Momento 项目消息结构设计文档

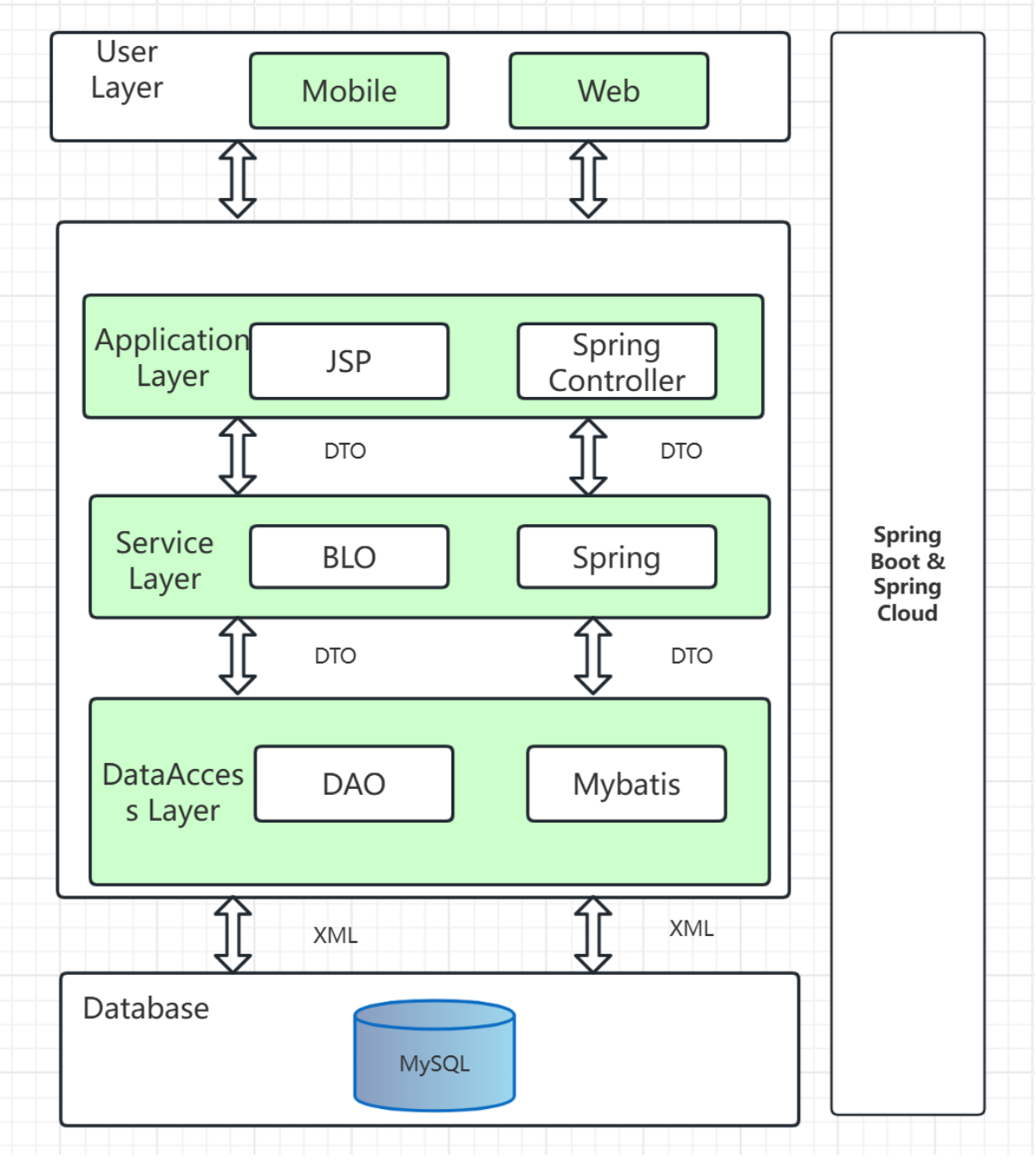
一、概述

Momento 项目采用 **RocketMQ** 作为消息队列中间件，旨在处理高并发场景下的用户交互事件、数据计数服务、缓存同步以及数据异步持久化等关键业务。通过消息队列架构，系统实现了高可用性、可扩展性和各组件之间的松耦合，确保在大规模用户操作下依然保持稳定高效的性能。

主要处理场景包括：

1. **用户交互事件**：如点赞、收藏、关注等操作。
2. **数据计数服务**：统计各类用户交互数据，如点赞数、收藏数等。
3. **缓存同步**：确保缓存与数据库之间的数据一致性。
4. **数据异步持久化**：将关键数据异步保存至数据库，提升系统响应速度。

