10 - 图库分析 - 智能协同云图库项目教程 - 编程导航教程

> 本节重点为进一步提升用户 / 管理员在平台上对空间图库的管理与分析能力，我们本节将重点扩展空间图库分析功能，包括：用户空间图库分析 管理员全空间分析通过这些分析功能，用户和管理员能够快速掌握空间的使用情。

本节重点

为进一步提升用户 / 管理员在平台上对空间图库的管理与分析能力，我们本节将重点扩展空间图库分析功能，包括：

- 用户空间图库分析
- 管理员全空间分析

通过这些分析功能，用户和管理员能够快速掌握空间的使用情况，提升管理效率。

一、需求分析

根据我们空间表和图片表的已有字段，可以挖掘出很多分析需求，整体分为用户空间图库分析和管理员全空间分析两类。

用户空间图库分析

用户可以对自己的空间图库进行分析，包括以下几个重点功能：

1）空间资源使用分析：通过统计当前空间已使用大小与总配额的比例，以及图片数量与最大允许数量的占比，帮助用户直观了解空间使用状态，及时清理或升级空间。图表形式推荐使用 仪表盘，类似进度条，可以更直观地了解比例。

2）空间图片分类分析：统计不同分类下图片的数量和总大小占比，帮助用户清晰了解各分类的资源分布，优化存储策略。由于同一个分类要展示多个信息，可以选择 分组条形图 来展示。

3）空间图片标签分析：解析用户图库中的标签，统计每个标签的关联图片数量。由于标签比较多，可以用 词云图 展示所有的标签，并突出常用标签，便于优化管理和图片搜索。

4）空间图片大小分析：按图片大小（如 <100 KB、100 KB-1 MB、>1 MB）分段统计图片数量，帮助用户识别大体积图片，合理分配存储资源。由于按图片大小分类的数量不多，可以使用 饼图 展示，能够体现每类大小图片的数量占比。

5）用户上传行为分析：统计用户每月、每周、每日上传图片的数量趋势，帮助用户识别上传高峰期并优化管理策略（虽然对目前这个阶段没有用，但之后我们要开发团队空间，可以给团队管理员使用）。推荐使用 折线图 呈现时间序列趋势。

管理员全空间分析

管理员全空间分析的核心是面向公共图库、以及所有用户空间的统计和管理：

1）全空间资源使用分析：统计公共图库、以及系统内所有空间的总存储量和总图片数，并且也支持任意空间的图片分类、图片标签、图片大小、用户上传行为的分析，便于管理员了解系统资源分配和利用情况。

其实跟用户分析自己空间的需求一致，只不过分析的范围更大罢了。

2）空间使用排行分析：按存储使用量排序，统计占用存储最多的前 N 个空间，帮助管理员快速定位高占用空间，并识别

潜在的资源滥用或异常情况。可以选用 柱状图，直观地展示排名和存储使用量。

二、方案设计

1、分析类需求的实现流程

对于分析类需求，实现流程几乎都是一致的，包括：

1）数据采集：从数据源（比如 MySQL 数据库或者大数据仓库）获取原始数据。要提前明确涉及的表和字段，必要时采用分页查询处理大数据量。

2）数据预处理：对数据进行清洗、加工和格式化，包括过滤无效数据（比如逻辑删除或审核未通过）、解析复杂字段（比如 JSON 格式的 tags），以及通过字段关联补充上下文信息。

3）数据计算：根据需求进行分组、聚合、排序等，从而计算关键指标，比如计算空间各分类图片的占用比例、用户上传图片的时间趋势。可以根据场景调整计算方案，比如对于大数据量的计算，可以采用 Spark 之类的大数据计算组件做离线计算；对于数据实时性要求较高的实时分析场景，可以用 Flink 做流式处理。

4）数据存储（可选）：针对频繁查询的分析结果，可将结果数据存储为单独的表或缓存，减少重复计算，提高查询效率。

5）数据接口设计：为前端提供统一接口，从而支持查询和展示。需要考虑到数据量较大导致前端渲染卡顿的情况，可以按需精简返回的字符串、分页查询等。

6）数据可视化：通过图表直观展示分析结果，前端可以使用 Apache ECharts 等可视化库渲染。当然也可以让后端生成图表图片并返回，但这种实现方法的灵活度有限。

后续还可以根据用户的反馈持续优化分析逻辑、增加指标或改进性能。

2、本项目实现方案

通过需求分析，我们发现，管理员对公共图库及全空间的分析需求，与用户对自己空间的分析需求在本质上是相同的，**唯一的区别在于图片范围的选择**。

下面以 “空间图片分类分析” 为例。

1）用户分析自己的空间，SQL 示例：

```
SELECT category, SUM(picSize) AS totalSize
FROM picture
WHERE spaceId = xxx
GROUP BY category;
```

2）管理员分析公共图库，SQL 示例：

```
SELECT category, SUM(picSize) AS totalSize
FROM picture
WHERE spaceId IS NULL
GROUP BY category;
```

3）管理员分析全部空间，SQL 示例：

```
SELECT category, SUM(picSize) AS totalSize
FROM picture
GROUP BY category;
```

你会发现，除了 where 查询条件不同之外，其他的计算方式都是一致的。

所以我们可以设计统一的接口，通过传递不同的请求参数，同时满足上述需求。参数含义和优先级如下（优先级从高到低）：

1. queryAll 字段：为 true 时表示查询全空间，仅管理员可使用。
2. queryPublic 字段：为 true 时表示查询公共图库，仅管理员可使用。

3. spaceId 字段：仅在 queryAll 和 queryPublic 均为 false 时生效，表示对特定空间进行分析，仅空间创建者和管理员可使用。

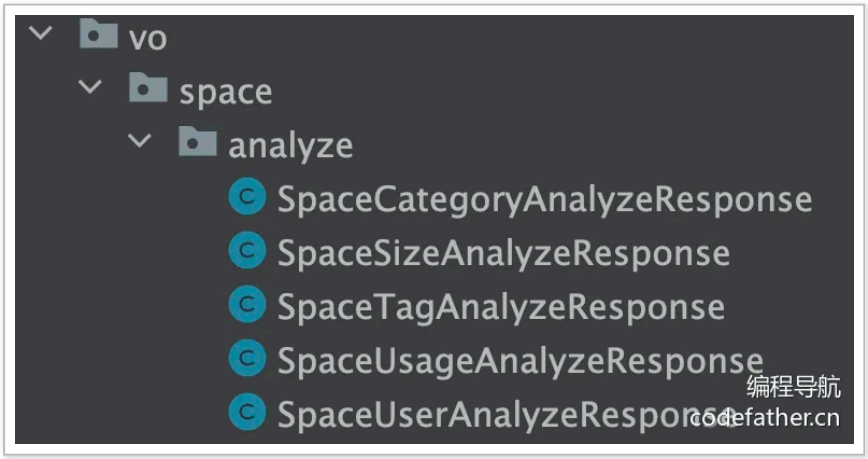
对应的后端伪代码如下，可以将这段逻辑封装为单独的方法：

```
QueryWrapper<Picture> queryWrapper = new QueryWrapper<>();
if (queryAll) {

} else if (queryPublic) {

    queryWrapper.isNull("spaceId");
} else if (spaceId != null) {

    queryWrapper.eq("spaceId", spaceId);
} else {
    throw new BusinessException(ErrorCode.PARAMS_ERROR, "未指定查询范围
}
```

通过这种方式，就不用多针对不同的查询范围编写一套接口了，可以大幅减少重复代码。

三、后端开发

下面我们依次开发每个具体的分析需求，由于分析类需求较多，我们可以编写单独的空间分析服务类（Service）、单独的空间分析接口（Controller），并且统一将分析需求相关的 DTO 和 VO 数据模型放到 `analyze` 包下，如图：



