类chatshare跨模态平台

软件架构文档

版本 <1.0>

修订历史记录

|  |  |  |  |
| --- | --- | --- | --- |
| **日期** | **版本** | **说明** | **作者** |
| <日/月/年> | <x.x> | <详细信息> | <姓名> |
| 4/7/25 | 1.0 | 技术原型迭代 | 杨可凡 |
|  |  |  |  |
|  |  |  |  |

目录

1. 简介 4

1.1 目的 4

1.2 参考资料 4

2. 用例视图 4

3. 逻辑视图 4

4. 部署视图（可选） 4

5. 进程视图（可选） 4

6. 实现视图（可选） 4

7. 技术视图 4

8. 数据视图（可选） 5

9. 算法视图（可选） 5

10. 性能视图（可选） 5

11. 可靠性视图（可选） 5

12. 安全性视图（可选） 5

13. 易用性视图（可选） 5

14. 可维护性视图（可选） 5

软件架构文档

# 简介

## 目的

本文档将从构架方面对系统进行综合概述，其中会使用多种不同的构架视图来描述系统的各个方面。它用于记录并表述已对系统的构架方面作出的重要决策。

本文档将从架构角度对本系统进行全面的概述，采用多种架构视图（如逻辑视图、开发视图、部署视图、数据视图、算法视图等）对系统的结构、组件、交互与运行环境进行描述。该文档的主要目的是：

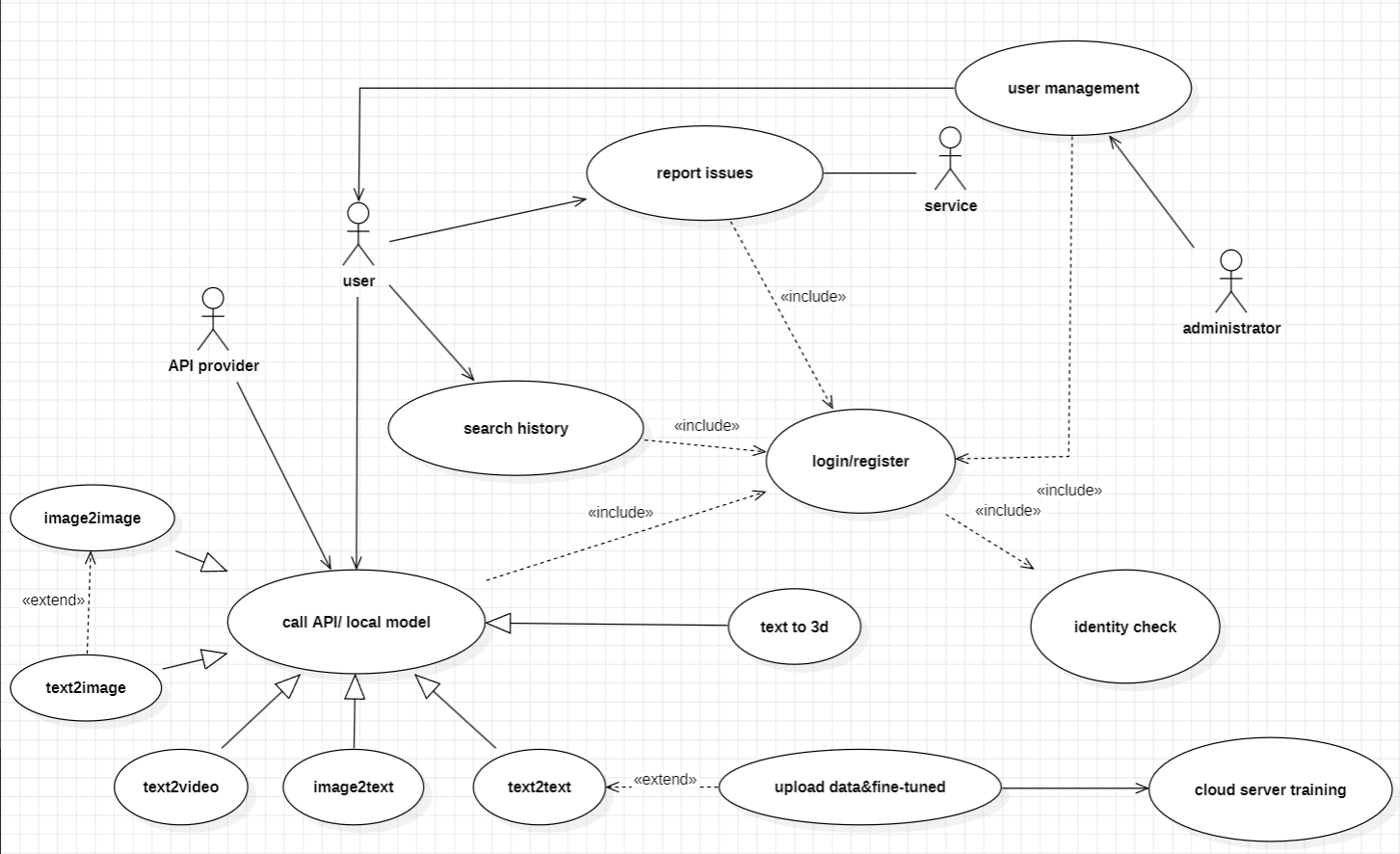
* 明确系统架构的整体设计思想与技术路线；
* 表述对系统在性能、安全性、可维护性、扩展性、可部署性等方面的关键设计决策；
* 为项目开发团队提供架构层面的指导，确保开发过程的一致性；
* 为后期系统的维护、扩展与优化提供参考依据；
* 为项目干系人（如产品经理、测试人员、运维人员等）提供系统结构的理解基础；
* 作为验收评审时的系统架构依据。

