tagName : ele.split(",")) {
411                             tagName = tagName.trim();
412                             tagName = StringUtils.strip(tagName, "\\");
413                             tagName = StringUtils.strip(tagName, "\\");
414                             tag = tagService.getByName(tagName);
415                             String slug = SlugUtils.slug(tagName);
416                             if (null == tag) {
417                                 tag = tagService.getBySlug(slug);
418                             }
419                             if (null == tag) {
420                                 tag = new Tag();
```

```

421         tag.setName(tagName);
422         tag.setSlug(slug);
423         tag = tagService.create(tag);
424     }
425     tagIds.add(tag.getId());
426 }
427 break;
428 case "categories":
429     Integer lastCategoryId = null;
430     for (String categoryName : ele.split(",")) {
431         categoryName = categoryName.trim();
432         categoryName = StringUtils.strip(categoryName, "\"");
433         categoryName = StringUtils.strip(categoryName, "'");
434         Category category = categoryService.getBy_name(categoryName);
435         if (null == category) {
436             category = new Category();
437             category.setName(categoryName);
438             category.setSlug(StringUtils.slug(categoryName));
439             category.setDescription(categoryName);
440             if (lastCategoryId != null) {
441                 category.setParentId(lastCategoryId);
442             }
443             category = categoryService.create(category);
444         }
445         lastCategoryId = category.getId();
446         categoryIds.add(lastCategoryId);
447     }
448     break;
449 default:
450     break;
451 }
452 }
453 }
454 }
455
456 if (null == post.getStatus()) {
457     post.setStatus(PostStatus.PUBLISHED);
458 }
459
460 if (StringUtils.isEmpty(post.getTitle())) {
461     post.setTitle(filename);
462 }
463
464 if (StringUtils.isEmpty(post.getSlug())) {
465     post.setSlug(StringUtils.slug(post.getTitle()));
466 }
467
468 post.setOriginalContent(markdown);
469

```

```

470         return createBy(post.convertTo(), tagIds, categoryIds, false);
471     }
472
473     @Override
474     public String exportMarkdown(Integer id) {
475         Assert.notNull(id, "Post id must not be null");
476         Post post = getById(id);
477         return exportMarkdown(post);
478     }
479
480     @Override
481     public String exportMarkdown(Post post) {
482         Assert.notNull(post, "Post must not be null");
483
484         StringBuilder content = new StringBuilder("---\n");
485
486         content.append("type: ").append("post").append("\n");
487         content.append("title: ").append(post.getTitle()).append("\n");
488         content.append("permalink: ").append(post.getSlug()).append("\n");
489         content.append("thumbnail: ").append(post.getThumbnail()).append("\n");
490         content.append("status: ").append(post.getStatus()).append("\n");
491         content.append("date: ").append(post.getCreateTime()).append("\n");
492         content.append("updated: ").append(post.getEditTime()).append("\n");
493         content.append("comments: ").append(!post.getDisallowComment()).append("\n");
494
495         List<Tag> tags = postTagService.listTagsBy(post.getId());
496
497         if (tags.size() > 0) {
498             content.append("tags:").append("\n");
499             for (Tag tag : tags) {
500                 content.append(" - ").append(tag.getName()).append("\n");
501             }
502         }
503
504         List<Category> categories = postCategoryService.listCategoriesBy(post.getId());
505
506         if (categories.size() > 0) {
507             content.append("categories:").append("\n");
508             for (Category category : categories) {
509                 content.append(" - ").append(category.getName()).append("\n");
510             }
511         }
512
513         List<PostMeta> metas = postMetaService.listBy(post.getId());
514
515         if (metas.size() > 0) {
516             content.append("metas:").append("\n");
517             for (PostMeta postMeta : metas) {
518                 content.append(" - ").append(postMeta.getKey()).append(" : ")

```

