11 - 团队空间 - 智能协同云图库项目教程 - 编程导航教程

> 本节重点从本节开始我们将进行项目第三阶段 —— 团队空间的开发，让项目能够面向 B 端（企业）提供服务，比如作为团队共享素材、团队活动相册等，增强项目的商业价值。

本节重点

从本节开始我们将进行项目第三阶段 —— 团队空间的开发，让项目能够面向 B 端（企业）提供服务，比如作为团队共享素材、团队活动相册等，增强项目的商业价值。

本节先给项目增加团队共享空间的能力，大纲：

- 团队空间需求分析
- 团队空间方案设计
- 团队空间后端开发
- 团队空间前端开发

本节学完后，你应该能够掌握一个团队协作系统的方案设计和开发。

⭐ 友情提示，本节涉及的后端新技术较多，学习难度略大，而且细节很多，请勿必仔细学习！

一、需求分析

之前我们已经完成了私有空间模块，团队空间和它类似，我们可以拆分为 4 个需求：

1）创建团队共享空间

用户可以创建 **最多一个** 团队共享空间，用于团队协作和资源共享，空间管理员拥有私有空间的所有能力，包括自由上传图片、检索图片、管理图片、分析空间等。

2）空间成员管理

- 成员邀请：空间管理员可以邀请新成员加入团队，共享空间内的图片。
- 设置权限：空间管理员可以为成员设置不同的角色（如查看者、编辑者、管理员），控制成员的权限范围。

3）空间成员权限控制：仅特定角色的成员可访问或操作团队空间内的图片。

4）空间数据管理：考虑到团队空间的图片数量可能比较多，可以对特定空间的数据进行单独的管理，而不是和公共图库、私有空间的图片混在一起。

二、方案设计

让我们先依次分析上述需求，并思考对应的解决方案。

创建团队共享空间

之前已经开发了空间模块，团队空间可以直接复用私有空间的大多数能力。因此可以给空间表新增一个 spaceType 字段，用于区分私有和团队空间。

```
ALTER TABLE space
    ADD COLUMN spaceType int default 0 not null comment '空间类型：0-私

CREATE INDEX idx_spaceType ON space (spaceType);
```

空间成员管理

1、业务流程

为了让项目更容易扩展，减少原有代码的修改，我们约定 只有团队空间才有成员的概念。

1）成员邀请：空间管理员可以直接输入成员 id 来添加新成员，无需该用户确认，这样可以提高开发效率。

2）设置权限：空间管理员可以为已加入成员设置不同的角色，控制成员的权限范围，类似于编辑成员信息。

2、库表设计

由于空间和用户是多对多的关系，还要同时记录用户在某空间的角色，所以需要新建关联表：

```
create table if not exists space_user
(
    id         bigint auto_increment comment 'id' primary key,
    spaceId    bigint                                not null commen
    userId     bigint                                not null commen
    spaceRole  varchar(128) default 'viewer'         null comment '
    createTime datetime     default CURRENT_TIMESTAMP not null commen
    updateTime datetime     default CURRENT_TIMESTAMP not null on upd

    UNIQUE KEY uk_spaceId_userId (spaceId, userId),
    INDEX idx_spaceId (spaceId),
    INDEX idx_userId (userId)
) comment '空间用户关联' collate = utf8mb4_unicode_ci;
```

注意几个细节：

1. 给 spaceId 和 userId 添加唯一索引，确保同一用户在同一空间中只能有一个角色（不能重复加入）。由于有唯一键，不需要使用逻辑删除字段，否则无法退出后再重新加入。

2. 给关联字段添加索引，提高查询效率

3. 为了跟用户自身在项目中的角色 userRole 区分开，空间角色的名称使用 spaceRole

为保证逻辑的统一，创建团队空间时要自动将创建人作为空间管理员，保存到空间成员表中。

空间成员权限控制

仅特定角色的成员可访问或操作团队空间内的图片。

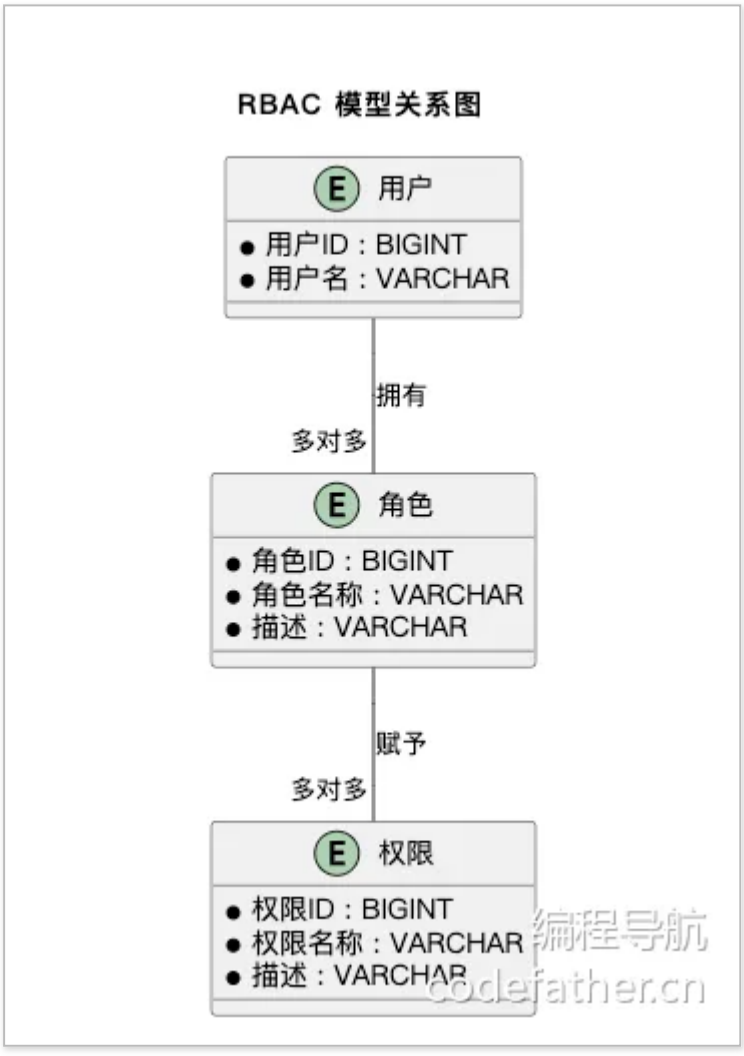
团队空间的权限管理可比私有空间的权限复杂多了，除了创建人外还有其他成员，涉及到查看图片、上传图片、管理空间图片、管理空间等多种不同的权限。

1、RBAC 权限控制

对于复杂的权限控制场景，我们可以采用经典的 RBAC 权限控制模型（基于角色的访问控制，Role-Based Access Control），核心概念包括 用户、角色、权限。

- 一个用户可以有多个角色
- 一个角色可以有多个权限

这样一来，就可以灵活地配置用户具有的权限了。



一般来说，标准的 RBAC 实现需要 5 张表：用户表、角色表、权限表、用户角色关联表、角色权限关联表，还是有一

定开发成本的。由于我们的项目中，团队空间不需要那么多角色，可以简化 RBAC 的实现方式，比如将角色和权限直接定义到配置文件中。

2、角色和权限定义

本项目的角色：

角色	描述
浏览者	仅可查看空间中的图片内容
编辑者	可查看、上传和编辑图片内容
管理员	拥有管理空间和成员的所有权限

本项目的权限：

权限键	功能名称	描述
spaceUser:manage	成员管理	管理空间成员，添加或移除成员
picture:view	查看图片	查看空间中的图片内容
picture:upload	上传图片	上传图片到空间中
picture:edit	修改图片	编辑已上传的图片信息
picture:delete	删除图片	删除空间中的图片

角色与权限映射：

角色	对应权限键	可执行功能
浏览	picture:view	查看图片

角色	对应权限键	可执行功能
编辑者	picture:view, picture:upload, picture:edit, picture:delete	查看图片、上传图片、修改图片、删除图片
管理员	spaceUser:manage, picture:view, picture:upload, picture:edit, picture:delete	成员管理、查看图片、上传图片、修改图片、删除图片

3、权限校验实现方案

RBAC 只是一种权限设计模型，我们在 Java 代码中如何实现权限校验呢？

1）最直接的方案是像之前校验私有空间权限一样，封装个团队空间的权限校验方法；或者类似用户权限校验一样，写个注解 + AOP 切面。

2）对于复杂的角色和权限管理，可以选用现成的第三方权限校验框架来实现，编写一套权限校验规则代码后，就能整体管理系统的权限校验逻辑了。

其实在本项目中，由于角色和权限不多，采用方案 1 实现会更方便一些，我也建议大家优先选择这种方案。方案 2 的代码量虽然未必比方案 1 少，但是会让整个系统的权限校验逻辑更加清晰，为了让大家后续能够应对更复杂的权限管理需求，此处鱼皮给大家讲解方案 2，并选用国内主流的 权限校验框架 Sa-Token 实现。

空间数据管理

考虑到团队空间的图片数量可能比较多，可以对特定空间的数据进行单独的管理。

如何对数据进行单独的管理呢？

1、图片信息数据

可以给每个团队空间单独创建一张图片表 `picture_{spaceId}` ，也就是分库分表中的 `分表` ，而不是和公共图库、私有空间的图片混在一起。这样不仅查询空间内的图片效率更高，还便于整体管理和清理空间。**但是要注意，仅对旗舰版空间生效，否则分表的数量会特别多，反而可能影响性能。**

注意，我们要实现的，还不是普通的静态分表，而是会随着新增空间不断增加分表数量的动态分表，会使用分库分表框架 Apache ShardingSphere 带大家实现。

2、图片文件数据

已经将每个空间的图片存到不同的路径中了，实现了隔离，无需额外开发。

💡 你会发现，我们在设计上就将团队空间和私有空间隔离，仅对团队空间应用成员管理、权限控制、动态分表。这样可以尽量减少对原有代码的改动，避免出现问题。

三、后端开发

创建团队共享空间

1、数据模型

Space、SpaceVO、SpaceAddRequest、SpaceQueryRequest 补充 spaceType 字段：

```
private Integer spaceType;
```

定义空间类型枚举：

```
@Getter
public enum SpaceTypeEnum {

    PRIVATE("私有空间", 0),
    TEAM("团队空间", 1);

    private final String text;

    private final int value;

    SpaceTypeEnum(String text, int value) {
```

```
        this.text = text;
        this.value = value;
    }


    public static SpaceTypeEnum getEnumByValue(Integer value) {
        if (ObjUtil.isEmpty(value)) {
            return null;
        }
        for (SpaceTypeEnum spaceTypeEnum : SpaceTypeEnum.values()) {
            if (spaceTypeEnum.value == value) {
                return spaceTypeEnum;
            }
        }
        return null;
    }
}
```

2、新建团队空间

可以直接复用创建空间的方法，只需要做一些改动即可。

1）创建空间时为空间类型指定默认值：

```
if (StrUtil.isBlank(spaceAddRequest.getSpaceName())) {
    spaceAddRequest.setSpaceName("默认空间");
}
if (spaceAddRequest.getSpaceLevel() == null) {
    spaceAddRequest.setSpaceLevel(SpaceLevelEnum.COMMON.getValue());
}
if (spaceAddRequest.getSpaceType() == null) {
    spaceAddRequest.setSpaceType(SpaceTypeEnum.PRIVATE.getValue());
}

Space space = new Space();
BeanUtils.copyProperties(spaceAddRequest, space);

this.fillSpaceBySpaceLevel(space);
```

2）validSpace 方法补充对空间类型的校验：

```
public void validSpace(Space space, boolean add) {
    Integer spaceType = space.getSpaceType();
    SpaceTypeEnum spaceTypeEnum = SpaceTypeEnum.getEnumByValue(spaceT

    if (add) {
        if (spaceType == null) {
            throw new BusinessException(ErrorCode.PARAMS_ERROR, "空间
        }
    }

    if (spaceType != null && spaceTypeEnum == null) {
```

```
            throw new BusinessException(ErrorCode.PARAMS_ERROR, "空间类型
        }
    }
```

3）限制每个普通用户仅能创建一个团队空间（管理员可以创建多个），由于普通用户也仅能创建一个私有空间，相当于 ** 普通用户每类空间只能创建 1 个。** 因此，只要在判断是否已创建空间时，补充 spaceType 作为查询条件即可：

```
Long newSpaceId = transactionTemplate.execute(status -> {
    if (!userService.isAdmin(loginUser)) {
        boolean exists = this.lambdaQuery()
                .eq(Space::getUserId, userId)
                .eq(Space::getSpaceType, spaceAddRequest.getSpaceType
                .exists();
        ThrowUtils.throwIf(exists, ErrorCode.OPERATION_ERROR, "每个用
    }

    boolean result = this.save(space);
    ThrowUtils.throwIf(!result, ErrorCode.OPERATION_ERROR);

    return space.getId();
});
```

