4 - 图片模块 - 智能协同云图库项目教程 - 编程导航教程

> 本节重点本节我们将开发实现公共图库平台的核心业务流程。

本节重点

本节我们将开发实现公共图库平台的核心业务流程。本节教程可以当做一个 图片分享平台 独立学习，适合新手入门。本节重点需要掌握文件上传下载功能的开发，后端和前端部分也可以按需独立学习。

本节大纲：

- 需求分析
- 方案设计
- 后端开发
  - 文件上传下载能力
- 前端开发

一、需求分析

在设计图库系统时，要优先确保用户能够查看图片功能的实现，而上传功能暂时仅限管理员使用，以保证系统的安全性和稳定性。

基于这一原则，我们将优先实现以下功能，并按优先级排列如下：

1）管理员功能

- 图片上传与创建

- 图片管理
- 图片修改（编辑信息）

2）用户功能

- 查看与搜索图片列表（主页）
- 查看图片详情（详情页）
- 图片下载

具体分析每个需求：

1）图片上传与创建：仅管理员可用，支持选择本地图片上传，并填写相关信息，如名称、简介、标签、分类等。系统会自动解析图片的基础信息（如宽高和格式等），便于检索。

2）图片管理：管理员可以对图库内的图片资源进行管理，包括查询和删除。

3）图片修改：管理员可以对图片信息进行编辑，例如修改名称、简介、标签、分类等。

4）查看与搜索图片列表：用户在主页上可按关键词搜索图片，并支持按分类、标签等筛选条件分页查看图片列表。

5）查看图片详情：用户点击列表中的图片后，可进入详情页，查看图片的大图及相关信息，如名称、简介、分类、标签、其他图片信息（如宽高和格式等）。

6）图片下载：用户在详情页可点击下载图片按钮，将图片保存至本地。

二、方案设计

方案设计阶段我们需要确认：

- 库表设计
- 如何实现图片上传和下载？

- 创建图片的业务流程
- 如何解析图片的信息?

库表设计

表名：picture（图片表），根据需求可以做出如下 SQL 设计：

```
create table if not exists picture
(
    id           bigint auto_increment comment 'id' primary key,
    url          varchar(512)                       not null comment
    name         varchar(128)                       not null comment
    introduction varchar(512)                       null comment '简介
    category     varchar(64)                        null comment '分类
    tags         varchar(512)                      null comment '标签
    picSize      bigint                             null comment '图片
    picWidth     int                                null comment '图片
    picHeight    int                                null comment '图片
    picScale     double                             null comment '图片
    picFormat    varchar(32)                        null comment '图片
    userId       bigint                             not null comment
    createTime   datetime default CURRENT_TIMESTAMP not null comment
    editTime     datetime default CURRENT_TIMESTAMP not null comment
    updateTime   datetime default CURRENT_TIMESTAMP not null on updat
    isDelete     tinyint  default 0                 not null comment
    INDEX idx_name (name),
    INDEX idx_introduction (introduction),
    INDEX idx_category (category),
    INDEX idx_tags (tags),
    INDEX idx_userId (userId)
) comment '图片' collate = utf8mb4_unicode_ci;
```

几个注意事项：

1）字段设计：

- 基础信息：包括图片的 URL、名称、简介、分类、标签等，满足图片管理和分类筛选的基本需求。
- 图片属性：记录图片大小、分辨率（宽度、高度）、宽高比和格式，方便后续按照多种维度筛选图片。
- 用户关联：通过 `userId` 字段关联用户表，表示由哪个用户创建了该图片。

- 多个标签：由于标签支持多个值，使用 JSON 数组字符串来维护，而不是单独新建一个表，可以提高开发效率。示例格式：`["标签1", "标签2"]`

2）时间字段区分：

- updateTime：任何字段的修改都会触发数据库自动更新，便于记录最新变动。**该字段可以不让用户看到**
- editTime：专用于记录图片信息被编辑的时间，需要通过业务逻辑主动更新。**该字段可以对用户公开**

3）索引设计：为高频查询的字段（如图片名称、简介、分类、标签、用户 id）添加索引，提高查询效率。

4）逻辑删除：也就是“软删除”，使用 `isDelete` 字段标记是否删除，避免直接删除数据导致数据不可恢复问题

💡 索引是数据库优化的核心手段，对于有大量查询需求的字段，不要吝啬索引的添加。

如何选择合适的字段添加索引？这是一道经典的面试题，可以 在面试鸭上查看 。

如何实现图片上传和下载？

图片本质上是一种“小型”文件，那么我们要思考：将文件上传到哪里？从哪里下载？

最简单的方式就是上传到后端项目所在的服务器，直接使用 Java 自带的文件读写 API 就能实现。但是，这种方式存在不少缺点，比如：

1. 不利于扩展：单个服务器的存储是有限的，如果存满了，只能再新增存储空间或者清理文件。
2. 不利于迁移：如果后端项目要更换服务器部署，之前所有的文件都要迁移到新服务器，非常麻烦。
3. 不够安全：如果忘记控制权限，用户很有可能通过恶意代码访问服务器上的文件，而且想控制权限也比较麻烦，需

要自己实现。

4. 不利于管理：只能通过一些文件管理器进行简单的管理操作，但是缺乏数据处理、流量控制等多种高级能力。

因此，除了存储一些需要清理的临时文件之外，我们通常不会将用户上传并保存的文件（比如用户头像和图片）直接上传到服务器，而是更推荐大家使用专业的第三方存储服务，专业的工具做专业的事。其中，最常用的便是 对象存储 。

什么是对象存储?

对象存储是一种存储 海量文件 的 分布式 存储服务，具有高扩展性、低成本、可靠安全等优点。

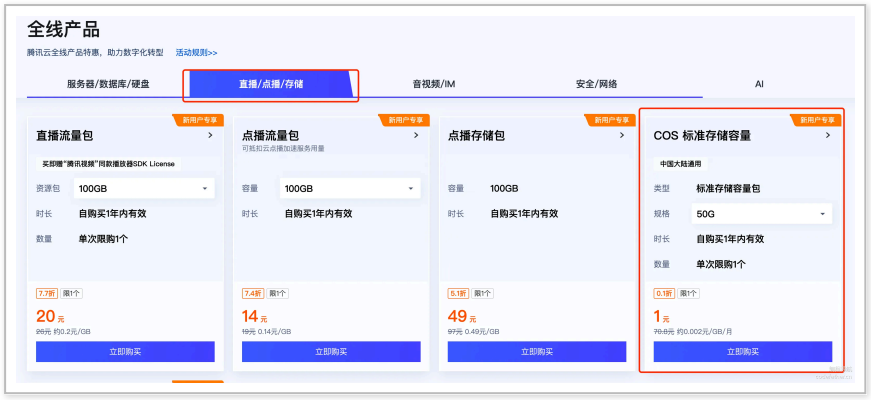
比如开源的对象存储服务 MinIO，还有商业版的云服务，像亚马逊 S3（Amazon S3）、阿里云对象存储（OSS）、腾讯云对象存储（COS）等等。

我个人更推荐学习者和小型团队使用第三方云服务，不要自己再去搭建 MinIO 之类的，主打一个快速！

鱼皮使用最多的对象存储服务当属 腾讯云的 COS，除了基本的对象存储的优点外，还可以通过控制台、API、SDK 和工具等多样化方式，简单快速地接入 COS，进行多格式文件的上传、下载和管理，实现海量数据存储和管理。

本节教程中，就将使用腾讯云 COS 带大家实现文件的上传和下载。鱼皮之前搭建的图床就是使用了 COS 对象存储实现，很简单。

💡 对象存储等第三方云服务通常是付费功能，按照存储量、流量等方式计费。不过对于大家学习来说，由于图片存储量和访问量不大，价格非常便宜（几元 ~ 几十元），而且还有一定免费额度。可以点击 鱼皮的分享链接 优惠购买。下拉找到 全线产品，点击存储页签，就能看到了：



创建图片的业务流程

创建图片其实包括了 2 个过程：上传图片文件 + 补充图片信息并保存到数据库中

有 2 种常见的处理方式：

1）先上传再提交数据：用户直接上传图片，系统生成图片的存储 URL；然后在用户填写其他相关信息并提交后，才保存图片记录到数据库中。

2）上传图片时直接保存记录：在用户上传图片后，系统立即生成图片的完整数据记录（包括图片 URL 和其他元信息），无需等待用户点击提交，图片信息就立刻存入了数据库中。之后用户再填写其他图片信息，相当于编辑了已有图片记录的信息。

方案 1 的优点是流程简单，但缺点是如果用户不提交，图片会残留在存储中，导致空间浪费；方案 2 则可以理解为保存了“图片草稿”，即使用户不填写任何额外信息，也能找到之前的创建记录。

在我们的系统中，由于图片是核心资源，所以此处选择方案 2。 便于对图片进行溯源，还可以对图片上传做一些限制 —— 比如发现用户上传资源过多，就禁止上传。

如何解析图片的信息?

根据需求，我们要获取的图片信息包括：宽度、高度、宽高比、大小、格式、名称。

主流的获取图片信息的方法主要有 2 种：

1. 在后端服务器直接处理图片，比如 Java 库 ImageIO、Python 库 Pillow，还有更成熟的专业图像处理库 OpenCV 等。

2. 通过第三方云存储服务（如腾讯云 COS、AWS S3）或图像处理 API（如 ImageMagick、ExifTool）直接提取图片的元数据。

由于本教程中使用腾讯云 COS 对象存储来实现文件的上传和下载，腾讯云 COS 对象存储支持在图片上传时通过 数据万象 ** ** 服务直接获取到图片的各种基础信息：

这样一来，我们不用再单独引入一个库或者自己编写解析代码了，更方便；而且提供的免费额度足够用了，所以采用这种方式。

免费范围

按照地域范围，免费额度只适用于**公有云地域**，地域划分可前往 地域和访问域名 进行了解。

免费额度详情

图片处理

操作类型	免费额度	有效期	发放频次
基础图片处理	10TB	1个月	每月发放
盲水印	6000次	2个月	一次性发放
Guetzli 压缩	6000次	2个月	一次性发放

OK，有了实现方案后，我们先来开发后端接口。

三、后端开发

先准备好项目所需的依赖 —— 对象存储，然后再开发服务和接口。

创建并使用对象存储

首先进入 对象存储的控制台 ，创建存储桶。

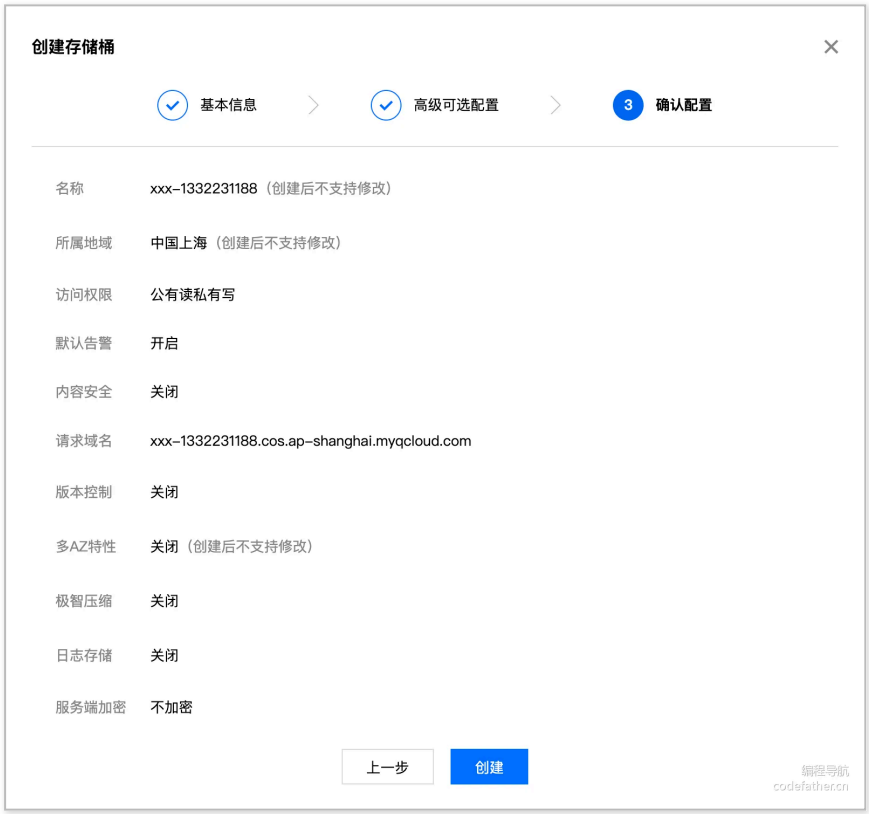
可以把存储桶理解为一个存储空间，和文件系统类似，都是根据路径找到文件或目录（比如 `/test/aaa.jpg` ）。可以多个项目共用一个存储桶，也可以每个项目一个。