```

519         .append(postMeta.getValue()).append("\n");
520     }
521 }
522
523     content.append("---\n\n");
524     content.append(post.getOriginalContent());
525     return content.toString();
526 }
527
528 @Override
529 public PostDetailVo convertToDetailVo(Post post) {
530     return convertToDetailVo(post, false);
531 }
532
533 @Override
534 public PostDetailVo convertToDetailVo(Post post, boolean queryEncryptCategory) {
535     // List tags
536     List<Tag> tags = postTagService.listTagsBy(post.getId());
537     // List categories
538     List<Category> categories = postCategoryService
539         .listCategoriesBy(post.getId(), queryEncryptCategory);
540     // List metas
541     List<PostMeta> metas = postMetaService.listBy(post.getId());
542     // Convert to detail vo
543     return convertTo(post, tags, categories, metas);
544 }
545
546 @Override
547 public Page<PostDetailVo> convertToDetailVo(Page<Post> postPage) {
548     Assert.notNull(postPage, "Post page must not be null");
549     return postPage.map(this::convertToDetailVo);
550 }
551
552 @Override
553 public Post removeById(Integer postId) {
554     Assert.notNull(postId, "Post id must not be null");
555
556     log.debug("Removing post: [{}]", postId);
557
558     // Remove post tags
559     List<PostTag> postTags = postTagService.removeByPostId(postId);
560
561     log.debug("Removed post tags: [{}]", postTags);
562
563     // Remove post categories
564     List<PostCategory> postCategories = postCategoryService.removeByPostId(postId);
565
566     log.debug("Removed post categories: [{}]", postCategories);
567

```

```

568 // Remove metas
569 List<PostMeta> metas = postMetaService.removeByPostId(postId);
570 log.debug("Removed post metas: [{}]", metas);
571
572 // Remove post comments
573 List<PostComment> postComments = postCommentService.removeByPostId(postId);
574 log.debug("Removed post comments: [{}]", postComments);
575
576 Post deletedPost = super.removeById(postId);
577
578 // Log it
579 eventPublisher.publishEvent(new LogEvent(this, postId.toString(), LogType.POST_DELETED,
580     deletedPost.getTitle()));
581
582 return deletedPost;
583 }
584
585 @Override
586 public Page<PostListVO> convertToListVo(Page<Post> postPage) {
587     return convertToListVo(postPage, false);
588 }
589
590 @Override
591 public Page<PostListVO> convertToListVo(Page<Post> postPage, boolean queryEncryptCategory) {
592     Assert.notNull(postPage, "Post page must not be null");
593
594     List<Post> posts = postPage.getContent();
595
596     Set<Integer> postIds = ServiceUtils.fetchProperty(posts, Post::getId);
597
598     // Get tag list map
599     Map<Integer, List<Tag>> tagListMap = postTagService.listTagListMapBy(postIds);
600
601     // Get category list map
602     Map<Integer, List<Category>> categoryListMap = postCategoryService
603         .listCategoryListMap(postIds, queryEncryptCategory);
604
605     // Get comment count
606     Map<Integer, Long> commentCountMap = postCommentService.countByStatusAndPostIds(
607         CommentStatus.PUBLISHED, postIds);
608
609     // Get post meta list map
610     Map<Integer, List<PostMeta>> postMetaListMap = postMetaService.listPostMetaAsMap(postIds);
611
612     return postPage.map(post -> {
613         PostListVO postListVO = new PostListVO().convertFrom(post);
614
615         if (StringUtils.isBlank(postListVO.getSummary())) {
616             postListVO.setSummary(generateSummary(post.getFormatContent()));

```