当然，这里的逻辑你可以自由调整，比如不允许用户创建团队空间，需要联系管理员或付费开通。

3、查询团队空间

给 SpaceService 的 getQueryWrapper 方法补充 spaceType 的查询条件：

```
Integer spaceType = spaceQueryRequest.getSpaceType();
queryWrapper.eq(ObjUtil.isNotEmpty(spaceType), "spaceType", spaceType
```

之后前端就能够按照空间类别获取空间列表了。

空间成员管理

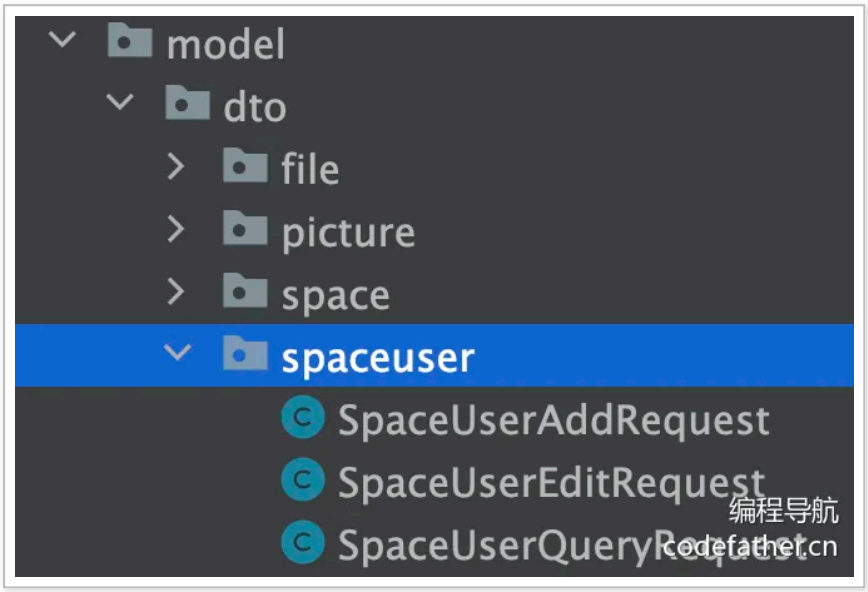
空间成员管理的开发比较简单，其实就是 “增删改查”。

1、数据模型

1）首先利用 MyBatisX 插件生成空间成员表相关的基础代码，包括实体类、Mapper、Service。

用户模块中有讲解详细流程，此处不再赘述。

2）每个操作都需要提供一个请求类，都放在 `model.dto.spaceuser` 包下。



添加空间成员请求，给空间管理员使用：

```
@Data
public class SpaceUserAddRequest implements Serializable {

    private Long spaceId;

    private Long userId;

    private String spaceRole;

    private static final long serialVersionUID = 1L;
}
```

编辑空间成员请求，给空间管理员使用，可以设置空间成员的角色：

```
@Data
public class SpaceUserEditRequest implements Serializable {

    private Long id;
```

```
    private String spaceRole;

    private static final long serialVersionUID = 1L;
}
```

查询空间成员请求，可以不用分页：

```
@Data
public class SpaceUserQueryRequest implements Serializable {


    private Long id;


    private Long spaceId;


    private Long userId;


    private String spaceRole;

    private static final long serialVersionUID = 1L;
}
```

3）在 `model.dto.vo` 下新建空间成员的视图包装类，可以额外关联空间信息和创建空间的用户信息。还可以编写 SpaceUser 实体类和该 VO 类的转换方法，便于后续快速传值。

```
@Data
public class SpaceUserVO implements Serializable {


    private Long id;


    private Long spaceId;


    private Long userId;


    private String spaceRole;


    private Date createTime;


    private Date updateTime;
```

```
    private UserVO user;


    private SpaceVO space;

    private static final long serialVersionUID = 1L;


    public static SpaceUser voToObj(SpaceUserVO spaceUserVO) {
        if (spaceUserVO == null) {
            return null;
        }
        SpaceUser spaceUser = new SpaceUser();
        BeanUtils.copyProperties(spaceUserVO, spaceUser);
        return spaceUser;
    }


    public static SpaceUserVO objToVo(SpaceUser spaceUser) {
        if (spaceUser == null) {
            return null;
        }
        SpaceUserVO spaceUserVO = new SpaceUserVO();
        BeanUtils.copyProperties(spaceUser, spaceUserVO);
        return spaceUserVO;
    }
}
```

4）在 `model.enums` 包下新建空间角色枚举：

```
@Getter
public enum SpaceRoleEnum {

    VIEWER("浏览者", "viewer"),
    EDITOR("编辑者", "editor"),
    ADMIN("管理员", "admin");

    private final String text;

    private final String value;

    SpaceRoleEnum(String text, String value) {
        this.text = text;
        this.value = value;
    }


    public static SpaceRoleEnum getEnumByValue(String value) {
        if (ObjUtil.isEmpty(value)) {
            return null;
        }
        for (SpaceRoleEnum anEnum : SpaceRoleEnum.values()) {
            if (anEnum.value.equals(value)) {
                return anEnum;
            }
        }
        return null;
    }
```

```
    public static List<String> getAllTexts() {
        return Arrays.stream(SpaceRoleEnum.values())
                .map(SpaceRoleEnum::getText)
                .collect(Collectors.toList());
    }


    public static List<String> getAllValues() {
        return Arrays.stream(SpaceRoleEnum.values())
                .map(SpaceRoleEnum::getValue)
                .collect(Collectors.toList());
    }
}
```

2、基础服务开发

可以参考图片服务的开发方法，完成 SpaceUserService 和实现类，大多数代码可以直接复用。

我们主要开发下列方法：

1）添加空间成员：

```
@Override
public long addSpaceUser(SpaceUserAddRequest spaceUserAddRequest) {

    ThrowUtils.throwIf(spaceUserAddRequest == null, ErrorCode.PARAMS_
    SpaceUser spaceUser = new SpaceUser();
    BeanUtils.copyProperties(spaceUserAddRequest, spaceUser);
    validSpaceUser(spaceUser, true);

    boolean result = this.save(spaceUser);
    ThrowUtils.throwIf(!result, ErrorCode.OPERATION_ERROR);
    return spaceUser.getId();
}
```

2）校验空间成员对象，增加 add 参数用来区分是创建数据时校验还是编辑时校验，判断条件是不一样的。比如创建成员时要检查用户是否存在。

```
@Override
public void validSpaceUser(SpaceUser spaceUser, boolean add) {
    ThrowUtils.throwIf(spaceUser == null, ErrorCode.PARAMS_ERROR);

    Long spaceId = spaceUser.getSpaceId();
    Long userId = spaceUser.getUserId();
    if (add) {
        ThrowUtils.throwIf(ObjectUtil.hasEmpty(spaceId, userId), Erro
        User user = userService.getById(userId);
        ThrowUtils.throwIf(user == null, ErrorCode.NOT_FOUND_ERROR, "
        Space space = spaceService.getById(spaceId);
```

```
        ThrowUtils.throwIf(space == null, ErrorCode.NOT_FOUND_ERROR,
    }

    String spaceRole = spaceUser.getSpaceRole();
    SpaceRoleEnum spaceRoleEnum = SpaceRoleEnum.getEnumByValue(spaceR
    if (spaceRole != null && spaceRoleEnum == null) {
        throw new BusinessException(ErrorCode.PARAMS_ERROR, "空间角色不
    }
}
```

还可以校验是否已添加该成员，可自行实现。

3）将查询请求对象转换为 MyBatis-Plus 的查询封装对象：

```
@Override
public QueryWrapper<SpaceUser> getQueryWrapper(SpaceUserQueryRequest
    QueryWrapper<SpaceUser> queryWrapper = new QueryWrapper<>();
    if (spaceUserQueryRequest == null) {
        return queryWrapper;
    }

    Long id = spaceUserQueryRequest.getId();
    Long spaceId = spaceUserQueryRequest.getSpaceId();
    Long userId = spaceUserQueryRequest.getUserId();
    String spaceRole = spaceUserQueryRequest.getSpaceRole();
    queryWrapper.eq(ObjUtil.isNotEmpty(id), "id", id);
    queryWrapper.eq(ObjUtil.isNotEmpty(spaceId), "spaceId", spaceId);
    queryWrapper.eq(ObjUtil.isNotEmpty(userId), "userId", userId);
    queryWrapper.eq(ObjUtil.isNotEmpty(spaceRole), "spaceRole", space
    return queryWrapper;
}
```

4）获取空间成员封装类，需要关联查询用户和空间的信息。

查询单个封装类：

```
@Override
public SpaceUserVO getSpaceUserVO(SpaceUser spaceUser, HttpServletReq

    SpaceUserVO spaceUserVO = SpaceUserVO.objToVo(spaceUser);

    Long userId = spaceUser.getUserId();
    if (userId != null && userId > 0) {
        User user = userService.getById(userId);
        UserVO userVO = userService.getUserVO(user);
        spaceUserVO.setUser(userVO);
    }

    Long spaceId = spaceUser.getSpaceId();
    if (spaceId != null && spaceId > 0) {
        Space space = spaceService.getById(spaceId);
        SpaceVO spaceVO = spaceService.getSpaceVO(space, request);
```

```
            spaceUserVO.setSpace(spaceVO);
        }
        return spaceUserVO;
    }
```

查询封装类列表：

```
    @Override
    public List<SpaceUserVO> getSpaceUserVOList(List<SpaceUser> spaceUser

        if (CollUtil.isEmpty(spaceUserList)) {
            return Collections.emptyList();
        }

        List<SpaceUserVO> spaceUserVOList = spaceUserList.stream().map(Sp

        Set<Long> userIdSet = spaceUserList.stream().map(SpaceUser::getUs
        Set<Long> spaceIdSet = spaceUserList.stream().map(SpaceUser::getS

        Map<Long, List<User>> userIdUserListMap = userService.listByIds(u
                .collect(Collectors.groupingBy(User::getId));
        Map<Long, List<Space>> spaceIdSpaceListMap = spaceService.listByI
                .collect(Collectors.groupingBy(Space::getId));

        spaceUserVOList.forEach(spaceUserVO -> {
            Long userId = spaceUserVO.getUserId();
            Long spaceId = spaceUserVO.getSpaceId();

            User user = null;
            if (userIdUserListMap.containsKey(userId)) {
                user = userIdUserListMap.get(userId).get(0);
            }
            spaceUserVO.setUser(userService.getUserVO(user));

            Space space = null;
            if (spaceIdSpaceListMap.containsKey(spaceId)) {
                space = spaceIdSpaceListMap.get(spaceId).get(0);
            }
            spaceUserVO.setSpace(SpaceVO.objToVo(space));
        });
        return spaceUserVOList;
    }
```

3、接口开发

参考图片接口的开发方法，完成 SpaceUserController 类，大多数代码可以直接复用。

需要开发的接口包括：

- 添加成员到空间：仅拥有成员管理权限的用户可使用。

- 从空间移除成员：仅拥有成员管理权限的用户可使用。
- 查询某个成员在空间的信息：仅拥有成员管理权限的用户可使用。
- 查询空间成员列表：仅拥有成员管理权限的用户可使用。
- 编辑成员信息：仅拥有成员管理权限的用户可使用。
- 查询我加入的团队空间列表：所有已登录用户可使用。

