图片管理系统 (PhotoMS) 软件设计说明书

1. 引言

1.1 编写目的

本设计说明书旨在为“图片管理网站”项目提供详细的架构设计、功能模块划分及数据库设计方案，作为后续开发（Coding）、测试及部署的指导依据。

1.2 项目背景

本项目为《BS体系软件设计》课程大作业。目标是构建一个基于 B/S 架构的图片管理平台，支持多端（PC/Mobile）访问，具备图片上传、管理、EXIF 自动解析、AI 智能分析及 LLM 对话检索（MCP）等功能。

2. 系统总体设计

2.1 技术架构选型

本项目采用前后端分离架构。前端使用 **TypeScript** 保证交互逻辑的健壮性，后端使用 **Go (Golang)** 利用其静态强类型和高性能并发特性。

- **前端 (Client)**:
  - **核心框架**: React 18 + TypeScript
  - **构建工具**: Vite
  - **UI 框架**: Tailwind CSS (移动端优先策略) + Shadcn/UI (基于 Radix 的组件库)
  - **状态管理**: Zustand (轻量级全局状态管理)
  - **网络请求**: Axios + TanStack Query (管理服务端状态与缓存)
- **后端 (Server)**:
  - **语言**: Go (Golang) 1.21+
  - **Web 框架**: **Gin** (轻量级、高性能，生态丰富，适合课程作业快速开发)
  - **ORM 框架**: **GORM** (Go 语言中事实标准的 ORM，支持 MySQL，开发效率高)
  - **图像处理**:
    - `disintegration/imaging`：纯 Go 实现的图像处理库（缩放、裁剪），避免了 CGO 带来的 Docker 部署复杂性。
    - `rwcarlsen/goexif`：用于提取 EXIF 元数据。
- **数据库 (Database)**:
  - **主库**: MySQL 8.0 (存储元数据)
- **基础设施**:
  - **容器化**: Docker (多阶段构建) + Docker Compose

3. 功能模块设计

3.1 用户认证与安全模块 (Auth & Security)

- **注册流程**:
  - **输入验证**: 后端使用正则表达式严格校验 Email 格式及密码复杂度（至少包含字母和数字，长度 >6）。
  - **唯一性检查**: 利用数据库 `UNIQUE` 索引处理并发注册时的冲突。
  - **密码存储**: 使用 `bcrypt` 算法进行加盐哈希（Cost=10），严禁明文存储。
- **鉴权机制**:
  - 采用 **JWT (JSON Web Token)** 标准。
  - **Payload 设计**: 包含 `UserID`、`Role`（用于未来扩展）和 `Exp`（过期时间，设为 24 小时）。
  - **中间件**: 实现 Gin Middleware，拦截非公开接口，验证 Header 中的 `Authorization: Bearer <token>`，校验失败直接返回 HTTP 401。
- **安全策略**:
  - **CORS**: 配置允许的前端域名白名单。
  - **Rate Limiting**: 针对登录接口增加 IP 维度的限流，防止暴力破解。

3.2 图片全生命周期管理模块 (Image Lifecycle)

- **上传子系统**:
  - **文件校验**: 不仅校验文件扩展名，还需读取文件头的前几个字节（Magic Number）校验 MIME 类型，防止伪装成图片的可执行文件上传。
  - **存储策略**:
    - 物理路径采用 `/uploads/{yyyy}/{mm}/{dd}/{uuid}.ext` 结构，避免单文件夹下文件过多导致文件系统性能下降。
    - 文件名使用 UUID v4 重命名，防止中文乱码及文件名冲突。
  - **秒传机制**: 上传前计算文件 SHA-256 哈希，查询数据库是否存在相同 Hash。若存在，直接复用文件路径，仅新增数据库引用记录。
- **EXIF 解析与同步**:
  - **关键字段**: 解析 `DateTimeOriginal`（拍摄时间），`Make/Model`（设备），`GPS`（经纬度），`FNumber`（光圈），`ExposureTime`（快门）。
  - **数据清洗**: 将 GPS 的度分秒格式转换为十进制经纬度，若无拍摄时间则回退使用文件创建时间。
- **图片处理**:
  - **缩略图生成**: 上传成功后，异步生成宽 300px 的 WebP 格式缩略图（体积比 JPG 小 30% 以上），用于列表页快速加载。
  - **编辑功能**: 前端提供裁剪框（`react-image-crop`）和滤镜参数（CSS filter），用户确认后，将参数传回后端或在前端 Canvas 处理完后作为新图片上传（非破坏性编辑）。

3.3 检索与展示模块 (Search & Visualization)