消息流动设计图



二、消息队列主题设计

项目中定义了多个消息主题（Topic），每个主题负责特定类型的消息传递。以下是详细的主题设计。

2.1 笔记相关主题

```java
public interface MQConstants {

    // 删除笔记本地缓存
    String TOPIC_DELETE_NOTE_LOCAL_CACHE = "DeleteNoteLocalCacheTopic";

    // 点赞与取消点赞共用主题
    String TOPIC_LIKE_OR_UNLIKE = "LikeUnlikeTopic";
    String TAG_LIKE = "Like";
    String TAG_UNLIKE = "Unlike";
    String TOPIC_COUNT_NOTE_LIKE = "CountNoteLikeTopic";

    // 收藏与取消收藏共用主题
    String TOPIC_COLLECT_OR_UN_COLLECT = "CollectUnCollectTopic";
    String TAG_COLLECT = "Collect";
    String TAG_UN_COLLECT = "UnCollect";
    String TOPIC_COUNT_NOTE_COLLECT = "CountNoteCollectTopic";

    // 笔记操作（发布、删除）
    String TOPIC_NOTE_OPERATE = "NoteOperateTopic";
    String TAG_NOTE_PUBLISH = "publishNote";
    String TAG_NOTE_DELETE = "deleteNote";
}
```

主要主题及描述：

- `DeleteNoteLocalCacheTopic`：用于删除笔记的本地缓存。
- `LikeUnlikeTopic`：处理笔记的点赞和取消点赞操作。
- `CountNoteLikeTopic`：统计笔记的点赞数量。
- `CollectUnCollectTopic`：处理笔记的收藏和取消收藏操作。
- `CountNoteCollectTopic`：统计笔记的收藏数量。
- `NoteOperateTopic`：管理笔记的发布和删除操作。

2.2 用户关系相关主题

```java
public interface MQConstants {

    // 关注与取关共用主题
    String TOPIC_FOLLOW_OR_UNFOLLOW = "FollowUnfollowTopic";
    String TAG_FOLLOW = "Follow";
    String TAG_UNFOLLOW = "Unfollow";

    // 关注数与粉丝数计数
    String TOPIC_COUNT_FOLLOWING = "CountFollowingTopic";
    String TOPIC_COUNT_FANS = "CountFansTopic";
}
```

主要主题及描述：

- `FollowUnfollowTopic`：处理用户的关注和取关操作。

- `CountFollowingTopic`：统计用户的关注数量。
- `CountFansTopic`：统计用户的粉丝数量。

2.3 计数服务相关主题

```java
public interface MQConstants {

    // 关注数与粉丝数计数
    String TOPIC_COUNT_FOLLOWING = "CountFollowingTopic";
    String TOPIC_COUNT_FANS = "CountFansTopic";

    // 关注数与粉丝数计数入库
    String TOPIC_COUNT_FANS_2_DB = "CountFans2DBTopic";
    String TOPIC_COUNT_FOLLOWING_2_DB = "CountFollowing2DBTopic";

    // 笔记点赞数与收藏数计数
    String TOPIC_COUNT_NOTE_LIKE = "CountNoteLikeTopic";
    String TOPIC_COUNT_NOTE_LIKE_2_DB = "CountNoteLike2DBTTopic";
    String TOPIC_COUNT_NOTE_COLLECT = "CountNoteCollectTopic";
    String TOPIC_COUNT_NOTE_COLLECT_2_DB = "CountNoteCollect2DBTTopic";

    // 笔记操作（发布、删除）
    String TOPIC_NOTE_OPERATE = "NoteOperateTopic";
    String TAG_NOTE_PUBLISH = "publishNote";
    String TAG_NOTE_DELETE = "deleteNote";
}
```

主要主题及描述：

- `CountFans2DBTopic`：将粉丝数量统计结果持久化到数据库。
- `CountFollowing2DBTopic`：将关注数量统计结果持久化到数据库。
- `CountNoteLike2DBTTopic`：将笔记点赞数量统计结果持久化到数据库。
- `CountNoteCollect2DBTTopic`：将笔记收藏数量统计结果持久化到数据库。

三、消息消费者实现

消息消费者负责订阅并处理特定主题的消息，执行相应的业务逻辑。以下是主要的消费者实现概述。

3.1 关注/取关消费者

职责：处理用户的关注和取关操作，更新相关的关注数和粉丝数。

关键实现点：

- **顺序消费模式**：确保同一用户的操作按顺序处理。
- **流量控制**：使用令牌桶算法限制每秒的处理请求数，防止过载。
- **幂等性设计**：通过联合唯一索引和状态检查，防止重复处理同一消息。