```

617     }
618
619     Optional.ofNullable(tagListMap.get(post.getId())).orElseGet(LinkedList::new);
620
621     // Set tags
622     postListVO.setTags(Optional.ofNullable(tagListMap.get(post.getId()))
623         .orElseGet(LinkedList::new)
624         .stream()
625         .filter(Objects::nonNull)
626         .map(tagService::convertTo)
627         .collect(Collectors.toList()));
628
629     // Set categories
630     postListVO.setCategories(Optional.ofNullable(categoryListMap.get(post.getId()))
631         .orElseGet(LinkedList::new)
632         .stream()
633         .filter(Objects::nonNull)
634         .map(categoryService::convertTo)
635         .collect(Collectors.toList()));
636
637     // Set post metas
638     List<PostMeta> metas = Optional.ofNullable(postMetaListMap.get(post.getId()))
639         .orElseGet(LinkedList::new);
640     postListVO.setMetas(postMetaService.convertToMap(metas));
641
642     // Set comment count
643     postListVO.setCommentCount(commentCountMap.getOrDefault(post.getId(), 0L));
644
645     postListVO.setFullPath(buildFullPath(post));
646
647     return postListVO;
648 });
649 }
650
651 @Override
652 public List<PostListVO> convertToListVo(List<Post> posts) {
653     return convertToListVo(posts, false);
654 }
655
656 @Override
657 public List<PostListVO> convertToListVo(List<Post> posts, boolean queryEncryptCategory) {
658     Assert.notNull(posts, "Post page must not be null");
659
660     Set<Integer> postIds = ServiceUtils.fetchProperty(posts, Post::getId);
661
662     // Get tag list map
663     Map<Integer, List<Tag>> tagListMap = postTagService.listTagListMapBy(postIds);
664
665     // Get category list map

```

```

666     Map<Integer, List<Category>> categoryListMap = postCategoryService
667         .listCategoryListMap(postIds, queryEncryptCategory);
668
669     // Get comment count
670     Map<Integer, Long> commentCountMap =
671         postCommentService.countByStatusAndPostIds(CommentStatus.PUBLISHED, postIds);
672
673     // Get post meta list map
674     Map<Integer, List<PostMeta>> postMetaListMap = postMetaService.listPostMetaAsMap(postIds);
675
676     return posts.stream().map(post -> {
677         PostListVO postListVO = new PostListVO().convertFrom(post);
678
679         if (StringUtils.isBlank(postListVO.getSummary())) {
680             postListVO.setSummary(generateSummary(post.getFormatContent()));
681         }
682
683         Optional.ofNullable(tagListMap.get(post.getId())).orElseGet(LinkedList::new);
684
685         // Set tags
686         postListVO.setTags(Optional.ofNullable(tagListMap.get(post.getId()))
687             .orElseGet(LinkedList::new)
688             .stream()
689             .filter(Objects::nonNull)
690             .map(tagService::convertTo)
691             .collect(Collectors.toList()));
692
693         // Set categories
694         postListVO.setCategories(Optional.ofNullable(categoryListMap.get(post.getId()))
695             .orElseGet(LinkedList::new)
696             .stream()
697             .filter(Objects::nonNull)
698             .map(categoryService::convertTo)
699             .collect(Collectors.toList()));
700
701         // Set post metas
702         List<PostMeta> metas = Optional.ofNullable(postMetaListMap.get(post.getId()))
703             .orElseGet(LinkedList::new);
704         postListVO.setMetas(postMetaService.convertToMap(metas));
705
706         // Set comment count
707         postListVO.setCommentCount(commentCountMap.getOrDefault(post.getId(), 0L));
708
709         postListVO.setFullPath(buildFullPath(post));
710
711         return postListVO;
712     }).collect(Collectors.toList());
713 }
714

```

```

715     @Override
716     public BasePostMinimalDTO convertToMinimal(Post post) {
717         Assert.notNull(post, "Post must not be null");
718         BasePostMinimalDTO basePostMinimalDTO = new BasePostMinimalDTO().convertFrom(post);
719
720         basePostMinimalDTO.setFullPath(buildFullPath(post));
721
722         return basePostMinimalDTO;
723     }
724
725     @Override
726     public List<BasePostMinimalDTO> convertToMinimal(List<Post> posts) {
727         if (CollectionUtils.isEmpty(posts)) {
728             return Collections.emptyList();
729         }
730
731         return posts.stream()
732             .map(this::convertToMinimal)
733             .collect(Collectors.toList());
734     }
735
736     @Override
737     public BasePostSimpleDTO convertToSimple(Post post) {
738         Assert.notNull(post, "Post must not be null");
739
740         BasePostSimpleDTO basePostSimpleDTO = new BasePostSimpleDTO().convertFrom(post);
741
742         // Set summary
743         if (StringUtils.isBlank(basePostSimpleDTO.getSummary())) {
744             basePostSimpleDTO.setSummary(generateSummary(post.getFormatContent()));
745         }
746
747         basePostSimpleDTO.setFullPath(buildFullPath(post));
748
749         return basePostSimpleDTO;
750     }
751
752     /**
753      * Converts to post detail vo.
754      *
755      * @param post post must not be null
756      * @param tags tags
757      * @param categories categories
758      * @param postMetaList postMetaList
759      * @return post detail vo
760      */
761     @NonNull
762     private PostDetailVO convertTo(@NonNull Post post, @Nullable List<Tag> tags,
763         @Nullable List<Category> categories, List<PostMeta> postMetaList) {

```