- **查询引擎**:
  - **多维过滤**: 支持 AND 组合查询，如 `TakenTime > '2023-01-01' AND Tag IN ('Landscape')`。

    - **分页策略**: 采用基于游标（Cursor-based）或标准 Offset 分页。考虑到作业规模，使用标准 Offset 分页即可。
- **前端交互优化**:
    - **虚拟滚动 (Virtual Scrolling)**: 当图片数量超过 1000 张时，仅渲染视口内的 DOM 节点，保证页面流畅度。
    - **懒加载 (Lazy Load)**: 图片进入视口前显示占位色块或低清模糊图。
- **移动端适配**:
    - **响应式布局**:
        - Desktop: 4-5 列瀑布流。
        - Tablet: 3 列。
        - Mobile: 1-2 列（根据屏幕宽度动态计算）。
    - **手势操作**: 支持左右滑动切换大图查看。

3.4 智能增强模块 (AI & MCP)

- **AI 异步分析**:
    - **Worker Pool 模式**: 后端维护一个带缓冲的 Channel（`chan PhotoTask`）和一组 Worker Goroutine。
    - **处理流程**: 图片上传完成 -> 发送任务 ID 到 Channel -> Worker 获取任务 -> 调用 Vision API -> 写入 Tags 表。避免外部 API 延迟阻塞上传接口。
- **MCP (Model Context Protocol) 服务端**:
    - **接口定义**: 实现符合 MCP 规范的工具描述端点。
    - **Schema**: 暴露 `search_photos` 工具，接受 `query`（SQL where 子句的自然语言描述）或结构化参数 `{"keyword": "cat", "start_date": "..."}`。

4. 数据库设计

使用 GORM Struct 模型定义，自动迁移生成表结构。

4.1 ERD 关系

User (1) <--> (N) Photo <--> (N) Tag, (1) <--> (1) Exif

4.2 数据表定义 (Go Struct 映射)

User (用户表)

```
type User struct {
    gorm.Model
    Username string `gorm:"uniqueIndex;type:varchar(100);not null"`
    Password string `gorm:"type:varchar(255);not null"` // Bcrypt hash
    Email    string `gorm:"uniqueIndex;type:varchar(100);not null"`
    Photos   []Photo
}
```

Photo (图片表)

```
type Photo struct {
    ID          uint           `gorm:"primaryKey"`
    CreatedAt   time.Time
    UpdatedAt   time.Time
    DeletedAt   gorm.DeletedAt `gorm:"index"`
    UserID      uint           `gorm:"index;not null"`
    Title       string         `gorm:"type:varchar(255)"`
    Description string         `gorm:"type:text"`
    FileName    string         `gorm:"type:varchar(255);not null"`
    Path        string         `gorm:"type:varchar(512);not null"` // 原图存储路径
    ThumbPath   string         `gorm:"type:varchar(512);not null"` // 缩略图路径
    Hash        string         `gorm:"index;type:char(64);not null"` // SHA-256
    Size        int64          // 文件大小
    MimeType    string         `gorm:"type:varchar(50)"`
    ExifData    ExifData       // Has One 关系
    Tags        []Tag          `gorm:"many2many:photo_tags;"`
}
```

ExifData (EXIF信息表)

```
type ExifData struct {
    PhotoID      uint      `gorm:"primaryKey"`
    CameraMake   string    `gorm:"type:varchar(100)"`
    CameraModel  string    `gorm:"type:varchar(100)"`
    LensModel    string    `gorm:"type:varchar(100)"`
    ISO          int
    Aperture     float64   // F值
    ShutterSpeed string    // 快门速度字符串，如 "1/100"
    FocalLength  float64
    Latitude     float64   // 纬度
    Longitude    float64   // 经度
    TakenAt      time.Time `gorm:"index"` // 拍摄时间，这是查询热点，必须加索引
}
```

Tag (标签表)

```
type Tag struct {
    ID   uint   `gorm:"primaryKey"`
    Name string `gorm:"uniqueIndex;type:varchar(50);not null"`
    Type string `gorm:"type:enum('USER','AI');default:'USER'"` // 区分来源
}
```

5. 接口设计 (API Design)

遵循 RESTful 风格。

方法	路径	说明	请求体/参数	响应
POST	/api/v1/auth/register	注册	{username, email, password}	{id, token}
POST	/api/v1/auth/login	登录	{email, password}	{token, user}

POST	/api/v1/photos	上传图片	Multipart: file	{id, url}
GET	/api/v1/photos	图片列表	?page=1&limit=20&tag=风景&start_date=...	{data:[], meta:{total}}
GET	/api/v1/photos/:id	详情	-	{photo_obj}
PUT	/api/v1/photos/:id	编辑元数据	{title, description, tags:[]}	{status}
DELETE	/api/v1/photos/:id	软删除	-	{status}
POST	/api/v1/mcp/query	MCP检索	{prompt: "string"}	{results: []}

