KnowForge

先进的 Python CLI 智能知识库与知识图谱 Agent 开发平台

设计文档全集

含：解决方案文档 · 架构设计文档 · 详细设计文档 · 具体实现文档

附：架构图 · 设计图 · 流程图 · UML（共 11 张）

灵感来源：Yuxi（github.com/xerrors/Yuxi）

深度优化 · 纯 Python · CLI 优先 · 嵌入式零依赖

目 录

附录 图表索引（11 张）

第一部分 解决方案文档

> 先进的 Python CLI 智能知识库与知识图谱 Agent 开发平台

1. 项目背景

1.1 行业现状

随着大语言模型（LLM）的快速演进，企业对"让 AI 真正理解私有知识并完成复杂任务"的需求急剧增长。单纯的对话式 LLM 存在三大短板：知识时效性差、缺乏可溯源引用、无法执行真实任务。检索增强生成（RAG）与知识图谱（KG）成为业界公认的关键补丁，但现有方案普遍存在以下问题：

第一，部署门槛过高。主流方案（如 Dify、FastGPT、Yuxi）依赖 PostgreSQL、Redis、Milvus、Neo4j、MinIO 等多个外部服务，需要 Docker Compose 编排，对个人开发者与中小企业极不友好。冷启动一个完整环境往往需要 5-10 分钟，消耗数 GB 内存。

第二，前后端耦合严重。这些平台均采用 Vue/React 前端 + FastAPI 后端的架构，前端代码量通常占整体 40% 以上，但对于命令行优先的开发者、CI/CD 集成场景、自动化脚本场景而言，前端是纯粹的负担。

第三，知识图谱与 RAG 割裂。多数平台要么只做向量检索（无图谱推理能力），要么把图谱作为独立模块（与 RAG 检索脱节）。Yuxi 率先尝试融合二者，但其图谱构建依赖 Neo4j 外部服务，部署成本依然高昂。

第四，扩展机制封闭。Skills、Tools、Parsers 通常硬编码在主仓库中，第三方难以贡献。缺乏标准的插件协议，导致生态难以繁荣。

1.2 Yuxi 项目分析

Yuxi（语析）是一个值得深入研究的参考实现。它由江南大学博士研究生开发，定位为"结合知识库、知识图谱管理的多租户 Agent Harness 平台"，技术栈如下：

层	Yuxi 技术选型
前端	Vue 3 + Vite + Pinia
后端	FastAPI + LangGraph + ARQ
存储	PostgreSQL + Redis + MinIO + Milvus + Neo4j
文档解析	MinerU + PaddleX + RapidOCR
部署	Docker Compose

Yuxi 的核心亮点包括：Agentic RAG（智能体自主决定检索时机）、沙箱文件系统（每会话独立 workspace）、Skills 系统（图像生成、深度报告等）、MCP 集成、子智能体编排、中间件链（检索注入、附件处理、历史摘要、动态工具注入）、15+ LLM Provider 支持。

但 Yuxi 也存在明显不足：

- 部署重：5 个外部服务（PG/Redis/MinIO/Milvus/Neo4j），冷启动慢
- 前端依赖：Vue 前端对 CLI 场景是冗余
- 图谱外部化：Neo4j 需独立部署，单机用户负担重
- 扩展性弱：Skills/Tools 硬编码，无标准插件协议
- 可观测性不足：缺乏 OpenTelemetry 集成
- 本地优先缺失：无 Ollama 一键离线模式

1.3 KnowForge 的定位

KnowForge 旨在深度优化 Yuxi 的核心理念，以"纯 Python、CLI 优先、嵌入式零依赖"的形态，提供一个开发者友好、部署极简、扩展开放的智能知识库与知识图谱 Agent 平台。它不是 Yuxi 的复刻，而是对其架构的重新思考与进化。

2. 设计目标

2.1 核心目标

目标	量化指标
零外部依赖	`pip install knowforge && knowforge config init` 即可用，无需 Docker
冷启动快	< 3 秒（不含模型加载）
内存占用低	空载 < 200MB
检索质量高	混合检索 + 重排，Top-5 命中率 > 90%
图谱推理	支持 2-3 跳子图检索，参与 RAG 增强
扩展开放	pip entry_points 插件机制，第三方零侵入
可观测	structlog + OpenTelemetry-ready + LangSmith 兼容
离线优先	可选 Ollama 集成，完全离线运行