点击创建存储桶，注意地域选择国内（离用户较近的位置）。此处访问权限先选择“公有读私有写”，因为我们的存储桶要存储允许用户公开访问图片。而如果整个存储桶要存储的文件都不允许用户访问，建议选择私有读写，更安全。

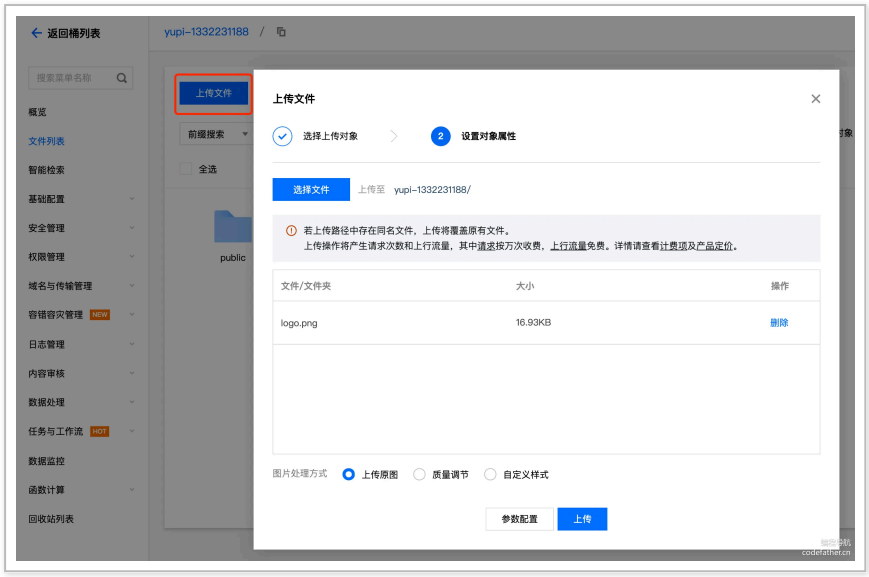
默认告警一定要勾选！因为对象存储服务的存储和访问流量都是计费的，超限后我们要第一时间得到通知并进行相应的处理。

> 不过也不用太担心，自己做项目的话一般是没人攻击你的，而且对象存储很便宜，正常情况下消耗的费用寥寥无几。

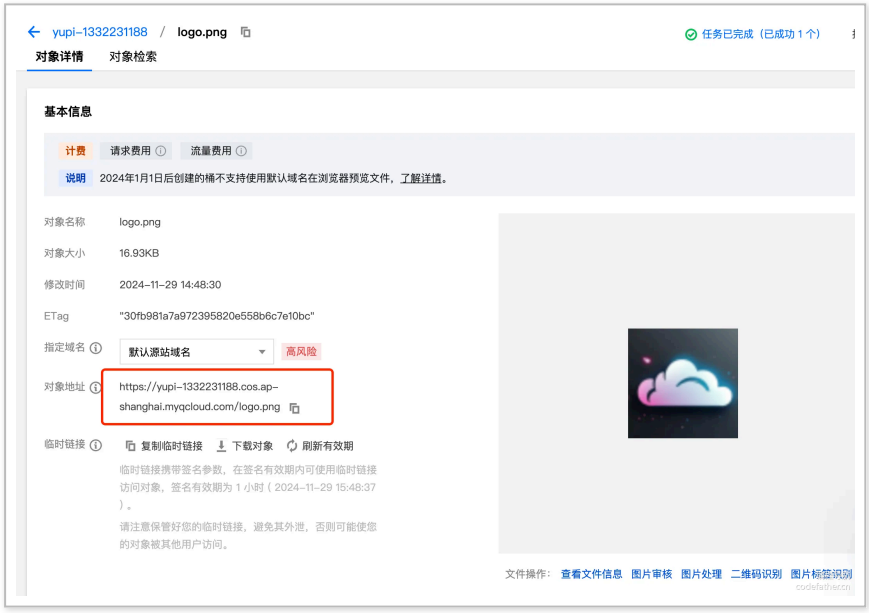
然后一直点击“下一步”即可：



开通成功后，我们可以试着使用 web 控制台上传和浏览文件：



上传文件后，可以使用对象存储服务为我们生成的默认域名，在线访问图片：



当然，一般情况下我们会使用程序来操作存储桶，下面就来实现。

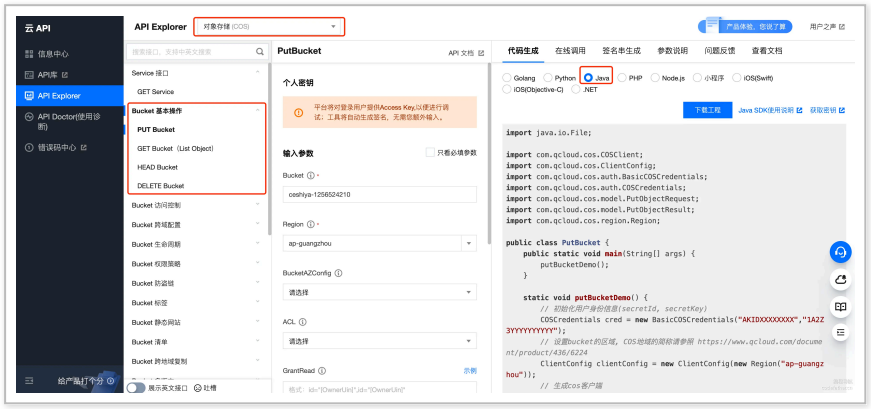
💡 你可能注意到了，系统提示我们“使用默认域名”是高风险的，因为对象存储的源站域名默认是不支持更换的，如果暴露出去被攻击者盯上了，可能你就只能迁移到一个新的存储桶了。本项目教程后续会给大家分享更安全的使用方式。

后端操作对象存储

如何在 Java 程序中使用对象存储呢？

其实非常简单，一般情况下，第三方服务都会提供比较贴心的文档教程，比如这里我们参考 官方的快速入门或 Java SDK 文档 ，就能快速入门基本操作（增删改查都有）。

还有更高级的学习操作方法，如果你是腾讯云熟练用户，可以直接使用 API Explorer ，在线寻找操作和示例代码。



1、初始化客户端

参考官方文档，我们要先初始化一个 COS 客户端对象，和对象存储服务进行交互。

对于我们的项目，只需要复用一个 COS 客户端对象即可，所以我们可以通过编写配置类初始化客户端对象。

1）引入 COS 依赖：

```
<dependency>
    <groupId>com.qcloud</groupId>
    <artifactId>cos_api</artifactId>
    <version>5.6.227</version>
```

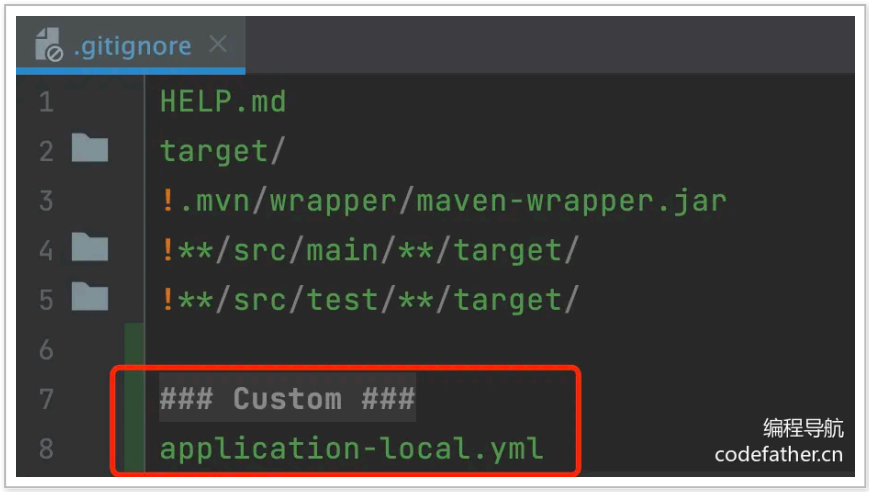
```
</dependency>
```

2）在项目的 `config` 包下新建 `CosClientConfig` 类。负责读取配置文件，并创建一个 COS 客户端的 Bean。代码如下：

```
@Configuration
@ConfigurationProperties(prefix = "cos.client")
@Data
public class CosClientConfig {


    private String host;


    private String secretId;


    private String secretKey;


    private String region;


    private String bucket;

    @Bean
    public COSClient cosClient() {

        COSCredentials cred = new BasicCOSCredentials(secretId, secre

        ClientConfig clientConfig = new ClientConfig(new Region(regio

        return new COSClient(cred, clientConfig);
    }
}
```

3）填写配置文件。

一定要注意防止密码泄露！ 所以我们新建 `application-local.yml` 文件，并且在 `.gitignore` 中忽略该文件的提交，这样就不会将代码等敏感配置提交到代码仓库了。



`application-local.yml` 配置代码如下：

```
cos:
  client:
    host: xxx
    secretId: xxx
    secretKey: xxx
    region: xxx
    bucket: xxx
```

可以通过如下方式分别获取需要的配置。

host 为存储桶域名，也就是我们前面访问测试上传的图片的域名，可以在 COS 控制台的域名信息部分找到：

secretId、secretKey 密钥对：在 腾讯云访问管理 => 密钥管理中获取。

region 表示地域名，可以 点此获取 。

bucket 是存储桶名，可以点进存储桶详情页获取：

2、通用能力类

在 `manager` 包下新建 `CosManager` 类，提供通用的对象存储操作，比如文件上传、文件下载等。

💡 Manager 也是人为约定的一种写法，表示通用的、可复用的能力，可供其他代码（比如 Service）调用。

该类需要引入对象存储配置和 COS 客户端，用于和 COS 进行交互。代码如下：

```
@Component
public class CosManager {

    @Resource
    private CosClientConfig cosClientConfig;

    @Resource
```

```
    private COSClient cosClient;


}
```

3、文件上传

参考 官方文档 的“上传对象”部分，可以编写出文件上传的代码。

1） `CosManager` 新增上传对象的方法，代码如下：

```
public PutObjectResult putObject(String key, File file) {
    PutObjectRequest putObjectRequest = new PutObjectRequest(cosClien
            file);
    return cosClient.putObject(putObjectRequest);
}
```

2）为了方便测试，在 `FileController` 中编写测试文件上传接口。

核心流程是先接受用户上传的文件，指定上传的路径，然后调用 `cosManager.putObject` 方法上传文件到 COS 对象存储；上传成功后，会返回一个文件的 key（其实就是文件路径），便于我们访问和下载文件。