示例代码：

```
@Component
@RocketMQMessageListener(
```

```
    consumerGroup = "momento_group_FollowUnfollowTopic",
    topic = "FollowUnfollowTopic",
    consumeMode = ConsumeMode.ORDERLY
)
@Slf4j
public class FollowUnfollowConsumer implements RocketMQListener<Message> {

    @Autowired
    private FollowingDOMapper followingDOMapper;
    @Autowired
    private FansDOMapper fansDOMapper;
    @Autowired
    private RateLimiter rateLimiter;

    @Override
    public void onMessage(Message message) {
        // 流量削峰
        rateLimiter.acquire();

        // 解析消息
        String body = new String(message.getBody(), StandardCharsets.UTF_8);
        String tags = message.getTags();

        log.info("FollowUnfollowConsumer 消费了消息: {}, 标签: {}", body,
tags);

        // 业务逻辑处理（伪代码）
        // FollowUnfollowDTO dto = parseMessage(body);
        // if (tags.equals("Follow")) { handleFollow(dto); }
        // else if (tags.equals("Unfollow")) { handleUnfollow(dto); }
    }
}
```

3.2 点赞/取消点赞消费者

职责：处理笔记的点赞和取消点赞操作，更新点赞数。

关键实现点：

- **顺序消费模式**：确保点赞和取消点赞操作按顺序处理。
- **流量控制**：使用令牌桶算法限制处理速率。
- **幂等性设计**：通过数据库约束和状态检查，防止重复点赞或取消。

示例代码：

```
@Component
@RocketMQMessageListener(
    consumerGroup = "momento_group_LikeUnlikeTopic",
    topic = "LikeUnlikeTopic",
    consumeMode = ConsumeMode.ORDERLY
)
@Slf4j
public class LikeUnlikeNoteConsumer implements RocketMQListener<Message> {

    @Autowired
```

```
    private NoteLikeDOMapper noteLikeDOMapper;
    @Autowired
    private RateLimiter rateLimiter;

    @Override
    public void onMessage(Message message) {
        // 流量削峰
        rateLimiter.acquire();

        // 解析消息
        String body = new String(message.getBody(), StandardCharsets.UTF_8);
        String tags = message.getTags();

        log.info("LikeUnlikeNoteConsumer 消费了消息: {}, 标签: {}", body,
tags);

        // 业务逻辑处理（伪代码）
        // LikeUnlikeDTO dto = parseMessage(body);
        // if (tags.equals("Like")) { handleLike(dto); }
        // else if (tags.equals("Unlike")) { handleUnlike(dto); }
    }
}
```

3.3 收藏/取消收藏消费者

职责：处理笔记的收藏和取消收藏操作，更新收藏数。

关键实现点：

- **顺序消费模式**：确保收藏和取消收藏操作按顺序处理。
- **流量控制**：使用令牌桶算法限制处理速率。
- **幂等性设计**：通过数据库约束和状态检查，防止重复收藏或取消。

示例代码：

```
@Component
@RocketMQMessageListener(
    consumerGroup = "momento_group_CollectUnCollectTopic",
    topic = "CollectUnCollectTopic",
    consumeMode = ConsumeMode.ORDERLY
)
@Slf4j
public class CollectUnCollectNoteConsumer implements
RocketMQListener<Message> {

    @Autowired
    private NoteCollectionDOMapper noteCollectionDOMapper;
    @Autowired
    private RateLimiter rateLimiter;

    @Override
    public void onMessage(Message message) {
        // 流量削峰
        rateLimiter.acquire();

```

```
        // 解析消息
        String body = new String(message.getBody(), StandardCharsets.UTF_8);
        String tags = message.getTags();

        log.info("CollectUnCollectNoteConsumer 消费了消息: {}, 标签: {}", body,
tags);

        // 业务逻辑处理（伪代码）
        // CollectUnCollectDTO dto = parseMessage(body);
        // if (tags.equals("Collect")) { handleCollect(dto); }
        // else if (tags.equals("UnCollect")) { handleUnCollect(dto); }
    }
}
```

四、消息发送示例

以下以 **笔记删除** 操作为例，展示如何构建消息体并发送到相应的主题。

4.1 笔记删除示例

步骤：

1. **构建消息体 DTO**：封装需要传递的数据。
2. **创建消息对象**：将 DTO 转换为 JSON 字符串并构建消息体。
3. **指定发送目标**：结合 Topic 和 Tag，确保消息被正确路由。
4. **异步发送消息**：提升接口响应速度，通过回调处理发送结果。