2.2 设计原则

原则一：CLI 优先，终端即界面。 所有交互通过 Typer + Rich 提供的终端体验完成，流式输出、语法高亮、引用面板、进度条一应俱全。这并非退步，而是对开发者工作流的回归——终端是可脚本化、可管道化、可版本控制的界面。

原则二：嵌入式一切。 SQLite 替代 PostgreSQL，ChromaDB 替代 Milvus，KùzuDB 替代 Neo4j，本地文件系统替代 MinIO，diskcache 替代 Redis（缓存场景）。所有存储都是嵌入式进程内组件，零外部服务。

原则三：插件化扩展。 Skills、Tools、Parsers、Embedders、LLM Providers 全部通过 pip entry_points 注册，第三方包只需在 pyproject.toml 声明 entry_points 即可被 KnowForge 自动发现与加载，无需修改主仓库。

原则四：可观测优先。 从第一天起就内置结构化日志（structlog）、分布式追踪（OpenTelemetry）、指标采集（Meter），并与 LangSmith 兼容，方便调试与优化。

原则五：渐进式复杂度。 默认配置极简（单命令初始化），但所有高级特性（多租户、异步任务、MCP、子智能体）都可通过配置开启，不强制用户面对全部复杂度。

3. 与 Yuxi 的深度对比

维度	Yuxi	KnowForge
部署形态	Docker Compose（5+ 外部服务）	pip install（零外部服务）
界面	Vue 3 前端 + FastAPI 后端	纯 CLI（Typer + Rich）
元数据存储	PostgreSQL	SQLite（嵌入式）
向量库	Milvus（独立服务）	ChromaDB（嵌入式）
图数据库	Neo4j（独立服务）	KùzuDB（嵌入式）
对象存储	MinIO	本地文件系统
缓存	Redis	diskcache（嵌入式）
异步任务	ARQ + Redis	asyncio + SQLite-backed 队列
Agent 编排	LangGraph	LangGraph（保留）
文档解析	MinerU + PaddleX + RapidOCR	PyMuPDF + python-docx/openpyxl/pptx + BeautifulSoup（可选 MinerU）
嵌入模型	OpenAI/智谱等云端	本地 sentence-transformers 优先 + 云端可选
重排器	无明确	BGE cross-encoder
检索策略	向量 + 图谱	向量 + BM25 + 图谱（RRF 融合 + 重排）
插件机制	硬编码	pip entry_points
多租户	数据库层	Profile + Tenant 双层
可观测	基础日志	structlog + OpenTelemetry
离线模式	无	Ollama 集成
代码量	前后端 ~5 万行	纯 Python ~7 千行
学习曲线	需懂前后端	仅 Python

4. 技术选型理由

4.1 为什么选 SQLite 而非 PostgreSQL

SQLite 是世界上部署量最大的数据库，单文件持久化、零配置、ACID 事务、WAL 模式支持并发读。对于单机 CLI 场景，SQLite 的性能完全够用（单机写入 > 10K TPS）。PostgreSQL 的优势在于多客户端并发与高级特性（如 JSONB 索引、全文检索），但这些对 CLI 场景并非必需。选择 SQLite 让用户免去数据库运维负担。

4.2 为什么选 ChromaDB 而非 Milvus

ChromaDB 是嵌入式向量数据库，纯 Python 实现，单进程内运行，数据持久化到本地目录。Milvus 是分布式向量数据库，适合大规模生产场景，但部署需要 etcd + MinIO + Milvus 至少三个服务。对于 10 万级以下的向量数据，ChromaDB 的 HNSW 索引性能与 Milvus 差距不大，但部署成本天壤之别。

4.3 为什么选 KùzuDB 而非 Neo4j

Kùzu 是一个嵌入式图数据库（类似 DuckDB 之于关系数据库），由滑铁卢大学开发，单文件持久化，支持 Cypher 查询语言。Neo4j Community 版本虽免费但需独立部署，Enterprise 版本收费。Kùzu 让知识图谱能力成为"开箱即用"的基础设施，而非需要额外运维的组件。