```

764     Assert.notNull(post, "Post must not be null");
765
766     // Convert to base detail vo
767     PostDetailVO postDetailVO = new PostDetailVO().convertFrom(post);
768
769     if (StringUtils.isBlank(postDetailVO.getSummary())) {
770         postDetailVO.setSummary(generateSummary(post.getFormatContent()));
771     }
772
773     // Extract ids
774     Set<Integer> tagIds = ServiceUtils.fetchProperty(tags, Tag::getId);
775     Set<Integer> categoryIds = ServiceUtils.fetchProperty(categories, Category::getId);
776     Set<Long> metaIds = ServiceUtils.fetchProperty(postMetaList, PostMeta::getId);
777
778     // Get post tag ids
779     postDetailVO.setTagIds(tagIds);
780     postDetailVO.setTags(tagService.convertTo(tags));
781
782     // Get post category ids
783     postDetailVO.setCategoryIds(categoryIds);
784     postDetailVO.setCategories(categoryService.convertTo(categories));
785
786     // Get post meta ids
787     postDetailVO.setMetaIds(metaIds);
788     postDetailVO.setMetas(postMetaService.convertTo(postMetaList));
789
790     postDetailVO.setCommentCount(postCommentService.countByStatusAndPostId(
791         CommentStatus.PUBLISHED, post.getId()));
792
793     postDetailVO.setFullPath(buildFullPath(post));
794
795     return postDetailVO;
796 }
797
798 /**
799  * Build specification by post query.
800  *
801  * @param postQuery post query must not be null
802  * @return a post specification
803  */
804 @NonNull
805 private Specification<Post> buildSpecByQuery(@NonNull PostQuery postQuery) {
806     Assert.notNull(postQuery, "Post query must not be null");
807
808     return (root, query, criteriaBuilder) -> {
809         List<Predicate> predicates = new LinkedList<>();
810
811         if (postQuery.getStatus() != null) {
812             predicates.add(criteriaBuilder.equal(root.get("status"), postQuery.getStatus()));

```

```

813     }
814
815     if (postQuery.getCategoryId() != null) {
816         Subquery<Post> postSubquery = query.subquery(Post.class);
817         Root<PostCategory> postCategoryRoot = postSubquery.from(PostCategory.class);
818         postSubquery.select(postCategoryRoot.get("postId"));
819         postSubquery.where(
820             criteriaBuilder.equal(root.get("id"), postCategoryRoot.get("postId")),
821             criteriaBuilder
822                 .equal(postCategoryRoot.get("categoryId"), postQuery.getCategoryId()));
823         predicates.add(criteriaBuilder.exists(postSubquery));
824     }
825
826     if (postQuery.getKeyword() != null) {
827         // Format like condition
828         String likeCondition = String
829             .format("%%%s%%", StringUtils.strip(postQuery.getKeyword()));
830
831         // Build like predicate
832         Predicate titleLike = criteriaBuilder.like(root.get("title"), likeCondition);
833         Predicate originalContentLike = criteriaBuilder
834             .like(root.get("originalContent"), likeCondition);
835
836         predicates.add(criteriaBuilder.or(titleLike, originalContentLike));
837     }
838
839     return query.where(predicates.toArray(new Predicate[0])).getRestriction();
840 };
841 }
842
843 private PostDetailVO createOrUpdate(@NonNull Post post, Set<Integer> tagIds,
844     Set<Integer> categoryIds, Set<PostMeta> metas) {
845     Assert.notNull(post, "Post param must not be null");
846
847     // Create or update post
848     Boolean needEncrypt = Optional.ofNullable(categoryIds)
849         .filter(HaloUtils::isNotEmpty)
850         .map(categoryIdSet -> {
851             for (Integer categoryId : categoryIdSet) {
852                 if (categoryService.categoryHasEncrypt(categoryId)) {
853                     return true;
854                 }
855             }
856             return false;
857         }).orElse(Boolean.FALSE);
858
859     // if password is not empty or parent category has encrypt, change status to intimate
860     if (post.getStatus() != PostStatus.DRAFT
861         && (StringUtils.isNotEmpty(post.getPassword()) || needEncrypt))

```