示例代码：

```
@Service
public class NoteServiceImpl implements NoteService {

    @Autowired
    private NoteDOMapper noteDOMapper;
    @Autowired
    private RocketMQTemplate rocketMQTemplate;

    @Override
    public Response deleteNote(Long noteId) {
        // 查询笔记
        NoteDO note = noteDOMapper.selectById(noteId);
        if (note == null) {
            return Response.error("笔记不存在");
        }

        // 删除笔记逻辑
        noteDOMapper.deleteById(noteId);

        // 构建消息体 DTO
        NoteOperateMqDTO dto = NoteOperateMqDTO.builder()
                .creatorId(note.getCreatorId())
                .noteId(noteId)
                .type(NoteOperateEnum.DELETE.getCode())
```

```
                .build();

        // 构建消息对象
        Message<String> message =
MessageBuilder.withPayload(JsonUtils.toJsonString(dto))
                .build();

        // 指定发送目标（Topic + Tag）
        String destination = "NoteOperateTopic:deleteNote";

        // 异步发送消息
        rocketMQTemplate.asyncSend(destination, message, new SendCallback()
{
            @Override
            public void onSuccess(SendResult sendResult) {
                log.info("【笔记删除】MQ 发送成功，SendResult: {}", sendResult);
            }

            @Override
            public void onException(Throwable throwable) {
                log.error("【笔记删除】MQ 发送异常: ", throwable);
            }
        });

        return Response.success();
    }
}
```

五、主要消息体设计

1. 用户相关代码

1.1 用户注册请求 DTO

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class RegisterUserReqDTO {

    /**
     * 邮箱
     */
    @NotBlank(message = "邮箱不能为空")
    @Email(message = "邮箱格式不正确")
    private String email;

}

```

1.2 用户服务接口

```
public interface UserService {

    /**
     * 更新用户信息
     *
     * @param updateUserInfoReqVO
     * @return
     */
    Response<?> updateUserInfo(UpdateUserInfoReqVO updateUserInfoReqVO);

    /**
     * 用户注册
     *
     * @param registerUserReqDTO
     * @return
     */
    Response<Long> register(RegisterUserReqDTO registerUserReqDTO);

    /**
     * 根据邮箱查询用户信息
     *
     * @param findUserByEmailReqDTO
     * @return
     */
    Response<FindUserByEmailRspDTO> findByEmail(FindUserByEmailReqDTO
findUserByEmailReqDTO);


    //根据昵称查找
    Response<FindUserByNicknameRspDTO>
findByNickname(FindUserByNicknameReqDTO findUserByNicknameReqDTO);

    /**
     * 更新密码
     *
     * @param updateUserPasswordReqDTO
     * @return
     */
    Response<?> updatePassword(UpdateUserPasswordReqDTO
updateUserPasswordReqDTO);

    /**
     * 根据用户 ID 查询用户信息
     *
     * @param findUserByIdReqDTO
     * @return
     */
    Response<FindUserByIdRspDTO> findById(FindUserByIdReqDTO
findUserByIdReqDTO);

    /**
     * 批量根据用户 ID 查询用户信息
     *
```

```
     * @param findUsersByIdsReqDTO
     * @return
     */
    Response<List<FindUserByIdRspDTO>> findByIds(FindUsersByIdsReqDTO findUsersByIdsReqDTO);


    Response<UserDO> getCurrentUserInfo();
}
```

1.3 用户登录请求 VO

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class UserLoginReqVO {

    /**
     * 邮箱
     */
    @NotBlank(message = "邮箱不能为空")
    @Email
    private String email;

    /**
     * 验证码
     */
    private String code;

    /**
     * 密码
     */
    private String password;

    /**
     * 登录类型：邮箱验证码，或者是账号密码
     */
    @NotNull(message = "登录类型不能为空")
    private Integer type;
}
```