通用分析请求

1）由于我们的很多分析需求都需要传递空间查询范围，可以先写一个公共的图片分析请求封装类：

```
@Data
public class SpaceAnalyzeRequest implements Serializable {


    private Long spaceId;


    private boolean queryPublic;


    private boolean queryAll;

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

然后各个具体的分析请求封装类就能直接继承了，这样也便于后续编写通用的分析请求处理方法。

2）我们可以新建 SpaceAnalyzeService 和对应实现类，开发校验空间分析权限、根据分析范围填充查询对象这两个方法，后续的需求也都会用到。

校验空间分析权限：

```
private void checkSpaceAnalyzeAuth(SpaceAnalyzeRequest spaceAnalyzeRe

    if (spaceAnalyzeRequest.isQueryAll() || spaceAnalyzeRequest.isQue

        ThrowUtils.throwIf(!userService.isAdmin(loginUser), ErrorCode
    } else {

        Long spaceId = spaceAnalyzeRequest.getSpaceId();
        ThrowUtils.throwIf(spaceId == null || spaceId <= 0, ErrorCode
        Space space = spaceService.getById(spaceId);
        ThrowUtils.throwIf(space == null, ErrorCode.NOT_FOUND_ERROR,
        spaceService.checkSpaceAuth(loginUser, space);
    }
}
```

根据分析范围填充查询对象：

```
private static void fillAnalyzeQueryWrapper(SpaceAnalyzeRequest space
    if (spaceAnalyzeRequest.isQueryAll()) {
        return;
    }
    if (spaceAnalyzeRequest.isQueryPublic()) {
```

```
            queryWrapper.isNull("spaceId");
            return;
        }
        Long spaceId = spaceAnalyzeRequest.getSpaceId();
        if (spaceId != null) {
            queryWrapper.eq("spaceId", spaceId);
            return;
        }
        throw new BusinessException(ErrorCode.PARAMS_ERROR, "未指定查询范围
    }
```

需求开发

1、空间资源使用分析

1）开发请求封装类，用于接收前端请求的数据。此处直接继承通用的图片分析请求封装类即可，不需要传递其他字段：

```
@EqualsAndHashCode(callSuper = true)
@Data
public class SpaceUsageAnalyzeRequest extends SpaceAnalyzeRequest {

}
```

2）开发响应视图类，用于将分析结果返回给前端：

```
@Data
public class SpaceUsageAnalyzeResponse implements Serializable {


    private Long usedSize;


    private Long maxSize;


    private Double sizeUsageRatio;


    private Long usedCount;


    private Long maxCount;


    private Double countUsageRatio;

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

3）开发 SpaceAnalyzeService 业务逻辑层，编写分析业务的实现逻辑。

注意，如果是分析全空间或公共图库的使用情况，需要编写“仅管理员可访问”的权限校验逻辑，并且更改查询图片表的范围；如果只是分析单个空间的使用情况，直接从空间表查询出单个空间的数据即可。

代码如下：

```
@Override
public SpaceUsageAnalyzeResponse getSpaceUsageAnalyze(SpaceUsageAnaly
    ThrowUtils.throwIf(spaceUsageAnalyzeRequest == null, ErrorCode.PA
    if (spaceUsageAnalyzeRequest.isQueryAll() || spaceUsageAnalyzeReq


        boolean isAdmin = userService.isAdmin(loginUser);
        ThrowUtils.throwIf(!isAdmin, ErrorCode.NO_AUTH_ERROR, "无权访

        QueryWrapper<Picture> queryWrapper = new QueryWrapper<>();
        queryWrapper.select("picSize");
        if (!spaceUsageAnalyzeRequest.isQueryAll()) {
            queryWrapper.isNull("spaceId");
        }
        List<Object> pictureObjList = pictureService.getBaseMapper().
        long usedSize = pictureObjList.stream().mapToLong(result -> r
        long usedCount = pictureObjList.size();

        SpaceUsageAnalyzeResponse spaceUsageAnalyzeResponse = new Spa
        spaceUsageAnalyzeResponse.setUsedSize(usedSize);
        spaceUsageAnalyzeResponse.setUsedCount(usedCount);

        spaceUsageAnalyzeResponse.setMaxSize(null);
        spaceUsageAnalyzeResponse.setSizeUsageRatio(null);
        spaceUsageAnalyzeResponse.setMaxCount(null);
        spaceUsageAnalyzeResponse.setCountUsageRatio(null);
        return spaceUsageAnalyzeResponse;
    } else {

        Long spaceId = spaceUsageAnalyzeRequest.getSpaceId();
        ThrowUtils.throwIf(spaceId == null || spaceId <= 0, ErrorCode

        Space space = spaceService.getById(spaceId);
        ThrowUtils.throwIf(space == null, ErrorCode.NOT_FOUND_ERROR,


        spaceService.checkSpaceAuth(loginUser, space);


        SpaceUsageAnalyzeResponse response = new SpaceUsageAnalyzeRes
        response.setUsedSize(space.getTotalSize());
        response.setMaxSize(space.getMaxSize());

        double sizeUsageRatio = NumberUtil.round(space.getTotalSize()
        response.setSizeUsageRatio(sizeUsageRatio);
        response.setUsedCount(space.getTotalCount());
        response.setMaxCount(space.getMaxCount());
```

```
            double countUsageRatio = NumberUtil.round(space.getTotalCount
            response.setCountUsageRatio(countUsageRatio);
            return response;
        }
    }
```

上述代码中，有一个很重要的优化细节，由于我们只需要获取图片存储大小，从数据库中查询时要指定 **只查询需要的列**，并且使用 mapper 的 `selectObjs` 方法直接返回 Object 对象，而不用封装为 Picture 对象，可以提高性能并节约存储空间。

```
QueryWrapper<Picture> queryWrapper = new QueryWrapper<>();
queryWrapper.select("picSize");
if (!spaceUsageAnalyzeRequest.isQueryAll()) {
    queryWrapper.isNull("spaceId");
}
List<Object> pictureObjList = pictureService.getBaseMapper().selectOb
long usedSize = pictureObjList.stream().mapToLong(result -> result in
```

可以在 SpaceService 中封装空间权限校验方法，其他的分析需求也会用到：

```
@Override
public void checkSpaceAuth(User loginUser, Space space) {

    if (!space.getUserId().equals(loginUser.getId()) && !userService.
        throw new BusinessException(ErrorCode.NO_AUTH_ERROR);
    }
}
```

然后可以将 SpaceController 中编辑和删除操作的权限校验代码替换为 checkSpaceAuth 方法，统一空间校验逻辑。

4）开发 SpaceAnalyzeController 接口：

```
@RestController
@RequestMapping("/space/analyze")
public class SpaceAnalyzeController {

    @Resource
    private SpaceAnalyzeService spaceAnalyzeService;

    @Resource
```

```
    private UserService userService;


    @PostMapping("/usage")
    public BaseResponse<SpaceUsageAnalyzeResponse> getSpaceUsageAnaly
            @RequestBody SpaceUsageAnalyzeRequest spaceUsageAnalyzeRe
            HttpServletRequest request
    ) {
        ThrowUtils.throwIf(spaceUsageAnalyzeRequest == null, ErrorCod
        User loginUser = userService.getLoginUser(request);
        SpaceUsageAnalyzeResponse spaceUsageAnalyze = spaceAnalyzeSer
        return ResultUtils.success(spaceUsageAnalyze);
    }
}
```

2、空间图片分类分析

1）开发请求封装类。分类分析只需要传递空间范围相关参数，因此可以直接继承公共的 `SpaceAnalyzeRequest` ：

```
@EqualsAndHashCode(callSuper = true)
@Data
public class SpaceCategoryAnalyzeRequest extends SpaceAnalyzeRequest

}
```

2）开发响应视图类。分类分析的结果需要返回图片分类、分类图片数量和分类图片总大小：

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class SpaceCategoryAnalyzeResponse implements Serializable {


    private String category;


    private Long count;


    private Long totalSize;

