图片管理系统 (PhotoMS) 软件设计说明书

1. 引言
   1.1 编写目的
   1.2 项目背景
2. 系统总体设计
   2.1 技术架构选型
3. 功能模块设计 (详细版)
   3.1 用户认证与安全模块 (Auth & Security)
   3.2 图片全生命周期管理模块 (Image Lifecycle)
   3.3 检索与展示模块 (Search & Visualization)
   3.4 智能增强模块 (AI & MCP)
4. 数据库设计
   4.1 集合设计 (Collections)
   4.2 文档定义 (Go Struct 映射 BSON)
5. 接口设计 (API Design)
6. 详细设计与实现要点
   6.1 架构分层与数据流向设计
   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 (IO 流式优化版)

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、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)
  - **数据库驱动**: **MongoDB Go Driver** (官方驱动 `go.mongodb.org/mongo-driver`，提供原生 BSON 支持与高性能连接池，适合处理非结构化图片元数据)
  - **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**: MongoDB 6.0+ (文档型数据库，利用其 Schema-less 特性灵活存储 EXIF 和 AI 标签)
- **基础设施**:
  - **容器化**: Docker (多阶段构建) + Docker Compose

3. 功能模块设计 (详细版)

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 MongoDB 的 `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`（Hex String）、`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**: 支持组合查询，如 `TakenTime > '2023-01-01' AND tags.name IN ('Landscape')`。
  - **分页策略**: 采用 `skip/limit` 或基于 `_id` 的游标分页。
- **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 –> **原子更新 (Atomic Update)** Tags 字段。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`（MongoDB Filter 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 MongoDB 文档模型设计。相比于关系型数据库，利用**嵌入式文档 (Embedded Documents)** 特性，将 EXIF 和 Tags 直接存储在图片文档中，减少联表查询，提高读取性能。

4.1 集合设计 (Collections)

主要包含两个集合：`users` 和 `photos`。

4.2 文档定义 (Go Struct 映射 BSON)

User (用户文档)

```
import "go.mongodb.org/mongo-driver/bson/primitive"

type User struct {
    ID        primitive.ObjectID `bson:"_id,omitempty" json:"id"`
    Username  string             `bson:"username" json:"username"` // 唯一索引
    Password  string             `bson:"password" json:"-"`        // Bcrypt hash
    Email     string             `bson:"email" json:"email"`       // 唯一索引
    CreatedAt primitive.DateTime `bson:"created_at" json:"created_at"`
    UpdatedAt primitive.DateTime `bson:"updated_at" json:"updated_at"`
}
```

Photo (图片文档)

设计说明：将 `Exif` 和 `Tags` 设为内嵌字段。`Tags` 采用对象数组，以便区分标签来源（用户/AI）。

```
type Photo struct {
    ID          primitive.ObjectID `bson:"_id,omitempty" json:"id"`
    UserID      primitive.ObjectID `bson:"user_id" json:"user_id"` // 关联字段，建立索引
    Title       string             `bson:"title" json:"title"`
    Description string             `bson:"description" json:"description"`

    // 文件元信息
    FileName    string             `bson:"file_name" json:"file_name"`
    Path        string             `bson:"path" json:"path"`
    ThumbPath   string             `bson:"thumb_path" json:"thumb_path"`
    Hash        string             `bson:"hash" json:"hash"`       // 建立索引用于秒传
    Size        int64              `bson:"size" json:"size"`
    MimeType    string             `bson:"mime_type" json:"mime_type"`

    // 内嵌 EXIF 信息 (1:1)
    Exif        ExifInfo           `bson:"exif,omitempty" json:"exif,omitempty"`

    // 内嵌标签信息 (1:N)
    Tags        []Tag              `bson:"tags,omitempty" json:"tags"`

    CreatedAt   primitive.DateTime `bson:"created_at" json:"created_at"` // 建立索引用于时间范围查询
    UpdatedAt   primitive.DateTime `bson:"updated_at" json:"updated_at"`
}

type ExifInfo struct {
    Make         string    `bson:"make,omitempty" json:"make"`
    Model        string    `bson:"model,omitempty" json:"model"`
    Lens         string    `bson:"lens,omitempty" json:"lens"`
    ISO          int       `bson:"iso,omitempty" json:"iso"`
    Aperture     float64   `bson:"aperture,omitempty" json:"aperture"`
    ShutterSpeed string    `bson:"shutter_speed,omitempty" json:"shutter_speed"`
    FocalLength  float64   `bson:"focal_length,omitempty" json:"focal_length"`
    GPS          GPSInfo   `bson:"gps,omitempty" json:"gps"`
    TakenAt      primitive.DateTime `bson:"taken_at,omitempty" json:"taken_at"`
}

type GPSInfo struct {
    Latitude  float64 `bson:"latitude" json:"latitude"`
```

```
    Longitude float64 `bson:"longitude" json:"longitude"`
}

type Tag struct {
    Name   string `bson:"name" json:"name"`
    Source string `bson:"source" json:"source"` // "USER" 或 "AI"
    Score  float64 `bson:"score,omitempty" json:"score"` // 可选：AI置信度
}
```