1.4 用户服务实现

```
        String introduction = updateUserInfoReqVO.getIntroduction();
        if (StringUtils.isNotBlank(introduction)) {
            Preconditions.checkArgument(ParamUtils.checkLength(introduction, 100), ResponseCodeEnum.INTRODUCTION_VALID_FAIL.getErrorMessage());
            userDO.setIntroduction(introduction);
            needUpdate = true;
        }
```

```
7
8           // 背景图
9           MultipartFile backgroundImgFile =
updateUserInfoReqVO.getBackgroundImg();
10          if (Objects.nonNull(backgroundImgFile)) {
11              String backgroundImg =
ossRpcService.uploadFile(backgroundImgFile);
12              log.info("==> 调用 oss 服务成功，上传背景图，url：{}",
backgroundImg);
13
14              if (StringUtils.isBlank(backgroundImg)) {
15                  throw new
BizException(ResponseCodeEnum.UPLOAD_BACKGROUND_IMG_FAIL);
16              }
17
18              userDO.setBackgroundImg(backgroundImg);
19              needUpdate = true;
20          }
21
22          if (needUpdate) {
23              // 更新用户信息
24              userDO.setUpdateTime(LocalDateTime.now());
25              userDOMapper.updateByPrimaryKeySelective(userDO);
26          }
27          return Response.success();
28      }
29
30      /**
31       * 用户注册
32       *
33       * @param registerUserReqDTO
34       * @return
35       */
36      @Override
37      @Transactional(rollbackFor = Exception.class)
38      public Response<Long> register(RegisterUserReqDTO registerUserReqDTO) {
39          String email = registerUserReqDTO.getEmail();
40
41          // 先判断该邮箱是否已被注册
42          UserDO userDO1 = userDOMapper.selectByEmail(email);
43
44          log.info("==> 用户是否注册，email: {}, userDO: {}", email,
JsonUtils.toJsonString(userDO1));
45
46          // 若已注册，则直接返回用户 ID
47          if (Objects.nonNull(userDO1)) {
48              return Response.success(userDO1.getId());
49          }
50
51          // 否则注册新用户
52          // RPC: 调用分布式 ID 生成服务生成小哈书 ID
53          String momentoId = distributedIdGeneratorRpcService.getMomentoId();
54
55          // RPC: 调用分布式 ID 生成服务生成用户 ID
56          String userIdStr = distributedIdGeneratorRpcService.getUserId();
57          Long userId = Long.valueOf(userIdStr);
```

```
58
59          UserDO userDO = UserDO.builder()
60                  .id(userId)
61                  .email(email)
62                  .momentoId(momentoId)
63                  .nickname("小红薯" + userIdStr)
```

1.5 根据邮箱查询用户请求 DTO

```
1  @Data
2  @AllArgsConstructor
3  @NoArgsConstructor
4  @Builder
5  public class FindUserByEmailReqDTO {
6
7      /**
8       * 邮箱
9       */
10     @NotBlank(message = "邮箱不能为空")
11     @Email(message = "邮箱格式不正确")
12     private String email;
13
14 }
15
```

1.6 用户响应 DTO

```
1  @Data
2  @AllArgsConstructor
3  @NoArgsConstructor
4  @Builder
5  public class FindUserByIdRspDTO {
6
7      /**
8       * 用户 ID
9       */
10     private Long id;
11
12     /**
13      * 昵称
14      */
15     private String nickName;
16
17     /**
18      * 头像
19      */
20     private String avatar;
21
22     /**
23      * 简介
24      */
25     private String introduction;
26 }
27
```

```

```

1.7 用户控制器

```
@RestController
@RequestMapping("/user")
@Slf4j
public class UserController {

    @Resource
    private UserService userService;

    /**
     * 用户信息修改
     *
     * @param updateUserInfoReqVO
     * @return
     */
    @PostMapping(value = "/update", consumes =
MediaType.MULTIPART_FORM_DATA_VALUE)
    public Response<?> updateUserInfo(@Validated UpdateUserInfoReqVO
updateUserInfoReqVO) {
        return userService.updateUserInfo(updateUserInfoReqVO);
    }

    // ==================================== 对其他服务提供的接口
====================================
    @PostMapping("/register")
    @ApiOperationLog(description = "用户注册")
    public Response<Long> register(@Validated @RequestBody RegisterUserReqDTO
registerUserReqDTO) {
        return userService.register(registerUserReqDTO);
    }

    @PostMapping("/findByEmail")
    @ApiOperationLog(description = "邮箱查询用户信息")
    public Response<FindUserByEmailRspDTO> findByEmail(@Validated
@RequestBody FindUserByEmailReqDTO findUserByEmailReqDTO) {
        return userService.findByEmail(findUserByEmailReqDTO);
    }

    @PostMapping("/findByNickname")
    @ApiOperationLog(description = "昵称查询用户信息")
    public Response<FindUserByNicknameRspDTO> findByNickname(@Validated
@RequestBody FindUserByNicknameReqDTO findUserByNicknameReqDTO) {
        return userService.findByNickname(findUserByNicknameReqDTO);
    }

    @PostMapping("/password/update")
    @ApiOperationLog(description = "密码更新")
    public Response<?> updatePassword(@Validated @RequestBody
UpdateUserPasswordReqDTO updateUserPasswordReqDTO) {
        return userService.updatePassword(updateUserPasswordReqDTO);
    }
```

```
44
45      @PostMapping("/findById")
46      @ApiOperationLog(description = "查询用户信息")
47      public Response<FindUserByIdRspDTO> findById(@Validated @RequestBody
    FindUserByIdReqDTO findUserByIdReqDTO) {
48          return userService.findById(findUserByIdReqDTO);
49      }
50
51      @PostMapping("/findByIds")
52      @ApiOperationLog(description = "批量查询用户信息")
53      public Response<List<FindUserByIdRspDTO>> findByIds(@Validated
    @RequestBody FindUsersByIdsReqDTO findUsersByIdsReqDTO) {
54          return userService.findByIds(findUsersByIdsReqDTO);
55      }
56
57      @GetMapping("/current")
58      public ResponseEntity<Response<UserDO>> getCurrentUserInfo() {
59          return ResponseEntity.ok(userService.getCurrentUserInfo());
60      }
61
62  }
63
```