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

3）开发 Service 服务。按照分类分组查询图片表的数据，注意查询数据库时只获取需要的字段即可：

```
@Override
public List<SpaceCategoryAnalyzeResponse> getSpaceCategoryAnalyze(Spa
    ThrowUtils.throwIf(spaceCategoryAnalyzeRequest == null, ErrorCode


    checkSpaceAnalyzeAuth(spaceCategoryAnalyzeRequest, loginUser);


    QueryWrapper<Picture> queryWrapper = new QueryWrapper<>();

    fillAnalyzeQueryWrapper(spaceCategoryAnalyzeRequest, queryWrapper


    queryWrapper.select("category AS category",
                    "COUNT(*) AS count",
                    "SUM(picSize) AS totalSize")
            .groupBy("category");


    return pictureService.getBaseMapper().selectMaps(queryWrapper)
            .stream()
            .map(result -> {
                String category = result.get("category") != null ? re
                Long count = ((Number) result.get("count")).longValue
                Long totalSize = ((Number) result.get("totalSize")).l
                return new SpaceCategoryAnalyzeResponse(category, cou
            })
            .collect(Collectors.toList());
}
```

💡 建议在编写具体的代码前，先编写示例 SQL 语句，并通过数据库查询客户端来验证。

4）开发接口：

```
@PostMapping("/category")
public BaseResponse<List<SpaceCategoryAnalyzeResponse>> getSpaceCateg
    ThrowUtils.throwIf(spaceCategoryAnalyzeRequest == null, ErrorCode
    User loginUser = userService.getLoginUser(request);
    List<SpaceCategoryAnalyzeResponse> resultList = spaceAnalyzeServi
    return ResultUtils.success(resultList);
}
```

3、空间图片标签分析

1）开发请求封装类，标签分析同样需要继承 `SpaceAnalyzeRequest` ：

```
@EqualsAndHashCode(callSuper = true)
@Data
```

```
public class SpaceTagAnalyzeRequest extends SpaceAnalyzeRequest {

}
```

2）开发响应视图类。标签分析的结果需要返回标签名称和关联的图片数量：

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class SpaceTagAnalyzeResponse implements Serializable {


    private String tag;


    private Long count;

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

3）开发 Service 服务。从数据库获取标签数据，统计每个标签的图片数量，并按使用次数降序排序：

```
@Override
public List<SpaceTagAnalyzeResponse> getSpaceTagAnalyze(SpaceTagAnaly
    ThrowUtils.throwIf(spaceTagAnalyzeRequest == null, ErrorCode.PARA


    checkSpaceAnalyzeAuth(spaceTagAnalyzeRequest, loginUser);


    QueryWrapper<Picture> queryWrapper = new QueryWrapper<>();
    fillAnalyzeQueryWrapper(spaceTagAnalyzeRequest, queryWrapper);


    queryWrapper.select("tags");
    List<String> tagsJsonList = pictureService.getBaseMapper().select
            .stream()
            .filter(ObjUtil::isNotNull)
            .map(Object::toString)
            .collect(Collectors.toList());


    Map<String, Long> tagCountMap = tagsJsonList.stream()
            .flatMap(tagsJson -> JSONUtil.toList(tagsJson, String.cla
            .collect(Collectors.groupingBy(tag -> tag, Collectors.cou


    return tagCountMap.entrySet().stream()
            .sorted((e1, e2) -> Long.compare(e2.getValue(), e1.getVal
            .map(entry -> new SpaceTagAnalyzeResponse(entry.getKey(),
            .collect(Collectors.toList());
```

```
}
```

4）开发接口：

```
@PostMapping("/tag")
public BaseResponse<List<SpaceTagAnalyzeResponse>> getSpaceTagAnalyze
    ThrowUtils.throwIf(spaceTagAnalyzeRequest == null, ErrorCode.PARA
    User loginUser = userService.getLoginUser(request);
    List<SpaceTagAnalyzeResponse> resultList = spaceAnalyzeService.ge
    return ResultUtils.success(resultList);
}
```

4、空间图片大小分析

1）开发请求封装类。图片大小分析也继承 `SpaceAnalyzeRequest` ：

```
@EqualsAndHashCode(callSuper = true)
@Data
public class SpaceSizeAnalyzeRequest extends SpaceAnalyzeRequest {

}
```

2）开发响应视图类。大小分析结果需要返回图片大小范围和对应的图片数量：

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class SpaceSizeAnalyzeResponse implements Serializable {


    private String sizeRange;


    private Long count;

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

3）开发 Service 服务，分段统计图片大小：

```
@Override
public List<SpaceSizeAnalyzeResponse> getSpaceSizeAnalyze(SpaceSizeAn
    ThrowUtils.throwIf(spaceSizeAnalyzeRequest == null, ErrorCode.PAR
```

```
    checkSpaceAnalyzeAuth(spaceSizeAnalyzeRequest, loginUser);


    QueryWrapper<Picture> queryWrapper = new QueryWrapper<>();
    fillAnalyzeQueryWrapper(spaceSizeAnalyzeRequest, queryWrapper);


    queryWrapper.select("picSize");
    List<Long> picSizes = pictureService.getBaseMapper().selectObjs(q
            .stream()
            .map(size -> ((Number) size).longValue())
            .collect(Collectors.toList());


    Map<String, Long> sizeRanges = new LinkedHashMap<>();
    sizeRanges.put("<100KB", picSizes.stream().filter(size -> size <
    sizeRanges.put("100KB-500KB", picSizes.stream().filter(size -> si
    sizeRanges.put("500KB-1MB", picSizes.stream().filter(size -> size
    sizeRanges.put(">1MB", picSizes.stream().filter(size -> size >= 1


    return sizeRanges.entrySet().stream()
            .map(entry -> new SpaceSizeAnalyzeResponse(entry.getKey()
            .collect(Collectors.toList());
}
```

上述代码其实还可以进一步优化，只需要遍历一次 picSizes 列表就可以按大小分别统计了。

4）开发接口：

```
@PostMapping("/size")
public BaseResponse<List<SpaceSizeAnalyzeResponse>> getSpaceSizeAnaly
    ThrowUtils.throwIf(spaceSizeAnalyzeRequest == null, ErrorCode.PAR
    User loginUser = userService.getLoginUser(request);
    List<SpaceSizeAnalyzeResponse> resultList = spaceAnalyzeService.g
    return ResultUtils.success(resultList);
}
```

5、用户上传行为分析

1）开发请求封装类。用户上传行为分析需要增加时间维度（日、周、月）和用户 ID 参数，支持只分析某个用户上传图片的情况。

```
@EqualsAndHashCode(callSuper = true)
@Data
public class SpaceUserAnalyzeRequest extends SpaceAnalyzeRequest {
```

```
    private Long userId;


    private String timeDimension;
}
```

2）开发响应视图类。用户行为分析结果需要返回时间区间和对应的图片数量：

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class SpaceUserAnalyzeResponse implements Serializable {


    private String period;


    private Long count;

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

3）开发 Service 服务，基于图片的创建时间维度统计用户的上传行为，并按照时间升序排序：

```
@Override
public List<SpaceUserAnalyzeResponse> getSpaceUserAnalyze(SpaceUserAn
    ThrowUtils.throwIf(spaceUserAnalyzeRequest == null, ErrorCode.PAR

    checkSpaceAnalyzeAuth(spaceUserAnalyzeRequest, loginUser);


    QueryWrapper<Picture> queryWrapper = new QueryWrapper<>();
    Long userId = spaceUserAnalyzeRequest.getUserId();
    queryWrapper.eq(ObjUtil.isNotNull(userId), "userId", userId);
    fillAnalyzeQueryWrapper(spaceUserAnalyzeRequest, queryWrapper);


    String timeDimension = spaceUserAnalyzeRequest.getTimeDimension()
    switch (timeDimension) {
        case "day":
            queryWrapper.select("DATE_FORMAT(createTime, '%Y-%m-%d')
            break;
        case "week":
            queryWrapper.select("YEARWEEK(createTime) AS period", "CO
            break;
        case "month":
            queryWrapper.select("DATE_FORMAT(createTime, '%Y-%m') AS
            break;
        default:
            throw new BusinessException(ErrorCode.PARAMS_ERROR, "不支
    }
```

```
    queryWrapper.groupBy("period").orderByAsc("period");


    List<Map<String, Object>> queryResult = pictureService.getBaseMap
    return queryResult.stream()
            .map(result -> {
                String period = result.get("period").toString();
                Long count = ((Number) result.get("count")).longValue
                return new SpaceUserAnalyzeResponse(period, count);
            })
            .collect(Collectors.toList());
}
```