由于我们后续会使用统一的权限管理框架，这个阶段可以先只实现功能，不进行权限校验。

代码如下：

```
@RestController
@RequestMapping("/spaceUser")
@Slf4j
public class SpaceUserController {

    @Resource
    private SpaceUserService spaceUserService;

    @Resource
    private UserService userService;


    @PostMapping("/add")
    public BaseResponse<Long> addSpaceUser(@RequestBody SpaceUserAddR
        ThrowUtils.throwIf(spaceUserAddRequest == null, ErrorCode.PAR
        long id = spaceUserService.addSpaceUser(spaceUserAddRequest);
        return ResultUtils.success(id);
    }


    @PostMapping("/delete")
    public BaseResponse<Boolean> deleteSpaceUser(@RequestBody DeleteR
                                                 HttpServletRequest r
        if (deleteRequest == null || deleteRequest.getId() <= 0) {
            throw new BusinessException(ErrorCode.PARAMS_ERROR);
        }
        long id = deleteRequest.getId();

        SpaceUser oldSpaceUser = spaceUserService.getById(id);
        ThrowUtils.throwIf(oldSpaceUser == null, ErrorCode.NOT_FOUND_

        boolean result = spaceUserService.removeById(id);
        ThrowUtils.throwIf(!result, ErrorCode.OPERATION_ERROR);
        return ResultUtils.success(true);
    }


    @PostMapping("/get")
    public BaseResponse<SpaceUser> getSpaceUser(@RequestBody SpaceUse
```

```
        ThrowUtils.throwIf(spaceUserQueryRequest == null, ErrorCode.P
        Long spaceId = spaceUserQueryRequest.getSpaceId();
        Long userId = spaceUserQueryRequest.getUserId();
        ThrowUtils.throwIf(ObjectUtil.hasEmpty(spaceId, userId), Erro

        SpaceUser spaceUser = spaceUserService.getOne(spaceUserServic
        ThrowUtils.throwIf(spaceUser == null, ErrorCode.NOT_FOUND_ERR
        return ResultUtils.success(spaceUser);
    }


    @PostMapping("/list")
    public BaseResponse<List<SpaceUserVO>> listSpaceUser(@RequestBody
                                                         HttpServletR
        ThrowUtils.throwIf(spaceUserQueryRequest == null, ErrorCode.P
        List<SpaceUser> spaceUserList = spaceUserService.list(
                spaceUserService.getQueryWrapper(spaceUserQueryReques
        );
        return ResultUtils.success(spaceUserService.getSpaceUserVOLis
    }


    @PostMapping("/edit")
    public BaseResponse<Boolean> editSpaceUser(@RequestBody SpaceUser
                                               HttpServletRequest req
        if (spaceUserEditRequest == null || spaceUserEditRequest.getI
            throw new BusinessException(ErrorCode.PARAMS_ERROR);
        }

        SpaceUser spaceUser = new SpaceUser();
        BeanUtils.copyProperties(spaceUserEditRequest, spaceUser);

        spaceUserService.validSpaceUser(spaceUser, false);

        long id = spaceUserEditRequest.getId();
        SpaceUser oldSpaceUser = spaceUserService.getById(id);
        ThrowUtils.throwIf(oldSpaceUser == null, ErrorCode.NOT_FOUND_

        boolean result = spaceUserService.updateById(spaceUser);
        ThrowUtils.throwIf(!result, ErrorCode.OPERATION_ERROR);
        return ResultUtils.success(true);
    }


    @PostMapping("/list/my")
    public BaseResponse<List<SpaceUserVO>> listMyTeamSpace(HttpServle
        User loginUser = userService.getLoginUser(request);
        SpaceUserQueryRequest spaceUserQueryRequest = new SpaceUserQu
        spaceUserQueryRequest.setUserId(loginUser.getId());
        List<SpaceUser> spaceUserList = spaceUserService.list(
                spaceUserService.getQueryWrapper(spaceUserQueryReques
        );
        return ResultUtils.success(spaceUserService.getSpaceUserVOLis
    }
}
```

4、创建团队空间时自动新增成员记录

根据需求，用户在创建团队空间时，会默认作为空间的管理员，需要在空间成员表中新增一条记录。

修改 addSpace 方法，在事务中补充插入空间成员记录：

```
boolean result = this.save(space);
ThrowUtils.throwIf(!result, ErrorCode.OPERATION_ERROR);

if (SpaceTypeEnum.TEAM.getValue() == spaceAddRequest.getSpaceType())
    SpaceUser spaceUser = new SpaceUser();
    spaceUser.setSpaceId(space.getId());
    spaceUser.setUserId(userId);
    spaceUser.setSpaceRole(SpaceRoleEnum.ADMIN.getValue());
    result = spaceUserService.save(spaceUser);
    ThrowUtils.throwIf(!result, ErrorCode.OPERATION_ERROR, "创建团队成
}

return space.getId();
```

扩展

1）添加成员到空间时，可以支持发送邀请和审批。

实现思路：给空间成员表新增一个邀请确认状态的字段

2）由于空间管理员可能有多个，空间成员表可以补充添加成员至空间的邀请人字段（createUserId）

3）空间成员操作执行前可以补充一些校验，比如：

- 只有已经是空间成员，才能被移除或编辑
- 如果编辑后的角色跟之前一致，就不用更新

空间成员权限控制

引入团队空间后，需要给空间操作、图片操作、空间成员操作添加权限控制逻辑。为了简化开发，同时防止一些空间重要信息的修改冲突，空间操作（空间信息的增删改查）仍然复用之前私有空间的校验逻辑 —— 仅创建人可操作。

由于权限校验属于整个项目的公共服务，统一放在 `manager.auth` 包中。

1、权限定义

根据 RBAC 权限模型，需要定义角色和权限。

1）此处选用 JSON 配置文件来定义角色、权限、角色和权限之间的关系，相比从数据库表中获取，实现更方便，查询也更高效。

在 `resources/biz` 目录下新建 JSON 配置文件 `spaceUserAuthConfig.json` ：

```
{
  "permissions": [
    {
      "key": "spaceUser:manage",
      "name": "成员管理",
      "description": "管理空间成员，添加或移除成员"
    },
    {
      "key": "picture:view",
      "name": "查看图片",
      "description": "查看空间中的图片内容"
    },
    {
      "key": "picture:upload",
      "name": "上传图片",
      "description": "上传图片到空间中"
    },
    {
      "key": "picture:edit",
      "name": "修改图片",
      "description": "编辑已上传的图片信息"
    },
    {
      "key": "picture:delete",
      "name": "删除图片",
      "description": "删除空间中的图片"
    }
  ],
  "roles": [
    {
      "key": "viewer",
      "name": "浏览者",
      "permissions": [
        "picture:view"
      ],
      "description": "查看图片"
    },
    {
      "key": "editor",
      "name": "编辑者",
      "permissions": [
        "picture:view",
        "picture:upload",
        "picture:edit",
        "picture:delete"
      ],
      "description": "查看图片、上传图片、修改图片、删除图片"
    },
    {
```

```
      "key": "admin",
      "name": "管理员",
      "permissions": [
        "spaceUser:manage",
        "picture:view",
        "picture:upload",
        "picture:edit",
        "picture:delete"
      ],
      "description": "成员管理、查看图片、上传图片、修改图片、删除图片"
    }
  ]
}
```

2）在 `auth.model` 包下新建数据模型，用于接收配置文件的值。

权限配置类：

```
@Data
public class SpaceUserAuthConfig implements Serializable {


    private List<SpaceUserPermission> permissions;


    private List<SpaceUserRole> roles;

    private static final long serialVersionUID = 1L;
}
```

空间成员权限：

```
@Data
public class SpaceUserPermission implements Serializable {


    private String key;


    private String name;


    private String description;

    private static final long serialVersionUID = 1L;

}
```

空间成员角色：

```
@Data
public class SpaceUserRole implements Serializable {


    private String key;


    private String name;


    private List<String> permissions;


    private String description;

    private static final long serialVersionUID = 1L;
}
```

3）定义空间成员权限常量类，便于后续校验权限时使用：

```
public interface SpaceUserPermissionConstant {

    String SPACE_USER_MANAGE = "spaceUser:manage";


    String PICTURE_VIEW = "picture:view";


    String PICTURE_UPLOAD = "picture:upload";


    String PICTURE_EDIT = "picture:edit";


    String PICTURE_DELETE = "picture:delete";
}
```

4）在 `auth` 包下新建 SpaceUserAuthManager，可加载配置文件到对象，并提供根据角色获取权限列表的方法。

```
@Component
public class SpaceUserAuthManager {

    @Resource
    private SpaceUserService spaceUserService;

    @Resource
    private UserService userService;

    public static final SpaceUserAuthConfig SPACE_USER_AUTH_CONFIG;

    static {
        String json = ResourceUtil.readUtf8Str("biz/spaceUserAuthConf
        SPACE_USER_AUTH_CONFIG = JSONUtil.toBean(json, SpaceUserAuthC
    }
```

```
    public List<String> getPermissionsByRole(String spaceUserRole) {
        if (StrUtil.isBlank(spaceUserRole)) {
            return new ArrayList<>();
        }

        SpaceUserRole role = SPACE_USER_AUTH_CONFIG.getRoles().stream
                .filter(r -> spaceUserRole.equals(r.getKey()))
                .findFirst()
                .orElse(null);
        if (role == null) {
            return new ArrayList<>();
        }
        return role.getPermissions();
    }
}
```

2、Sa-Token 入门

Sa-Token 是一个轻量级 Java 权限认证框架，相比 Spring Security 等更加简单易学，用作者的话说，使用该框架可以让鉴权变得简单、优雅~

框架的学习并不难，参考 官方文档 就好，等下我们要学习实战 Sa-Token 的主流特性和高级用法。

1）引入 Sa-Token：

```
<dependency>
    <groupId>cn.dev33</groupId>
    <artifactId>sa-token-spring-boot-starter</artifactId>
    <version>1.39.0</version>
</dependency>
```

Sa-Token 默认将数据（比如用户登录态）保存在内存中，此模式读写速度最快，且避免了序列化与反序列化带来的性能消耗，但缺点是重启后数据会丢失、无法在分布式环境中共享数据。

我们项目中既然已经使用了 Redis，那么可以 参考官方文档 让 Sa-Token 整合 Redis，将用户的登录态等内容保存在 Redis 中。

此处选择 jackson 序列化方式整合 Redis，这样存到 Redis 的数据是可读的：

```
<dependency>
    <groupId>cn.dev33</groupId>
    <artifactId>sa-token-redis-jackson</artifactId>
    <version>1.39.0</version>
</dependency>

<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-pool2</artifactId>
</dependency>
```

2）了解 Sa-Token 的基本用法

Sa-Token 的使用方式比较简单，首先是用户登录时调用 login 方法，产生一个新的会话：

```
StpUtil.login(10001);
```

还可以给会话保存一些信息，比如登录用户的信息：

```
StpUtil.getSession().set("user", user)
```

接下来你就可以判断用户是否登录、获取用户信息了，可以通过代码进行判断：

```
StpUtil.checkLogin();

StpUtil.getSession().get("user");
```

也可以参考 官方文档 ，使用注解进行鉴权：

```
@SaCheckLogin
@RequestMapping("info")
public String info() {
    return "查询用户信息";
}
```

这是 Sa-Token 最基本的用法，下面我们正式在项目中使用 Sa-Token。

3、新建空间账号体系

目前，我们的项目中其实存在两套权限校验体系。一套是最开始就有的，对 user 表的角色进行校验，分为普通用户和管理员；另一套是本节新学习的，对团队空间的权限进行校验。

为了更轻松地扩展项目，减少对原有代码的改动，我们原有的 user 表权限校验依然使用自定义注解 + AOP 的方式实现。而团队空间权限校验，采用 Sa-Token 来管理。

相当于我们不是整个项目都交给 Sa-Token，只是把 Sa-Token 当做实现团队空间权限管理的工具罢了。

这种同一项目有多账号体系的情况下，不建议使用 Sa-Token 默认的账号体系，而是使用 Sa-Token 提供的 多账号认证特性 ，可以将多套账号的授权给区分开，让它们互不干扰。

1）可以参考官方文档，使用 Kit 模式 实现多账号认证，在 `auth` 包下新建 `StpKit.java` ，定义空间账号体系：

```
@Component
public class StpKit {

    public static final String SPACE_TYPE = "space";

    public static final StpLogic DEFAULT = StpUtil.stpLogic;

    public static final StpLogic SPACE = new StpLogic(SPACE_TYPE);
}
```

之后就可以在代码中使用账号体系，以下是示例代码：

```
StpKit.SPACE.login(10001);

StpKit.SPACE.checkPermission("picture:edit");

StpKit.SPACE.getSession().set("user", "程序员鱼皮");
```

2）修改用户服务的 userLogin 方法，用户登录成功后，保存登录态到 Sa-Token 的空间账号体系中：

```
request.getSession().setAttribute(USER_LOGIN_STATE, user);

StpKit.SPACE.login(user.getId());
StpKit.SPACE.getSession().set(USER_LOGIN_STATE, user);
return this.getLoginUserVO(user);
```

4、权限认证逻辑

Sa-Token 开发的核心是编写权限认证类，我们需要在该类中实现 “如何根据登录用户 id 获取到用户已有的角色和权限列表” 方法。当要判断某用户是否有某个角色或权限时，Sa-Token 会先执行我们编写的方法，得到该用户的角色或权限列表，然后跟需要的角色权限进行比对。

参考 官方文档 ，示例权限认证类如下：

```
@Component
public class StpInterfaceImpl implements StpInterface {


    @Override
    public List<String> getPermissionList(Object loginId, String logi

        List<String> list = new ArrayList<String>();
        list.add("user.add");
        list.add("user.update");
        list.add("user.get");
        list.add("art.*");
        return list;
    }


    @Override
    public List<String> getRoleList(Object loginId, String loginType)

        List<String> list = new ArrayList<String>();
        list.add("admin");
        list.add("super-admin");
        return list;
    }
}
```

Sa-Token 支持按照角色和权限校验，对于权限不多的项目，基于角色校验即可；对于权限较多的项目，建议根据权限校验。对于本项目，虽然权限并不多，但是考虑到扩展性，还是 选择更细粒度的权限校验，业务含义会更明确。

观察上述代码我们会发现，`getPermissionList` 方法只提供了 loginId（登录用户 id）和 loginType（账号体系）两个参数。这会给我们造成很大的难度：

- 我们光有用户 id 是没办法进行权限校验的，因为我们要给图片操作和空间成员操作增加权限校验逻辑，还需要获取到空间 id，才知道用户是否具有某个团队空间的权限。**那么如何获取到空间 id 呢？**
- 如果要进行统一的权限校验，也包括了公共图库和私有空间，更要命的是，公共图库是没有空间 id 的！**这就意味着要根据操作的图片情况动态判断。**