1.8 用户数据对象

```
1   @Data
2   @AllArgsConstructor
3   @NoArgsConstructor
4   @Builder
5   public class UserDO {
6       private Long id;
7
8       private String momentoId;
9
10      private String password;
11
12      private String nickname;
13
14      private String avatar;
15
16      private LocalDate birthday;
17
18      private String backgroundImg;
19
20      private String phone;
21
22      private Integer sex;
23
24      private Integer status;
25
26      private String introduction;
27
28      private LocalDateTime createTime;
29
30      private LocalDateTime updateTime;
```

```
31
32      private Boolean isDeleted;
33
34      private String email;
35  }
```

1.9 根据用户 ID 查询请求 DTO

```
1   @Data
2   @AllArgsConstructor
3   @NoArgsConstructor
4   @Builder
5   public class FindUserByIdReqDTO {
6
7       /**
8        * ID
9        */
10      @NotNull(message = "用户 ID 不能为空")
11      private Long id;
12
13  }
14
15
16
```

1.10 根据昵称查询用户请求 DTO

```
1   @Data
2   @AllArgsConstructor
3   @NoArgsConstructor
4   @Builder
5   public class FindUserByNicknameReqDTO {
6
7       /**
8        * 昵称
9        */
10      @NotBlank(message = "昵称不能为空")
11      private String nickname;
12  }
13
```

2. 笔记相关代码

2.1 笔记控制器

```
1       public Response<?> getNoteList(@RequestBody NoteListReqVO noteListReqVO)
    {
2           // 通过请求体获取 page 和 size 参数
3           return noteService.getNoteList(noteListReqVO.getPage(),
    noteListReqVO.getSize());
4       }
5
6       /**
```

```
     * 获取当前用户的笔记列表
     *
     * @param request 请求体参数（包括用户ID、分页、筛选条件等）
     * @return 当前用户的笔记列表
     */
    @PostMapping("/UserNoteList")
    public Response<?> getUserNoteList(@RequestBody GetUserNotesRequest request) {
        return noteService.getUserNotes(
                request.getUserId(),
                request.getPage(),
                request.getSize(),
                request.getIsTop(),
                request.getVisible(),
                request.getStatus()
        );
    }

    @PostMapping("/liked/list")
    @ApiOperationLog(description = "分页查询用户点赞过的笔记")
    public Response<List<LikedNoteVO>> getUserLikedNotes(@Validated @RequestBody GetUserLikedNotesRequest request) {
        return noteService.getUserLikedNotes(
                request.getUserId(),
                request.getPage(),
                request.getSize()
        );
    }

    @PostMapping("/collected/list")
```

2.2 笔记操作 MQ DTO

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class NoteOperateMqDTO {

    /**
     * 笔记发布者 ID
     */
    private Long creatorId;

    /**
     * 笔记 ID
     */
    private Long noteId;

    /**
     * 操作类型： 0 - 笔记删除； 1：笔记发布；
     */
    private Integer type;

}
```

```

```

2.3 笔记列表请求 VO

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class NoteListReqVO {

    private int page;   // 页码，表示请求哪一页
    private int size;   // 每页显示多少条记录
}