上述代码中，我们使用 MySQL 的日期时间函数对图片的创建时间进行了格式化，使得同一天（周 / 月）的值相同，就能够统一按照一个字段（period）进行分组和排序了。

4）开发接口：

```
@PostMapping("/user")
public BaseResponse<List<SpaceUserAnalyzeResponse>> getSpaceUserAnaly
    ThrowUtils.throwIf(spaceUserAnalyzeRequest == null, ErrorCode.PAR
    User loginUser = userService.getLoginUser(request);
    List<SpaceUserAnalyzeResponse> resultList = spaceAnalyzeService.g
    return ResultUtils.success(resultList);
}
```

上述的这些需求，可以同时给用户和管理员使用，已经满足了管理员 “全空间资源使用分析” 的需求。接下来我们只需要单独开发一个 **仅管理员可使用的功能** —— 空间使用排行分析。

6、空间使用排行分析

该功能仅管理员可使用，返回值就是前 N 个空间的信息。由于已经有现成的 Space 空间对象，就不用编写响应视图类了。

1）开发请求封装类。空间使用排行需要接收一个参数 `topN` ，指定要返回的前 N 名空间信息，默认值为 10：

```
@Data
public class SpaceRankAnalyzeRequest implements Serializable {
```

```
    private Integer topN = 10;

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

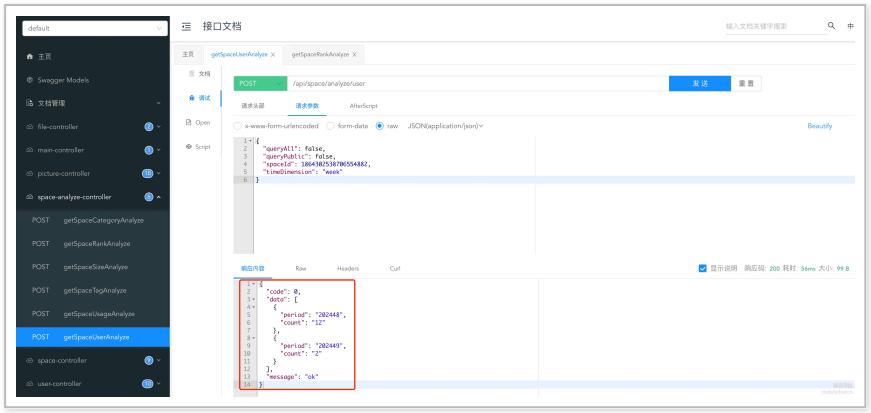
2）开发 Service 服务，按存储使用量排序查询前 N 个空间。注意，只有管理员可以查看空间排行：

```
@Override
public List<Space> getSpaceRankAnalyze(SpaceRankAnalyzeRequest spaceR
ThrowUtils.throwIf(spaceRankAnalyzeRequest == null, ErrorCode.PARAMS_


ThrowUtils.throwIf(!userService.isAdmin(loginUser), ErrorCode.NO_AUTH


QueryWrapper<Space> queryWrapper = new QueryWrapper<>();
queryWrapper.select("id", "spaceName", "userId", "totalSize")
        .orderByDesc("totalSize")
        .last("LIMIT " + spaceRankAnalyzeRequest.getTopN());


return spaceService.list(queryWrapper);
}
```

3）开发接口：

```
@PostMapping("/rank")
public BaseResponse<List<Space>> getSpaceRankAnalyze(@RequestBody Spa
    ThrowUtils.throwIf(spaceRankAnalyzeRequest == null, ErrorCode.PAR
    User loginUser = userService.getLoginUser(request);
    List<Space> resultList = spaceAnalyzeService.getSpaceRankAnalyze(
    return ResultUtils.success(resultList);
}
```

至此，分析需求的后端接口就开发完成了，可以通过 Swagger 接口文档测试一波~ 尤其注意验证查询范围的准确性：



扩展知识 - 自定义 SQL

上述的需求我们是通过 MyBatis Plus 提供的方法实现数据库的分组排序查询，对于更复杂多样的分析需求，其实我们还可以自己在代码中编写 SQL 语句。

可能有部分同学会好奇，MyBatis 还能自定义 SQL？不都是直接调用 `xxx.select` 之类的方法么？

这就是典型的“用框架习惯了”，其实为了提高开发效率、避免自己写 SQL，我们之前一直使用的是 MyBatis Plus 框架。但别忘了，MyBatis Plus 是 MyBatis 的增强版，本质还是基于 MyBatis 的一些能力进行的一些封装简化，自定义 SQL 可是 MyBatis 最最最基础的能力之一。

在 MyBatis 一般会以两种方式来实现自定义 SQL：

1、Java 注解实现

基于 Java 注解写在 xxxMapper.java 中。

注解使用很简单， 在 mapper 层的接口类方法利用 `@Select、@Update、@Insert、@Delete` 等注解，在注解内填写自定义 SQL 语句，即可实现查询、更新、存储、删除。

例如下面两个方法：

```
public interface SpaceMapper extends BaseMapper<Space> {


    @Select("SELECT id, spaceName, userId, totalSize " +
            "FROM space " +
            "ORDER BY totalSize DESC " +
            "LIMIT #{topN}")
    List<Space> getTopNSpaceUsage(int topN);
```

```
    @Delete("DELETE FROM space WHERE userId = #{userId}")
    int deleteByUserId(Long userId);
}
```

完整语句 = SQL 语句模板 + 设置动态参数。方法的参数可以作为动态参数自动填充到 SQL 模板中，得到最终的 SQL 语句，结果也会自动转成方法返回值的 Java 类型。

💡 通过 #{} 和 ${} 都可以实现 SQL 参数绑定，但是两者是有区别的。#{} 是预编译参数，可以防止 SQL 注入，而 ${} 是直接替换，会导致 SQL 注入。

2、XML 配置实现

基于 XML 配置文件写在 xxxMapper.xml 中。

之前通过代码生成器，项目里面已经有很多 xxxMapper.xml 配置文件了。比如 SpaceMapper.xml，里面定义了表和 Java 类的字段映射、SQL 字段列表片段。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.yupi.yupicturebackend.mapper.SpaceMapper">

    <resultMap type="com.yupi.yupicturebackend.model.entity.Space">
        <id property="id" column="id" jdbcType="BIGINT"/>
        <result property="spaceName" column="spaceName" jdbcType="VAR
        <result property="spaceLevel" column="spaceLevel" jdbcType="I
        <result property="maxSize" column="maxSize" jdbcType="BIGINT"
        <result property="maxCount" column="maxCount" jdbcType="BIGIN
        <result property="totalSize" column="totalSize" jdbcType="BIG
        <result property="totalCount" column="totalCount" jdbcType="B
        <result property="userId" column="userId" jdbcType="BIGINT"/>
        <result property="createTime" column="createTime" jdbcType="T
        <result property="editTime" column="editTime" jdbcType="TIMES
        <result property="updateTime" column="updateTime" jdbcType="T
        <result property="isDelete" column="isDelete" jdbcType="TINYI
    </resultMap>

    <sql>
        id,spaceName,spaceLevel,
        maxSize,maxCount,totalSize,
        totalCount,userId,createTime,
        editTime,updateTime,isDelete
    </sql>
</mapper>
```