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，并引入**依赖注入 (DI)** 思想以解耦层级依赖。

- **Controller 层 (接口适配)**:
  - **职责**: 仅处理 HTTP 协议相关逻辑（参数绑定 BindJSON/BindQuery、JWT 解析、统一响应封装）。
  - **设计模式**: 使用 DTO (Data Transfer Object) 接收前端参数，避免直接将数据库 Model 暴露给 HTTP 接口。
- **Service 层 (业务编排)**:
  - **职责**: 实现核心业务逻辑，如“上传流程控制”、“权限校验”。
  - **上下文管理**: 必须接收 `context.Context`，以便在 HTTP 请求取消时（如用户关闭浏览器）能级联取消数据库查询或耗时计算，防止资源泄露。
- **Repository 层 (数据访问)**:
  - **职责**: 封装 MongoDB Driver 原生操作 (`Find`，`InsertOne`，`Aggregate`)。
  - **连接池管理**: 在全局初始化时通过 `mongo.Connect` 创建 Client，并设置 `SetMaxPoolSize(100)` 。Repository 复用该 Client，避免频繁握手。
  - **BSON 构建**: 负责将 Go Struct 转换为 `bson.M` 或 `bson.D`，屏蔽数据库细节。

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 (IO 流式优化版)

利用 Go 的 `io` 接口特性，实现“零拷贝”或“单次遍历”处理，最大化吞吐量。

1. **同步阶段 (极致性能)**:
   - **流式 Hash 计算与落盘 (TeeReader)**:
     - 不将整个文件读入 RAM（防止大文件 OOM）。
     - 使用 `io.TeeReader(fileStream, hashWriter)` 包装文件流，在将文件流写入磁盘 (`io.Copy`) 的同时，自动计算 SHA-256 哈希值。
   - **秒传逻辑 (逻辑分支)**:
     - Hash 计算完成后，先查询 DB。若 Hash 命中 -> 删除刚写入的临时文件 -> 复用旧文件路径 -> 插入新 Photo 文档（指向旧路径）。

        - 若 Hash 未命中 -> 保留文件 -> 解析 EXIF (利用 `goexif` 读取文件头) -> 插入新 Photo 文档。
    - **原子写入**:
        - 构建包含 `Exif` 内嵌文档的 `Photo` 结构体。
        - 调用 `collection.InsertOne(ctx, photo)` 。MongoDB 的单文档原子性保证了无需开启复杂事务即可保证数据一致性。
2. **异步阶段 (可靠性设计)**:
    - **任务投递**: 使用带缓冲的 Channel `chan string` (传递 PhotoID) 作为任务队列。
    - **并发控制 (Worker Pool)**: 启动固定数量 (e.g., 5个) 的 Goroutine 消费 Channel，防止瞬间上传流量冲垮 CPU 或内存（特别是图片缩放操作极其消耗 CPU）。
    - **原子更新**:
        - 缩略图生成后，调用 `UpdateOne` 更新 `thumb_path` 。
        - AI 分析完成后，使用 `$addToSet` 操作符更新 `tags` 数组，确保即使有多个并发 AI 服务也不会覆盖彼此的标签。