4.4 为什么选 LangGraph 保留

LangGraph 是 LangChain 出品的状态机式 Agent 编排框架，支持循环、条件分支、并行节点、人机协同（human-in-the-loop）。它是目前最成熟的 Agent 编排抽象，社区活跃、文档完善。KnowForge 保留 LangGraph 作为编排核心，避免重复造轮子，同时聚焦于知识引擎与 CLI 体验的创新。

4.5 为什么 CLI 优先

CLI 不是退步，而是对开发者工作流的回归。CLI 的优势：可脚本化（管道、重定向）、可版本控制（配置即代码）、可远程执行（SSH）、可 CI/CD 集成、资源占用低。对于知识库管理、批量摄入、自动化评测等场景，CLI 比 Web 界面高效得多。Rich 库让现代终端 UI 同样美观——流式输出、语法高亮、表格、进度条、面板一应俱全。

5. 适用场景

5.1 个人开发者知识管理

开发者可将技术文档、博客、笔记摄入知识库，通过 CLI 快速检索与问答，构建个人"第二大脑"。所有数据本地存储，隐私可控。

5.2 企业内部知识库

中小企业无需运维复杂基础设施，单机部署即可服务团队。多 Profile 实现项目隔离，RBAC 控制访问权限。异步任务支持批量摄入大型文档集。

5.3 研究与教育

研究人员可摄入论文、数据集、实验记录，利用知识图谱发现实体间隐含关系。学生可基于此平台学习 RAG、KG、Agent 编排的工程实践。

5.4 CI/CD 集成

将 KnowForge 嵌入 CI 流水线，实现：代码文档自动摄入、PR 问答机器人、技术决策报告自动生成。CLI 原生支持 JSON 输出，便于与其他工具链集成。

5.5 离线场景

通过 Ollama 集成本地 LLM，配合本地嵌入模型（sentence-transformers），实现完全离线的智能知识库，适用于内网、保密、无网络环境。

6．方案优势总结

KnowForge 相比 Yuxi 与同类方案的核心优势可归纳为五点：

部署极简：一条 pip install + 一条 config init，3 秒内可用，无需 Docker、无需外部服务。

体验现代：Rich TUI 提供流式输出、语法高亮、引用面板、进度条，终端体验不输 Web 界面，且可脚本化。

检索先进：向量 + BM25 + 知识图谱三路融合，RRF 排序 + Cross-Encoder 重排，引用可溯源到文档与页码。

扩展开放：pip entry_points 插件机制，Skills/Tools/Parsers/Embedders/LLM Providers 全可第三方扩展，零侵入。

可观测：structlog 结构化日志 + OpenTelemetry 分布式追踪 + LangSmith 兼容，调试与优化有据可依。

7．风险与缓解

风险	缓解措施
SQLite 并发写入瓶颈	WAL 模式 + busy_timeout；写入串行化到单 Worker
ChromaDB 大规模性能	提供 Qdrant/Milvus 适配器（可选依赖）
KùzuDB 生态不如 Neo4j	保留 Neo4j 适配器接口；Cypher 兼容降低迁移成本
CLI 不适合非技术用户	后续可叠加 Web UI 作为可选插件，不污染核心
本地嵌入模型质量	默认 BGE 中文模型，可切换 OpenAI/智谱嵌入
LangGraph 版本变动	锁定版本 + 抽象层隔离