需要注意，测试接口一定要加上管理员权限！防止任何用户随意上传文件。

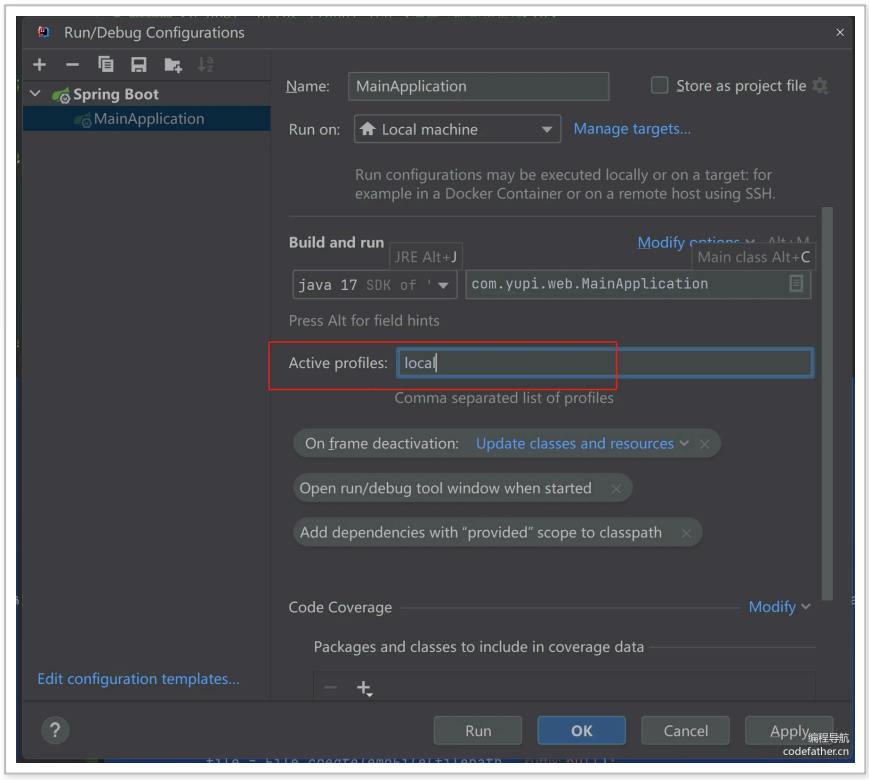
测试文件上传接口代码如下：

```
@AuthCheck(mustRole = UserConstant.ADMIN_ROLE)
@PostMapping("/test/upload")
public BaseResponse<String> testUploadFile(@RequestPart("file") Multi

    String filename = multipartFile.getOriginalFilename();
    String filepath = String.format("/test/%s", filename);
    File file = null;
    try {

        file = File.createTempFile(filepath, null);
        multipartFile.transferTo(file);
        cosManager.putObject(filepath, file);

        return ResultUtils.success(filepath);
```

```
        } catch (Exception e) {
            log.error("file upload error, filepath = " + filepath, e);
            throw new BusinessException(ErrorCode.SYSTEM_ERROR, "上传失败'
        } finally {
            if (file != null) {

                boolean delete = file.delete();
                if (!delete) {
                    log.error("file delete error, filepath = {}", filepat
                }
            }
        }
    }
```

4）测试接口。

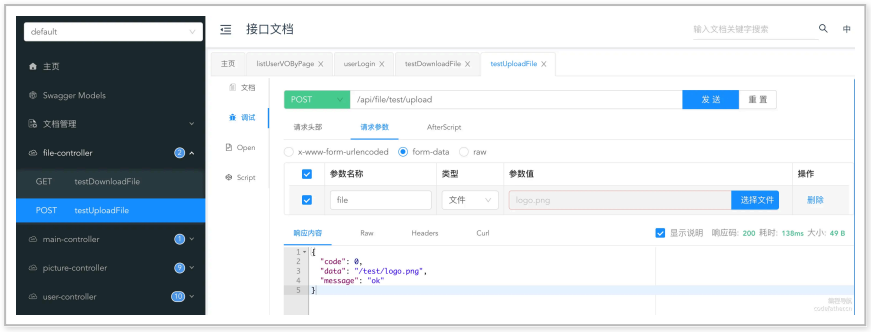
使用 local 配置启动项目：



也可以在主配置文件中指定激活的环境配置：

```
spring:
  profiles:
    active: local
```

打开 Swagger 接口文档，测试文件上传：



4、文件下载

官方文档介绍了 2 种文件下载方式。一种是直接下载 COS 的文件到后端服务器（适合服务器端处理文件），另一种是获取到文件下载输入流（适合返回给前端用户）。

参考官方文档：

- https://cloud.tencent.com/document/product/436/65937
- https://cloud.tencent.com/document/product/436/10199#.E4.B8.8B.E8.BD.BD.E5.AF.B9.E8.B1.A1

其实还有第三种“下载方式”，直接通过 URL 路径链接访问，适用于单一的、可以被用户公开访问的资源，比如用户头像、本项目中的公开图片。

💡 对于安全性要求较高的场景，建议先通过后端服务器进行权限校验，然后从 COS 下载文件到服务器，再返回给前端，这样可以在后端限制只有登录用户才能下载。

不过还有更巧妙的方式，先通过后端服务器进行权限校验，然后返回给前端一个临时秘钥，之后前端可以凭借该秘钥直接从对象存储下载，不用经过服务端中转，性能更高。

对于我们目前的项目，图片本身就是公开的，直接使用第三种方式，凭借 URL 连接访问即可。

但是作为一个小知识，还是给大家演示如何将对象存储的文件下载到服务器中。

1）首先在 `CosManager` 中新增对象下载方法，根据对象的 key 获取存储信息：

```
public COSObject getObject(String key) {
    GetObjectRequest getObjectRequest = new GetObjectRequest(cosClien
    return cosClient.getObject(getObjectRequest);
}
```

2）为了方便测试，在 `FileController` 中编写测试文件下载接口。

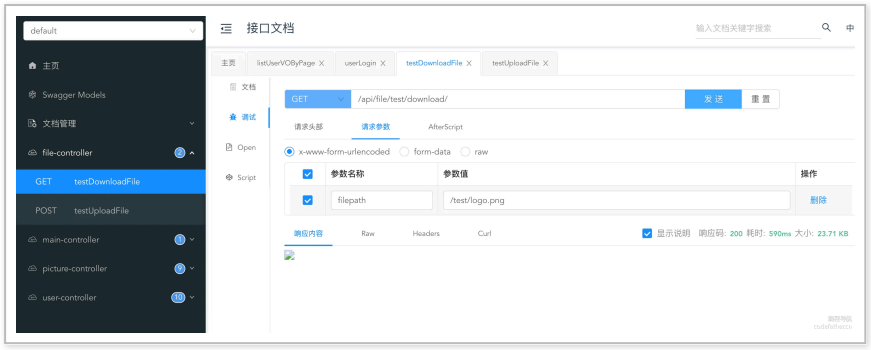
核心流程是根据路径获取到 COS 文件对象，然后将文件对象转换为文件流，并写入到 Servlet 的 Response 对象中。注意要设置文件下载专属的响应头。

同上，测试接口一定要加上管理员权限！防止任何用户随意上传文件。

测试文件下载接口代码如下：

```
@AuthCheck(mustRole = UserConstant.ADMIN_ROLE)
@GetMapping("/test/download/")
public void testDownloadFile(String filepath, HttpServletResponse res
    COSObjectInputStream cosObjectInput = null;
    try {
        COSObject cosObject = cosManager.getObject(filepath);
        cosObjectInput = cosObject.getObjectContent();

        byte[] bytes = IOUtils.toByteArray(cosObjectInput);

        response.setContentType("application/octet-stream;charset=UTF
        response.setHeader("Content-Disposition", "attachment; filena

        response.getOutputStream().write(bytes);
        response.getOutputStream().flush();
    } catch (Exception e) {
        log.error("file download error, filepath = " + filepath, e);
        throw new BusinessException(ErrorCode.SYSTEM_ERROR, "下载失败"
    } finally {
        if (cosObjectInput != null) {
            cosObjectInput.close();
        }
    }
}
```

3）启动项目，打开 Swagger 接口文档，测试文件下载：



在某些操作系统（浏览器）中，虽然图片没有显示，但通过响应码和响应大小，也能判断出图片是成功下载了。

至此，后端操作对象存储的代码已编写完成，下面开发接口。

图片基础代码

首先利用 MyBatisX 插件生成图片表相关的基础代码，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。

然后根据需求优化 Picture 实体类：

```
@TableName(value ="picture")
@Data
public class Picture implements Serializable {

    @TableId(type = IdType.ASSIGN_ID)
    private Long id;


    private String url;


    private String name;


    private String introduction;


    private String category;


    private String tags;


    private Long picSize;
```

```
    private Integer picWidth;

    private Integer picHeight;

    private Double picScale;

    private String picFormat;

    private Long userId;

    private Date createTime;

    private Date editTime;

    private Date updateTime;

    @TableLogic
    private Integer isDelete;

    @TableField(exist = false)
    private static final long serialVersionUID = 1L;
}
```

图片上传

1、数据模型

在 `model.dto.picture` 下新建用于接受请求参数的类。由于图片需要支持重复上传（基础信息不变，只改变图片文件），所以要添加图片 id 参数：

```
@Data
public class PictureUploadRequest implements Serializable {

    private Long id;

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

在 `model.vo` 下新建上传成功后返回给前端的响应类，这是一个视图包装类，可以额外关联上传图片的用户信息。还可以编写 Picture 实体类和该 VO 类的转换方法，便于后续快速传值。

```
@Data
public class PictureVO implements Serializable {

    private Long id;

    private String url;

    private String name;

    private String introduction;

    private List<String> tags;

    private String category;

    private Long picSize;

    private Integer picWidth;

    private Integer picHeight;

    private Double picScale;

    private String picFormat;

    private Long userId;

    private Date createTime;

    private Date editTime;

    private Date updateTime;

    private UserVO user;

    private static final long serialVersionUID = 1L;

    public static Picture voToObj(PictureVO pictureVO) {
        if (pictureVO == null) {
            return null;
        }
        Picture picture = new Picture();
        BeanUtils.copyProperties(pictureVO, picture);

        picture.setTags(JSONUtil.toJsonStr(pictureVO.getTags()));
```

```
        return picture;
    }


    public static PictureVO objToVo(Picture picture) {
        if (picture == null) {
            return null;
        }
        PictureVO pictureVO = new PictureVO();
        BeanUtils.copyProperties(picture, pictureVO);

        pictureVO.setTags(JSONUtil.toList(picture.getTags(), String.c
        return pictureVO;
    }
}
```

2、通用文件上传服务

之前虽然我们已经编写了通用的对象存储操作类 CosManager，但这个类并不能直接满足我们的图片上传需求。

比如：

- 图片是否符合要求？需要校验
- 将图片上传到哪里？需要指定路径
- 如何解析图片？需要使用数据万象服务

所以，可以针对我们的项目，编写一个更贴合业务的文件上传服务 FileManager（这里用 Service 也可以），该服务提供一个上传图片并返回图片解析信息的方法。

```
@Service
@Slf4j
public class FileManager {

    @Resource
    private CosClientConfig cosClientConfig;

    @Resource
    private CosManager cosManager;


}
```

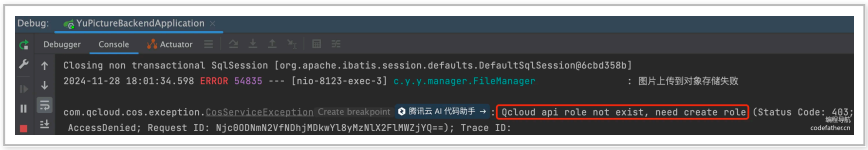
1）在 `model.dto.file` 中新增用于接受图片解析信息的包装类：

```
@Data
public class UploadPictureResult {

    private String url;

    private String picName;

    private Long picSize;

    private int picWidth;

    private int picHeight;

    private Double picScale;

    private String picFormat;

}
```

2）参考 数据万象 的文档，在 CosManager 中添加上传图片并解析图片的方法：

```
public PutObjectResult putPictureObject(String key, File file) {
    PutObjectRequest putObjectRequest = new PutObjectRequest(cosClien
            file);

    PicOperations picOperations = new PicOperations();

    picOperations.setIsPicInfo(1);

    putObjectRequest.setPicOperations(picOperations);
    return cosClient.putObject(putObjectRequest);
}
```

如果你之前没有使用过数据万象，需要先 开通数据万象并授权 ，否则会报错：



3）在 FileManager 中编写上传图片的方法：

```
public UploadPictureResult uploadPicture(MultipartFile multipartFile,

    validPicture(multipartFile);