本架构文档适用于系统的开发人员、架构师、测试人员、运维人员及相关技术决策人员。

## 参考资料

Vision文档1.0 5组成员蒋宸 23/JUN/25

# 用例视图



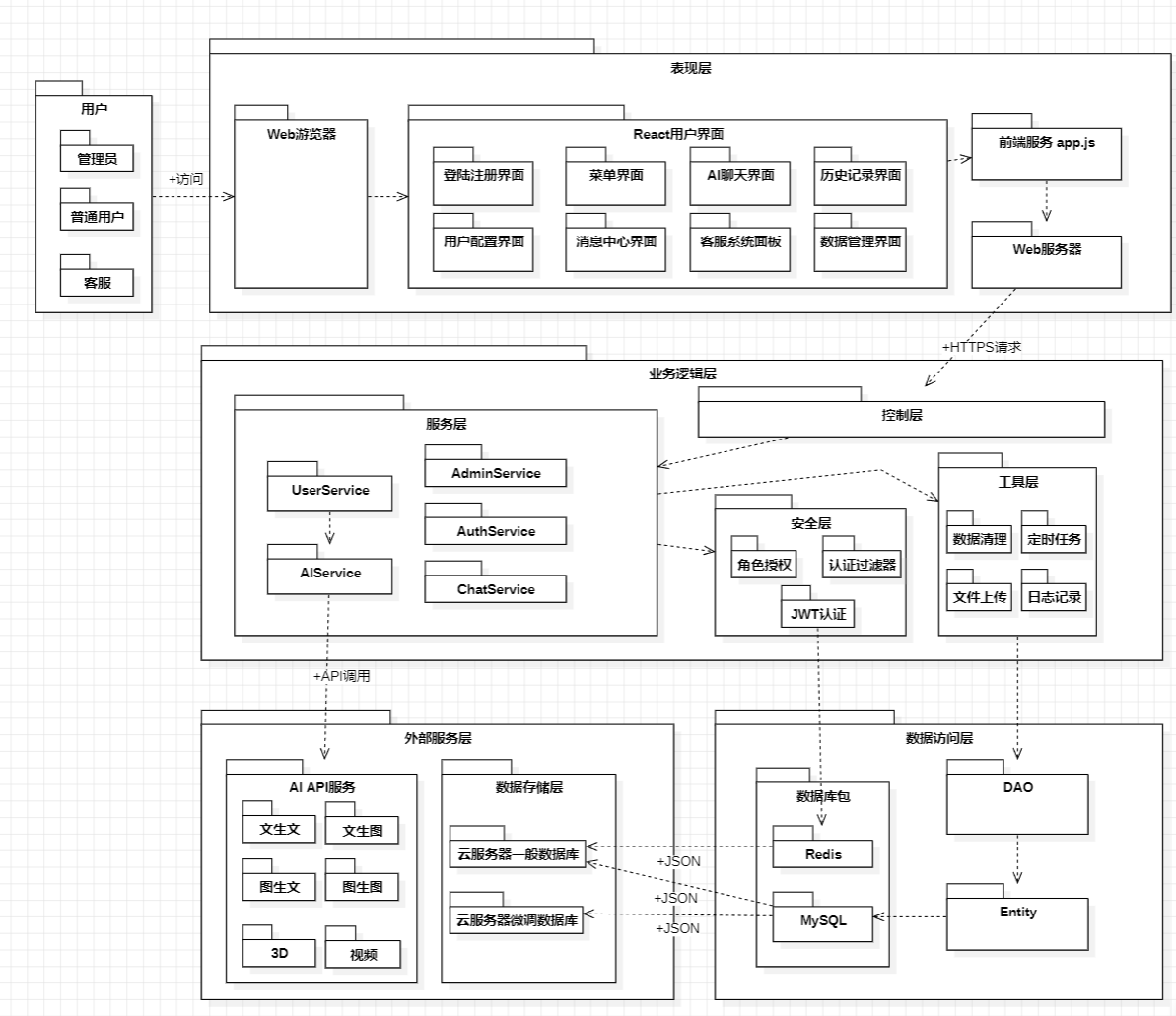
1. **调用多模态大模型进行生成（Call API/Local Model）**  
该用例是系统最核心的功能之一，涵盖了文生文、图生文、文生图、文生3D 等子功能。它涉及前端、后端、模型推理服务、任务队列（如 Celery）和结果回传等多个构架组件，体现了系统对异步请求调度、多模型路由与负载均衡的支持。