所以我们要解决的关键问题有 2 个：

1. 如何在 Sa-Token 中获取当前请求操作的参数？
2. 如何编写一套权限校验逻辑，同时兼容公共图库、私有空间和团队空间？

1）先看第一个问题，使用 Sa-Token 有 2 种方式 —— 注解式和编程式。

如果使用注解式，那么在接口被调用时就会立刻触发 Sa-Token 的权限校验，此时参数只能通过 Servlet 的请求对象传递。

如果使用编程式，可以在代码任意位置执行权限校验，只要在执行前将参数放到当前线程的上下文 ThreadLocal 对象中，就能在鉴权时获取到了。

为了后续我们给接口添加鉴权更直观方便，我们选择注解式鉴权。那就有一个关键问题，不同接口的请求参数是不同的，有的请求参数有 spaceId、有的只有 pictureId，怎么办呢？

我们可以定义一个 **上下文类**，用于统一接收请求中传递来的参数：

```
@Data
public class SpaceUserAuthContext {
```

```
    private Long id;

    private Long pictureId;

    private Long spaceId;

    private Long spaceUserId;

    private Picture picture;

    private Space space;

    private SpaceUser spaceUser;
}
```

如何知道哪个请求包含了哪些字段呢？别忘了，我们每类操作（图片 / 空间成员）的请求前缀都是固定的，可以从请求路径中提取到要访问的是哪个 Controller，而每类 Controller 的请求参数，都是一致的。

举个例子，如果访问地址是 `/api/picture/xxx` ，那么一定是要调用 PictureController 的接口，这些接口的 id 字段都表示 pictureId。我们就可以通过访问地址来决定应该给上下文传递哪些字段，代码如下：

```
@Value("${server.servlet.context-path}")
private String contextPath;


private SpaceUserAuthContext getAuthContextByRequest() {
    HttpServletRequest request = ((ServletRequestAttributes) RequestC
    String contentType = request.getHeader(Header.CONTENT_TYPE.getVal
    SpaceUserAuthContext authRequest;

    if (ContentType.JSON.getValue().equals(contentType)) {
        String body = ServletUtil.getBody(request);
        authRequest = JSONUtil.toBean(body, SpaceUserAuthContext.clas
    } else {
        Map<String, String> paramMap = ServletUtil.getParamMap(reques
        authRequest = BeanUtil.toBean(paramMap, SpaceUserAuthContext.
    }

    Long id = authRequest.getId();
    if (ObjUtil.isNotNull(id)) {
        String requestUri = request.getRequestURI();
        String partUri = requestUri.replace(contextPath + "/", "");
        String moduleName = StrUtil.subBefore(partUri, "/", false);
        switch (moduleName) {
```

```
            case "picture":
                authRequest.setPictureId(id);
                break;
            case "spaceUser":
                authRequest.setSpaceUserId(id);
                break;
            case "space":
                authRequest.setSpaceId(id);
                break;
            default:
        }
    }
    return authRequest;
}
```

注意，上述代码中，我们使用 Hutool 的工具类 `ServletUtil` 从 HttpServletRequest 中获取到了参数信息，但是坑爹的是，HttpServletRequest 的 body 值是个流，** 只支持读取一次，读完就没了！** 所以为了解决这个问题，我们还要在 `config` 包下自定义请求包装类和请求包装类过滤器。这些就是样板代码了，大家直接复制粘贴即可，不用编码。

RequestWrapper 请求包装类：

```
@Slf4j
public class RequestWrapper extends HttpServletRequestWrapper {

    private final String body;

    public RequestWrapper(HttpServletRequest request) {
        super(request);
        StringBuilder stringBuilder = new StringBuilder();
        try (InputStream inputStream = request.getInputStream(); Buff
            char[] charBuffer = new char[128];
            int bytesRead = -1;
            while ((bytesRead = bufferedReader.read(charBuffer)) > 0)
                stringBuilder.append(charBuffer, 0, bytesRead);
            }
        } catch (IOException ignored) {
        }
        body = stringBuilder.toString();
    }

    @Override
    public ServletInputStream getInputStream() throws IOException {
        final ByteArrayInputStream byteArrayInputStream = new ByteArr
        return new ServletInputStream() {
            @Override
            public boolean isFinished() {
                return false;
            }

            @Override
            public boolean isReady() {
```

```
                return false;
            }

            @Override
            public void setReadListener(ReadListener readListener) {
            }

            @Override
            public int read() throws IOException {
                return byteArrayInputStream.read();
            }
        };

    }

    @Override
    public BufferedReader getReader() throws IOException {
        return new BufferedReader(new InputStreamReader(this.getInput
    }

    public String getBody() {
        return this.body;
    }

}
```

HttpRequestWrapperFilter 请求包装过滤器：

```
@Order(1)
@Component
public class HttpRequestWrapperFilter implements Filter {

    @Override
    public void doFilter(ServletRequest request, ServletResponse resp
        if (request instanceof HttpServletRequest) {
            HttpServletRequest servletRequest = (HttpServletRequest)
            String contentType = servletRequest.getHeader(Header.CONT
            if (ContentType.JSON.getValue().equals(contentType)) {

                chain.doFilter(new RequestWrapper(servletRequest), re
            } else {
                chain.doFilter(request, response);
            }
        }
    }

}
```

这样我们就能正常获取到请求参数了~

2）编写通用的权限校验逻辑，兼容公共图库、私有空间和团队空间

这个没啥好说的，就是写业务逻辑，而且是比较复杂的业务逻辑，所以建议一定要先把业务流程梳理清楚，再编写代码。

业务流程如下：

1. 校验登录类型：如果 `loginType` 不是 `"space"` ，直接返回空权限列表。
2. 管理员权限处理：如果当前用户为管理员，直接返回管理员权限列表。
3. 获取上下文对象：从请求中获取 `SpaceUserAuthContext` 上下文，检查上下文字段是否为空。**如果上下文中所有字段均为空（如没有空间或图片信息），视为公共图库操作，直接返回管理员权限列表。**
4. 校验登录状态：通过 `loginId` 获取当前登录用户信息。如果用户未登录，抛出未授权异常；否则获取用户的唯一标识 `userId` ，用于后续权限判断。
5. 从上下文中优先获取 `SpaceUser` 对象：如果上下文中存在 `SpaceUser` 对象，直接根据其角色获取权限码列表。
6. 通过 `spaceUserId` 获取空间用户信息：如果上下文中存在 `spaceUserId` ：

- 查询对应的 `SpaceUser` 数据。如果未找到，抛出数据未找到异常。
- 校验当前登录用户是否属于该空间，如果不是，返回空权限列表。
- 否则，根据登录用户在该空间的角色，返回相应的权限码列表。

7. 通过 `spaceId` 或 `pictureId` 获取空间或图片信息

- 如果 `spaceId` 不存在：使用 `pictureId` 查询图片信息，并通过图片的 `spaceId` 继续判断权限；如果 `pictureId` 和 `spaceId` 均为空，默认视为管理员权限。

- 对于公共图库：**如果图片是当前用户上传的，或者当前用户为管理员**，返回管理员权限列表；如果图片不是当前用户上传的，返回仅允许查看的权限码。

8. 获取 `Space` 对象并判断空间类型：查询 `Space` 信息，如果未找到空间数据，抛出数据未找到异常。否则根据空间类型进行判断

- 私有空间：仅空间所有者和管理员有权限（即返回全部权限），其他用户返回空权限列表。
- 团队空间：查询登录用户在该空间的角色，并返回对应的权限码列表。如果用户不属于该空间，返回空权限列表。

根据业务流程编写代码：

```
public List<String> getPermissionList(Object loginId, String loginTyp

    if (!StpKit.SPACE_TYPE.equals(loginType)) {
        return new ArrayList<>();
    }

    List<String> ADMIN_PERMISSIONS = spaceUserAuthManager.getPermissi

    SpaceUserAuthContext authContext = getAuthContextByRequest();

    if (isAllFieldsNull(authContext)) {
        return ADMIN_PERMISSIONS;
    }

    User loginUser = (User) StpKit.SPACE.getSessionByLoginId(loginId)
    if (loginUser == null) {
        throw new BusinessException(ErrorCode.NO_AUTH_ERROR, "用户未登
    }
    Long userId = loginUser.getId();

    SpaceUser spaceUser = authContext.getSpaceUser();
    if (spaceUser != null) {
        return spaceUserAuthManager.getPermissionsByRole(spaceUser.ge
    }

    Long spaceUserId = authContext.getSpaceUserId();
    if (spaceUserId != null) {
        spaceUser = spaceUserService.getById(spaceUserId);
        if (spaceUser == null) {
            throw new BusinessException(ErrorCode.NOT_FOUND_ERROR, "未
        }

        SpaceUser loginSpaceUser = spaceUserService.lambdaQuery()
                .eq(SpaceUser::getSpaceId, spaceUser.getSpaceId())
                .eq(SpaceUser::getUserId, userId)
                .one();
        if (loginSpaceUser == null) {
            return new ArrayList<>();
```

```
        }

        return spaceUserAuthManager.getPermissionsByRole(loginSpaceUs
    }

    Long spaceId = authContext.getSpaceId();
    if (spaceId == null) {

        Long pictureId = authContext.getPictureId();

        if (pictureId == null) {
            return ADMIN_PERMISSIONS;
        }
        Picture picture = pictureService.lambdaQuery()
                .eq(Picture::getId, pictureId)
                .select(Picture::getId, Picture::getSpaceId, Picture:
                .one();
        if (picture == null) {
            throw new BusinessException(ErrorCode.NOT_FOUND_ERROR, "未
        }
        spaceId = picture.getSpaceId();

        if (spaceId == null) {
            if (picture.getUserId().equals(userId) || userService.isA
                return ADMIN_PERMISSIONS;
            } else {

                return Collections.singletonList(SpaceUserPermissionC
            }
        }
    }

    Space space = spaceService.getById(spaceId);
    if (space == null) {
        throw new BusinessException(ErrorCode.NOT_FOUND_ERROR, "未找到
    }

    if (space.getSpaceType() == SpaceTypeEnum.PRIVATE.getValue()) {

        if (space.getUserId().equals(userId) || userService.isAdmin(l
            return ADMIN_PERMISSIONS;
        } else {
            return new ArrayList<>();
        }
    } else {

        spaceUser = spaceUserService.lambdaQuery()
                .eq(SpaceUser::getSpaceId, spaceId)
                .eq(SpaceUser::getUserId, userId)
                .one();
        if (spaceUser == null) {
            return new ArrayList<>();
        }
        return spaceUserAuthManager.getPermissionsByRole(spaceUser.ge
    }
}
```

上述代码依赖 “判断所有字段都为空” 的方法，通过反射获取对象的所有字段，进行判空：

```
private boolean isAllFieldsNull(Object object) {
    if (object == null) {
        return true;
    }

    return Arrays.stream(ReflectUtil.getFields(object.getClass()))

            .map(field -> ReflectUtil.getFieldValue(object, field))

            .allMatch(ObjectUtil::isEmpty);
}
```

OK，这就是 Sa-Token 动态权限校验的核心代码，你会发现编写一套统一的权限校验逻辑并不容易，所以实际项目中要**按需使用** 第三方权限校验框架。

💡 注意，采用注解式鉴权 + 通过请求对象获取参数时，可能会重复查询数据库。比如业务代码中已经有根据 id 查询空间信息的代码了，但为了权限校验，也查库获取了一次空间信息，会对性能造成影响。如果想更灵活、更高性能地实现鉴权，可以考虑使用编程式鉴权。获取权限的方法和上下文类都是可以复用的，只需要将 `getAuthContextByRequest` 方法的逻辑改为从 ThreadLocal 上下文中获取即可。

基于 ThreadLocal 实现上下文管理的示例代码：

```
public class SaTokenContextHolder {

    private static final ThreadLocal<Map<String, Object>> CONTEXT = T


    public static void set(String key, Object value) {
        CONTEXT.get().put(key, value);
    }


    public static Object get(String key) {
        return CONTEXT.get().get(key);
    }


    public static void clear() {
        CONTEXT.remove();
    }
}
```

5、权限校验注解

默认情况下使用 注解式鉴权 ，需要新建配置类：

1、注册拦截器

以 `SpringBoot2.0` 为例，新建配置类 `SaTokenConfigure.java`

```java
@Configuration
public class SaTokenConfigure implements WebMvcConfigurer {
    // 注册 Sa-Token 拦截器，打开注解式鉴权功能
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        // 注册 Sa-Token 拦截器，打开注解式鉴权功能
        registry.addInterceptor(new SaInterceptor()).addPathPatterns("/**");
    }
}
```


但由于我们使用了多账号体系，每次使用注解时都要指定账号体系的 loginType，会比较麻烦：

```
@SaCheckLogin(type = StpUserUtil.TYPE)
```

所以可以参考官方文档，使用 注解合并 简化代码。在 `auth.annotation` 包下新建 Sa-Token 配置类，开启注解鉴权和注解合并：

```
@Configuration
public class SaTokenConfigure implements WebMvcConfigurer {


    @Override
    public void addInterceptors(InterceptorRegistry registry) {

        registry.addInterceptor(new SaInterceptor()).addPathPatterns(
    }

    @PostConstruct
    public void rewriteSaStrategy() {

        SaAnnotationStrategy.instance.getAnnotation = (element, annot
            return AnnotatedElementUtils.getMergedAnnotation(element,
        };
    }
}
```