    String uuid = RandomUtil.randomString(16);
    String originFilename = multipartFile.getOriginalFilename();
    String uploadFilename = String.format("%s_%s.%s", DateUtil.format
            FileUtil.getSuffix(originFilename));
    String uploadPath = String.format("/%s/%s", uploadPathPrefix, upl
    File file = null;
    try {

        file = File.createTempFile(uploadPath, null);
        multipartFile.transferTo(file);

        PutObjectResult putObjectResult = cosManager.putPictureObject
        ImageInfo imageInfo = putObjectResult.getCiUploadResult().get

        UploadPictureResult uploadPictureResult = new UploadPictureRe
        int picWidth = imageInfo.getWidth();
        int picHeight = imageInfo.getHeight();
        double picScale = NumberUtil.round(picWidth * 1.0 / picHeight
        uploadPictureResult.setPicName(FileUtil.mainName(originFilena
        uploadPictureResult.setPicWidth(picWidth);
        uploadPictureResult.setPicHeight(picHeight);
        uploadPictureResult.setPicScale(picScale);
        uploadPictureResult.setPicFormat(imageInfo.getFormat());
        uploadPictureResult.setPicSize(FileUtil.size(file));
        uploadPictureResult.setUrl(cosClientConfig.getHost() + "/" +
        return uploadPictureResult;
    } catch (Exception e) {
        log.error("图片上传到对象存储失败", e);
        throw new BusinessException(ErrorCode.SYSTEM_ERROR, "上传失败"
    } finally {
        this.deleteTempFile(file);
    }
}


public void validPicture(MultipartFile multipartFile) {
    ThrowUtils.throwIf(multipartFile == null, ErrorCode.PARAMS_ERROR,

    long fileSize = multipartFile.getSize();
    final long ONE_M = 1024 * 1024L;
    ThrowUtils.throwIf(fileSize > 2 * ONE_M, ErrorCode.PARAMS_ERROR,

    String fileSuffix = FileUtil.getSuffix(multipartFile.getOriginalF

    final List<String> ALLOW_FORMAT_LIST = Arrays.asList("jpeg", "jpg
    ThrowUtils.throwIf(!ALLOW_FORMAT_LIST.contains(fileSuffix), Error
}
```

```
    public void deleteTempFile(File file) {
        if (file == null) {
            return;
        }

        boolean deleteResult = file.delete();
        if (!deleteResult) {
            log.error("file delete error, filepath = {}", file.getAbsolut
        }
    }
```

上述代码中有几个实现关键：

1. 由于文件校验规则较复杂，单独抽象为 validPicture 方法，对文件大小、类型进行校验。
2. 文件上传时，会先在本地创建临时文件，无论上传是否成功，都要记得删除临时文件，否则会导致资源泄露。
3. 可以根据自己的需求定义文件上传地址，比如此处鱼皮给文件名前增加了上传日期和 16 位 uuid 随机数，便于了解文件上传时间并防止文件重复。还预留了一个 uploadPathPrefix 参数，由调用方指定上传文件到哪个目录。

💡 如果多个项目共享存储桶，可以给上传文件路径再加一个 ProjectName 前缀。不过建议还是每个项目独立分配资源。

3、服务开发

在 PictureService 中编写上传图片的方法：

接口：

```
    PictureVO uploadPicture(MultipartFile multipartFile,
                            PictureUploadRequest pictureUploadRequest,
                            User loginUser);
```

实现类：

```
    @Override
    public PictureVO uploadPicture(MultipartFile multipartFile, PictureUp
```

```
        ThrowUtils.throwIf(loginUser == null, ErrorCode.NO_AUTH_ERROR);

        Long pictureId = null;
        if (pictureUploadRequest != null) {
            pictureId = pictureUploadRequest.getId();
        }

        if (pictureId != null) {
            boolean exists = this.lambdaQuery()
                    .eq(Picture::getId, pictureId)
                    .exists();
            ThrowUtils.throwIf(!exists, ErrorCode.NOT_FOUND_ERROR, "图片不
        }


        String uploadPathPrefix = String.format("public/%s", loginUser.ge
        UploadPictureResult uploadPictureResult = fileManager.uploadPictu

        Picture picture = new Picture();
        picture.setUrl(uploadPictureResult.getUrl());
        picture.setName(uploadPictureResult.getPicName());
        picture.setPicSize(uploadPictureResult.getPicSize());
        picture.setPicWidth(uploadPictureResult.getPicWidth());
        picture.setPicHeight(uploadPictureResult.getPicHeight());
        picture.setPicScale(uploadPictureResult.getPicScale());
        picture.setPicFormat(uploadPictureResult.getPicFormat());
        picture.setUserId(loginUser.getId());

        if (pictureId != null) {

            picture.setId(pictureId);
            picture.setEditTime(new Date());
        }
        boolean result = this.saveOrUpdate(picture);
        ThrowUtils.throwIf(!result, ErrorCode.OPERATION_ERROR, "图片上传失
        return PictureVO.objToVo(picture);
    }
```

上述代码中，注意：

1. 我们将所有图片都放到了 public 目录下，并且每个用户的图片存储到对应用户 id 的目录下，便于管理。

2. 如果 pictureId 不为空，表示更新已有图片的信息，需要判断对应 id 的图片是否存在，并且更新时要指定 editTime 编辑时间。可以调用 MyBatis Plus 提供的 saveOrUpdate 方法兼容创建和更新操作。

4、接口开发

在 PictureController 中编写上传图片接口，注意仅管理员可用：

```
@PostMapping("/upload")
@AuthCheck(mustRole = UserConstant.ADMIN_ROLE)
public BaseResponse<PictureVO> uploadPicture(
        @RequestPart("file") MultipartFile multipartFile,
        PictureUploadRequest pictureUploadRequest,
        HttpServletRequest request) {
    User loginUser = userService.getLoginUser(request);
    PictureVO pictureVO = pictureService.uploadPicture(multipartFile,
    return ResultUtils.success(pictureVO);
}
```

5、测试

使用 Swagger 进行测试，发现当上传图片过大时，会触发一段报错。但这个报错不是我们自定义的异常导致的，而是由于 Tomcat 服务器默认限制了请求中文件上传的大小。

需要在 `application.yml` 中更改配置，调大允许上传的文件大小：

```
spring:

  servlet:
    multipart:
      max-file-size: 10MB
```

扩展思路

1）可以用枚举类（FileUploadBizEnum）支持根据业务场景区分文件上传路径、校验规则等，从而复用 FileManager。

2）目前在文件上传时，会先在本地创建临时文件。如果你不需要对文件进行额外的处理、想进一步提高性能，可以直接用流的方式将请求中的文件上传到 COS。以下代码仅供参考：

```
public static String uploadToCOS(MultipartFile multipartFile, String

    COSClient cosClient = createCOSClient();

    try (InputStream inputStream = multipartFile.getInputStream()) {

        ObjectMetadata metadata = new ObjectMetadata();
        metadata.setContentLength(multipartFile.getSize());
        metadata.setContentType(multipartFile.getContentType());
```

```
            PutObjectRequest putObjectRequest = new PutObjectRequest(buck


            cosClient.putObject(putObjectRequest);


            return "https://" + bucketName + ".cos." + cosClient.getClien
                    + ".myqcloud.com/" + key;
        } finally {
            cosClient.shutdown();
        }
    }
```

3）补充更严格的校验，比如为支持的图片格式定义枚举，仅允许上传枚举定义的格式。

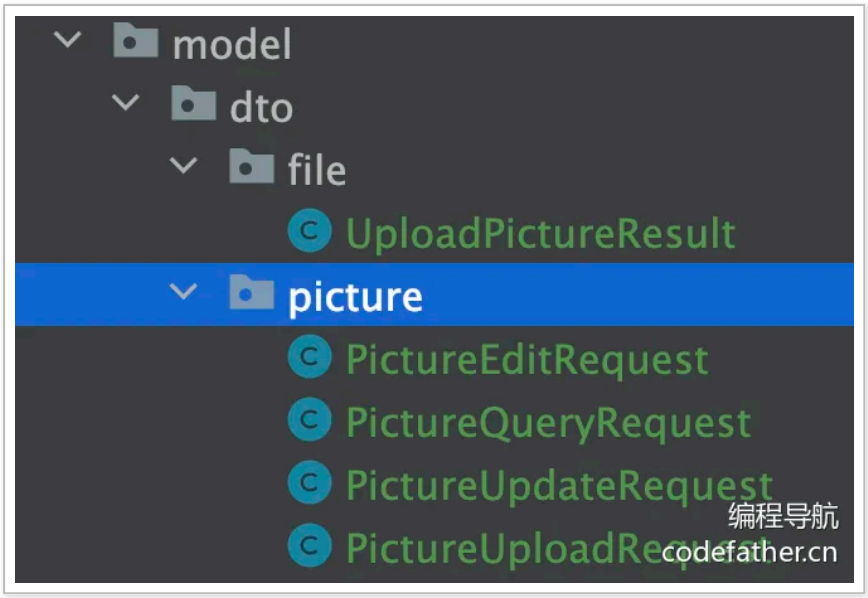
图片管理

图片管理功能具体可以拆分为：

- 【管理员】根据 id 删除图片
- 【管理员】更新图片
- 【管理员】分页获取图片列表（不需要脱敏和限制条数）
- 【管理员】根据 id 获取图片（不需要脱敏）
- 分页获取图片列表（需要脱敏和限制条数）
- 根据 id 获取图片（需要脱敏）
- 修改图片

1、数据模型

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



1）图片更新请求，给管理员使用。注意要将 tags 的类型改为 `List<String>` ，便于前端上传：

```
@Data
public class PictureUpdateRequest implements Serializable {

    private Long id;

    private String name;

    private String introduction;

    private String category;

    private List<String> tags;

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

2）图片修改请求，一般情况下给普通用户使用，可修改的字段范围小于更新请求：

```
@Data
public class PictureEditRequest implements Serializable {

    private Long id;

    private String name;
```

```
    private String introduction;

    private String category;

    private List<String> tags;

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

3）图片查询请求，需要继承公共包中的 `PageRequest` 来支持分页查询：

```
@EqualsAndHashCode(callSuper = true)
@Data
public class PictureQueryRequest extends PageRequest implements Seria

    private Long id;

    private String name;

    private String introduction;

    private String category;

    private List<String> tags;

    private Long picSize;

    private Integer picWidth;

    private Integer picHeight;

    private Double picScale;

    private String picFormat;

    private String searchText;

    private Long userId;

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

```
}
```

2、服务开发

1）在 UserService 中编写判断用户是否为管理员的方法，后续开发中会用到。

接口代码：

```
boolean isAdmin(User user);
```

实现类：

```
@Override
public boolean isAdmin(User user) {
    return user != null && UserRoleEnum.ADMIN.getValue().equals(user.
}
```

2）对于分页查询接口，需要根据用户传入的参数来构造 SQL 查询。由于使用 MyBatis Plus 框架，不用自己拼接 SQL 了，而是通过构造 QueryWrapper 对象来生成 SQL 查询。

可以在 PictureService 中编写一个方法，专门用于将查询请求转为 QueryWrapper 对象：

```
@Override
public QueryWrapper<Picture> getQueryWrapper(PictureQueryRequest pict
    QueryWrapper<Picture> queryWrapper = new QueryWrapper<>();
    if (pictureQueryRequest == null) {
        return queryWrapper;
    }

    Long id = pictureQueryRequest.getId();
    String name = pictureQueryRequest.getName();
    String introduction = pictureQueryRequest.getIntroduction();
    String category = pictureQueryRequest.getCategory();
    List<String> tags = pictureQueryRequest.getTags();
    Long picSize = pictureQueryRequest.getPicSize();
    Integer picWidth = pictureQueryRequest.getPicWidth();
    Integer picHeight = pictureQueryRequest.getPicHeight();
    Double picScale = pictureQueryRequest.getPicScale();
    String picFormat = pictureQueryRequest.getPicFormat();
    String searchText = pictureQueryRequest.getSearchText();
    Long userId = pictureQueryRequest.getUserId();
    String sortField = pictureQueryRequest.getSortField();
```

```
        String sortOrder = pictureQueryRequest.getSortOrder();

        if (StrUtil.isNotBlank(searchText)) {

            queryWrapper.and(qw -> qw.like("name", searchText)
                    .or()
                    .like("introduction", searchText)
            );
        }
        queryWrapper.eq(ObjUtil.isNotEmpty(id), "id", id);
        queryWrapper.eq(ObjUtil.isNotEmpty(userId), "userId", userId);
        queryWrapper.like(StrUtil.isNotBlank(name), "name", name);
        queryWrapper.like(StrUtil.isNotBlank(introduction), "introduction
        queryWrapper.like(StrUtil.isNotBlank(picFormat), "picFormat", pic
        queryWrapper.eq(StrUtil.isNotBlank(category), "category", categor
        queryWrapper.eq(ObjUtil.isNotEmpty(picWidth), "picWidth", picWidt
        queryWrapper.eq(ObjUtil.isNotEmpty(picHeight), "picHeight", picHe
        queryWrapper.eq(ObjUtil.isNotEmpty(picSize), "picSize", picSize);
        queryWrapper.eq(ObjUtil.isNotEmpty(picScale), "picScale", picScal

        if (CollUtil.isNotEmpty(tags)) {
            for (String tag : tags) {
                queryWrapper.like("tags", "\"" + tag + "\"");
            }
        }

        queryWrapper.orderBy(StrUtil.isNotEmpty(sortField), sortOrder.equ
        return queryWrapper;
    }
```