```

2.4 评论服务实现

```
        CommentDO commentDO = CommentDO.builder()
                .id(Long.valueOf(snowflakeIdId))
                .noteId(commentReqVO.getNoteId())
                .userId(currentUserId) // 使用当前用户ID
                .parentId(commentReqVO.getParentId())
                .content(commentReqVO.getContent().trim())
                .status((byte) 0) // 默认状态为正常
                .createTime(LocalDateTime.now())
                .updateTime(LocalDateTime.now())
                .build();

        // 插入数据库
        int result = commentDOMapper.insertSelective(commentDO);

        if (result > 0) {
            log.info("评论添加成功: {}", commentDO);
            return Response.success();
        } else {
            log.error("评论添加失败: {}", commentDO);
            throw new BizException(ResponseCodeEnum.COMMENT_ERROR);
        }
    }

    //删除评论
    @Override
    public Response<?> deleteComment(Long commentId) {
        // 从上下文中获取当前用户ID
        Long currentUserId = LoginUserContextHolder.getUserId();

        // 查询评论信息
        CommentDO comment = commentDOMapper.selectByPrimaryKey(commentId);
...
        // 参数校验
        Preconditions.checkArgument(noteId != null && noteId > 0, "笔记ID不能为空且必须大于0");
        Preconditions.checkArgument(page > 0, "页码必须大于0");
```

```
        Preconditions.checkArgument(size > 0, "每页大小必须大于0");

        // 计算分页参数
        int offset = (page - 1) * size;

        // 查询评论及用户信息
        List<CommentWithUserVO> commentWithUserVOList =
                commentDOMapper.selectCommentsWithUserByNoteId(noteId,
offset, size);

        // 返回结果
        return Response.success(commentWithUserVOList);
    }

    //查看二级评论
    @Override
    public Response<List<CommentRspVO>> getRepliesByCommentId(Long commentId)
{
        // 参数校验
        Preconditions.checkArgument(commentId != null && commentId > 0, "父评
论ID不能为空且必须大于0");

        // 查询子评论列表（包含用户信息）
        List<CommentRspVO> replies =
commentDOMapper.selectRepliesWithUserByCommentId(commentId)
                .stream()
                .map(commentDO -> {
                    CommentRspVO commentRspVO = new CommentRspVO();
                    commentRspVO.setId(commentDO.getId());
                    commentRspVO.setNoteId(commentDO.getNoteId());
                    commentRspVO.setUserId(commentDO.getUserId());
                    commentRspVO.setUserName(commentDO.getUserName());
                    commentRspVO.setUserAvatar(commentDO.getUserAvatar());
                    commentRspVO.setContent(commentDO.getContent());
                    commentRspVO.setCreateTime(commentDO.getCreateTime());
                    commentRspVO.setUpdateTime(commentDO.getUpdateTime());
                    commentRspVO.setParentId(commentDO.getParentId());

                    // 递归查询子评论
                    List<CommentRspVO> childReplies =
getRepliesByCommentId(commentDO.getId()).getData();
                    commentRspVO.setReplies(childReplies);

                    return commentRspVO;
                })
                .collect(Collectors.toList());

        return Response.success(replies);
    }

    //统计某笔记的评论数
    @Override
    public Response<Integer> countCommentsByNoteId(Long noteId) {
        // 参数校验
        Preconditions.checkArgument(noteId != null && noteId > 0, "笔记ID不能
为空且必须大于0");
```

```
        // 查询评论总数
        int count = commentDOMapper.countCommentsByNoteId(noteId);

```

3. 用户关系相关代码

3.1 关注/取关 MQ DTO

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class CountFollowUnfollowMqDTO {

    /**
     * 原用户
     */
    private Long userId;

    /**
     * 目标用户
     */
    private Long targetUserId;

    /**
     * 1:关注 0:取关
     */
    private Integer type;

}
```

4. 收藏相关代码

4.1 收藏/取消收藏 MQ DTO

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class CountCollectUnCollectNoteMqDTO {

    private Long userId;

    private Long noteId;

    /**
     * 0: 取消收藏， 1：收藏
     */
    private Integer type;

    private LocalDateTime createTime;

    /**
```

```
     * 笔记发布者 ID
     */
    private Long noteCreatorId;
}
```