然后参考 官方提供的示例代码 ，在 `auth.annotation` 包下新建空间账号体系的鉴权注解：

```
@SaCheckPermission(type = StpKit.SPACE_TYPE)
@Retention(RetentionPolicy.RUNTIME)
```

```
@Target({ElementType.METHOD, ElementType.TYPE})
public @interface SaSpaceCheckPermission {


    @AliasFor(annotation = SaCheckPermission.class)
    String[] value() default {};


    @AliasFor(annotation = SaCheckPermission.class)
    SaMode mode() default SaMode.AND;


    @AliasFor(annotation = SaCheckPermission.class)
    String[] orRole() default {};

}
```

之后就可以直接使用该注解了。

6、应用权限注解

认真核对一遍各个操作接口的代码、以及接口调用的 Service 代码，包括图片操作 PictureController 和 PictureService、空间成员操作 SpaceUserController 和 SpaceUserService。

1）给 Controller 接口补充上合适的权限注解，PictureController 图片接口：

```
@PostMapping("/upload")
@SaSpaceCheckPermission(value = SpaceUserPermissionConstant.PICTURE_U
public BaseResponse<PictureVO> uploadPicture() {
}


@PostMapping("/upload/url")
@SaSpaceCheckPermission(value = SpaceUserPermissionConstant.PICTURE_U
public BaseResponse<PictureVO> uploadPictureByUrl() {
}


@PostMapping("/delete")
@SaSpaceCheckPermission(value = SpaceUserPermissionConstant.PICTURE_D
public BaseResponse<Boolean> deletePicture() {
}


@PostMapping("/edit")
@SaSpaceCheckPermission(value = SpaceUserPermissionConstant.PICTURE_E
public BaseResponse<Boolean> editPicture() {
}


@PostMapping("/search/color")
@SaSpaceCheckPermission(value = SpaceUserPermissionConstant.PICTURE_V
public BaseResponse<List<PictureVO>> searchPictureByColor() {
}
```

```
@PostMapping("/edit/batch")
@SaSpaceCheckPermission(value = SpaceUserPermissionConstant.PICTURE_E
public BaseResponse<Boolean> editPictureByBatch() {
}


@PostMapping("/out_painting/create_task")
@SaSpaceCheckPermission(value = SpaceUserPermissionConstant.PICTURE_E
public BaseResponse<CreateOutPaintingTaskResponse> createPictureOutPa
}
```

SpaceUserController 接口：

```
@PostMapping("/add")
@SaSpaceCheckPermission(value = SpaceUserPermissionConstant.SPACE_USE
public BaseResponse<Long> addSpaceUser() {
}


@PostMapping("/delete")
@SaSpaceCheckPermission(value = SpaceUserPermissionConstant.SPACE_USE
public BaseResponse<Boolean> deleteSpaceUser() {
}


@PostMapping("/get")
@SaSpaceCheckPermission(value = SpaceUserPermissionConstant.SPACE_USE
public BaseResponse<SpaceUser> getSpaceUser() {
}


@PostMapping("/list")
@SaSpaceCheckPermission(value = SpaceUserPermissionConstant.SPACE_USE
public BaseResponse<List<SpaceUserVO>> listSpaceUser() {
}


@PostMapping("/edit")
@SaSpaceCheckPermission(value = SpaceUserPermissionConstant.SPACE_USE
public BaseResponse<Boolean> editSpaceUser() {
}
```

2）移除这些接口和相关服务原本的权限校验逻辑，比如 `PictureService#checkPictureAuth` ，确保该方法变成了灰色（未被使用）。

还有 PictureServiceImpl 的 uploadPicture 方法中的权限校验，也要注释掉：

3）注意，只要加上了 Sa-Token 注解，框架就会强制要求用户登录，未登录会抛出异常。所以针对未登录也可以调用的接口，需要改为编程式权限校验，比如 getPictureVOById 和 listPictureVOByPage 方法。

```
@GetMapping("/get/vo")
public BaseResponse<PictureVO> getPictureVOById(long id, HttpServletR
    ThrowUtils.throwIf(id <= 0, ErrorCode.PARAMS_ERROR);

    Picture picture = pictureService.getById(id);
    ThrowUtils.throwIf(picture == null, ErrorCode.NOT_FOUND_ERROR);

    Space space = null;
    Long spaceId = picture.getSpaceId();
    if (spaceId != null) {
        boolean hasPermission = StpKit.SPACE.hasPermission(SpaceUserP
        ThrowUtils.throwIf(!hasPermission, ErrorCode.NO_AUTH_ERROR);
    }
    PictureVO pictureVO = pictureService.getPictureVO(picture, reques

    return ResultUtils.success(pictureVO);
}

@PostMapping("/list/page/vo")
public BaseResponse<Page<PictureVO>> listPictureVOByPage(@RequestBody
    long current = pictureQueryRequest.getCurrent();
    long size = pictureQueryRequest.getPageSize();

    ThrowUtils.throwIf(size > 20, ErrorCode.PARAMS_ERROR);

    Long spaceId = pictureQueryRequest.getSpaceId();

    if (spaceId == null) {

        pictureQueryRequest.setReviewStatus(PictureReviewStatusEnum.P
        pictureQueryRequest.setNullSpaceId(true);
    } else {
        boolean hasPermission = StpKit.SPACE.hasPermission(SpaceUserP
        ThrowUtils.throwIf(!hasPermission, ErrorCode.NO_AUTH_ERROR);
    }

    Page<Picture> picturePage = pictureService.page(new Page<>(curren

    return ResultUtils.success(pictureService.getPictureVOPage(pictur
}
```

7、全局异常处理

如果 Sa-Token 校验用户没有符合要求的权限、或者用户未登录，就会抛出它定义的异常， 参考文档 。

需要将框架的异常全局处理为我们自己定义的业务异常，在全局异常处理器中添加代码：

```
@ExceptionHandler(NotLoginException.class)
public BaseResponse<?> notLoginException(NotLoginException e) {
    log.error("NotLoginException", e);
    return ResultUtils.error(ErrorCode.NOT_LOGIN_ERROR, e.getMessage(
}

@ExceptionHandler(NotPermissionException.class)
public BaseResponse<?> notPermissionExceptionHandler(NotPermissionExc
    log.error("NotPermissionException", e);
    return ResultUtils.error(ErrorCode.NO_AUTH_ERROR, e.getMessage())
}
```

8、补充获取权限的接口

前面写的都是后端权限校验的代码，但对于用户来说，如果没有空间图片的编辑权限，进入空间详情页时不应该能看到编辑按钮。也就是说，前端也需要根据用户的权限来进行一些页面内容的展示和隐藏。

因此，后端需要将用户具有的权限返回给前端，帮助前端进行判断，这样就不用让前端编写复杂的角色和权限校验逻辑了。

思考下具体的使用场景：如果是团队空间（空间详情页）或团队空间的图片（图片详情页），返回给前端用户具有的权限（比如能否编辑、能否上传、能否删除、能否管理成员）。

1）比起新写一个获取权限的接口，我们可以直接在返回图片或空间详情时，额外传递权限列表。给 SpaceVO 和 PictureVO 新增权限列表字段：

```
private List<String> permissionList = new ArrayList<>();
```

2）在 SpaceUserAuthManager 中新增获取权限列表的方法，注意要区分公共图库、私有空间和团队空间，对于有权限的情况，可以返回 “管理员权限” 列表。

```
public List<String> getPermissionList(Space space, User loginUser) {
    if (loginUser == null) {
        return new ArrayList<>();
    }

    List<String> ADMIN_PERMISSIONS = getPermissionsByRole(SpaceRoleEn

    if (space == null) {
        if (userService.isAdmin(loginUser)) {
            return ADMIN_PERMISSIONS;
        }
        return new ArrayList<>();
    }
    SpaceTypeEnum spaceTypeEnum = SpaceTypeEnum.getEnumByValue(space.
    if (spaceTypeEnum == null) {
        return new ArrayList<>();
    }

    switch (spaceTypeEnum) {
        case PRIVATE:

            if (space.getUserId().equals(loginUser.getId()) || userSe
                return ADMIN_PERMISSIONS;
            } else {
                return new ArrayList<>();
            }
        case TEAM:

            SpaceUser spaceUser = spaceUserService.lambdaQuery()
                    .eq(SpaceUser::getSpaceId, space.getId())
                    .eq(SpaceUser::getUserId, loginUser.getId())
                    .one();
            if (spaceUser == null) {
                return new ArrayList<>();
            } else {
                return getPermissionsByRole(spaceUser.getSpaceRole())
            }
    }
    return new ArrayList<>();
}
```

3）修改获取空间详情和图片详情的接口 getSpaceVOById、getPictureVOById，增加获取权限列表的逻辑。

获取空间详情接口：

```
@GetMapping("/get/vo")
public BaseResponse<SpaceVO> getSpaceVOById(long id, HttpServletReque
    ThrowUtils.throwIf(id <= 0, ErrorCode.PARAMS_ERROR);

    Space space = spaceService.getById(id);
    ThrowUtils.throwIf(space == null, ErrorCode.NOT_FOUND_ERROR);
    SpaceVO spaceVO = spaceService.getSpaceVO(space, request);
    User loginUser = userService.getLoginUser(request);
    List<String> permissionList = spaceUserAuthManager.getPermissionL
    spaceVO.setPermissionList(permissionList);
```

```
    return ResultUtils.success(spaceVO);
}
```

获取图片详情接口，注意即使空间 id 不存在（公共图库）也要获取权限列表，管理员会获取到全部权限，这样前端才能顺利展示出操作按钮：

```
@GetMapping("/get/vo")
public BaseResponse<PictureVO> getPictureVOById(long id, HttpServletR
    ThrowUtils.throwIf(id <= 0, ErrorCode.PARAMS_ERROR);

    Picture picture = pictureService.getById(id);
    ThrowUtils.throwIf(picture == null, ErrorCode.NOT_FOUND_ERROR);

    Space space = null;
    Long spaceId = picture.getSpaceId();
    if (spaceId != null) {
        boolean hasPermission = StpKit.SPACE.hasPermission(SpaceUserP
        ThrowUtils.throwIf(!hasPermission, ErrorCode.NO_AUTH_ERROR);
        space = spaceService.getById(spaceId);
        ThrowUtils.throwIf(space == null, ErrorCode.NOT_FOUND_ERROR,
    }

    User loginUser = userService.getLoginUser(request);
    List<String> permissionList = spaceUserAuthManager.getPermissionL
    PictureVO pictureVO = pictureService.getPictureVO(picture, reques
    pictureVO.setPermissionList(permissionList);

    return ResultUtils.success(pictureVO);
}
```

9、接口测试

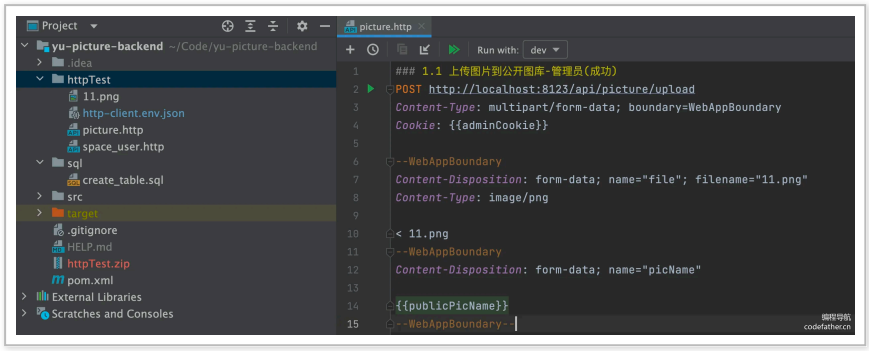
终于开发完了，我们会发现，细节实在是太多了，所以 **一定要进行严格的测试！！！**

用不同权限的用户去验证不同的空间类别（公共图库、私有空间、团队空间）。

如何测试呢？

大家用的比较多的就是单元测试，但是单元测试想要测试携带登录态的 Controller 接口是比较麻烦的。所以我们可以采用自动化接口测试，比如 Postman 等。

此处为了方便，我们直接使用 IDEA 自带的 REST API 测试，可以将测试参数和测试接口保存为文件，每次修改代码后，改改参数，执行文件就能整体测试了。



由于要测试的情况较多，鱼皮给大家准备好了测试代码，直接下载使用即可：httpTest.zip

至此，空间成员权限控制开发完成，大家会发现还是挺麻烦的。其实如果没有公共图库的概念的话，开发起来就轻松很多。因此 Sa-Token 等权限框架要按需使用，更适合复杂的、企业内部的权限管理系统。

如果你想开发起来更轻松一些，推荐其他的实现方式：

1. 直接封装权限校验方法，在业务代码中调用
2. 将团队空间图片的增删改查提取为独立的接口，单独进行权限校验，不影响公共图库

扩展

1）可以给空间操作（SpaceController）、空间分析操作（SpaceAnalyzeController）增加统一的权限校验

空间数据管理

根据需求和方案设计，我们要将旗舰版团队空间的图片数据进行单独管理，每个团队空间的图片数据存储到一张单独的表中，也就是 分表。

1、什么是分库分表?