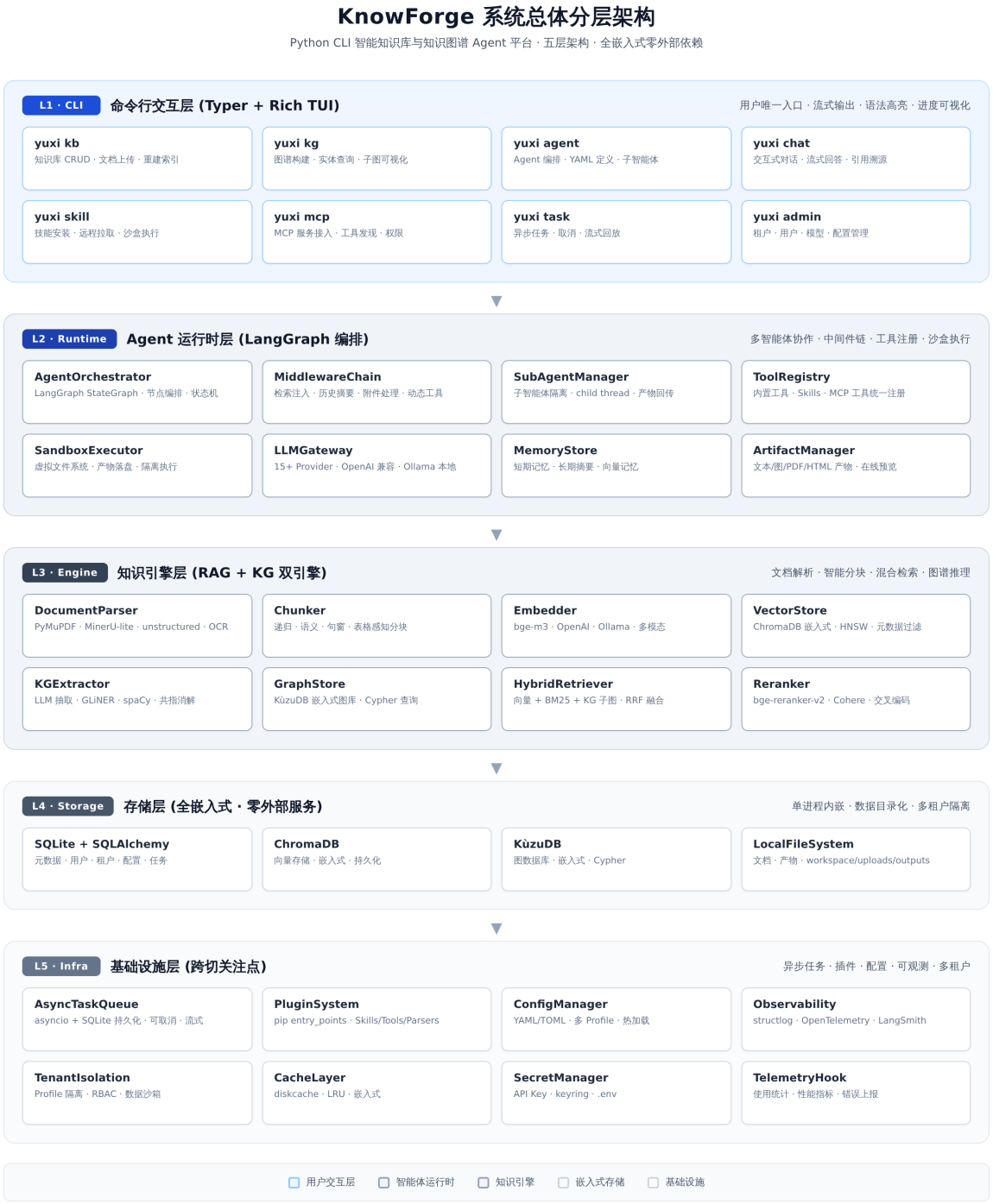
8．后续路线图

- v0.2：Web UI 插件（可选）、Qdrant 适配器、Neo4j 适配器
- v0.3：多模态文档解析（图片 OCR、表格结构化）、Agent 评测框架
- v0.4：联邦学习（多 KnowForge 节点知识共享）、知识图谱可视化 TUI
- v0.5：MCP Server 模式（KnowForge 作为 MCP 服务器被其他 Agent 调用）

第二部分 架构设计文档

1. 架构总览

KnowForge 采用六层分层架构，自顶向下依次为：CLI 表现层、核心服务层、Agent 运行时层、知识引擎层、知识图谱层、存储层，辅以异步任务与插件系统横切关注点。整体架构如图 1 所示。



KnowForge 系统总体架构图

图 1：KnowForge 系统总体架构图

1.1 架构设计哲学

KnowForge 的架构设计遵循三条核心哲学：

关注点分离。每一层有明确职责：CLI 层只管交互，核心层只管基础设施，Agent 层只管编排，知识引擎层只管检索，图谱层只管图结构，存储层只管持久化。层间通过抽象接口通信，互不耦合具体实现。

依赖单向。上层依赖下层，下层不感知上层。例如知识引擎层不知道 Agent 层的存在，Agent 层通过依赖注入调用知识引擎。这使得每一层可独立测试与替换。

嵌入式优先。所有存储组件都是进程内嵌入式（SQLite/ChromaDB/KùzuDB），不依赖任何外部服务进程。这让部署从"编排多个服务"简化为"运行一个 Python 进程"。