不用自己新建，仅需在里面添加自定义的 SQL 代码即可。

跟注解类似，MyBatis XML 中提供了 `<select>` 、 `<update>` 、 `<insert>` 、 `<delete>` 等语法，在内部添加自定义 SQL ，即可实现查询、更新、存储、删除。

移除上述 Mapper 的 SQL 注解，然后在 XML 文件中编写 SQL 片段，示例代码如下：

```
<select resultType="com.yupi.Space">
  SELECT id, spaceName, userId, totalSize
  FROM space
  ORDER BY totalSize DESC
  LIMIT #{topN}
</select>
```

```
<delete>
  DELETE FROM space WHERE userId = #{userId}
</delete>
```

需要注意 2 点：

1. Mapper 接口中的方法名称必须与 XML 文件中定义的 SQL 片段的 id 对应，MyBatis 才能正确解析和匹配方法。
2. Mapper 接口方法的返回类型需要与 XML 文件中 resultType（或 resultMap）的定义保持一致，以确保查询结果能够正确映射到返回对象。

这样一来，MyBatis 在运行时会根据 Mapper 接口解析对应的 XML 文件，通过动态代理机制，将接口方法与 SQL 执行逻辑关联起来。

扩展知识 - 查询加速

数据分析通常有 2 种处理方式，实时分析和离线分析。

实时分析是指在数据生成的同时，立即对其进行处理和分析，以提供即时的结果，这种方式适用于需要快速决策的场景，比如监控系统中的异常检测或电商的实时推荐；离线数据分析则是在批量收集和存储数据后，进行复杂计算和深度分析，适合数据量极大、不需要即时结果的场景，比如生成历史报表或挖掘数据中的潜在特征。

即使我们没学过大数据技术，也可以通过业务逻辑层的编码加速数据查询和分析，典型的解决方案就是缓存。利用 Redis 分布式缓存或本地缓存来存储往期的查询结果，并设定一定的过期时间，就能避免重复计算并快速响应。

当然，对于定期的分析诉求（比如计算每日的排行榜）还有一种典型的方案，是通过定时任务计算每日的结果并存储在数据库中，之后就可以按照日期来直接查询某天的结果了。

比如上述需求中的 “用户上传行为分析”，就可以每日计算某个空间的用户上传情况，查询时直接范围查询日期。

排名	统计日期	空间 ID	空间名称	用户 ID	总大小 (MB)
1	2024-12-13	1001	鱼皮的个人空间	2001	2048
2	2024-12-13	1002	张三的个人空间	2002	1832
3	2024-12-13	1003	李四的个人空间	2003	1456
4	2024-12-13	1004	孙五的个人空间	2004	1387
5	2024-12-13	1005	老六的个人空间	2005	1203

四、前端开发

前端开发将分为几个步骤：

1. 引入数据可视化组件
2. 开发分析组件
3. 开发分析页面

数据可视化组件

Apache ECharts 是主流的开源图表库，vue-echarts 是基于 Echarts 的封装，简化了在 Vue 项目中的使用，所以推荐使用 vue-echarts 实现数据可视化。

引入类库，注意必须同时引入 Echarts：

```
npm i echarts vue-echarts
```

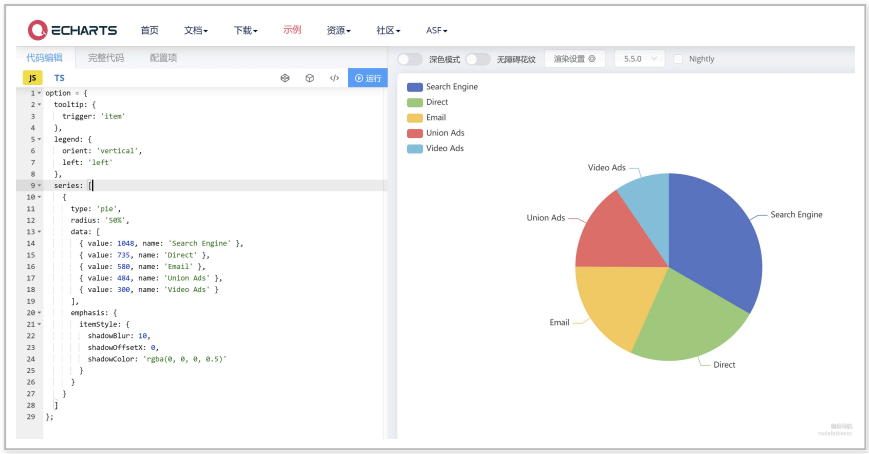
在需要使用图表的页面 JS 引入：

```
import VChart from "vue-echarts";
import "echarts";
```

然后就可以使用组件了，示例代码：

```
<v-chart :option="options" />
```

上述代码中，option 的值包括了图表的基本信息、图表的数据等，可以从 ECharts 官网的示例 中快速学习和获取。Echarts 提供了 Playground 练习网站，建议先在网站上调试出自己想要的效果，再尝试在程序中构造 option 对象：



ECharts 参数可能非常多，建议使用 AI 工具搭配官方文档去了解，比如让 AI 帮忙解释 option 各个参数的含义，或者根据后端接口返回值来生成 option 选项。

开发分析组件

由于分析需求较多，如果都在一个页面中编写所有的分析组件代码，会让页面过于复杂。所以我们把每个分析需求的图表展示和数据获取逻辑都封装为一个独立的组件，统一放在 `components/analyze` 目录下，之后分析页面（父页面）引入这些组件即可。

每个组件的开发模式都是类似的，先定义属性，每个组件都要接受父组件传来的查询范围参数，这样父页面可以灵活指定要查询的空间范围，并统一让所有分析图表重新加载。

```
interface Props {
  queryAll?: boolean
  queryPublic?: boolean
  spaceId?: number
}
```

```
const props = withDefaults(defineProps<Props>(), {
  queryAll: false,
  queryPublic: false,
})
```

每个组件的样式风格可以统一，比如都用卡片进行包装、指定最大高度、给图表应用 loading 效果：

```
<template>
  <div>
    <a-card title="分析需求名称">
      <v-chart :option="options" :loading="loading" />
    </a-card>
  </div>
</template>
```

每个组件也都需要在加载时调用后端接口获取数据，并且计算展示图表需要的 option，不同的需求对应的代码不同，需要定制开发。

1、空间资源使用分析

通过统计当前空间已使用大小与总配额的比例，以及图片数量与最大允许数量的占比，帮助用户直观了解空间使用状态，及时清理或升级空间。图表形式推荐使用 仪表盘 来展示比例，类似进度条，可以更直观地了解比例。

我们使用的 Ant Design 组件库中就自带了 进度条组件 ，支持仪表盘的展示方式，无需使用 ECharts。

1）编写获取数据的逻辑：

```
const data = ref<API.SpaceUsageAnalyzeResponse>({})
const loading = ref(true)


const fetchData = async () => {
  loading.value = true
  const res = await getSpaceUsageAnalyzeUsingPost({
    queryAll: props.queryAll,
    queryPublic: props.queryPublic,
    spaceId: props.spaceId,
  })
  if (res.data.code === 0 && res.data.data) {
    data.value = res.data.data
  } else {
    message.error('获取数据失败，' + res.data.message)
  }
  loading.value = false
```

```
}


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

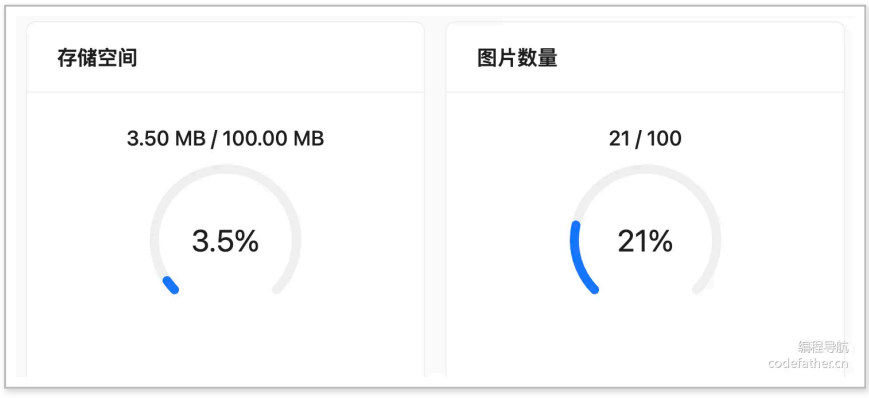
和之前不同的是，为了让组件的属性变化时重新加载图表，我们使用 watchEffect 来监听所有动态变量，只要有任何一个值发生变化，都会重新执行封装的函数。

2）编写图表结构。该组件要展示存储空间使用比例和图片数量使用比例，因此采用一行两列的 Flex 布局：

