中山大学计算机学院

软件工程课程项目

LifeMaster系统架构设计文档

项 目 名 称: LifeMaster

组 员 姓 名: 刘昊、彭怡萱、马福泉

: 林炜东、刘贤彬、刘明宇

专 业: 软件工程

课 程 教 师: 郑贵锋

起 始 日 期: 2025年3月1日

结 束 日 期: 2025年7月6日

学 院: 计算机学院

目录

1 系统概述

LifeMaster是一个集成待办事项管理、记账管理和手账管理功能的个人生活管理系统。系统采用经典三层架构设计，实现前后端分离。

1.1 系统目标

LifeMaster致力于为用户提供一站式的生活管理服务，主要目标包括：

- **统一管理**：将待办事项、财务记录、生活手账等功能整合到一个平台
- **用户友好**：提供直观易用的界面和流畅的交互体验
- **数据安全**：确保用户数据的安全性和隐私保护
- **高性能**：保证系统的响应速度和稳定性
- **可扩展**：采用模块化设计，便于功能扩展和维护

1.2 核心功能

- **待办事项管理**：任务创建、分类、优先级设置、进度跟踪
- **记账管理**：收支记录、分类统计、预算管理
- **手账管理**：日记记录、图片上传、情绪标记
- **数据分析**：生活数据可视化、趋势分析、报告生成

2 架构设计

2.1 整体架构

系统采用分层架构模式，从上至下分为：

- **表示层（前端层）**：负责用户界面展示和用户交互
- **业务逻辑层（后端层）**：处理业务逻辑和数据处理
- **数据访问层（数据层）**：负责数据存储和访问

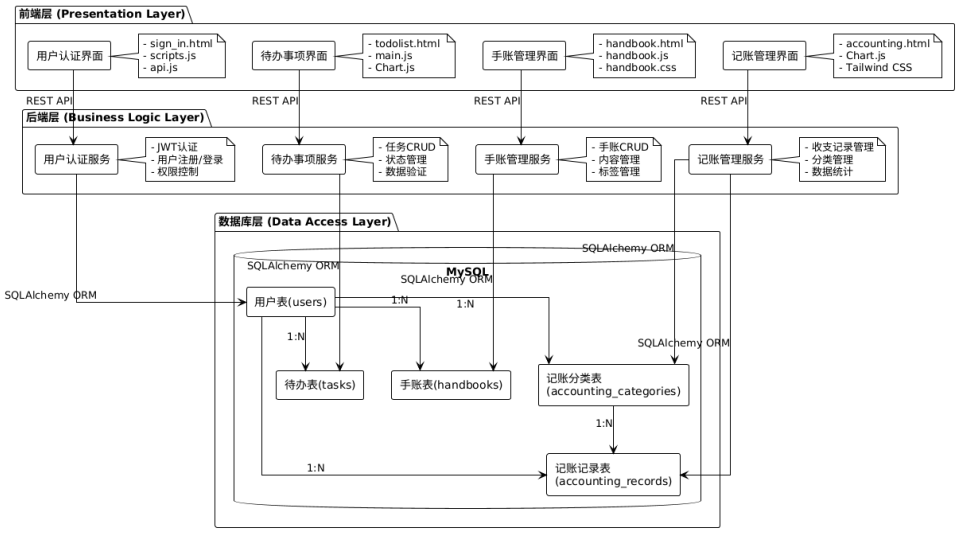

图 1: LifeMaster系统架构图

该架构具有以下优势：

- **分层清晰**：各层职责明确，便于开发和维护
- **解耦合**：层与层之间通过标准接口通信，降低耦合度
- **可扩展**：可以独立扩展各层功能，不影响其他层
- **可维护**：便于定位问题和进行代码维护

2.2 技术栈选择

2.2.1 前端技术栈

技术	版本	用途
HTML5	-	页面结构定义和语义化标记
CSS3	-	样式设计和动画效果
JavaScript	ES6+	页面交互逻辑和数据处理
Tailwind CSS	3.0+	响应式布局和组件样式
Chart.js	3.0+	数据可视化和图表展示
Fetch API	-	网络请求处理和数据交互

表 1: 前端技术栈

2.2.2 后端技术栈

技术	版本	用途
Python	3.8+	后端开发语言
Flask	2.0+	Web应用框架
JWT	-	身份认证和授权
SQLAlchemy	1.4+	对象关系映射（ORM）框架
RESTful API	-	API设计规范

表 2: 后端技术栈

2.2.3 数据库技术

技术	版本	用途
MySQL	8.0+	关系型数据库管理系统
事务支持	-	保证数据一致性和完整性
外键约束	-	维护数据引用完整性

表 3: 数据库技术栈

3 详细设计

3.1 前端层设计

3.1.1 模块划分

前端采用模块化设计，按功能划分为不同的模块：
前端项目结构：

- main.html - 主页面
- handbook.html - 手账模块页面
- accounting.html - 记账模块页面
- todolist.html - 待办模块页面
- sign_in.html - 登录注册页面
- main.js - 主页面逻辑
- handbook.js - 手账模块逻辑
- accounting.js - 记账模块逻辑
- api.js - 前端API请求封装
- styles.css - 全局样式
- handbook.css - 手账模块样式
- 图片素材/ - 图片与字体等资源
- fonts/ - 字体文件