2. **上传数据并发起模型微调（Upload Data for Fine-tuning）**  
本用例体现了系统在模型适配方面的高级能力。用户可上传特定行业的数据集（如医疗、教育、游戏），通过后台触发基于 Ray 和 DeepSpeed 的分布式微调流程，实现模型在指定场景下的知识迁移。该流程涉及数据预处理、微调任务调度、资源分配、模型存储等多个架构模块，是体现系统训练能力的重要用例。

3. **用户权限与订阅管理（User Access & Membership Control）**  
本用例说明了系统对用户权限控制的架构设计。部分高级功能（如文生3D、视频生成）仅对已订阅高级会员开放。该场景涉及用户角色认证、权限拦截、付费接口校验等关键安全性设计点，保障系统功能分级和资源合理使用。

4. **任务状态查询与结果下载（Job Tracking & Result Access）**  
多模态模型任务具有一定计算时间，用户提交请求后可随时查询状态（如等待中、处理中、失败、完成），并在任务完成后获取结果（图片、视频、文本等）。该用例体现了任务管理系统与数据库、缓存系统的交互逻辑，是系统可用性和交互性的关键体现。

# 逻辑视图



本系统采用典型的分层逻辑架构，主要分为控制层（controller）、业务逻辑层（service）、数据模型层（model）、数据访问层（repository）以及任务调度层（scheduler）等核心模块。

控制层负责处理来自前端的 API 请求，并将其转发至业务逻辑层。典型的控制器包括 TextGenController、ImageGenController 和 FineTuneController，分别用于处理文本生成、图像生成及模型微调请求。控制层还承担参数验证、异常捕获与路由分发的职责。

业务逻辑层负责系统核心功能的实现，如模型路由、微调任务构造、权限校验等，封装了主要的应用逻辑。其中，ModelRoutingService 负责根据请求类型和用户订阅权限选择适配模型；FineTuneService 实现微调流程的组织与调用，结合 Ray 与 DeepSpeed 进行分布式训练；JobTrackingService 则用于追踪生成或训练任务的执行状态，供前端轮询展示。

数据模型层定义了系统的核心数据结构，例如 User（表示用户实体，包含用户名、邮箱、订阅等级等）、GenTask（表示一次多模态生成或微调任务，包含输入、输出、状态等）以及 ModelMeta（用于记录模型元信息如名称、类型、参数量等）。这些模型类通常与数据库表结构一一对应。