```
<a-flex gap="middle">
  <a-card title="存储空间">
    <div>
      <h3>{{ formatSize(data.usedSize) }} / {{ data.maxSize ? formatS
      <a-progress type="dashboard" :percent="data.sizeUsageRatio ?? 0
    </div>
  </a-card>
  <a-card title="图片数量">
    <div>
      <h3>{{ data.usedCount }} / {{ data.maxCount ?? '无限制' }} </h3>
      <a-progress type="dashboard" :percent="data.countUsageRatio ??
    </div>
  </a-card>
</a-flex>
```

注意，要给 percent 百分比的值设置默认值，否则会影响页面的加载。

效果如图：



2、空间图片分类分析

统计不同分类下图片的数量和总大小占比，帮助用户清晰了解各分类的资源分布，优化存储策略。由于同一个分类要展示多个信息，可以选择 分组条形图 来展示。

1）编写获取数据的逻辑：

```
const dataList = ref<API.SpaceCategoryAnalyzeResponse[]>([])
const loading = ref(true)


const fetchData = async () => {
  loading.value = true
  const res = await getSpaceCategoryAnalyzeUsingPost({
    queryAll: props.queryAll,
    queryPublic: props.queryPublic,
    spaceId: props.spaceId,
  })
  if (res.data.code === 0) {
    dataList.value = res.data.data ?? []
  } else {
    message.error('获取数据失败，' + res.data.message)
  }
  loading.value = false
}
```

2）编写图表结构：

```
<div>
  <a-card title="图库分类占用">
      <v-chart :option="options" :loading="loading" />
  </a-card>
</div>
```

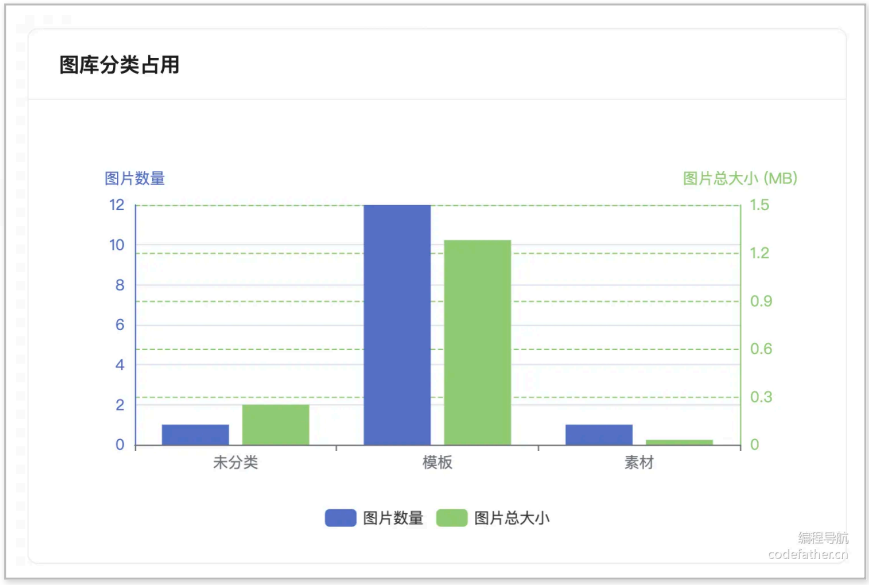
3）编写图表选项。注意，由于 dataList 是有一个加载过程的，要使用 `computed` 计算属性，始终根据 dataList 的值来计算选项：

```
const options = computed(() => {
  const categories = dataList.value.map((item) => item.category)
  const countData = dataList.value.map((item) => item.count)
  const sizeData = dataList.value.map((item) => (item.totalSize / (10

  return {
    tooltip: { trigger: 'axis' },
    legend: { data: ['图片数量', '图片总大小'], top: 'bottom' },
    xAxis: { type: 'category', data: categories },
    yAxis: [
      {
        type: 'value',
        name: '图片数量',
        axisLine: { show: true, lineStyle: { color: '#5470C6' } },
```

```
      },
      {
        type: 'value',
        name: '图片总大小 (MB)',
        position: 'right',
        axisLine: { show: true, lineStyle: { color: '#91CC75' } },
        splitLine: {
          lineStyle: {
            color: '#91CC75',
            type: 'dashed',
          },
        },
      },
    ],
    series: [
      { name: '图片数量', type: 'bar', data: countData, yAxisIndex: 0
      { name: '图片总大小', type: 'bar', data: sizeData, yAxisIndex: 1
    ],
  }
})
```

💡 这段选项代码完全可以先利用 AI 生成，再根据自己的需求微调样式。

效果如图：



3、空间图片标签分析

解析用户图库中的标签，统计每个标签的关联图片数量。由于标签比较多，可以用 词云图 展示所有的标签，并突出常用标签，便于优化管理和图片搜索。

注意，Apache ECharts 默认不会引入词云图组件，需要我们安装 词云图 依赖并引入：

```
import VChart from 'vue-echarts'
import 'echarts'
import 'echarts-wordcloud'
```

1）编写获取数据的逻辑：

```
const fetchData = async () => {
  loading.value = true
  const res = await getSpaceTagAnalyzeUsingPost({
    queryAll: props.queryAll,
    queryPublic: props.queryPublic,
    spaceId: props.spaceId,
  })
  if (res.data.code === 0) {
    dataList.value = res.data.data ?? []
  } else {
    message.error('获取数据失败，' + res.data.message)
  }
  loading.value = false
}
```

2）编写图表结构：

```
<div>
  <a-card title="图库标签词云">
    <v-chart :option="options" :loading="loading" />
  </a-card>
</div>
```

3）编写图表选项：

```
const options = computed(() => {
  const tagData = dataList.value.map((item) => ({
    name: item.tag,
    value: item.count,
  }))

  return {
    tooltip: {
      trigger: 'item',
      formatter: (params: any) => `${params.name}: ${params.value} 次
    },
    series: [
      {
        type: 'wordCloud',
        gridSize: 10,
        sizeRange: [12, 50],
        rotationRange: [-90, 90],
        shape: 'circle',
        textStyle: {
          color: () =>
            `rgb(${Math.round(Math.random() * 255)}, ${Math.round(
```

```
              Math.random() * 255,
            )}, ${Math.round(Math.random() * 255)})`,
        },
        data: tagData,
      },
    ],
  }
})
```

效果如图：

4、空间图片大小分析

按图片大小（如 <100 KB、100 KB-1 MB、>1 MB）分段统计图片数量，帮助用户识别大体积图片，合理分配存储资源。由于按图片大小分类的数量不多，可以使用 饼图 展示，能够体现每类大小图片的数量占比。

1）编写获取数据的逻辑：

```
const fetchData = async () => {
  loading.value = true
  const res = await getSpaceSizeAnalyzeUsingPost({
    queryAll: props.queryAll,
    queryPublic: props.queryPublic,
    spaceId: props.spaceId,
  })
  if (res.data.code === 0) {
    dataList.value = res.data.data ?? []
  } else {
    message.error('获取数据失败，' + res.data.message)
  }
  loading.value = false
```