3.1.2 关键设计决策

1. **模块化设计**：将不同功能的代码分离到独立文件中，提高代码的可维护性和可重用性
2. **统一API接口层**：通过api.js文件统一处理所有后端通信，包括：
   - 请求封装和响应处理
   - 错误处理和重试机制
   - 认证token管理
   - 请求缓存策略
3. **响应式设计**：使用Tailwind CSS实现响应式布局，适配多种设备：
   - 移动端：手机和小屏设备
   - 平板端：中等屏幕设备
   - 桌面端：大屏幕设备

3.2 后端层设计

3.2.1 模块结构

后端采用Flask框架的蓝图（Blueprint）模式进行模块化设计：
后端项目结构：

- api/ - 后端API接口（Flask蓝图）
- api/index.py - API主入口
- migrations/ - 数据库迁移脚本
- test/ - 单元测试与数据库测试
- app.py - Flask应用入口
- requirements.txt - Python依赖
- Dockerfile - 容器部署配置

3.2.2 关键设计决策

1. RESTful API设计
API设计遵循RESTful规范，具有以下特点：

HTTP方法	用途	示例
GET	获取资源	GET /api/tasks
POST	创建资源	POST /api/tasks
PUT	更新资源	PUT /api/tasks/1
DELETE	删除资源	DELETE /api/tasks/1

表 4: RESTful API HTTP方法使用规范

2. 身份认证机制

系统采用JWT（JSON Web Token）认证机制：

- **Token生成**：用户登录成功后生成JWT token
- **Token验证**：每个API请求都需要验证token有效性
- **Token过期**：token过期时间设置为24小时
- **Token刷新**：提供token刷新机制，避免频繁重新登录

3. 数据验证策略

实施多层数据验证：

- **请求参数验证**：验证参数类型、格式、长度等
- **业务规则验证**：验证业务逻辑的合理性
- **数据一致性检查**：确保数据库操作的一致性

3.3 数据层设计

3.3.1 数据库表结构

系统主要包含以下核心数据表：

用户表（users）

```
CREATE TABLE users (
id VARCHAR(36) PRIMARY KEY,
username VARCHAR(50) NOT NULL,
email VARCHAR(120) UNIQUE NOT NULL,
password_hash VARCHAR(128) NOT NULL,
created_at DATETIME NOT NULL,
updated_at DATETIME NOT NULL
);
```

待办表（tasks）

```
CREATE TABLE tasks (
id INTEGER PRIMARY KEY AUTO_INCREMENT,
user_id VARCHAR(36) NOT NULL,
text TEXT NOT NULL,
deadline DATETIME,
completed BOOLEAN DEFAULT FALSE,
created_at DATETIME NOT NULL,
updated_at DATETIME NOT NULL,
FOREIGN KEY (user_id) REFERENCES users(id)
);
```

记账表（accounts）

```
CREATE TABLE accounts (
id INTEGER PRIMARY KEY AUTO_INCREMENT,
```

```
user_id VARCHAR(36) NOT NULL,
amount DECIMAL(10,2) NOT NULL,
category VARCHAR(50) NOT NULL,
description TEXT,
type ENUM('income', 'expense') NOT NULL,
date DATE NOT NULL,
created_at DATETIME NOT NULL,
updated_at DATETIME NOT NULL,
FOREIGN KEY (user_id) REFERENCES users(id)
);
```

手账表（handbooks）

```
CREATE TABLE handbooks (
id INTEGER PRIMARY KEY AUTO_INCREMENT,
user_id VARCHAR(36) NOT NULL,
title VARCHAR(200) NOT NULL,
content TEXT,
images TEXT,
tags TEXT,
mood INTEGER,
date DATE NOT NULL,
created_at DATETIME NOT NULL,
updated_at DATETIME NOT NULL,
FOREIGN KEY (user_id) REFERENCES users(id)
);
```

3.3.2 关键设计决策

1. 数据库选择

选择MySQL作为主数据库的原因：

- **ACID特性**：保证数据一致性和完整性
- **事务支持**：支持复杂的业务操作
- **外键约束**：维护数据引用完整性
- **成熟稳定**：广泛使用，文档完善，社区支持好
- **性能优秀**：适合中小型应用的性能需求

2. ORM策略

使用SQLAlchemy ORM框架的优势：

- **对象映射**：将数据库表映射为Python对象
- **查询简化**：提供Pythonic的查询语法
- **连接管理**：自动管理数据库连接和连接池
- **迁移支持**：支持数据库版本管理和迁移