数据访问层封装了对数据库的 CRUD 操作，通过对应的 Repository 类（如 UserRepository、GenTaskRepository 等）访问持久化数据，通常基于 ORM 工具实现，与数据库保持良好的结构映射。

# 部署视图（可选）

图示

AI 生成的内容可能不正确。

# 进程视图（可选）

本系统采用典型的多进程 + 多线程架构，将系统划分为多个轻量级和重量级进程，并借助异步队列实现任务间的解耦与通信。整体架构以服务化部署为基础，不同功能模块运行于不同的服务进程中，以提高系统的可伸缩性和健壮性。以下是系统主要的进程划分及通信关系说明。

**前端客户端进程**负责发起多模态请求（如文生图、图生文、模型微调等），通过 HTTP 协议访问后端 API 接口。该进程通常运行在浏览器环境或移动应用容器中，为用户提供交互入口，所有操作均由前端页面驱动。

**Web 后端服务进程**是系统的核心协调者，运行在一个独立的重量级进程中，采用 Python（FastAPI）或 Node.js 实现。每个用户请求会被主进程接收，并通过多线程或协程分发给具体的业务处理函数（控制器）。该进程负责用户身份认证、请求参数解析、权限校验，并将推理或训练任务转交至异步任务队列处理。其内部线程或协程池承担请求处理和响应回传等操作。

**异步任务处理进程组**由 Celery worker 构成，是一组轻量级任务执行进程，负责处理文本生成、图像生成、文生3D、视频生成等多模态任务。这些进程通过 Redis 消息中间件与后端服务通信，采用消息队列的方式进行解耦。每个任务被封装为消息，由任务处理进程拉取并执行，执行完毕后写入数据库并通知后端回传结果。

**分布式微调训练进程组**在用户发起数据微调时，由后端启动一个独立的训练调度流程，基于 Ray 或 Deepspeed 启动多个分布式进程节点。这些训练节点可跨服务器调度，内部通过 MPI/NCCL 等高速通信协议进行梯度同步与模型共享。每个节点本质上由多个轻量级子进程构成，完成模型切分、数据加载、反向传播等任务。

**数据库进程**运行在单独的 PostgreSQL 服务中，以守护进程方式存在。它负责接收后端的存储请求，维护用户信息、任务记录、模型元数据等。数据库通过标准 SQL 接口与后端服务通信，属于被动型进程，按需响应请求。

**文件存储与日志进程**作为辅助模块，在微调训练、生成任务中产生的中间文件与结果文件会写入本地或云端存储，并由独立的进程负责文件管理、日志归档等任务。

各进程之间的通信方式主要包括：HTTP 请求/响应（前端与后端）、Redis 消息队列（后端与 Celery）、数据库连接池（后端与数据库）、RPC（Ray 或 Deepspeed 节点之间）。系统还支持任务状态轮询与回调通知机制，以提升交互性能与用户体验。

本系统整体遵循**主进程调度 + 异步执行 + 分布式训练**的运行机制，实现了任务解耦、模块独立和处理并发能力的良好平衡。

# 实现视图（可选）

图示

AI 生成的内容可能不正确。

# 技术视图

图示

AI 生成的内容可能不正确。

本系统面向多模态内容生成与模型微调平台的应用场景，技术架构采用前后端分离、多语言混合实现、分布式任务调度和容器化部署的设计，确保系统具有良好的可扩展性、可维护性与开发效率。

**编程语言与开发工具**  
前端使用 JavaScript 和 React 框架进行开发，构建工具为 Vite/Webpack，开发过程中配合 ESLint 与 Prettier 保证代码风格一致性。后端主服务采用 Java 语言，基于 Spring Boot 框架实现 RESTful API，同时负责用户认证、任务管理与数据库操作。模型服务及异步任务处理采用 Python 实现，主要依赖 FastAPI、Celery、Ray 和 DeepSpeed 等工具，支持多模态内容生成和模型微调等核心功能。