```
}
```

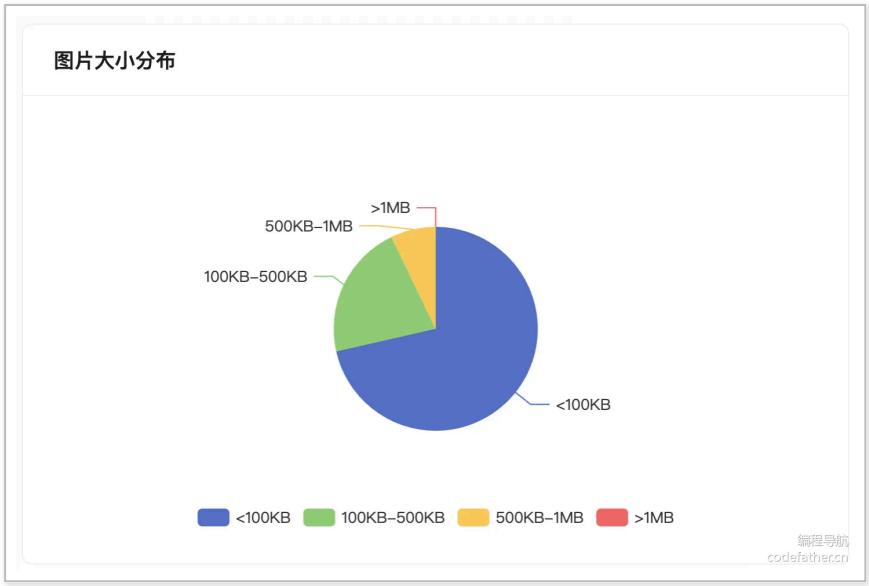
2）编写图表结构：

```
<div>
  <a-card title="空间图片大小分析">
    <v-chart :option="options" :loading="loading" />
  </a-card>
</div>
```

3）编写图表选项：

```
const options = computed(() => {
  const pieData = dataList.value.map((item) => ({
    name: item.sizeRange,
    value: item.count,
  }))

  return {
    tooltip: {
      trigger: 'item',
      formatter: '{a} <br/>{b}: {c} ({d}%)',
    },
    legend: {
      top: 'bottom',
    },
    series: [
      {
        name: '图片大小',
        type: 'pie',
        radius: '50%',
        data: pieData,
      },
    ],
  }
})
```

效果如图：



5、用户上传行为分析

统计用户每月、每周、每日上传图片的数量趋势，帮助用户识别上传高峰期并优化管理策略，推荐使用 折线图 呈现时间序列趋势。

1）编写获取数据的逻辑：

```
const fetchData = async () => {
  loading.value = true
  const res = await getSpaceUserAnalyzeUsingPost({
    queryAll: props.queryAll,
    queryPublic: props.queryPublic,
    spaceId: props.spaceId,
    timeDimension: timeDimension.value,
    userId: userId.value,
  })
  if (res.data.code === 0) {
    dataList.value = res.data.data ?? []
  } else {
    message.error('获取数据失败，' + res.data.message)
  }
  loading.value = false
}
```

2）编写图表结构：

```
<div>
  <a-card title="空间图片用户分析">
    <v-chart :option="options" :loading="loading" />
  </a-card>
</div>
```

3）编写图表选项：

```
const options = computed(() => {
  const periods = dataList.value.map((item) => item.period)
  const counts = dataList.value.map((item) => item.count)

  return {
    tooltip: { trigger: 'axis' },
    xAxis: { type: 'category', data: periods, name: '时间区间' },
    yAxis: { type: 'value', name: '上传数量' },
    series: [
      {
        name: '上传数量',
        type: 'line',
        data: counts,
        smooth: true,
        emphasis: {
          focus: 'series',
        },
      },
    ],
  }
})
```

4）支持用户选择统计的时间范围（日 / 周 / 月）并按照用户 id 筛选。

先开发页面结构，可以利用 Card 组件 的插槽功能，在卡片标题的右侧展示搜索表单：

```
<a-card title="用户上传分析">
  <v-chart :option="options" />
  <template #extra>
    <a-space>
      <a-segmented v-model:value="timeDimension" :options="timeDimens
      <a-input-search placeholder="请输入用户 id" enter-button="搜索用
    </a-space>
  </template>
</a-card>
```

定义变量，用于接受表单项的输入值，并且给下拉选择表单提供默认选项：

```
const userId = ref<string>()
const timeDimension = ref<string>('day')
const timeDimensionOptions = [
  {
    label: '日',
    value: 'day',
  },
```

```
  {
    label: '周',
    value: 'week',
  },
  {
    label: '月',
    value: 'month',
  },
]
```

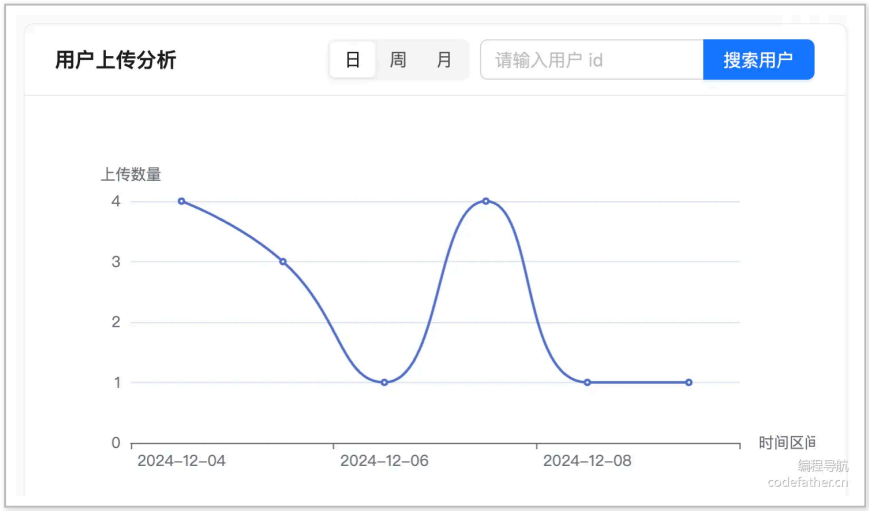
编写提交表单的函数，点击搜索时更改 userId 的值：

```
const doSearch = (value: string) => {
  userId.value = value
}
```

最后，补充搜索条件到获取数据的函数中，只要属性或者选项的值发生了修改，立刻就会重新加载：

```
const res = await getSpaceUserAnalyzeUsingPost({
  queryAll: props.queryAll,
  queryPublic: props.queryPublic,
  spaceId: props.spaceId,
  timeDimension: timeDimension.value,
  userId: userId.value,
})
```

效果如图：



6、空间使用排行分析

按存储使用量排序，统计占用存储最多的前 N 个空间，帮助管理员快速定位高占用空间，并识别潜在的资源滥用或异常

情况。可以选用 柱状图，直观地展示排名和存储使用量。

1）编写获取数据的逻辑：