1.2 模块依赖关系

模块间的依赖关系如图 2 所示。CLI 层依赖核心服务与 Agent 服务；Agent 运行时依赖知识引擎与图谱；知识引擎依赖存储层；异步任务横切知识引擎与图谱；插件系统通过 entry_points 注入各层。

KnowForge 模块依赖图

图 2：KnowForge 模块依赖图

2. 分层架构详解

2.1 CLI 表现层（knowforge.cli）

职责：提供终端交互入口，渲染输出，处理用户输入。

技术选型：Typer（命令行参数解析）+ Rich（终端渲染）+ prompt-toolkit（交互输入）。

核心组件：

- app.py：Typer 主应用，注册 9 个子命令，处理全局选项（--profile/--verbose/--json/--no-color）
- commands/：9 个子命令模块（kb/kg/agent/chat/skill/mcp/task/config/admin）
- tui/console.py：Rich Console 单例，封装面板、表格、Markdown、代码高亮渲染
- tui/streaming.py：流式输出渲染，使用 Rich Live 实时刷新

设计要点：

第一，全局选项通过 @app.callback() 注册，在所有子命令前执行，负责初始化日志、数据库、租户上下文。

第二，每个子命令是独立的 Typer app，通过 app.add_typer 挂载到主应用，实现命令分组与帮助文档自动生成。

第三，JSON 输出模式（--json）让所有命令输出机器可读格式，便于 CI/CD 集成与脚本化。

第四，流式渲染使用 Rich Live + Markdown 组合，LLM token 流实时渲染为格式化文本，体验接近 ChatGPT。

2.2 核心服务层（knowforge.core）

职责：提供配置、租户、密钥、缓存、日志、可观测性等基础设施。

核心组件：

- config.py：ConfigManager，多 Profile 隔离，YAML + 环境变量 + CLI 参数三层覆盖，热加载
- tenant.py：TenantContext + contextvars，Profile + Tenant 双层隔离
- secrets.py：SecretManager，环境变量 → keyring → secrets.yaml 三级回退
- cache.py：CacheLayer，基于 diskcache 的持久化 LRU，嵌入向量与 LLM 响应缓存
- logging.py：structlog 配置，console 与 JSON 双模式，自动注入 tenant_id 与 trace_id
- observability.py：OpenTelemetry 集成，span 与 metric，惰性初始化
- exceptions.py：KnowForgeError 基类 + 6 个子类异常

设计要点：

配置管理采用 dataclass + YAML 序列化，类型安全且人类可读。多 Profile 通过工作区目录隔离实现，每个 Profile 拥有独立的 SQLite/ChromaDB/KùzuDB 数据目录。