```

862     ) {
863         post.setStatus(PostStatus.INTIMATE);
864     }
865
866     post = super.createOrUpdateBy(post);
867
868     postTagService.removeByPostId(post.getId());
869
870     postCategoryService.removeByPostId(post.getId());
871
872     // List all tags
873     List<Tag> tags = tagService.listAllByIds(tagIds);
874
875     // List all categories
876     List<Category> categories = categoryService.listAllByIds(categoryIds, true);
877
878     // Create post tags
879     List<PostTag> postTags = postTagService.mergeOrCreateByIfAbsent(post.getId(),
880         ServiceUtils.fetchProperty(tags, Tag::getId));
881
882     log.debug("Created post tags: [{}]", postTags);
883
884     // Create post categories
885     List<PostCategory> postCategories =
886         postCategoryService.mergeOrCreateByIfAbsent(post.getId(),
887             ServiceUtils.fetchProperty(categories, Category::getId));
888
889     log.debug("Created post categories: [{}]", postCategories);
890
891     // Create post meta data
892     List<PostMeta> postMetaList = postMetaService
893         .createOrUpdateByPostId(post.getId(), metas);
894     log.debug("Created post metas: [{}]", postMetaList);
895
896     // Remove authorization every time an post is created or updated.
897     authorizationService.deletePostAuthorization(post.getId());
898
899     // Convert to post detail vo
900     return convertTo(post, tags, categories, postMetaList);
901 }
902
903 @Override
904 @Transactional
905 public Post updateStatus(PostStatus status, Integer postId) {
906     super.updateStatus(status, postId);
907     if (PostStatus.PUBLISHED.equals(status)) {
908         // When the update status is published, it is necessary to determine whether
909         // the post status should be converted to a intimate post
910         categoryService.refreshPostStatus(Collections.singletonList(postId));

```

```

911     }
912     return getById(postId);
913 }
914
915 @Override
916 @Transactional
917 public List<Post> updateStatusByIds(List<Integer> ids, PostStatus status) {
918     if (CollectionUtils.isEmpty(ids)) {
919         return Collections.emptyList();
920     }
921     return ids.stream().map(id -> updateStatus(status, id)).collect(Collectors.toList());
922 }
923
924 @Override
925 public void publishVisitEvent(Integer postId) {
926     eventPublisher.publishEvent(new PostVisitEvent(this, postId));
927 }
928
929 @Override
930 public @NotNull Sort getPostDefaultSort() {
931     String indexSort = optionService.getByPropertyOfNonNull(PostProperties.INDEX_SORT)
932         .toString();
933     return Sort.by(DESC, "topPriority").and(Sort.by(DESC, indexSort).and(Sort.by(DESC, "id")))
934 }
935
936 @Override
937 public List<PostMarkdownVO> listPostMarkdowns() {
938     List<Post> allPostList = listAll();
939     List<PostMarkdownVO> result = new ArrayList(allPostList.size());
940     for (int i = 0; i < allPostList.size(); i++) {
941         Post post = allPostList.get(i);
942         result.add(convertToPostMarkdownVo(post));
943     }
944     return result;
945 }
946
947 private PostMarkdownVO convertToPostMarkdownVo(Post post) {
948     PostMarkdownVO postMarkdownVO = new PostMarkdownVO();
949
950     StringBuilder frontMatter = new StringBuilder("---\n");
951     frontMatter.append("title: ").append(post.getTitle()).append("\n");
952     frontMatter.append("date: ").append(post.getCreateTime()).append("\n");
953     frontMatter.append("updated: ").append(post.getUpdateTime()).append("\n");
954
955     //set fullPath
956     frontMatter.append("url: ").append(buildFullPath(post)).append("\n");
957
958     //set category
959     List<Category> categories = postCategoryService.listCategoriesBy(post.getId());

```