上面的代码中，注意两点：

1. searchText 支持同时从 name 和 introduction 中检索，可以用 queryWrapper 的 or 语法构造查询条件。
2. 由于 tags 在数据库中存储的是 JSON 格式的字符串，如果前端要传多个 tag（必须同时存在才查出），需要遍历 tags 数组，每个标签都使用 like 模糊查询，将这些条件组合在一起。

💡 从 MySQL 5.7 开始，MySQL 提供了 `JSON_CONTAINS` 函数，可以用来检查一个 JSON 数组中是否包含某个元素：

```
SELECT * FROM picture
WHERE JSON_CONTAINS(tags, 'yupi');
```

需要在程序中编写 MyBatis 的自定义 SQL 实现。

3）编写获取图片封装的方法，可以为原有的图片关联创建用户的信息。

获取单个图片封装：

```
@Override
public PictureVO getPictureVO(Picture picture, HttpServletRequest req

    PictureVO pictureVO = PictureVO.objToVo(picture);

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

分页获取图片封装：

```
@Override
public Page<PictureVO> getPictureVOPage(Page<Picture> picturePage, Ht
    List<Picture> pictureList = picturePage.getRecords();
    Page<PictureVO> pictureVOPage = new Page<>(picturePage.getCurrent
    if (CollUtil.isEmpty(pictureList)) {
        return pictureVOPage;
    }

    List<PictureVO> pictureVOList = pictureList.stream().map(PictureV

    Set<Long> userIdSet = pictureList.stream().map(Picture::getUserId
    Map<Long, List<User>> userIdUserListMap = userService.listByIds(u
            .collect(Collectors.groupingBy(User::getId));

    pictureVOList.forEach(pictureVO -> {
        Long userId = pictureVO.getUserId();
        User user = null;
        if (userIdUserListMap.containsKey(userId)) {
            user = userIdUserListMap.get(userId).get(0);
        }
        pictureVO.setUser(userService.getUserVO(user));
    });
    pictureVOPage.setRecords(pictureVOList);
    return pictureVOPage;
}
```

注意，这里我们做了个小优化，不是针对每条数据都查询一次用户，而是先获取到要查询的用户 id 列表，只发送一次查

询用户表的请求，再将查到的值设置到图片对象中。

4）编写图片数据校验方法，用于更新和修改图片时进行判断：

```
@Override
public void validPicture(Picture picture) {
    ThrowUtils.throwIf(picture == null, ErrorCode.PARAMS_ERROR);

    Long id = picture.getId();
    String url = picture.getUrl();
    String introduction = picture.getIntroduction();

    ThrowUtils.throwIf(ObjUtil.isNull(id), ErrorCode.PARAMS_ERROR, "i
    if (StrUtil.isNotBlank(url)) {
        ThrowUtils.throwIf(url.length() > 1024, ErrorCode.PARAMS_ERRO
    }
    if (StrUtil.isNotBlank(introduction)) {
        ThrowUtils.throwIf(introduction.length() > 800, ErrorCode.PAR
    }
}
```

可以根据自己的需要，补充更多校验规则。

3、接口开发

上述功能其实都是样板代码，俗称 “增删改查”。

代码实现比较简单，注意添加对应的权限注解、做好参数校验即可：

```
@PostMapping("/delete")
public BaseResponse<Boolean> deletePicture(@RequestBody DeleteRequest
    if (deleteRequest == null || deleteRequest.getId() <= 0) {
        throw new BusinessException(ErrorCode.PARAMS_ERROR);
    }
    User loginUser = userService.getLoginUser(request);
    long id = deleteRequest.getId();

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

    if (!oldPicture.getUserId().equals(loginUser.getId()) && !userSer
        throw new BusinessException(ErrorCode.NO_AUTH_ERROR);
    }

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

```
@PostMapping("/update")
@AuthCheck(mustRole = UserConstant.ADMIN_ROLE)
public BaseResponse<Boolean> updatePicture(@RequestBody PictureUpdate
    if (pictureUpdateRequest == null || pictureUpdateRequest.getId()
        throw new BusinessException(ErrorCode.PARAMS_ERROR);
    }

    Picture picture = new Picture();
    BeanUtils.copyProperties(pictureUpdateRequest, picture);

    picture.setTags(JSONUtil.toJsonStr(pictureUpdateRequest.getTags()

    pictureService.validPicture(picture);

    long id = pictureUpdateRequest.getId();
    Picture oldPicture = pictureService.getById(id);
    ThrowUtils.throwIf(oldPicture == null, ErrorCode.NOT_FOUND_ERROR)

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


@GetMapping("/get")
@AuthCheck(mustRole = UserConstant.ADMIN_ROLE)
public BaseResponse<Picture> getPictureById(long id, HttpServletReque
    ThrowUtils.throwIf(id <= 0, ErrorCode.PARAMS_ERROR);

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

    return ResultUtils.success(picture);
}


@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);

    return ResultUtils.success(pictureService.getPictureVO(picture, r
}


@PostMapping("/list/page")
@AuthCheck(mustRole = UserConstant.ADMIN_ROLE)
public BaseResponse<Page<Picture>> listPictureByPage(@RequestBody Pic
    long current = pictureQueryRequest.getCurrent();
    long size = pictureQueryRequest.getPageSize();

    Page<Picture> picturePage = pictureService.page(new Page<>(curren
            pictureService.getQueryWrapper(pictureQueryRequest));
    return ResultUtils.success(picturePage);
}


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

```
    long size = pictureQueryRequest.getPageSize();

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

    Page<Picture> picturePage = pictureService.page(new Page<>(curren
            pictureService.getQueryWrapper(pictureQueryRequest));

    return ResultUtils.success(pictureService.getPictureVOPage(pictur
}


@PostMapping("/edit")
public BaseResponse<Boolean> editPicture(@RequestBody PictureEditRequ
    if (pictureEditRequest == null || pictureEditRequest.getId() <= 0
        throw new BusinessException(ErrorCode.PARAMS_ERROR);
    }

    Picture picture = new Picture();
    BeanUtils.copyProperties(pictureEditRequest, picture);

    picture.setTags(JSONUtil.toJsonStr(pictureEditRequest.getTags()))

    picture.setEditTime(new Date());

    pictureService.validPicture(picture);
    User loginUser = userService.getLoginUser(request);

    long id = pictureEditRequest.getId();
    Picture oldPicture = pictureService.getById(id);
    ThrowUtils.throwIf(oldPicture == null, ErrorCode.NOT_FOUND_ERROR)

    if (!oldPicture.getUserId().equals(loginUser.getId()) && !userSer
        throw new BusinessException(ErrorCode.NO_AUTH_ERROR);
    }

    boolean result = pictureService.updateById(picture);
    ThrowUtils.throwIf(!result, ErrorCode.OPERATION_ERROR);
    return ResultUtils.success(true);
}
```

注意，修改和编辑接口中，需要将请求包装对象转换为数据库实体类，便于操作数据库。转换的过程中，由于 tags 的类型不同，需要手动转换：

```
Picture picture = new Picture();
BeanUtils.copyProperties(pictureEditRequest, picture);

picture.setTags(JSONUtil.toJsonStr(pictureEditRequest.getTags()));
```

💡 如果觉得手动转换比较麻烦，也可以用一些工具提供的注解，让类库自动帮你转换。比如 JSON 类型的字段 tags 可以使用 MyBatis Plus 的 `@TableField(typeHandler = JacksonTypeHandler.class)` 标注， 参考文档 。

获取预置标签和分类

根据需求，要支持用户根据标签和分类搜索图片，我们可以给用户列举一些常用的标签和分类，便于筛选。

在项目前期规模不大的时候，我们没必要将标签和分类单独用数据表来维护了，直接在 PictureController 中写一个接口，返回预设的固定数据即可：

```
@GetMapping("/tag_category")
public BaseResponse<PictureTagCategory> listPictureTagCategory() {
    PictureTagCategory pictureTagCategory = new PictureTagCategory();
    List<String> tagList = Arrays.asList("热门", "搞笑", "生活", "高清
    List<String> categoryList = Arrays.asList("模板", "电商", "表情包"
    pictureTagCategory.setTagList(tagList);
    pictureTagCategory.setCategoryList(categoryList);
    return ResultUtils.success(pictureTagCategory);
}
```

随着系统规模和数据不断扩大，可以再改为使用配置中心或数据库动态管理这些数据，或者通过定时任务计算出热门的图片分类和标签。

至此，图片相关的后端接口开发完毕，大家可以按需完善上述代码。

四、前端开发

💡 记得执行一下 openapi 命令生成接口对应的请求代码，每次后端改动时都需要这么做。

图片上传和创建页面

1、新建路由和菜单

首先新建 `AddPicturePage.vue` 页面文件，在 `router/index.ts` 中定义路由：

```
{
  path: '/add_picture',
  name: '创建图片',
  component: AddPicturePage,
```

```
},
```

在 GlobalHeader 组件中补充菜单：

```
{
  key: '/add_picture',
  label: '创建图片',
  title: '创建图片',
}
```

2、图片上传组件

在开发页面前，先开发通用图片上传组件 PictureUpload（放到 components 目录下），支持上传图片并返回图片信息。

1）使用 Ant Design Vue 提供的 upload 组件 快速开发，引入示例代码后就能立刻测试效果了：

2）该组件为受控组件，由父组件（图片创建页面）来管理，需要定义属性：

```
interface Props {
  picture?: API.PictureVO
  onSuccess?: (newPicture: API.PictureVO) => void
}

const props = defineProps<Props>()
```

其中，picture 就是已上传的图片信息，会展示出来；onSuccess 是上传成功后，需要将得到的新图片信息返回给父组件，来更新 picture 的值。

父组件可以这样使用组件：

```
<PictureUpload :picture="picture" :onSuccess="onSuccess" />
```

```
const picture = ref<API.PictureVO>()
const onSuccess = (newPicture: API.PictureVO) => {
  picture.value = newPicture
}
```

3）修改页面结构代码，更改展示的图片地址、文案等：

```
<div>
  <a-upload
    list-type="picture-card"
    :show-upload-list="false"
    :custom-request="handleUpload"
    :before-upload="beforeUpload"
  >
    <img v-if="picture?.url" :src="picture?.url" alt="avatar" />
    <div v-else>
      <loading-outlined v-if="loading"></loading-outlined>
      <plus-outlined v-else></plus-outlined>
      <div>点击或拖拽上传图片</div>
    </div>
  </a-upload>
</div>
```

4）Upload 组件支持上传前校验和自定义请求处理逻辑，我们可以编写对应的函数并传递给组件。

上传前校验函数：

```
const beforeUpload = (file: UploadProps['fileList'][number]) => {
  const isJpgOrPng = file.type === 'image/jpeg' || file.type === 'ima
  if (!isJpgOrPng) {
    message.error('不支持上传该格式的图片，推荐 jpg 或 png')
  }
  const isLt2M = file.size / 1024 / 1024 < 2
  if (!isLt2M) {
    message.error('不能上传超过 2M 的图片')
  }
  return isJpgOrPng && isLt2M
```