租户上下文使用 contextvars 传递，避免线程/协程串扰。TenantGuard 上下文管理器封装 set/reset，确保异常时也能恢复。

密钥管理三级回退策略兼顾安全与便利：生产用 keyring（操作系统级安全存储），开发用 secrets.yaml（明文但方便），CI 用环境变量。

2.3 Agent 运行时层（knowforge.agent）

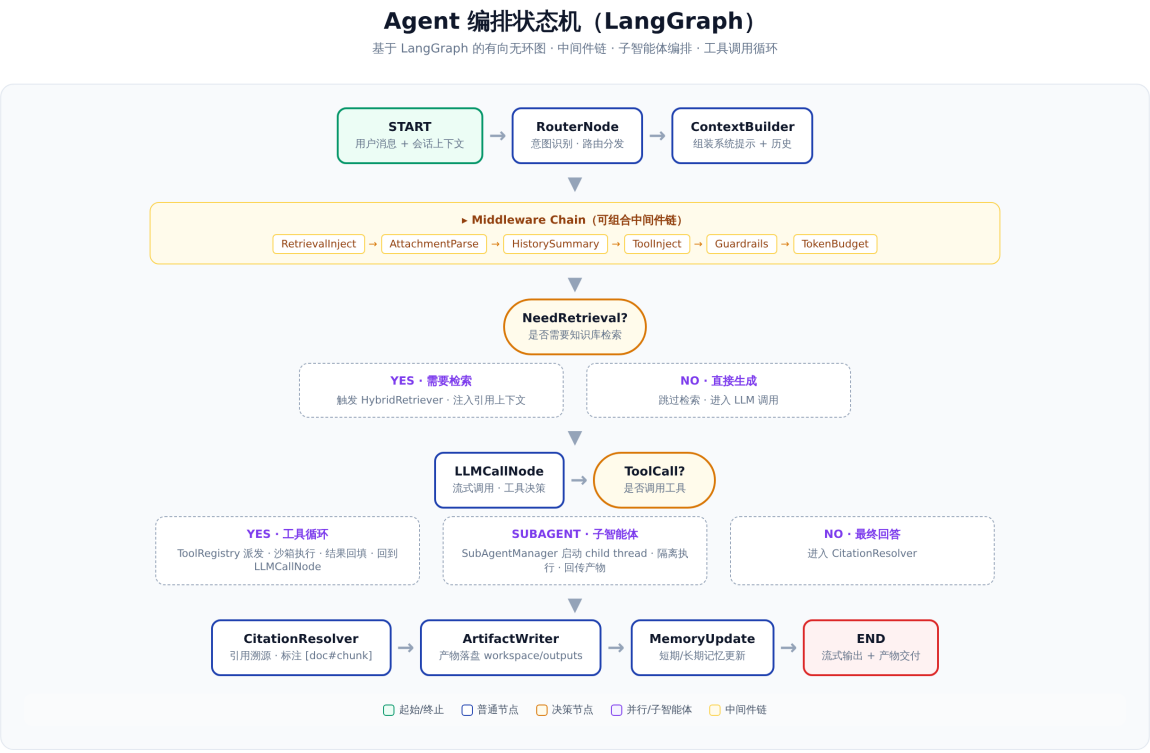
职责：编排智能体执行流程，管理中间件、工具、技能、子智能体、MCP、沙箱。

技术选型：LangGraph（状态机编排）+ 自研中间件链。

核心组件：

- orchestrator.py：AgentOrchestrator，基于状态机的编排核心
- middleware/：中间件链（检索注入、附件解析、历史摘要、动态工具注入）
- tools/：工具注册表 + 内置工具（present_artifacts/ask_user/web_search/install_skill）
- skills/：Skill 抽象 + 加载器 + 内置技能（深度报告、数据报告、图片生成）
- subagent/manager.py：子智能体管理器，支持主智能体派生隔离子任务
- mcp/client.py：MCP 客户端，stdio 与 SSE 两种传输
- sandbox/filesystem.py：沙箱文件系统，每会话独立 workspace/uploads/outputs
- agent_service.py：AgentService 门面，整合所有组件

编排流程如图 3 所示。Agent 执行遵循状态机：START → 中间件链 → 决策（是否检索？）→ 检索 → 生成 → 决策（是否调用工具？）→ 工具执行 → 引用解析 → 记忆更新 → END。



KnowForge Agent 编排状态机

图 3：KnowForge Agent 编排状态机

设计要点：

第一，AgentState 是不可变状态对象，包含消息历史、检索结果、工具调用、产物等所有运行时数据。每个节点接收状态并返回新状态，符合函数式编程思想。

第二，中间件链采用责任链模式，每个中间件独立处理一个关注点（检索、附件、历史、工具），可自由组合与替换。默认链通过 build_default_chain 工厂构建。

第三，工具调用支持同步与异步 handler，通过 asyncio.iscoroutinefunction 自动判断。工具执行结果回填到 messages，LLM 可基于结果继续推理。

第四，子智能体通过 asyncio.Task 实现隔离执行，拥有独立会话与上下文，完成后将结果回传主智能体。支持超时取消。

第五，沙箱文件系统通过路径隔离实现安全，每个会话的 sandbox_id 对应独立的物理目录，Agent 只能通过工具操作该目录内文件，无法越权访问。

2.4 知识引擎层（knowforge.knowledge）

职责：文档解析、分块、嵌入、向量存储、重排、混合检索。

核心组件：

- parser/：ParserRegistry 按扩展名路由，内置 PDF/Office/HTML/Markdown 解析器
- chunker/：递归分块、语义分块、句子窗口分块三种策略
- embedder/：本地 sentence-transformers + OpenAI 兼容嵌入
- vectorstore/：ChromaDB 嵌入式向量存储
- reranker/：BGE cross-encoder 重排
- retriever/：HybridRetriever，向量 + BM25 + KG 三路融合
- kb_service.py：KBService 门面，整合所有知识引擎组件