分库分表是一种将数据拆分到多个数据库或数据表中的设计策略，主要用于解决随着业务数据量和访问量增长带来的数

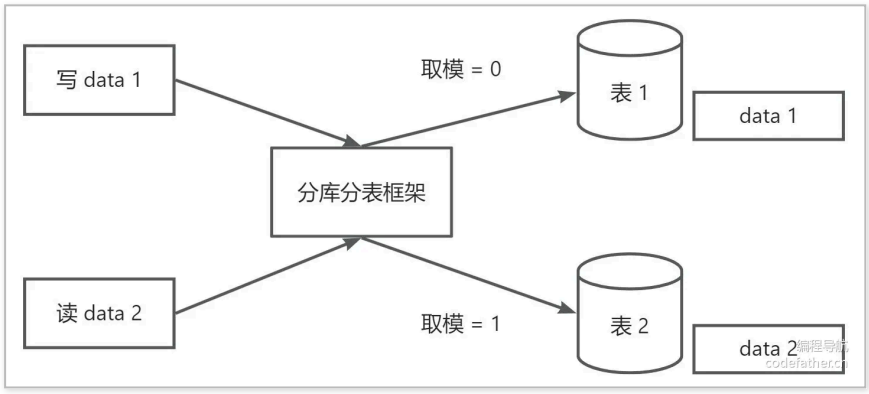
据库性能问题。

通过分库分表，可以减小单库或单表的数据量和访问压力，从而提高查询和写入效率、增强系统的高并发能力、优化大数据量下的性能表现；同时降低单点故障风险，实现更好的系统扩展性和容灾能力。

2、分库分表实现

如果让我们自己实现分库分表，应该怎么做呢？

思路主要是基于业务需求设计 **数据分片规则**，将数据按一定策略（如取模、哈希、范围或时间）分散存储到多个库或表中，同时开发路由逻辑来决定查询或写入操作的目标库表。



简单来说，就是将数据写到不同的表、并且从相同的表读取数据，其实通过给 SQL 表名拼接动态参数就能实现：

```
select * from table_${分片唯一标识}
```

但这只是最简单的情况，实际上，分库分表还涉及跨库表查询、事务一致性、分页聚合等复杂场景，还可能需要配套设计监控、扩容和迁移方案以确保系统的可维护性和扩展性。

所以，不建议自己实现分库分表。本项目中，鱼皮将使用主流的分库分表框架 Apache ShardingSphere 带大家实现。

3、ShardingSphere 分库分表

Apache ShardingSphere 提供了开箱即用的分片策略、灵活的配置能力以及对跨库查询、事务一致性、读写分离等复杂功能的全面支持。

它又分为 2 大核心模块 ShardingSphere-JDBC 和 ShardingSphere-Proxy，我用一张表格来列举 2 者的区别：

维度	ShardingSphere JDBC	ShardingSphere Proxy
运行方式	嵌入式运行在应用内部	独立代理，运行在应用与数据库之间
性能	低网络开销，性能较高	引入网络开销，性能略低
支持语言	仅支持 Java	支持多语言（Java、Python、Go 等）
配置管理	分布式配置较复杂	支持集中配置和动态管理
扩展性	随着应用扩展，需单独调整配置	代理服务集中化管理，扩展性强
适用场景	单体或小型系统，对性能要求高的场景	多语言、大型分布式系统或需要统一管理的场景

对大多数 Java 项目来说，选择 ShardingSphere-JDBC 就足够了；对于跨语言的大型分布式项目、或者公司内有技术部门统一管理基础设施的情况下，再考虑使用 ShardingSphere-Proxy。

本项目也将使用 ShardingSphere-JDBC，在依赖文件中引入：

```
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-jdbc-core-spring-boot-starter</artifac
    <version>5.2.0</version>
</dependency>
```

4、分库分表策略 - 静态分表

分库分表的策略总体分为 2 类：静态分表和动态分表，下面先讲静态分表。

在设计阶段，分表的数量和规则就是固定的，不会根据业务增长动态调整，比如 picture_0、picture_1。

分片规则通常基于某一字段（如图片 id）通过简单规则（如取模、范围）来决定数据存储在哪个表或库中。

这种方式的优点是简单、好理解；缺点是不利于扩展，随着数据量增长，可能需要手动调整分表数量并迁移数据。

举个例子，图片表按图片 id 对 4 取模拆分：

```
String tableName = "picture_" + (pictureId % 4);
```

静态分表的实现很简单，直接在 `application.yml` 中编写 ShardingSphere 的配置就能完成分库分表，比如：

```
rules:
  sharding:
    tables:
      picture:
        actualDataNodes: ds0.picture_${0..2}
        tableStrategy:
          standard:
            shardingColumn: pictureId
            shardingAlgorithmName: pictureIdMod
    shardingAlgorithms:
      pictureIdMod:
        type: INLINE
        props:
          algorithm-expression: picture_${pictureId % 3}
```

你甚至不需要修改任何业务代码，在查询 picture 表（一般叫逻辑表）时，框架会自动帮你修改 SQL，根据 pictureId 将查

询请求路由到不同的表中。如果要进行分页多条数据查询，你只需要写一条查询逻辑表的 SQL 语句即可：

```
SELECT * FROM picture;
```

实际上，ShardingSphere 将查询逻辑表 `picture` 的请求自动路由到所有实际分表 picture_1、picture_2 ... picture_N，获取到数据后，在中间件层自动合并结果并返回给应用程序。

5、分库分表策略 - 动态分表

动态分表是指分表的数量可以根据业务需求或数据量动态增加，表的结构和规则是运行时动态生成的。举个例子，根据时间动态创建 picture_2025_01、picture_2025_02。

```
String tableName = "picture_" + LocalDate.now().format(
    DateTimeFormatter.ofPattern("yyyy_MM")
);
```

显然，动态分表更灵活、扩展性强，适合数据量快速增长的场景；但缺点是实现更复杂，需要动态生成表并维护表的元信息。如果没有控制好，说不定分表特别多，反而影响了数据库的性能。

动态分表的实现就比较麻烦了，首先要自定义分表算法类，还要在代码中编写动态创建表的逻辑。

自定义分表算法类：

```
public class PictureShardingAlgorithm implements StandardShardingAlgo

    @Override
    public String doSharding(Collection<String> availableTargetNames,


    }

    @Override
    public Collection<String> doSharding(Collection<String> collectio
        return new ArrayList<>();
    }

    @Override
    public Properties getProps() {
        return null;
    }
```

```
    @Override
    public void init(Properties properties) {

    }
}
```

对于我们的项目，由于空间是用户动态创建的，显然要使用动态分表，下面来实现。

6、动态分表算法开发

根据需求，我们希望将每个旗舰版空间的图片单独存放在一起，显然是按照 `spaceId` 分表，那么分表的名称规则为 `picture_${spaceId}` 。

1）首先编写动态分表的配置，包括数据库连接、分表规则、分表算法：

```
spring:

  shardingsphere:
    datasource:
      names: yu_picture
      yu_picture:
        type: com.zaxxer.hikari.HikariDataSource
        driver-class-name: com.mysql.cj.jdbc.Driver
        url: jdbc:mysql://localhost:3306/yu_picture
        username: root
        password: 123456
    rules:
      sharding:
        tables:
          picture:
            actual-data-nodes: yu_picture.picture
            table-strategy:
              standard:
                sharding-column: spaceId
                sharding-algorithm-name: picture_sharding_algorithm
        sharding-algorithms:
          picture_sharding_algorithm:
            type: CLASS_BASED
            props:
              strategy: standard
              algorithmClassName: com.yupi.yupicturebackend.manager.s
    props:
      sql-show: true
```

其中，有几个细节需要注意：

1. `actual-data-nodes` 一般情况下是指定一段分表的范围，比如 `yu_picture.picture_${0..9999}` 表示有 picture_0 ~ picture_9999 这 10000 张分表。ShardingSphere 在执行分表查询时会校验要查询的表（比如 picture_123456789）是否在 actual-data-nodes 的配置范围内。但是由于 spaceId 是长整型，范围太大，无法通过指定范围将所有分表名称包含，导致无法通过框架内置的校验。所以此处将 actual-data-nodes 的值设置为逻辑表 `yu_picture.picture` 。
2. 指定分表字段为 spaceId、分表算法为自定义的分片算法 `picture_sharding_algorithm` 。
3. 配置自定义分片算法，采用基于自定义类的方式实现，算法的类名配置必须为类的绝对路径。

2）编写图片分表算法类，必须实现 `StandardShardingAlgorithm` 接口。核心是编写 doSharding 方法，根据 spaceId 获取到实际要查询的分表名，如果 spaceId 不存在分表（比如是私有空间）或者 spaceId 为空（公共图库），那么就从原表（逻辑表）picture 查询。

之所以要做兼容，是因为虽然我们设计上只对团队空间进行分库分表，但是一旦引入了分库分表框架，查询 picture 表时就会触发分表逻辑。

在 `manager.sharding` 包下新建分表算法类：

```
public class PictureShardingAlgorithm implements StandardShardingAlgo

    @Override
    public String doSharding(Collection<String> availableTargetNames,
        Long spaceId = preciseShardingValue.getValue();
        String logicTableName = preciseShardingValue.getLogicTableNam

        if (spaceId == null) {
            return logicTableName;
        }

        String realTableName = "picture_" + spaceId;
        if (availableTargetNames.contains(realTableName)) {
            return realTableName;
        } else {
            return logicTableName;
        }
    }
```

```
    @Override
    public Collection<String> doSharding(Collection<String> collectio
        return new ArrayList<>();
    }

    @Override
    public Properties getProps() {
        return null;
    }

    @Override
    public void init(Properties properties) {

    }
}
```

3）光有上述代码还不能完成动态分表，因为 availableTargetNames（可用的分表）始终为逻辑表名 `picture` ！原因在于 ShardingSphere 在分片逻辑初始化时默认获取的是配置的 `actual-data-nodes` 中的目标表名，也就是我们写的固定值。这样还是无法通过 ShardingSphere 的查询校验，我们也没办法判断 spaceId 是否要分表：

```
if (availableTargetNames.contains(realTableName)) {
    return realTableName;
} else {
    return logicTableName;
}
```

既然框架自身不支持动态维护分表，那我们可以写一个分表管理器，自己来维护分表列表，并更新到 ShardingSphere 的 `actual-data-nodes` 配置中。

在 `manager.sharding` 包下新建分表管理器类：

```
@Component
@Slf4j
public class DynamicShardingManager {

    @Resource
    private DataSource dataSource;

    @Resource
    private SpaceService spaceService;

    private static final String LOGIC_TABLE_NAME = "picture";

    private static final String DATABASE_NAME = "logic_db";
```

```
@PostConstruct
public void initialize() {
    log.info("初始化动态分表配置...");
    updateShardingTableNodes();
}


private Set<String> fetchAllPictureTableNames() {

    Set<Long> spaceIds = spaceService.lambdaQuery()
            .eq(Space::getSpaceType, SpaceTypeEnum.TEAM.getValue(
            .list()
            .stream()
            .map(Space::getId)
            .collect(Collectors.toSet());
    Set<String> tableNames = spaceIds.stream()
            .map(spaceId -> LOGIC_TABLE_NAME + "_" + spaceId)
            .collect(Collectors.toSet());
    tableNames.add(LOGIC_TABLE_NAME);
    return tableNames;
}


private void updateShardingTableNodes() {
    Set<String> tableNames = fetchAllPictureTableNames();
    String newActualDataNodes = tableNames.stream()
            .map(tableName -> "yu_picture." + tableName)
            .collect(Collectors.joining(","));
    log.info("动态分表 actual-data-nodes 配置: {}", newActualDataN

    ContextManager contextManager = getContextManager();
    ShardingSphereRuleMetaData ruleMetaData = contextManager.getM
            .getMetaData()
            .getDatabases()
            .get(DATABASE_NAME)
            .getRuleMetaData();

    Optional<ShardingRule> shardingRule = ruleMetaData.findSingle
    if (shardingRule.isPresent()) {
        ShardingRuleConfiguration ruleConfig = (ShardingRuleConfi
        List<ShardingTableRuleConfiguration> updatedRules = ruleC
                .stream()
                .map(oldTableRule -> {
                    if (LOGIC_TABLE_NAME.equals(oldTableRule.getL
                        ShardingTableRuleConfiguration newTableRu
                        newTableRuleConfig.setDatabaseShardingStr
                        newTableRuleConfig.setTableShardingStrate
                        newTableRuleConfig.setKeyGenerateStrategy
                        newTableRuleConfig.setAuditStrategy(oldTa
                        return newTableRuleConfig;
                    }
                    return oldTableRule;
                })
                .collect(Collectors.toList());
        ruleConfig.setTables(updatedRules);
        contextManager.alterRuleConfiguration(DATABASE_NAME, Coll
        contextManager.reloadDatabase(DATABASE_NAME);
        log.info("动态分表规则更新成功！");
    } else {
        log.error("未找到 ShardingSphere 的分片规则配置，动态分表更新
    }
}
```

```
    private ContextManager getContextManager() {
        try (ShardingSphereConnection connection = dataSource.getConn
            return connection.getContextManager();
        } catch (SQLException e) {
            throw new RuntimeException("获取 ShardingSphere ContextMa
        }
    }
}
```