```
}
```

上传图片至后端的函数：

```
const loading = ref<boolean>(false)


const handleUpload = async ({ file }: any) => {
  loading.value = true
  try {
    const params = props.picture ? { id: props.picture.id } : {};
    const res = await uploadPictureUsingPost(params, {}, file)
    if (res.data.code === 0 && res.data.data) {
      message.success('图片上传成功')

      props.onSuccess?.(res.data.data)
    } else {
      message.error('图片上传失败，' + res.data.message)
    }
  } catch (error) {
    message.error('图片上传失败')
  } finally {
    loading.value = false
  }
}
```

注意，调用后端上传图片接口时，如果已经有 pictureId，表示对已上传的图片进行更新，需要将该参数也添加到请求中，否则每次都会新增图片记录。

```
const params = props.picture ? { id: props.picture.id } : {};
```

5）最后，可以再按需优化下组件的 CSS 样式，比如让组件的宽高等同于父组件的宽高：

```
.picture-upload :deep(.ant-upload) {
  width: 100% !important;
  height: 100% !important;
  min-height: 152px;
  min-width: 152px;
}

.picture-upload img {
  max-width: 100%;
  max-height: 480px;
}

.ant-upload-select-picture-card i {
```

```
  font-size: 32px;
  color: #999;
}

.ant-upload-select-picture-card .ant-upload-text {
  margin-top: 8px;
  color: #666;
}
```

注意上述代码中，需要使用 `:deep` 语法来修改 Upload 组件内置的样式。

最终，组件的效果如图：

3、开发创建页面

1）先开发页面结构，从上到下分别是页面标题、图片上传组件和表单。表单支持填写名称、简介、分类和标签：

```
<div>
  <h2>创建图片</h2>
  <PictureUpload :picture="picture" :onSuccess="onSuccess" />
  <a-form layout="vertical" :model="pictureForm" @finish="handleSubmi
    <a-form-item label="名称" name="name">
      <a-input v-model:value="pictureForm.name" placeholder="请输入名
    </a-form-item>
    <a-form-item label="简介" name="introduction">
      <a-textarea
        v-model:value="pictureForm.introduction"
        placeholder="请输入简介"
        :rows="2"
        autoSize
        allowClear
      />
    </a-form-item>
    <a-form-item label="分类" name="category">
      <a-auto-complete
        v-model:value="pictureForm.category"
        placeholder="请输入分类"
        allowClear
      />
    </a-form-item>
    <a-form-item label="标签" name="tags">
      <a-select
        v-model:value="pictureForm.tags"
```

```
          mode="tags"
          placeholder="请输入标签"
          allowClear
        />
      </a-form-item>
      <a-form-item>
        <a-button type="primary" html-type="submit">创建</a-button>
      </a-form-item>
    </a-form>
  </div>
```

注意，我们针对不同的数据使用了不同的输入组件：

- 名称： 普通输入框
- 简介： TextArea 多行输入框
- 分类： AutoComplete 输入框 ， 可以自由输入，同时会给出搜索提示
- 标签： Select 选择框 ， 使用 mode="tags" 支持自由输入和多选

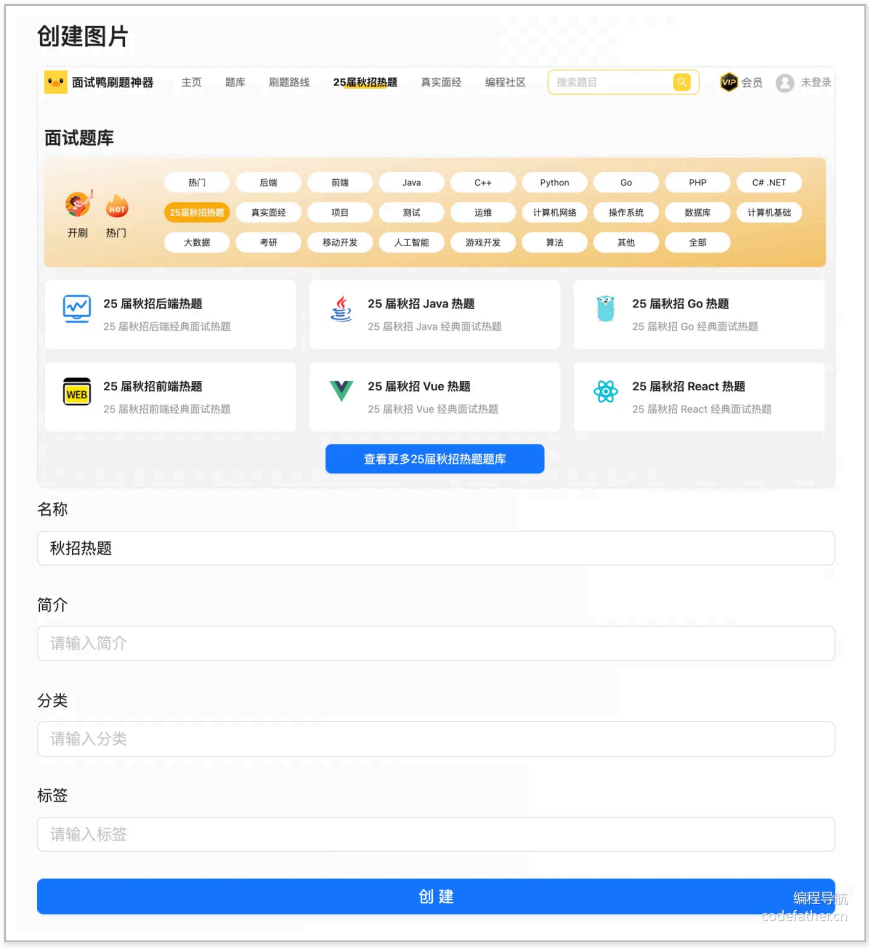
定义变量，接受上传的图片和填写的表单信息：

```
const picture = ref<API.PictureVO>()
const pictureForm = reactive<API.PictureEditRequest>({})
```

可以限制下页面最大宽度，让用户视角更集中：

```
#addPicturePage {
  max-width: 720px;
  margin: 0 auto;
}
```

效果如图：



2）上传图片后，可以将得到的图片信息（比如名称）填充到表单，直接赋值即可：

```
const onSuccess = (newPicture: API.PictureVO) => {
  picture.value = newPicture
  pictureForm.name = newPicture.name
}
```

3）编写提交表单的函数，需要带着上传图片得到的 pictureId 来调用后端 **修改图片接口**（而不是上传接口）：

```
const router = useRouter()


const handleSubmit = async (values: any) => {
  const pictureId = picture.value.id
  if (!pictureId) {
    return
  }
  const res = await editPictureUsingPost({
    id: pictureId,
    ...values,
  })
  if (res.data.code === 0 && res.data.data) {
    message.success('创建成功')
```

```
    router.push({
      path: `/picture/${pictureId}`,
    })
  } else {
    message.error('创建失败，' + res.data.message)
  }
}
```

创建成功后，需要跳转到图片详情页。

4）给分类和标签选择框补充选项数据，注意需要转换为输入组件接受的格式：

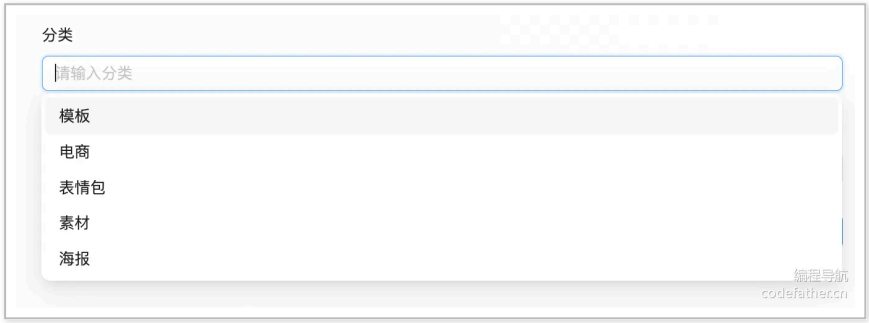
```
const categoryOptions = ref<string[]>([])
const tagOptions = ref<string[]>([])


const getTagCategoryOptions = async () => {
  const res = await listPictureTagCategoryUsingGet()
  if (res.data.code === 0 && res.data.data) {

    tagOptions.value = (res.data.data.tagList ?? []).map((data: strin
      return {
        value: data,
        label: data,
      }
    })
    categoryOptions.value = (res.data.data.categoryList ?? []).map((d
      return {
        value: data,
        label: data,
      }
    })
  } else {
    message.error('加载选项失败，' + res.data.message)
  }
}

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

给组件补充 options 选项：

```
<a-form-item label="分类" name="category">
  <a-auto-complete
    v-model:value="pictureForm.category"
    :options="categoryOptions"
    placeholder="请输入分类"
    allowClear
  />
</a-form-item>
<a-form-item label="标签" name="tags">
```

```
    <a-select
      v-model:value="pictureForm.tags"
      :options="tagOptions"
      mode="tags"
      placeholder="请输入标签"
      allowClear
    />
  </a-form-item>
```

效果如图：



5）最后，可以再做一些优化，比如像某视频网站投稿一样，上传图片后才展示更多表单项。

直接用 v-if 判断即可，picture 为空则表示图片未上传：

```
<a-form v-if="picture" layout="vertical" :model="pictureForm"@finish=
</a-form>
```

效果如图：

图片信息修改

可以直接复用创建页面，在页面后增加 URL 查询参数 `?id=xx` 表示要修改对应 id 的图片。

没传这个参数则表示创建新图片，有 id 的话就直接根据 id 查询到 picture 数据，并且将值填充到表单项中，其他的逻辑和创建图片是兼容的。

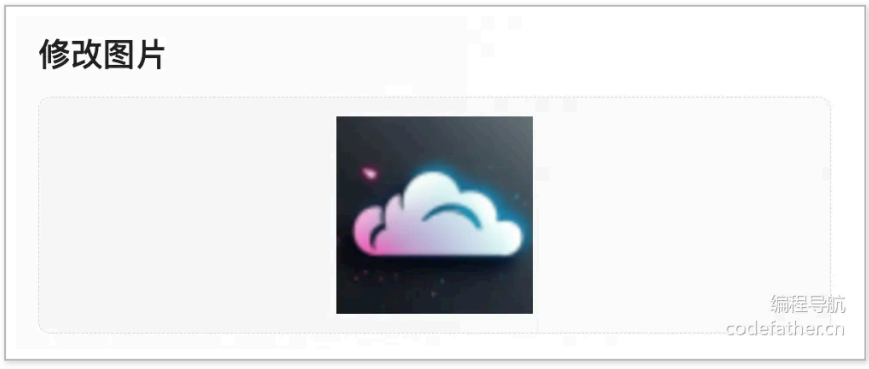
1）根据 id 查询要修改的图片数据，并设置到 picture 变量中：

```
const route = useRoute()


const getOldPicture = async () => {

  const id = route.query?.id
  if (id) {
    const res = await getPictureVoByIdUsingGet({
      id: id,
    })
    if (res.data.code === 0 && res.data.data) {
      const data = res.data.data
      picture.value = data
      pictureForm.name = data.name
      pictureForm.introduction = data.introduction
      pictureForm.category = data.category
      pictureForm.tags = data.tags
    }
  }
}

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

2）优化页面细节，比如设置合适的标题：

```
<h2>
  {{ route.query?.id ? '修改图片' : '创建图片' }}
</h2>
```

效果如图：



图片管理

1、新建路由和菜单

首先新建 `admin/PictureManagePage.vue` 页面文件，在 `router/index.ts` 中定义路由：

```
{
  path: '/admin/pictureManage',
  name: '图片管理',
  component: PictureManagePage,
}
```

在 GlobalHeader 组件中补充菜单：