**数据库**  
系统使用 MySQL 作为主数据库，存储用户信息、生成任务记录、模型元数据等。数据库访问在后端通过 Spring Data JPA 实现，支持复杂查询与事务管理。日志信息与中间结果存储则使用本地文件系统或云存储方案。

**中间件与异步框架**  
为了实现高性能异步任务处理和跨语言模块解耦，系统采用 Redis 作为消息队列中间件，配合 Celery 构建分布式任务队列体系。任务处理流程由 Python 编写的 Celery Worker 异步消费，后端通过 Redis 将任务分发给模型服务端处理，并通过回调通知或轮询机制获取执行状态。

**分布式训练与推理框架**  
模型微调部分集成了 Ray 和 DeepSpeed，支持在多节点、GPU 资源下的高效分布式训练，具备参数并行与数据并行能力。该部分与微调服务解耦，通过配置任务规格自动调度训练集群与资源。

**API 接口与通信协议**  
系统内部采用 HTTP 协议进行前后端通信，API 规范遵循 RESTful 风格，前端调用由 Java 后端提供的接口，后端与 Celery Worker 间通过 Redis 消息发布订阅机制（Pub/Sub）实现通信。后端与数据库间使用 JDBC/ORM 层通信，训练节点之间通过 Ray RPC 和 NCCL 等通信协议进行高效参数同步。

# 数据视图（可选）

图示

AI 生成的内容可能不正确。

# 算法视图（可选）

图示

AI 生成的内容可能不正确。

**模型微调算法解释**：

整个微调过程始于输入数据处理模块，该模块负责读取和预处理训练数据，包括数据清洗、格式转换及数据增强等操作，确保输入符合模型要求。接着，预训练模型加载模块载入基础模型的结构和权重，通常支持冻结部分参数，只对特定层或子集进行微调。微调策略模块则定义具体的调优方法，如全模型微调、参数高效微调（例如LoRA、Adapter或Prefix-tuning）以及冻结部分层的训练方式，旨在提高微调效率和性能。损失计算模块根据具体任务目标（分类、生成或回归等）计算相应的损失函数，如交叉熵或均方误差，并支持多任务加权。优化器模块负责根据计算出的梯度更新模型参数，常用的优化算法包括Adam和SGD，同时结合学习率调度和梯度裁剪等技术以保持训练稳定。整个训练过程由训练循环控制模块协调管理，执行前向传播、反向传播、参数更新，同时支持训练中断恢复和模型保存。评估模块则定期在验证集上检测微调效果，通过计算准确率、损失等指标为训练过程提供反馈，辅助调整训练策略。总体而言，该算法视图为理解和设计微调流程提供了清晰的结构框架，帮助开发者有效构建和优化微调系统。

# 性能视图（可选）

|  |  |
| --- | --- |
| 分类 | 设计战术 |
| **资源需求优化** | 1. 前端资源优化：代码分割、懒加载、图片压缩、CSS/JS压缩 2. API请求优化：减少HTTP请求、合并接口调用、数据压缩传输 3. 内存管理：及时清理事件监听器、避免内存泄漏、限制历史消息缓存 |
| **资源管理** | 1. 并发处理：异步请求处理、Web Workers、多线程文件处理 2. 资源池管理：数据库连接池、AI API调用池、文件上传队列 3. 缓存策略：Redis缓存、浏览器缓存、CDN缓存、API结果缓存 |
| **资源仲裁** | 1. 请求优先级：实时聊天优先、管理员请求优先、VIP用户优先 2. 队列管理：FIFO消息队列、AI请求排队、文件处理队列 3. 流量控制：API限流、用户请求频率控制、防止恶意请求 |

# 可靠性视图（可选）

|  |  |
| --- | --- |
| 分类 | 设计战术 |
| **故障预防** | 1. 操作确认机制：重要操作二次确认、危险操作警告提示 2. 权限预检查：功能访问权限验证、数据操作权限控制 3. 资源限制控制：上传文件大小限制、请求频率控制、存储空间管理 |
| **错误检测** | 1. 前端输入验证：表单实时验证、数据格式检查、必填项提醒 2. API状态监控：请求响应监控、服务可用性检测、超时处理 3. 行为状态监控：异常操作检测、频繁请求识别、数据库连接检测 |
| **错误恢复** | 1. 自动重试机制：网络请求重试、AI调用重试、文件上传重试 2. 数据恢复功能：对话历史恢复、用户设置恢复、草稿自动保存 3. 会话保持：登录状态恢复、对话上下文保持、操作状态还原 |
| **冗余容错** | 1. 多模型支持：GPT-4、Claude-3等多个AI模型备选方案 2. 服务降级策略：核心功能保障、非核心功能降级、用户体验保持 |