上述代码虽然看起来比较复杂，但其实不难理解，主要做了这么几件事：

1. 将管理器注册为 Bean，通过 `@PostConstruct` 注解，在 Bean 加载后获取所有的分表并更新配置。
2. 编写获取分表列表的方法，从数据库中查询符合要求的空间列表，再补充上逻辑表，就得到了完整的分表列表。
3. 更新 ShardingSphere 的 actual-data-nodes 动态表名配置。获取到 ShardingSphere 的 ContextManager，找到配置文件中的那条规则进行更新即可。

4）动态创建分表

在分表管理器中新增动态创建分表的方法，通过拼接 SQL 的方式创建出和 picture 表结构一样的分表，创建新的分表后记得更新分表节点。代码如下：

```
public void createSpacePictureTable(Space space) {


    if (space.getSpaceType() == SpaceTypeEnum.TEAM.getValue() && spac
        Long spaceId = space.getId();
        String tableName = "picture_" + spaceId;

        String createTableSql = "CREATE TABLE " + tableName + " LIKE
        try {
            SqlRunner.db().update(createTableSql);

            updateShardingTableNodes();
        } catch (Exception e) {
            log.error("创建图片空间分表失败，空间 id = {}", space.getId
        }
    }
}
```

注意，想要使用 MyBatis Plus 的 SqlRunner，必须要开启配置：

```
mybatis-plus:
  global-config:
    enable-sql-runner: true
```

然后在创建空间时，调用该方法：

```
if (SpaceTypeEnum.TEAM.getValue() == spaceAddRequest.getSpaceType())
    SpaceUser spaceUser = new SpaceUser();
    spaceUser.setSpaceId(space.getId());
    spaceUser.setUserId(userId);
    spaceUser.setSpaceRole(SpaceRoleEnum.ADMIN.getValue());
    result = spaceUserService.save(spaceUser);
    ThrowUtils.throwIf(!result, ErrorCode.OPERATION_ERROR, "创建团队成
}

dynamicShardingManager.createSpacePictureTable(space);

return space.getId();
```

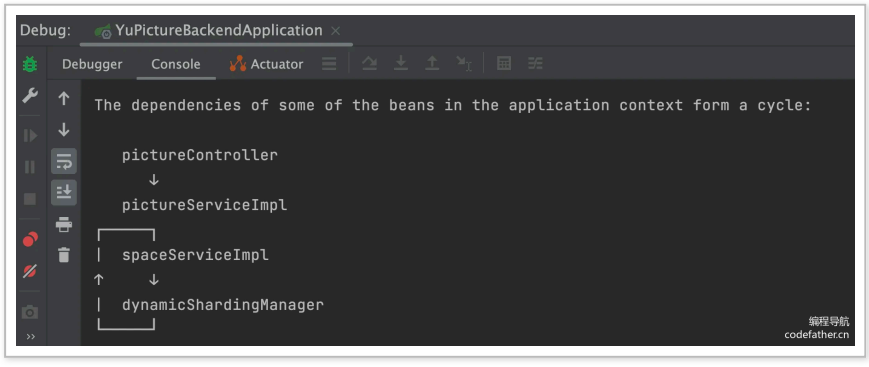
至此，动态分表就开发完成了。

💡 其实 ShardingSphere 还提供了 hint 强制分表路由机制来实现动态分表，允许在代码中强制指定具体的物理表，从而解决动态分表问题。但缺点是需要在每次查询或者操作数据时都显式设置表名，会给代码增加很多额外逻辑，不够优雅。所以不采用，大家了解一下即可。

7、测试

分表是个对系统影响很大的操作，所以要进行严格的测试。

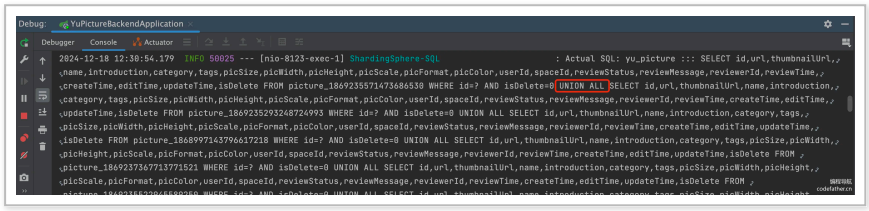
如果启动项目时出现了循环依赖：



可以添加 Lazy 注解解决：

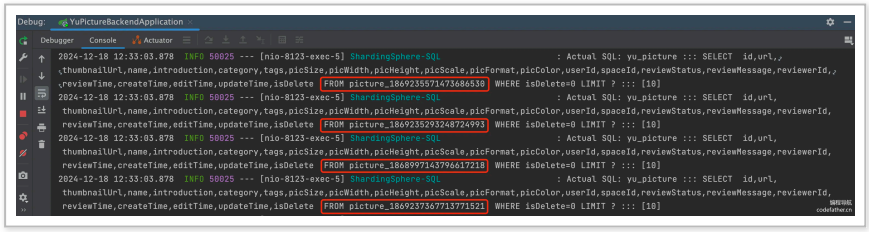
```
@Resource
@Lazy
private DynamicShardingManager dynamicShardingManager;
```

1）单独查询某个图片，不指定 spaceId 查询条件时，会自动查所有的 picture 表：

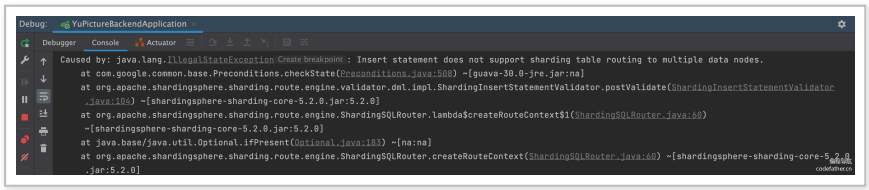


历史数据会自动兼容，只要查到的 spaceId 没有分表，都会查原来的 picture 表。只有指定 spaceId 且存在分表时，才会查询特定的单张分表。

2）查询图片列表，不指定 spaceId 或 nullSpaceId（查询 spaceId 为 null 的值）时，会自动查所有的 picture 表。所以查询时间会随着分表数增加：



3）测试数据插入。插入时如果想往公共空间插入（不指定 spaceId），就会报错，因为 ShardingSphere 不知道要把数据插入到哪个表中。



这就意味着，如果你要使用分表，spaceId 必须不能为 null！

为了解决这个问题，插入时一定要指定 spaceId，可以约定公共空间的 spaceId 都为 0，并且在插入时为 spaceId 设置默认值 0。

```
if (spaceId == null) {
    picture.setSpaceId(0L);
}
```

注意，增删改查时都要补充 spaceId，才能避免报错和多表查询影响效率。

比如查询单个图片，改为通过 QueryWrapper 指定 spaceId 查询：

```
QueryWrapper<Picture> queryWrapper = new QueryWrapper<>();
queryWrapper.eq("id", id)
            .eq("spaceId", spaceId);


Picture picture = pictureService.getOne(queryWrapper);
```

构造图片分页查询条件时，如果查询公共图库，spaceId 改为 0：

```
queryWrapper.eq(nullSpaceId, "spaceId", 0);
```

更新 / 批量更新图片时，设置 spaceId 作为查询条件：

```
UpdateWrapper<Picture> updateWrapper = new UpdateWrapper<>();
updateWrapper.eq("id", picture.getId())
             .eq("spaceId", xxx);


boolean result = pictureService.update(picture, updateWrapper);
```

删除图片时，设置 spaceId 作为查询条件：

```
QueryWrapper<Picture> queryWrapper = new QueryWrapper<>();
queryWrapper.eq("id", id)
            .eq("spaceId", spaceId);
```

```
boolean result = pictureService.remove(queryWrapper);
```

注意，分表后，picture 的 spaceId 将不能修改！！！

经过开发和测试，你会发现动态分库分表的实现非常麻烦。某些单表的查询性能是高了，但整体查询的性能可能会减少，所以原则上 非必要不分表，一定要找到合适的应用场景。

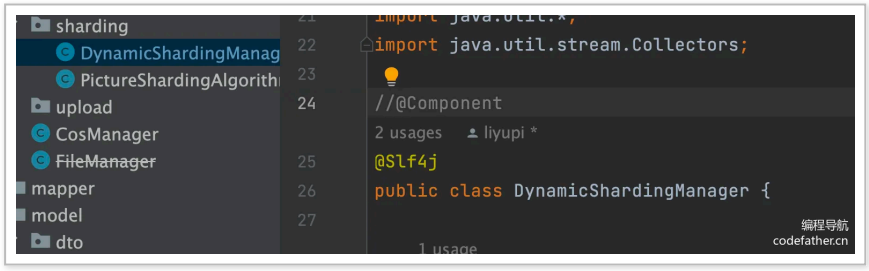
考虑到让更多同学后续直接部署项目，降低理解成本，教程中就不带大家实际执行上述改造细节了，并且我再教大家一种可以关闭分库分表的方法。

8、关闭分库分表（可选）

1）启动类排除依赖（配置文件可以不注释）：

```
@SpringBootApplication(exclude = {ShardingSphereAutoConfiguration.cla
```

2）注释掉分库分表管理器组件 DynamicShardingManager：



3）注释掉使用 DynamicShardingManager 方法的代码，比如空间服务中对其的引用、创建分表的代码：

参考文章

- 关于动态更新 actual-data-nodes：https://www.yuque.com/linghengqian/meve2v/cgi5en
- 相关 issue：https://github.com/apache/shardingsphere/issues/21503

- 开源社区的讨论：https://github.com/apache/shardingsphere/discussions/12258#discussioncomment-3917927

四、前端开发

团队空间的前端开发工作量不大，因为绝大多数页面都可以复用私有空间。

基础代码

首先根据后端的枚举类和常量，定义空间类型相关常量、空间角色相关常量、空间权限常量：

```
export const SPACE_TYPE_ENUM = {
  PRIVATE: 0,
  TEAM: 1,
}


export const SPACE_TYPE_MAP: Record<number, string> = {
  0: '私有空间',
  1: '团队空间',
}


export const SPACE_TYPE_OPTIONS = Object.keys(SPACE_TYPE_MAP).map((ke
  const value = Number(key)
  return {
    label: SPACE_TYPE_MAP[value],
    value,
  }
})


export const SPACE_ROLE_ENUM = {
  VIEWER: "viewer",
  EDITOR: "editor",
  ADMIN: "admin",
} as const;


export const SPACE_ROLE_MAP: Record<string, string> = {
  viewer: "浏览者",
  editor: "编辑者",
  admin: "管理员",
};


export const SPACE_ROLE_OPTIONS = Object.keys(SPACE_ROLE_MAP).map((ke
  return {
    label: SPACE_ROLE_MAP[key],
    value: key,
  };
```

```
});


export const SPACE_PERMISSION_ENUM = {
  SPACE_USER_MANAGE: "spaceUser:manage",
  PICTURE_VIEW: "picture:view",
  PICTURE_UPLOAD: "picture:upload",
  PICTURE_EDIT: "picture:edit",
  PICTURE_DELETE: "picture:delete",
} as const;
```

创建团队空间

1、创建团队空间页面

可以复用创建私有空间页面，通过请求参数的 type 字段来区分创建团队空间（type=1）还是私有空间（不传 type 或为 0）。

1）创建私有空间页面新增空间类别变量：

```
const spaceType = computed(() => {
  if (route.query?.type) {
    return Number(route.query.type)
  }
  return SPACE_TYPE_ENUM.PRIVATE
})
```

2）提交表单时，额外传递 spaceType 字段：

```
res = await addSpaceUsingPost({
  ...formData,
  spaceType: spaceType.value
})
```

3）还可以修改标题的展示，体现出空间类别：

```
<h2>
  {{ route.query?.id ? '修改' : '创建' }}{{ SPACE_TYPE_MAP[spaceType]
</h2>
```

效果如图：

2、创建团队空间入口

1）给全局侧边栏增加创建团队按钮：

```
const fixedMenuItems = [
  {
    key: '/',
    label: '公共图库',
    icon: () => h(PictureOutlined),
  },
  {
    key: '/my_space',
    label: '我的空间',
    icon: () => h(UserOutlined),
  },
  {
    key: '/add_space?type=' + SPACE_TYPE_ENUM.TEAM,
    label: '创建团队',
    icon: () => h(TeamOutlined),
  },
]
```

2）点击菜单事件要改为 `router.push(key)` ，否则无法携带参数跳转：

```
const doMenuClick = ({ key }: { key: string }) => {
  router.push(key)
}
```