```
const fetchData = async () => {
  loading.value = true
  const res = await getSpaceRankAnalyzeUsingPost({
    queryAll: props.queryAll,
    queryPublic: props.queryPublic,
    spaceId: props.spaceId,
  })
  if (res.data.code === 0) {
    dataList.value = res.data.data ?? []
  } else {
    message.error('获取数据失败，' + res.data.message)
  }
  loading.value = false
}
```

可以像开发用户上传行为分析图表一样，增加一个修改 topN 查询条数的表单项。这里我们简单一点，就先不传 topN，后端会填充默认值（10 条）。

2）编写图表结构：

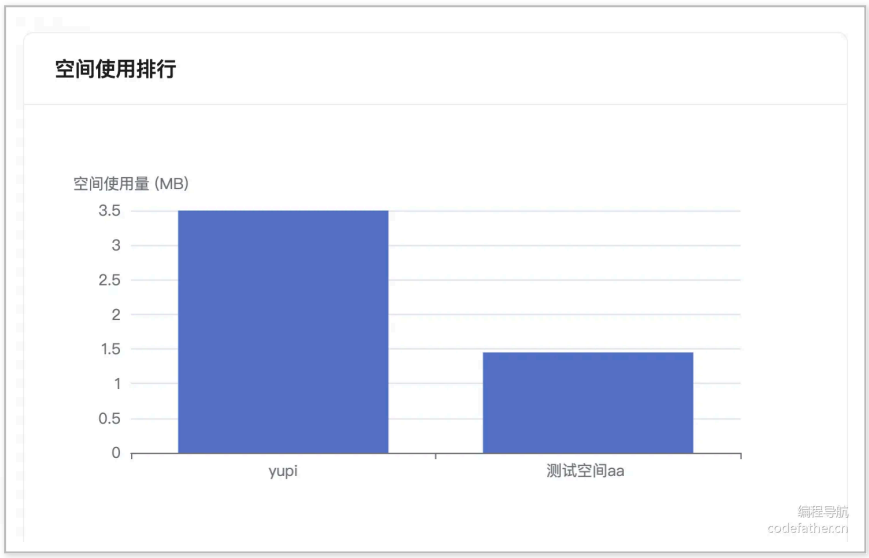
```
<div>
  <a-card title="空间使用排行">
    <v-chart :option="options" />
  </a-card>
</div>
```

3）编写图表选项：

```
const options = computed(() => {
  const spaceNames = dataList.value.map((item) => item.spaceName)
  const usageData = dataList.value.map((item) => (item.totalSize / (1

  return {
    tooltip: { trigger: 'axis' },
    xAxis: {
      type: 'category',
      data: spaceNames,
    },
    yAxis: {
      type: 'value',
      name: '空间使用量 (MB)',
    },
    series: [
      {
        name: '空间使用量 (MB)',
        type: 'bar',
```

```
        data: usageData,
        itemStyle: {
          color: '#5470C6',
        },
      },
    ],
  }
})
```

效果如图：



开发分析页面

新建一个分析页面，所有图表组件都放到该页面中。可以通过 URL 查询参数来触发不同范围的查询，比如分析某个特定空间（spaceId=xxx）、分析公共图库（queryPublic=1）、分析全部空间（queryAll=1）。这样无论是管理员分析全空间 / 公共图库，还是用户分析某个空间，都可以复用该页面。

1）新建页面文件和路由：

```
{
  path: '/space_analyze',
  name: '空间分析',
  component: SpaceAnalyzePage,
}
```

2）开发页面，先定义查询范围参数，从 URL 查询参数中获取：

```
const route = useRoute()


const spaceId = computed(() => {
  return route.query?.spaceId as string
})


const queryAll = computed(() => {
  return !!route.query?.queryAll
})


const queryPublic = computed(() => {
  return !!route.query?.queryPublic
})
```

3）开发页面结构，引入组件，并使用栅格系统一行两列布局：

```
<div>
  <h2>
    空间图库分析 -
    <span v-if="queryAll"> 全部空间 </span>
    <span v-else-if="queryPublic"> 公共图库 </span>
    <span v-else>
      <a :href="`/space/${spaceId}`" target="_blank">id：{{ spaceId }
    </span>
  </h2>
  <a-row :gutter="[16, 16]">
    <!-- 空间使用分析 -->
    <a-col :xs="24" :md="12">
      <SpaceUsageAnalyze :spaceId="spaceId" :queryAll="queryAll" :que
    </a-col>
    <!-- 空间分类分析 -->
    <a-col :xs="24" :md="12">
      <SpaceCategoryAnalyze :spaceId="spaceId" :queryAll="queryAll" :
    </a-col>
    <!-- 标签分析 -->
    <a-col :xs="24" :md="12">
      <SpaceTagAnalyze :spaceId="spaceId" :queryAll="queryAll" :query
    </a-col>
    <!-- 图片大小分段分析 -->
    <a-col :xs="24" :md="12">
      <SpaceSizeAnalyze :spaceId="spaceId" :queryAll="queryAll" :quer
    </a-col>
    <!-- 用户上传行为分析 -->
    <a-col :xs="24" :md="12">
      <SpaceUserAnalyze :spaceId="spaceId" :queryAll="queryAll" :quer
    </a-col>
    <!-- 空间使用排行分析 -->
    <a-col :xs="24" :md="12">
      <SpaceRankAnalyze :spaceId="spaceId" :queryAll="queryAll" :quer
    </a-col>
  </a-row>
```

```
</div>
```

4）权限控制，仅管理员才能看到“空间使用排行分析”。

先定义是否为管理员变量：

```
const loginUserStore = useLoginUserStore()
const loginUser = loginUserStore.loginUser

const isAdmin = computed(() => {
  return loginUser.userRole === 'admin'
})
```

组件添加 `v-if` 属性：

```
<SpaceRankAnalyze v-if="isAdmin"
  :spaceId="spaceId"
  :queryAll="queryAll"
  :queryPublic="queryPublic"
/>
```

其他的权限控制在后端已经实现了，比如普通用户不能访问其他人的空间，效果如图：

补充跳转入口

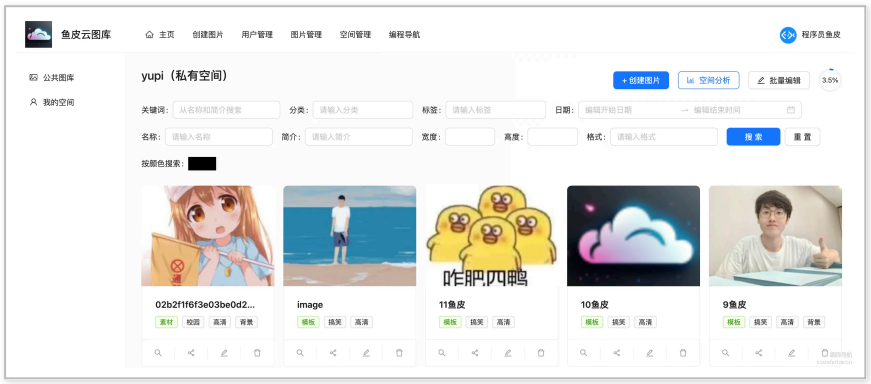
给用户空间详情页、空间管理页面增加跳转到分析页的入口。

1）用户空间详情页补充空间分析按钮：

```
<a-button
  type="primary"
```

```
    ghost
    :icon="h(BarChartOutlined)"
    :href="`/space_analyze?spaceId=${id}`"
    target="_blank"
  >
    空间分析
  </a-button>
```

效果如图：



2）空间管理新增公共图库分析按钮、全空间分析按钮，并且可以直接跳转到某个特定的空间分析页。

```
<a-space>
  <a-button type="primary" href="/add_space" target="_blank">+ 创建空
  <a-button type="primary" ghost href="/space_analyze?queryPublic=1"
    分析公共图库
  </a-button>
  <a-button type="primary" ghost href="/space_analyze?queryAll=1" tar
    分析全空间
  </a-button>
</a-space>
```

操作栏补充：

```
<a-button type="link" :href="`/space_analyze?spaceId=${record.id}`" t
  分析
</a-button>
```

效果如图：

五、扩展

1、用户行为分析支持同环比分析，在同一个图表中展示两条折线（比如一条是上周的，一条是这周的）。

2、【前端】空间排名分析图表中，支持点击某个空间快速跳转查看单个空间详情

3、新增分析需求，按照空间级别对空间进行分类统计，分析不同级别空间的使用情况。

4、新增分析需求，管理员可以对系统内图片的审核状态进行分类统计，还可以按时间维度分析图片审核量的变化趋势。

5、新增分析需求，管理员可以按时间统计用户的登录次数、图片上传量和活跃度的变化趋势，帮助管理员识别高活跃用户，对用户进行分层管理。

至此，本项目的第二阶段就结束了，这个阶段，我们学到了空间模块设计、多维图片搜索、AI 图片编辑、图片分析这一整套的业务流程，现在你已经能够独立完成一个网盘系统、私人相册、私人作品集、私人档案网站啦！

全文完