4 安全设计

4.1 认证与授权

4.1.1 JWT Token认证机制

系统采用JWT（JSON Web Token）进行用户认证：

组件	说明
Header	包含token类型和加密算法信息
Payload	包含用户ID、过期时间等claims信息
Signature	使用密钥对header和payload进行签名

表 5: JWT Token结构

4.1.2 密码安全策略

- **密码加密**：使用bcrypt进行密码哈希存储
- **密码强度**：要求密码长度至少8位，包含字母和数字
- **防暴力破解**：登录失败次数限制和账户锁定机制

4.2 数据安全

4.2.1 防注入攻击

- **SQL注入防护**：使用ORM参数化查询，避免直接拼接SQL
- **XSS防护**：对所有用户输入进行严格验证和过滤
- **CSRF防护**：为每个表单生成唯一的CSRF token

5 性能优化

5.1 前端优化策略

5.1.1 资源优化

优化策略	实现方式
静态资源缓存	设置合适的Cache-Control头，使用版本号控制缓存更新
按需加载	使用动态import()语法，实现代码分割和懒加载
图片懒加载	使用Intersection Observer API实现图片懒加载
资源压缩	CSS/JS文件压缩，图片压缩和格式优化

表 6: 前端性能优化策略

5.2 后端优化策略

5.2.1 数据库优化

优化策略	实现方式
索引优化	为频繁查询的字段建立合适的索引，避免全表扫描
查询缓存	使用Redis缓存热点数据和查询结果
连接池管理	配置合适的数据库连接池大小，复用连接
慢查询优化	监控和优化慢查询，使用EXPLAIN分析执行计划

表 7: 数据库性能优化策略

6 扩展性考虑

6.1 系统可扩展性

6.1.1 水平扩展

- **无状态设计**：应用服务器无状态，便于水平扩展
- **数据库分片**：支持数据库读写分离和分片策略
- **缓存集群**：使用Redis集群提供高可用缓存服务
- **CDN加速**：静态资源使用CDN分发，提高访问速度

6.1.2 功能扩展

- **模块化设计**：新功能可以作为独立模块开发和部署
- **插件机制**：预留插件接口，支持第三方功能扩展
- **API版本管理**：支持API版本控制，保证向后兼容
- **微服务架构**：为未来微服务改造预留架构空间

7 部署方案

7.1 环境要求

7.1.1 开发环境

组件	版本要求	说明
Python	3.8+	后端开发语言
MySQL	8.0+	数据库服务
Node.js	14+	前端构建工具（用于前端构建）
Redis	6.0+	缓存服务（可选）

表 8: 开发环境要求

7.1.2 生产环境

组件	配置要求	说明
CPU	2核心+	推荐4核心以上
内存	4GB+	推荐8GB以上
存储	50GB+	SSD推荐
网络	10Mbps+	稳定的网络连接

表 9: 生产环境配置要求

7.2 部署步骤

7.2.1 数据库初始化

1. 创建数据库：CREATE DATABASE lifemaster;
2. 创建用户：CREATE USER 'lifemaster'@'localhost' IDENTIFIED BY 'password';
3. 授权：GRANT ALL PRIVILEGES ON lifemaster.* TO 'lifemaster'@'localhost';
4. 执行数据库迁移：python -m flask db upgrade

7.2.2 后端服务部署

1. 安装依赖：pip install -r requirements.txt
2. 配置环境变量：设置FLASK_APP、DATABASE_URL等
3. 启动服务：gunicorn -w 4 -b 0.0.0.0:5000 app:app

7.2.3 前端资源部署

1. 构建前端资源：npm install && npm run build
2. 部署到Web服务器：cp -r dist/* /var/www/html/
3. 配置Nginx：配置反向代理和静态资源服务

7.2.4 Nginx配置

Nginx作为反向代理服务器，负责：

- 前端路由处理
- API请求代理到后端服务
- 静态资源缓存
- SSL/TLS终止

8 总结

本文档详细阐述了LifeMaster系统的架构设计，涵盖了系统概述、架构设计、详细设计、安全设计、性能优化、扩展性考虑和部署方案等各个方面。

8.1 架构优势

LifeMaster系统架构具有以下主要优势：

- **分层清晰**：三层架构职责明确，便于开发和维护
- **技术成熟**：选择了经过验证的成熟技术栈
- **安全可靠**：完善的安全防护机制和数据保护措施
- **性能优秀**：多层次的性能优化策略
- **易于扩展**：模块化设计便于功能扩展和技术升级
- **运维友好**：提供完整的部署和监控方案

8.2 设计原则

在架构设计过程中，我们始终遵循以下设计原则：

- **简单性**：保持架构简单清晰，避免过度设计
- **可维护性**：代码结构清晰，便于维护和调试
- **可扩展性**：预留扩展空间，支持业务增长
- **可靠性**：确保系统稳定运行，数据安全可靠
- **性能性**：优化系统性能，提供良好用户体验

此架构文档将作为系统开发、部署和维护的重要参考，为项目的成功实施提供技术保障。