# 安全性视图（可选）

|  |  |
| --- | --- |
| 分类 | 设计战术 |
| **身份认证** | 1. JWT令牌机制：安全token生成、过期时间控制、刷新机制 2. 密码安全策略：bcrypt加密存储、强密码要求、定期更换提醒 3. 会话管理：登录状态管理、并发登录控制、异地登录检测 |
| **权限访问** | 1. 基于角色的权限系统：admin/support/user角色分离、权限继承 2. 功能权限控制：AI功能访问权限、数据操作权限、管理功能权限 3. 数据访问隔离：用户数据隔离、对话记录隔离、敏感信息保护 4. API接口权限：接口访问控制、参数权限验证 |
| **安全防护** | 1. 输入验证过滤：SQL注入防护、XSS攻击防护 2. 速率限制控制：API请求限流、用户操作频率限制 3. 数据传输加密：HTTPS传输、敏感数据加密、通信协议安全 |

# 易用性视图（可选）

|  |  |
| --- | --- |
| 分类 | 设计战术 |
| **界面交互** | 1. 直观的AI功能选择界面：文生文、文生图、图生文等模块化设计 2. 实时响应反馈：消息发送状态、AI思考进度、错误提示显示 3. 响应式布局：适配桌面、平板、移动端的界面优化 |
| **用户体验** | 1. 个性化设置：主题切换、语言选择、数据微调 2. 智能推荐：历史对话推荐、常用功能推荐 3. 错误处理：友好的错误提示、操作恢复建议 |
| **功能可达性** | 1. 搜索功能：全文搜索、标签筛选、时间筛选 2. 界面布局：清晰的信息层次、直观的导航结构 |

# 可维护性视图（可选）

|  |  |
| --- | --- |
| 分类 | 设计战术 |
| **代码结构管理** | 1. 模块化设计：前端组件模块化、后端服务模块化、业务逻辑分层 2. 代码规范统一：代码格式化、命名规范、注释规范 3. 依赖管理：第三方库版本控制、依赖更新策略、安全依赖检查 4. 架构文档：技术架构文档、API文档、数据库设计文档 |
| **配置管理** | 1. 环境配置分离：开发/测试/生产环境配置、配置热更新 2. 数据库版本管理：数据库迁移脚本、版本控制、回滚机制 3. 部署配置：容器化部署、服务编排、环境一致性保证 |
| **运维支持** | 1. 日志系统：结构化日志、日志分级、日志聚合分析 2. 监控告警：系统指标监控、业务指标监控、异常告警 3. 故障诊断：错误追踪、性能分析、问题定位工具 4. 维护工具：数据库管理工具、系统管理脚本、备份恢复工具 |

# 可测试性视图

|  |  |
| --- | --- |
| 分类 | 设计战术 |
| **测试数据管理** | 1. 测试环境隔离：独立测试数据库、测试用户账户 2. 数据模拟生成：用户数据生成、对话记录生成、AI响应模拟 3. 测试数据清理：自动清理测试数据、测试环境重置、数据一致性检查 4. 边界条件测试：大数据量测试、极限参数测试、异常数据测试 |
| **测试接口设计** | 1. 单元测试支持：组件接口测试、函数接口测试、API接口测试 2. 集成测试框架：数据库集成测试、第三方服务集成测试 3. 端到端测试：用户流程测试、界面自动化测试、业务场景测试 4. 性能测试工具：负载测试、压力测试、并发测试、响应时间测试 |
| **监控测试能力** | 1. 测试覆盖率监控：代码覆盖率统计、功能覆盖率分析、测试用例覆盖 2. 自动化测试流程：自动回归测试、持续集成部署 3. 测试结果分析：测试报告生成、失败原因分析、性能指标对比 4. 测试环境监控：测试环境状态监控、资源使用监控、测试稳定性检查 |