5. 用户 API 接口

5.1 用户 Feign API

```
@FeignClient(name = ApiConstants.SERVICE_NAME)
public interface UserFeignApi {

    String PREFIX = "/user";

    /**
     * 用户注册
     *
     * @param registerUserReqDTO
     * @return
     */
    @PostMapping(value = PREFIX + "/register")
    Response<Long> registerUser(@RequestBody RegisterUserReqDTO registerUserReqDTO);

    /**
     * 根据邮箱查询用户信息
     *
     * @param findUserByEmailReqDTO
     * @return
     */
    @PostMapping(value = PREFIX + "/findByEmail")
    Response<FindUserByEmailRspDTO> findByEmail(@RequestBody FindUserByEmailReqDTO findUserByEmailReqDTO);

    /**
     * 更新密码
     *
     * @param updateUserPasswordReqDTO
     * @return
     */
    @PostMapping(value = PREFIX + "/password/update")
    Response<?> updatePassword(@RequestBody UpdateUserPasswordReqDTO updateUserPasswordReqDTO);

    /**
     * 根据用户 ID 查询用户信息
     *
     * @param findUserByIdReqDTO
     * @return
     */
    @PostMapping(value = PREFIX + "/findById")
    Response<FindUserByIdRspDTO> findById(@RequestBody FindUserByIdReqDTO findUserByIdReqDTO);

    /**
```

```
43          * 批量查询用户信息
44          *
45          * @param findUsersByIdsReqDTO
46          * @return
47          */
48         @PostMapping(value = PREFIX + "/findByIds")
49         Response<List<FindUserByIdRspDTO>> findByIds(@RequestBody
    FindUsersByIdsReqDTO findUsersByIdsReqDTO);
50     }
51
```

六、关键特性

6.1 消息顺序性保证

为了确保消息按预期顺序处理，项目在生产和消费端采取了以下措施：

1. **发送端保证**：
   - 使用 **同步发送** 方式，确保消息发送的顺序性。
   - 对相关操作使用相同的 **MessageQueue**，避免跨分区顺序混乱。
2. **消费端保证**：
   - 配置消费者为 **顺序消费模式**（`ConsumeMode.ORDERLY`），确保同一队列内的消息按顺序处理。

6.2 消息可靠性保证

系统通过多种机制确保消息的可靠传递和处理：

1. **生产端**：
   - **同步发送**：生产者在发送消息后等待服务器的确认，确保消息已被成功接收。
   - **异步发送回调**：通过回调函数处理发送成功或失败的情况。
   - **事务消息机制**：支持事务消息，确保消息与本地事务的一致性。
2. **消费端**：
   - **消费失败重试机制**：当消费者处理消息失败时，系统会自动重试，确保消息最终被处理。
   - **死信队列处理**：对于多次处理失败的消息，系统将其转入死信队列，避免无限重试。

6.3 性能优化

为应对高并发和大流量场景，Momento 项目在性能方面做了多项优化：

1. **流量控制**：
   - 使用 **令牌桶算法** 限制每秒的请求速率（如每秒5000个请求），防止系统过载。
   - 监控系统的 **吞吐量** 和 **延迟**，及时调整流量控制策略。
2. **异步处理**：
   - 将非核心业务流程采用 **异步化** 处理，提升接口响应速度和系统整体性能。
3. **消息批量处理**：
   - 通过 **批量发送** 和 **批量消费** 减少网络传输次数，降低 RocketMQ 服务器的负载。

6.4 幂等性保证

为了防止消息的重复消费导致的数据不一致，系统在多个层面实现了幂等性设计：

1. **数据库层**：
    - 使用 **联合唯一索引**，确保关键字段的唯一性，防止重复插入。
    - 采用 **乐观锁机制**，在并发更新时避免数据冲突。
2. **业务层**：
    - 在处理消息前进行 **状态检查**，确认操作是否已执行。
    - 实现 **去重处理**，过滤掉重复的消息请求。

七、监控与告警

为了保障系统的稳定运行，Momento 项目实现了全面的监控和告警机制。

7.1 日志监控

通过详细的日志记录，系统能够实时跟踪消息的处理状态和异常情况。

```
log.info("消息处理开始: {}", message);
log.error("消息处理异常: ", e);
```

监控内容包括：

- 消息消费的开始和结束。
- 消息处理过程中的关键步骤和状态。
- 消息处理异常和错误信息。

7.2 告警实现

系统支持多种告警方式，确保在异常发生时能够及时响应和处理。

```
public interface AlarmInterface {
    boolean send(String message);
}

// 邮件告警
@Slf4j
public class MailAlarmHelper implements AlarmInterface {
    @Override
    public boolean send(String message) {
        log.info("【邮件告警】: {}", message);
        return true;
    }
}

// 短信告警
@Slf4j
public class SmsAlarmHelper implements AlarmInterface {
    @Override
    public boolean send(String message) {
```

```
        log.info("【短信告警】: {}", message);
        return true;
    }
}
```

告警方式包括:

- **邮件告警**：发送详细的错误信息到指定的邮箱。
- **短信告警**：将关键的告警信息通过短信方式发送到管理员手机。