```
{
  key: '/admin/pictureManage',
  label: '图片管理',
  title: '图片管理',
}
```

由于之前已经编写了权限校验逻辑，地址以 /admin 开头的页面都会限制为仅管理员可见和可用，所以无需再编写额外的权限校验代码。

2、开发管理页面

跟用户管理页面类似，页面的上方是搜索栏，下方是表格，表格需要支持分页。

大多数的内容可以直接复用用户管理页面的代码，可以先复制过来，再进行修改。

1）给表格定义展示的列：

```
const columns = [
  {
    title: 'id',
    dataIndex: 'id',
    width: 80,
  },
  {
    title: '图片',
    dataIndex: 'url',
  },
  {
    title: '名称',
```

```
    dataIndex: 'name',
  },
  {
    title: '简介',
    dataIndex: 'introduction',
    ellipsis: true,
  },
  {
    title: '类型',
    dataIndex: 'category',
  },
  {
    title: '标签',
    dataIndex: 'tags',
  },
  {
    title: '图片信息',
    dataIndex: 'picInfo',
  },
  {
    title: '用户 id',
    dataIndex: 'userId',
    width: 80,
  },
  {
    title: '创建时间',
    dataIndex: 'createTime',
  },
  {
    title: '编辑时间',
    dataIndex: 'editTime',
  },
  {
    title: '操作',
    key: 'action',
  },
]
```

2）从后端获取数据，并支持搜索和分页：

```
const dataList = ref([])
const total = ref(0)


const searchParams = reactive<API.PictureQueryRequest>({
  current: 1,
  pageSize: 10,
  sortField: 'createTime',
  sortOrder: 'descend',
})


const pagination = computed(() => {
  return {
    current: searchParams.current ?? 1,
    pageSize: searchParams.pageSize ?? 10,
    total: total.value,
    showSizeChanger: true,
    showTotal: (total) => `共 ${total} 条`,
  }
```

```
})


const fetchData = async () => {
  const res = await listPictureByPageUsingPost({
    ...searchParams,
  })
  if (res.data.data) {
    dataList.value = res.data.data.records ?? []
    total.value = res.data.data.total ?? 0
  } else {
    message.error('获取数据失败，' + res.data.message)
  }
}


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


const doSearch = () => {

  searchParams.current = 1
  fetchData()
}


const doTableChange = (page: any) => {
  searchParams.current = page.current
  searchParams.pageSize = page.pageSize
  fetchData()
}
```

注意：

1. 跟用户管理页面不同的是，默认按照创建时间降序展示图片，最新的图片会被优先看到。
2. 获取数据时，调用的是仅管理员可用的查询接口 listPictureByPageUsingPost（不是给用户使用的查询包装类接口）

3）自定义列的展示，比如图片、标签、图片解析信息、创建时间、编辑时间等：

```
<template #bodyCell="{ column, record }">
  <template v-if="column.dataIndex === 'url'">
    <a-image :src="record.url" :width="120" />
  </template>
  <!-- 标签 -->
  <template v-if="column.dataIndex === 'tags'">
    <a-space wrap>
      <a-tag v-for="tag in JSON.parse(record.tags || '[]')" :key="tag
    </a-space>
```

```
    </template>
    <!-- 图片信息 -->
    <template v-if="column.dataIndex === 'picInfo'">
      <div>格式：{{ record.picFormat }}</div>
      <div>宽度：{{ record.picWidth }}</div>
      <div>高度：{{ record.picHeight }}</div>
      <div>宽高比：{{ record.picScale }}</div>
      <div>大小：{{ (record.picSize / 1024).toFixed(2) }}KB</div>
    </template>
    <template v-else-if="column.dataIndex === 'createTime'">
      {{ dayjs(record.createTime).format('YYYY-MM-DD HH:mm:ss') }}
    </template>
    <template v-else-if="column.dataIndex === 'editTime'">
      {{ dayjs(record.editTime).format('YYYY-MM-DD HH:mm:ss') }}
    </template>
    <template v-else-if="column.key === 'action'">
      <a-button type="link" danger @click="doDelete(record.id)">删除</a
    </template>
  </template>
```

注意，由于后端返回的 tags 类型是字符串，需要用 `JSON.parse` 转为 JS 数组。

4）开发搜索表单，支持按照关键词、类型和标签搜索：

```
  <a-form layout="inline" :model="searchParams" @finish="doSearch">
    <a-form-item label="关键词" name="searchText">
      <a-input
        v-model:value="searchParams.searchText"
        placeholder="从名称和简介搜索"
        allow-clear
      />
    </a-form-item>
    <a-form-item label="类型" name="category">
      <a-input v-model:value="searchParams.category" placeholder="请输入
    </a-form-item>
    <a-form-item label="标签" name="tags">
      <a-select
        v-model:value="searchParams.tags"
        mode="tags"
        placeholder="请输入标签"

        allow-clear
      />
    </a-form-item>
    <a-form-item>
      <a-button type="primary" html-type="submit">搜索</a-button>
    </a-form-item>
  </a-form>
```

5）补充操作按钮。

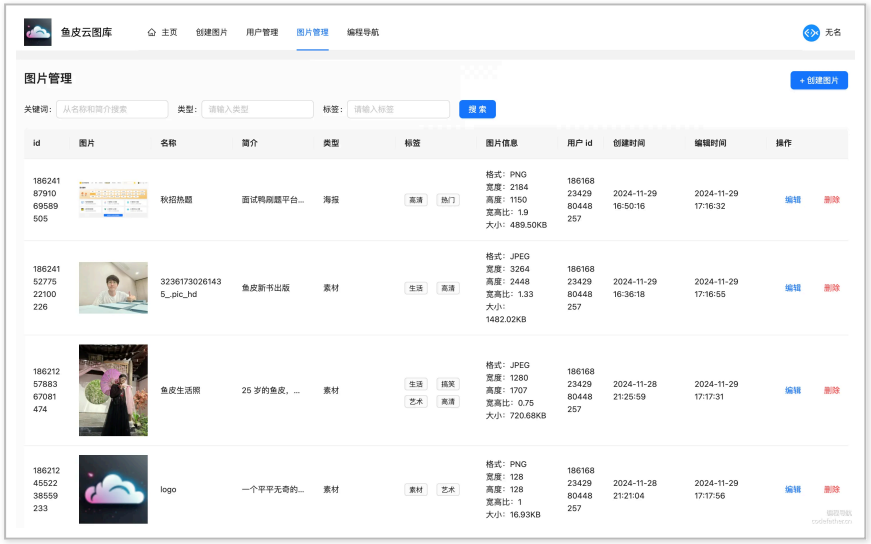
可以在搜索表单上新增一行，展示标题和创建图片按钮，点击按钮会打开创建图片页面：

```
<a-flex justify="space-between">
  <h2>图片管理</h2>
  <a-button type="primary" href="/add_picture" target="_blank">+ 创建
</a-flex>
```

在表格操作列中，可以补充编辑按钮，点击后打开编辑图片页面：

```
<a-space>
  <a-button type="link" :href="`/add_picture?id=${record.id}`" target
  <a-button type="link" danger @click="doDelete(record.id)">删除</a-b
</a-space>
```

最终页面效果如图：



💡 如果觉得表格的列在窄屏下会受到挤压，可以给 table 组件增加属性 `:scroll="{ x: 'max-content' }"` ，使表格支持横向滚动。

图片列表页（主页）

整个页面布局从上到下依次为：搜索框、分类选项、标签选项、图片列表、分页操作栏。

1、分页列表开发

1）使用 响应式的 List 组件 ，会根据屏幕大小调整每行展示的图片数：

```
<!-- 图片列表 -->
<a-list
  :grid="{ gutter: 16, xs: 1, sm: 2, md: 3, lg: 4, xl: 5, xxl: 6 }"
  :data-source="dataList"
  :pagination="pagination"
  :loading="loading"
>
  <template #renderItem="{ item: picture }">
    <a-list-item>
      <!-- 单张图片 -->
    </a-list-item>
  </template>
</a-list>
```

2）定义数据、搜索条件、分页参数，以及获取数据的函数：

```
const dataList = ref([])
const total = ref(0)
const loading = ref(true)


const searchParams = reactive<API.PictureQueryRequest>({
  current: 1,
  pageSize: 12,
  sortField: 'createTime',
  sortOrder: 'descend',
})


const pagination = computed(() => {
  return {
    current: searchParams.current ?? 1,
    pageSize: searchParams.pageSize ?? 10,
    total: total.value,

    onChange: (page, pageSize) => {
      searchParams.current = page
      searchParams.pageSize = pageSize
      fetchData()
    },
  }
})


const fetchData = async () => {
  loading.value = true
  const res = await listPictureVoByPageUsingPost(searchParams)
  if (res.data.data) {
    dataList.value = res.data.data.records ?? []
    total.value = res.data.data.total ?? 0
  } else {
```

```
      message.error('获取数据失败，' + res.data.message)
    }
    loading.value = false
  }


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

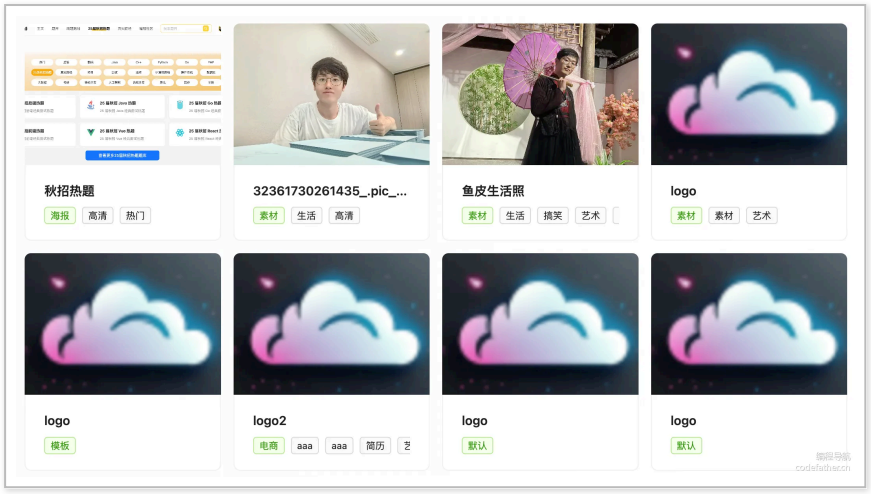
注意，上述代码中，鱼皮故意移除了数据的总数和切换每页数量的选择器，这些信息没必要对用户展示，可以让页面更精简。

3）展示图片，可以使用 Card 组件：

```
<a-list-item>
  <!-- 单张图片 -->
  <a-card hoverable>
    <template #cover>
      <img

        :alt="picture.name"
        :src="picture.url"
      />
    </template>
    <a-card-meta :title="picture.name">
      <template #description>
        <a-flex>
          <a-tag color="green">
            {{ picture.category ?? '默认' }}
          </a-tag>
          <a-tag v-for="tag in picture.tags" :key="tag">
            {{ tag }}
          </a-tag>
        </a-flex>
      </template>
    </a-card-meta>
  </a-card>
</a-list-item>
```

注意，由于图片的宽高都是不同的，为了防止页面“参差不齐”，给所有图片统一设置相同的高度、并使用 `object-fit: cover` 优化图片的展示效果，不会受到挤压。

效果如图：



2、搜索能力开发

1）开发搜索框，使用 Input.Search 组件，先定义页面结构：

```
<!-- 搜索框 -->
<div>
  <a-input-search
    placeholder="从海量图片中搜索"
    v-model:value="searchParams.searchText"
    enter-button="搜索"
    size="large"
    @search="doSearch"
  />
</div>
```

2）点击搜索按钮时，触发搜索事件：

```
const doSearch = () => {

  searchParams.current = 1
  fetchData()
}
```

3）优化 CSS 样式：

```
#homePage .search-bar {
  max-width: 480px;
  margin: 0 auto 16px;
}
```