6. 详细设计与实现要点

6.1 架构分层与数据流向设计

为保证系统的可维护性与扩展性，后端采用标准的**三层架构 (Controller-Service-Repository)** 模式，层与层之间通过接口定义边界。

- **Controller 层 (接口适配)**:
  - 负责 HTTP 协议解析、参数绑定 (Parsing & Binding) 与基础校验。
  - 负责将 HTTP 响应转换为统一的 JSON 格式。
  - **连接点**: 仅调用 Service 层接口，不包含任何 SQL 逻辑或复杂的业务判断。
- **Service 层 (业务逻辑)**:
  - 系统的核心大脑，负责事务控制 (Transaction Management)。
  - **编排逻辑**: 例如“上传图片”服务，需要依次调用文件存储模块、EXIF 解析模块、数据库仓库模块，并触发 AI 分析任务。
  - **连接点**: 向下调用 Repository 层进行数据持久化，向旁调用 AI Service 或 File Service。
- **Repository 层 (数据访问)**:
  - 封装 GORM 操作，对 Service 层屏蔽底层数据库差异（MySQL）。
  - 只负责 CRUD 操作，不包含业务逻辑。

6.2 核心业务流程的串联与实现

6.2.1 高并发图片上传流水线

上传模块需平衡响应速度与处理复杂度，采用**同步与异步结合**的策略。

1. **同步阶段 (即时响应)**:
   - **哈希计算与去重**: 文件流进入内存后，先计算 SHA-256。若命中秒传，立即建立 DB 关联并返回，不再进行 IO 操作。

- **IO 落盘与元数据提取**: 若为新文件，Service 层协调将文件流写入本地磁盘（或 OSS），并同步调用 `goexif` 提取关键 EXIF 信息。
- **事务提交**: 将文件路径、基础信息、EXIF 数据在一个 DB 事务中写入 Photo 表。成功后立即向前端返回 HTTP 201。

2. **异步阶段 (后台处理)**:
   - **生产者-消费者模型**: 上传成功后，Service 层将 PhotoID 投递到 Go Channel (任务队列)。
   - **缩略图生成**: 后台 Goroutine 监听队列，调用 `imaging` 库处理图片，生成 WebP 缩略图并更新 DB 字段。
   - **AI 智能打标**: 独立的 Goroutine 池负责调用外部 Vision API。这里需要实现**并发控制（Semaphore）**，防止大量上传瞬间耗尽 API Quota 或导致服务器 OOM。

6.2.2 前后端状态同步与交互

前端 React 应用不再仅作为展示层，而是通过状态管理深度参与业务流。

- **React Query 作为服务端状态桥梁**:
  - 所有 GET 请求的数据（如图片列表、详情）视为“服务端状态”，由 React Query 负责缓存、去重和后台刷新。
  - **乐观更新 (Optimistic Updates)**: 在用户修改图片标题或删除图片时，前端先直接修改内存中的缓存数据（UI 立即变更），再发送 API 请求。若 API 失败，则自动回滚 UI，从而提供“零延迟”的操作体验。
- **鉴权状态流转**:
  - JWT Token 存储在 LocalStorage 中。
  - Axios 拦截器负责在请求头注入 Token，并监听 HTTP 401 响应。一旦检测到 401，触发全局登出 Action，清理 Zustand Store 中的用户信息并重定向至登录页。

6.3 智能模块集成方案 (MCP & AI)

6.3.1 AI 服务的解耦设计

考虑到 AI 模型可能更换（如从 OpenAI 换为本地部署模型），后端需定义统一的 `AIService` 接口。

- **接口定义**: `Analyze(imageStream) -> []Tags` 。
- **容错处理**: 网络调用需设置超时（Timeout）和重试（Retry）机制。若 AI 服务不可用，不应影响主流程，仅在日志中记录错误，并标记该图片为“未分析”状态，允许后续通过定时任务补偿处理。

6.3.2 MCP 协议的转换逻辑

MCP 接口充当自然语言与结构化查询（SQL）之间的**翻译层**。

- **意图识别**: 接收用户的自然语言 Prompt（如“找一下去年在海边的照片”）。
- **Prompt Engineering**: 将 Prompt 包装后发给 LLM，要求 LLM 返回预定义的 JSON Schema（包含时间范围、地点关键词、物体标签）。
- **动态 SQL 构建**: 后端解析 LLM 返回的 JSON，通过 GORM 的 `Where` 条件动态拼接 SQL 语句，最终返回符合语义的图片列表。