3）在全局侧边栏中加载“我的团队空间列表”，每个团队空间作为一个菜单项展示。最终展示的菜单项 = 固定菜单 + 团队空间菜单，代码如下：

```
const teamSpaceList = ref<API.SpaceUserVO[]>([])
const menuItems = computed(() => {
```

```
  if (teamSpaceList.value.length < 1) {
    return fixedMenuItems;
  }

  const teamSpaceSubMenus = teamSpaceList.value.map((spaceUser) => {
    const space = spaceUser.space
    return {
      key: '/space/' + spaceUser.spaceId,
      label: space?.spaceName,
    }
  })
  const teamSpaceMenuGroup = {
    type: 'group',
    label: '我的团队',
    key: 'teamSpace',
    children: teamSpaceSubMenus,
  }
  return [...fixedMenuItems, teamSpaceMenuGroup]
})


const fetchTeamSpaceList = async () => {
  const res = await listMyTeamSpaceUsingPost()
  if (res.data.code === 0 && res.data.data) {
    teamSpaceList.value = res.data.data
  } else {
    message.error('加载我的团队空间失败，' + res.data.message)
  }
}


watchEffect(() => {

  if (loginUserStore.loginUser.id) {
    fetchTeamSpaceList()
  }
})
```

效果如图：

空间成员管理

1、成员管理页面入口

空间详情页的空间分析按钮左边增加成员管理按钮，点击后跳转到成员管理页面：

```
<a-button
  type="primary"
  ghost
  :icon="h(TeamOutlined)"
  :href="`/spaceUserManage/${id}`"
  target="_blank"
>
  成员管理
</a-button>
```

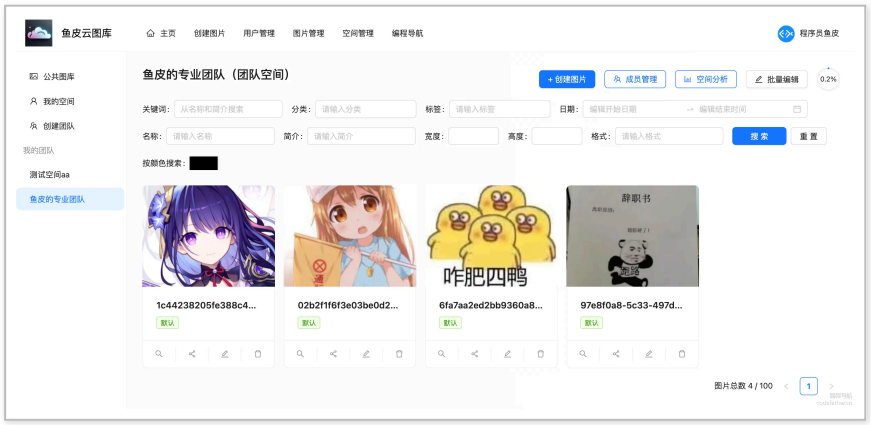
该页面还有一些细节可以优化，比如修改标题展示，区分空间类别：

```
<h2>{{ space.spaceName }}（{{ SPACE_TYPE_MAP[space.spaceType] }}）</h
```

切换空间时，应该重新获取空间信息和图片列表。可以使用 watch 来监听空间 id 变量实现：

```
watch(
  () => props.id,
  (newSpaceId) => {
    fetchSpaceDetail()
    fetchData()
  },
)
```

效果如图：



2、空间成员管理页面

参考语雀的空间成员管理，页面结构为添加成员表单 + 成员信息表格：

1）复制空间管理页面，新建路由，该页面接受空间 id 作为动态参数，展示某个空间下的成员列表：

```
{
  path: '/spaceUserManage/:id',
  name: '空间成员管理',
  component: SpaceUserManagePage,
  props: true,
},
```

该页面绝大多数代码都可以复用空间管理页面，只需要遵循流程修改即可。

2）定义表格列：

```
const columns = [
  {
    title: '用户',
    dataIndex: 'userInfo',
  },
  {
    title: '角色',
    dataIndex: 'spaceRole',
  },
  {
    title: '创建时间',
    dataIndex: 'createTime',
  },
  {
    title: '操作',
    key: 'action',
  },
```

```
]
```

3）调用接口以获取表格数据，此处不需要分页，直接展示所有成员：

```
interface Props {
  id: string
}

const props = defineProps<Props>()


const dataList = ref([])


const fetchData = async () => {
  const spaceId = props.id
  if (!spaceId) {
    return
  }
  const res = await listSpaceUserUsingPost({
    spaceId,
  })
  if (res.data.data) {
    dataList.value = res.data.data ?? []
  } else {
    message.error('获取数据失败，' + res.data.message)
  }
}


onMounted(() => {
  fetchData()
})
```

4）自定义表格列，展示用户信息、空间角色、创建时间和操作按钮。由于可修改的成员信息只有“角色”，所以可以直接将空间角色渲染为下拉框选择器组件，便于管理员操作。

```
<a-table :columns="columns" :data-source="dataList">
  <template #bodyCell="{ column, record }">
    <template v-if="column.dataIndex === 'userInfo'">
      <a-space>
        <a-avatar :src="record.user?.userAvatar" />
        {{ record.user?.userName }}
      </a-space>
    </template>
    <template v-if="column.dataIndex === 'spaceRole'">
      <a-select
        v-model:value="record.spaceRole"
        :options="SPACE_ROLE_OPTIONS"
        @change="(value) => editSpaceRole(value, record)"
      />
    </template>
```

```
    <template v-else-if="column.dataIndex === 'createTime'">
      {{ dayjs(record.createTime).format('YYYY-MM-DD HH:mm:ss') }}
    </template>
    <template v-else-if="column.key === 'action'">
      <a-space wrap>
        <a-button type="link" danger @click="doDelete(record.id)">删除
      </a-space>
    </template>
  </template>
</a-table>
```

编辑空间角色的函数：

```
const editSpaceRole = async (value, record) => {
  const res = await editSpaceUserUsingPost({
    id: record.id,
    spaceRole: value,
  })
  if (res.data.code === 0) {
    message.success('修改成功')
  } else {
    message.error('修改失败，' + res.data.message)
  }
}
```

删除成员的函数：

```
const doDelete = async (id: string) => {
  if (!id) {
    return
  }
  const res = await deleteSpaceUserUsingPost({ id })
  if (res.data.code === 0) {
    message.success('删除成功')

    fetchData()
  } else {
    message.error('删除失败')
  }
}
```

5）在表格上方编写添加成员表单，默认角色是“浏览者”

```
<a-form layout="inline" :model="formData" @finish="handleSubmit">
  <a-form-item label="用户 id" name="userId">
    <a-input v-model:value="formData.userId" placeholder="请输入用户 i
  </a-form-item>
  <a-form-item>
    <a-button type="primary" html-type="submit">添加用户</a-button>
  </a-form-item>
```

```
</a-form>
```

编写表单项变量和提交函数：

```
const formData = reactive<API.SpaceUserAddRequest>({})

const handleSubmit = async () => {
  const spaceId = props.id
  if (!spaceId) {
    return
  }
  const res = await addSpaceUserUsingPost({
    spaceId,
    ...formData,
  })
  if (res.data.code === 0) {
    message.success('添加成功')

    fetchData()
  } else {
    message.error('添加失败，' + res.data.message)
  }
}
```

页面效果如图：

成员权限控制

1、需求梳理

需求：用户没有某个操作权限时，不应该看到对应的操作按钮。

首先梳理一下页面和需要控制权限的按钮，以及对应的权限：

1）空间详情页

- 图片编辑按钮：需要 `picture:edit` 权限
- 图片删除按钮：需要 `picture:delete` 权限
- 成员管理按钮：需要 `spaceUser:manage` 权限
- 空间分析按钮：需要 `spaceUser:manage` 权限
- 上传图片按钮：需要 `picture:upload` 权限

2）图片详情页

- 图片编辑按钮：需要 `picture:edit` 权限
- 图片删除按钮：需要 `picture:delete` 权限

2、权限控制

1）空间详情页新增权限变量。由于每个权限检查的逻辑都是一致的（判断权限列表中是否包含需要的权限），可以编写一个通用的权限检查函数。

```
function createPermissionChecker(permission: string) {
  return computed(() => {
    return (space.value.permissionList ?? []).includes(permission)
  })
}
```

```
const canManageSpaceUser = createPermissionChecker(SPACE_PERMISSION_E
const canUploadPicture = createPermissionChecker(SPACE_PERMISSION_ENU
const canEditPicture = createPermissionChecker(SPACE_PERMISSION_ENUM.
const canDeletePicture = createPermissionChecker(SPACE_PERMISSION_ENU
```

💡 其实也可以让后端计算好 `canXXX` ，然后返回给前端直接用，不过差别不大。

2）给对应的操作按钮增加 `v-if` ，比如创建图片按钮：

```
<a-button
  v-if="canUploadPicture"
  type="primary"
  :href="`/add_picture?spaceId=${id}`"
  target="_blank"
```

```
>
  + 创建图片
</a-button>
```

3）图片列表组件支持控制编辑和删除按钮的隐藏，由父组件传递属性：

```
interface Props {
  dataList?: API.PictureVO[]
  loading?: boolean
  showOp?: boolean
  onReload?: () => void
  canEdit?: boolean
  canDelete?: boolean
}

const props = withDefaults(defineProps<Props>(), {
  dataList: () => [],
  loading: false,
  showOp: false,
  canEdit: false,
  canDelete: false,
})
```

页面代码：

```
<edit-outlined v-if="canEdit" @click="(e) => doEdit(picture, e)" />
<delete-outlined v-if="canDelete" @click="(e) => doDelete(picture, e)
```

空间详情页就可以将权限变量传递给该组件了：

```
<!-- 图片列表 -->
<PictureList
  :dataList="dataList"
  :loading="loading"
  :onReload="fetchData"
  showOp
  :canEdit="canEditPicture"
  :canDelete="canDeletePicture"
/>
```

4）图片详情页也按照上述方式进行修改，不再赘述：

```
function createPermissionChecker(permission: string) {
  return computed(() => {
    return (picture.value.permissionList ?? []).includes(permission)
  })
```

```
}
```

```
const canEdit = createPermissionChecker(SPACE_PERMISSION_ENUM.PICTURE
const canDelete = createPermissionChecker(SPACE_PERMISSION_ENUM.PICTU
```

3、前端测试

涉及到权限的改动都要认真测试，可以主要测试以下情况：

- 未登录操作公共图库、私有图库、团队图库
- 管理员操作公共图库、私有图库、团队图库
- 普通用户操作公共图库、私有图库、别人的私有图库
- 协作者操作团队图库，可以看到编辑和删除按钮，但看不到成员管理按钮
- 浏览者操作团队图库，仅能查看图片，看不到编辑和删除按钮

其他开发

1、问题修复 - 兼容多个空间

MySpacePage 获取我的空间时，改为获取 “私有空间” 的第一个：

```
const res = await listSpaceVoByPageUsingPost({
  userId: loginUser.id,
  current: 1,
  pageSize: 1,
  spaceType: 0,
})
```

2、空间管理补充空间类别

补充空间类别列的定义：

```
{
  title: '空间类别',
  dataIndex: 'spaceType',
},
```

自定义空间类别列的展示：

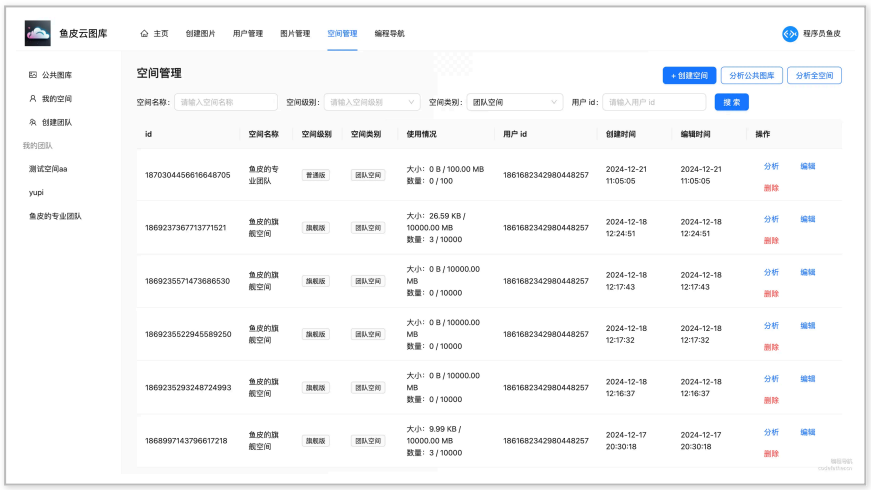
```
<!-- 空间类别 -->
<template v-if="column.dataIndex === 'spaceType'">
  <a-tag>{{ SPACE_TYPE_MAP[record.spaceType] }}</a-tag>
</template>
```

支持按类别搜索：

```
<a-form-item label="空间类别" name="spaceType">
  <a-select
      v-model:value="searchParams.spaceType"
      :options="SPACE_TYPE_OPTIONS"
      placeholder="请输入空间类别"

      allow-clear
  />
</a-form-item>
```

效果如图：



以上就是本节内容，细节非常多，希望大家能够掌握，最好是自己试着敲一遍。

全文完