3、分类和标签筛选能力

1）先开发页面结构。分类仅支持单选，可以使用 Tabs 组件；标签支持多选，可以使用 标签选择器组件 。

为了支持取消选中的分类，可以新增一个 “全部” 分类，页面代码如下：

```
<!-- 分类 + 标签 -->
<a-tabs v-model:activeKey="selectedCategory" @change="doSearch">
  <a-tab-pane key="all" tab="全部" />
  <a-tab-pane v-for="category in categoryList" :key="category" :tab="
</a-tabs>
<div>
  <span>标签：</span>
  <a-space :size="[0, 8]" wrap>
    <a-checkable-tag
      v-for="(tag, index) in tagList"
      :key="tag"
      v-model:checked="selectedTagList[index]"
      @change="doSearch"
    >
      {{ tag }}
    </a-checkable-tag>
  </a-space>
</div>
```

2）定义可选分类 / 标签列表、选中的分类 / 标签，并获取分类和标签选项：

```
const categoryList = ref<string[]>([])
const selectedCategory = ref<string>('all')
const tagList = ref<string[]>([])
const selectedTagList = ref<string[]>([])


const getTagCategoryOptions = async () => {
  const res = await listPictureTagCategoryUsingGet()
  if (res.data.code === 0 && res.data.data) {

    categoryList.value = res.data.data.categoryList ?? []
    tagList.value = res.data.data.tagList ?? []
  } else {
    message.error('加载分类标签失败，' + res.data.message)
  }
}

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

3）在搜索时，需要将选中的分类和标签转换为对应的请求参数：

```
const fetchData = async () => {
  loading.value = true

  const params = {
    ...searchParams,
    tags: [],
  }
  if (selectedCategory.value !== 'all') {
    params.category = selectedCategory.value
  }
  selectedTagList.value.forEach((useTag, index) => {
    if (useTag) {
      params.tags.push(tagList.value[index])
    }
  })
  const res = await listPictureVoByPageUsingPost(params)
  if (res.data.data) {
    dataList.value = res.data.data.records ?? []
    total.value = res.data.data.total ?? 0
  } else {
    message.error('获取数据失败，' + res.data.message)
  }
  loading.value = false
}
```

4）给图片卡片绑定点击时间，点击图片后会跳转到图片详情页。

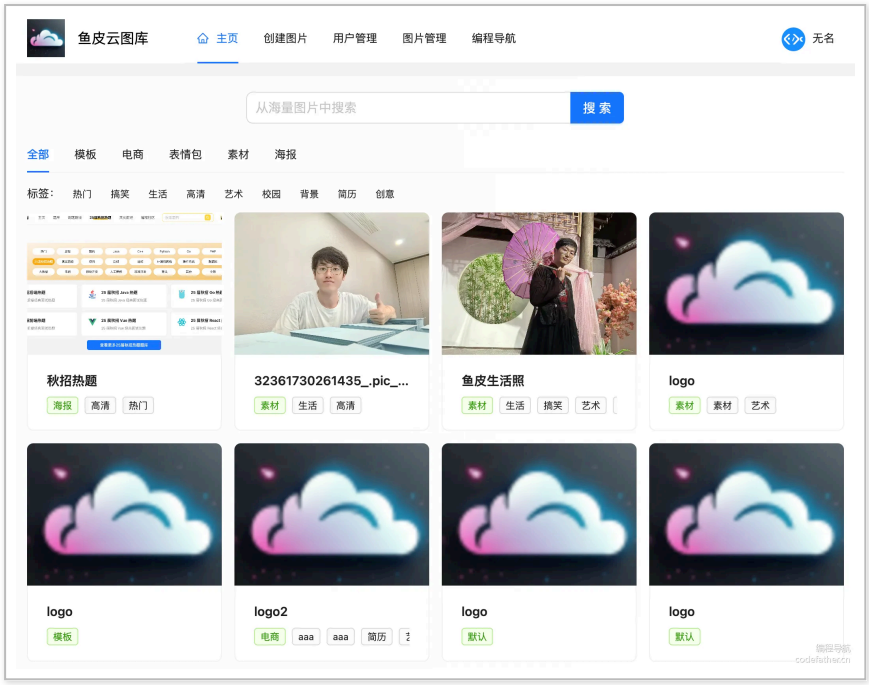
修改页面：

```
<!-- 单张图片 -->
<a-card hoverable @click="doClickPicture(picture)">
```

补充跳转事件：

```
const router = useRouter()

const doClickPicture = (picture) => {
  router.push({
    path: `/picture/${picture.id}`,
  })
}
```

最终的页面效果如图，是不是有内味儿了？



4、扩展思路

1）如果想保留当前用户之前的搜索参数，可以在修改搜索参数时，同步修改 URL 查询参数；并且在页面刷新时，将 URL 查询参数设置到搜索参数中。

2）优化图片列表的展示效果。可以用自适应瀑布流 + 下拉加载的方式实现，网上有很多的插件。但是注意，为了防止下拉加载时获得重复的数据，后端最好改为使用游标查询。

3）优化图片本身的展示效果。可以将卡片的额外信息折叠到图片内部，实现悬浮遮罩的效果，用 CSS 就能实现了，类似下图：

图片详情页

1、新建路由

首先新建 `PictureDetailPage.vue` 页面文件，在 `router/index.ts` 中定义路由：

```
{
  path: '/picture/:id',
  name: '图片详情',
  component: PictureDetailPage,
  props: true,
}
```

由于图片详情页要展示的图片是根据 id 而变化的，所以此处使用动态路由。在页面中可以使用 props 获取到动态的参数：

```
const props = defineProps<{
  id: string | number
```

```
}>()
```

然后就可以在页面加载时，通过 id 调用接口来获取到图片：

```
const picture = ref<API.PictureVO>({})


const fetchPictureDetail = async () => {
  try {
    const res = await getPictureVoByIdUsingGet({
      id: props.id,
    })
    if (res.data.code === 0 && res.data.data) {
      picture.value = res.data.data
    } else {
      message.error('获取图片详情失败，' + res.data.message)
    }
  } catch (e: any) {
    message.error('获取图片详情失败：' + e.message)
  }
}

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

2、页面开发

1）采用一行两列的响应式布局结构，左边使用 图片浏览组件 展示图片，右边使用 描述列表组件 展示图片信息。

```
<a-row :gutter="[16, 16]">
  <!-- 图片展示区 -->
  <a-col :sm="24" :md="16" :xl="18">
    <a-card title="图片预览">
      <a-image

        :src="picture.url"
      />
    </a-card>
  </a-col>
  <!-- 图片信息区 -->
  <a-col :sm="24" :md="8" :xl="6">
    <a-card title="图片信息">
      <a-descriptions :column="1">
        <a-descriptions-item label="作者">
          <a-space>
            <a-avatar :size="24" :src="picture.user?.userAvatar" />
            <div>{{ picture.user?.userName }}</div>
          </a-space>
        </a-descriptions-item>
        <a-descriptions-item label="名称">
          {{ picture.name ?? '未命名' }}
        </a-descriptions-item>
        <a-descriptions-item label="简介">
```

```
          {{ picture.introduction ?? '-' }}
        </a-descriptions-item>
        <a-descriptions-item label="分类">
          {{ picture.category ?? '默认' }}
        </a-descriptions-item>
        <a-descriptions-item label="标签">
          <a-tag v-for="tag in picture.tags" :key="tag">
            {{ tag }}
          </a-tag>
        </a-descriptions-item>
        <a-descriptions-item label="格式">
          {{ picture.picFormat ?? '-' }}
        </a-descriptions-item>
        <a-descriptions-item label="宽度">
          {{ picture.picWidth ?? '-' }}
        </a-descriptions-item>
        <a-descriptions-item label="高度">
          {{ picture.picHeight ?? '-' }}
        </a-descriptions-item>
        <a-descriptions-item label="宽高比">
          {{ picture.picScale ?? '-' }}
        </a-descriptions-item>
        <a-descriptions-item label="大小">
          {{ formatSize(picture.picSize) }}
        </a-descriptions-item>
      </a-descriptions>
    </a-card>
  </a-col>
</a-row>
```

注意，为了防止图片过高，给图片设置最大高度；并且设置 `object-fit: contain` 让图片能够完整展示。

可以将计算图片尺寸的代码移动到 `utils/index.ts` 中，作为工具类，可在其他位置复用：

```
export const formatSize = (size?: number) => {
  if (!size) return '未知'
  if (size < 1024) return size + ' B'
  if (size < 1024 * 1024) return (size / 1024).toFixed(2) + ' KB'
  return (size / (1024 * 1024)).toFixed(2) + ' MB'
}
```

2）在描述列表下补充操作按钮，对于图片上传者或管理员，可以编辑和删除图片：

```
<a-space wrap>
  <a-button v-if="canEdit" type="default" @click="doEdit">
    编辑
    <template #icon>
```

```
      <EditOutlined />
    </template>
  </a-button>
  <a-button v-if="canEdit" danger @click="doDelete">
    删除
    <template #icon>
      <DeleteOutlined />
    </template>
  </a-button>
</a-space>
```

编写权限判断逻辑，canEdit 的值为 true 表示有编辑和删除权限：

```
const loginUserStore = useLoginUserStore()

const canEdit = computed(() => {
  const loginUser = loginUserStore.loginUser;

  if (!loginUser.id) {
    return false
  }

  const user = picture.value.user || {}
  return loginUser.id === user.id || loginUser.userRole === 'admin'
})
```

编写对应的事件：

```
const doEdit = () => {
  router.push('/add_picture?id=' + picture.value.id)
}

const doDelete = async () => {
  const id = picture.value.id
  if (!id) {
    return
  }
  const res = await deletePictureUsingPost({ id })
  if (res.data.code === 0) {
    message.success('删除成功')
  } else {
    message.error('删除失败')
  }
}
```

图片下载

最后，我们来开发图片下载功能。之前方案设计中提到，为了实现方便，我们可以直接从对象存储的 URL 下载图片，无

需经过后端。

前端可以使用 `file-saver` 库，下载指定 URL 或者后端返回的 blob 内容为文件。

1）先安装 `file-saver` 库：

```
npm install file-saver
npm i --save-dev @types/file-saver
```

2）在图片详情页的操作区域补充下载按钮：

```
<a-button type="primary" @click="doDownload">
  免费下载
  <template #icon>
    <DownloadOutlined />
  </template>
</a-button>
```

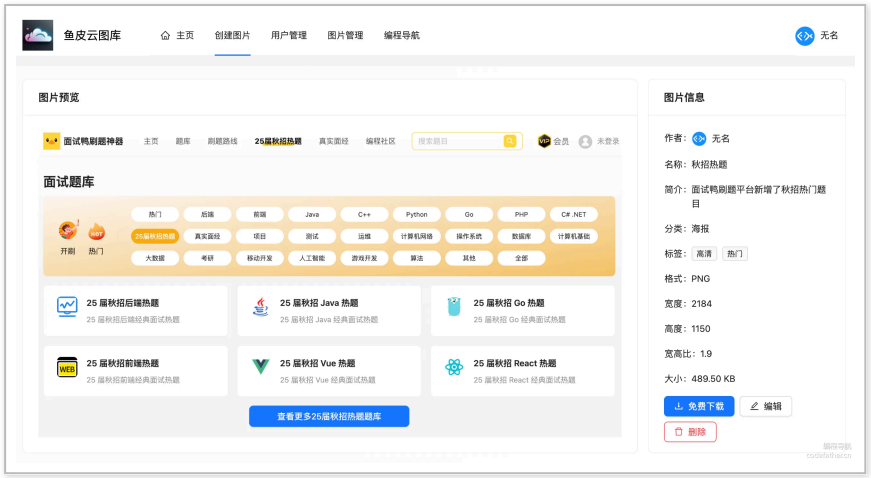
3）定义下载事件：

```
const doDownload = () => {
  downloadImage(picture.value.url)
}
```

4）在 `utils/index.ts` 中补充下载图片的工具函数：

```
export function downloadImage(url?: string, fileName?: string) {
  if (!url) {
    return
  }
  saveAs(url, fileName)
}
```

最终页面效果如图：



五、扩展思路

1）使用数据库表动态管理网站的标签和分类，前端也可以添加对应的管理界面，即标签管理和分类管理。

2）可以使用定时任务或者标签表增加 “使用数” 字段的方式统计标签的使用次数，给主页展示出热门标签，帮助用户更快地找到需要的内容。（分类同理）

3）可以在图片上传成功后，利用 AI 自动补充简介、标签和分类（较难）。