```

960     StringBuilder categoryContent = new StringBuilder();
961     for (int i = 0; i < categories.size(); i++) {
962         Category category = categories.get(i);
963         String categoryName = category.getName();
964         if (i == 0) {
965             categoryContent.append(categoryName);
966         } else {
967             categoryContent.append(" | ").append(categoryName);
968         }
969     }
970     frontMatter.append("categories: ").append(categoryContent.toString()).append("\n");
971
972     //set tags
973     List<Tag> tags = postTagService.listTagsBy(post.getId());
974     StringBuilder tagContent = new StringBuilder();
975     for (int i = 0; i < tags.size(); i++) {
976         Tag tag = tags.get(i);
977         String tagName = tag.getName();
978         if (i == 0) {
979             tagContent.append(tagName);
980         } else {
981             tagContent.append(" | ").append(tagName);
982         }
983     }
984     frontMatter.append("tags: ").append(tagContent.toString()).append("\n");
985
986     frontMatter.append("---\n");
987     postMarkdownVO.setFrontMatter(frontMatter.toString());
988     postMarkdownVO.setOriginalContent(post.getOriginalContent());
989     postMarkdownVO.setTitle(post.getTitle());
990     postMarkdownVO.setSlug(post.getSlug());
991     return postMarkdownVO;
992 }
993
994 private String buildFullPath(Post post) {
995
996     PostPermalinkType permalinkType = optionService.getPostPermalinkType();
997
998     String pathSuffix = optionService.getPathSuffix();
999
1000     String archivesPrefix = optionService.getArchivesPrefix();
1001
1002     int month = DateUtils.month(post.getCreateTime()) + 1;
1003
1004     String monthString = month < 10 ? "0" + month : String.valueOf(month);
1005
1006     int day = DateUtils.dayOfMonth(post.getCreateTime());
1007
1008     String dayString = day < 10 ? "0" + day : String.valueOf(day);

```

```
1009
1010     StringBuilder fullPath = new StringBuilder();
1011
1012     if (optionService.isEnabledAbsolutePath()) {
1013         fullPath.append(optionService.getBlogBaseUrl());
1014     }
1015
1016     fullPath.append(URL_SEPARATOR);
1017
1018     if (permalinkType.equals(PostPermalinkType.DEFAULT)) {
1019         fullPath.append(archivesPrefix)
1020             .append(URL_SEPARATOR)
1021             .append(post.getSlug())
1022             .append(pathSuffix);
1023     } else if (permalinkType.equals(PostPermalinkType.ID)) {
1024         fullPath.append("?p=")
1025             .append(post.getId());
1026     } else if (permalinkType.equals(PostPermalinkType.DATE)) {
1027         fullPath.append(DateUtils.year(post.getCreateTime()))
1028             .append(URL_SEPARATOR)
1029             .append(monthString)
1030             .append(URL_SEPARATOR)
1031             .append(post.getSlug())
1032             .append(pathSuffix);
1033     } else if (permalinkType.equals(PostPermalinkType.DAY)) {
1034         fullPath.append(DateUtils.year(post.getCreateTime()))
1035             .append(URL_SEPARATOR)
1036             .append(monthString)
1037             .append(URL_SEPARATOR)
1038             .append(dayString)
1039             .append(URL_SEPARATOR)
1040             .append(post.getSlug())
1041             .append(pathSuffix);
1042     } else if (permalinkType.equals(PostPermalinkType.YEAR)) {
1043         fullPath.append(DateUtils.year(post.getCreateTime()))
1044             .append(URL_SEPARATOR)
1045             .append(post.getSlug())
1046             .append(pathSuffix);
1047     } else if (permalinkType.equals(PostPermalinkType.ID_SLUG)) {
1048         fullPath.append(archivesPrefix)
1049             .append(URL_SEPARATOR)
1050             .append(post.getId())
1051             .append(pathSuffix);
1052     }
1053     return fullPath.toString();
1054 }
1055 }
```