检索流水线如图 4 所示。完整流程分六阶段：摄入 → 解析 → 分块 → 嵌入 → 检索 → 生成。

KnowForge RAG + KG 检索流水线

图 4：KnowForge RAG + 知识图谱检索流水线

设计要点：

第一，解析器注册表按文件扩展名路由，第三方可通过 entry_points 注册新解析器。所有解析器返回统一的 ParserResult（纯文本 + 结构化元数据）。

第二，分块器抽象统一，三种策略可互换：递归分块（默认，按分隔符层级递归）、语义分块（基于嵌入相似度切分）、句子窗口分块（单句存储，检索时扩展窗口）。

第三，混合检索是核心创新。三路并行检索：向量（语义相似）、BM25（关键词精确）、KG 子图（多跳关系推理）。通过 RRF（Reciprocal Rank Fusion）融合排序，再用 Cross-Encoder 重排 Top-K。

第四，引用溯源贯穿全流程。每个 chunk 携带 doc_id/page/ordinal 元数据，检索结果标注 [doc_name#chunk_id]，LLM 生成时引用约束 prompt，最终 CitationResolver 解析引用并溯源到文档与页码。

2.5 知识图谱层（knowforge.graph）

职责：实体/关系抽取、图存储、子图检索。

核心组件：

- extractor/：LLM 驱动抽取（结构化 prompt + JSON 输出）+ GLiNER 本地 NER
- store/：KùzuDB 嵌入式图存储，Cypher 查询
- retriever/subgraph.py：子图检索器，多跳扩展 + 文本序列化
- kg_service.py：KGService 门面

设计要点：

第一，抽取器双策略：LLM 抽取（高质量，需 API Key）与 GLiNER 本地 NER（零样本，离线）。可根据场景选择。

第二，KùzuDB 嵌入式图存储，单文件持久化，支持 Cypher 查询语言，与 Neo4j 语法兼容度高，迁移成本低。

第三，子图检索器将图结构序列化为 LLM 可读文本（"实体A --[关系]--> 实体B"），注入到 RAG 上下文，让 LLM 基于关系推理而非仅靠语义相似。

2.6 存储层（knowforge.storage）

职责：SQLite + SQLAlchemy ORM，管理元数据。

核心组件：

- database.py：Database 门面，连接池，WAL 模式，会话上下文管理
- models.py：10+ ORM 模型（Tenant/User/KnowledgeBase/Document/Chunk/Entity/Relation/Agent/Conversation/Message/Artifact/Job/JobLog）

设计要点：

第一，所有表带 tenant_id 实现逻辑隔离，配合 Profile 物理隔离形成双层防护。

第二，主键用 ULID（字符串，时间有序），便于排序与分布式生成。

第三，软删除（deleted_at）而非物理删除，支持数据恢复与审计。

第四，JSON 字段用于灵活扩展（metadata/settings/config），避免频繁 schema 变更。

2.7 异步任务（knowforge.tasks）

职责：长耗时任务异步执行，支持取消与流式输出。

核心组件：

- queue.py：JobQueue，SQLite-backed 持久化队列，优先级调度
- worker.py：AsyncWorker，asyncio 事件循环，并发执行，信号处理
- jobs/：具体任务（ingest/extract_kg/reindex）

设计要点：

第一，任务持久化到 SQLite，进程重启不丢失，状态可恢复。

第二，优先级队列：priority ASC + scheduled_at ASC，高优先级任务先执行。

第三，重试机制：max_retries 控制重试次数，失败任务标记 FAILED 并记录错误堆栈。

第四，进度上报：通过 JobLog 表记录任务执行日志，progress 字段实时更新。

2.8 插件系统（knowforge.plugins）

职责：通过 pip entry_points 加载第三方扩展。

五个 entry_points 组：

- knowforge.skills：第三方 Skill 包
- knowforge.tools：第三方 Tool 包
- knowforge.parsers：第三方 Parser 包
- knowforge.embedders：第三方 Embedder 包
- knowforge.llm_providers：第三方 LLM Provider 包

设计要点：

第三方包只需在 pyproject.toml 声明 entry_points，KnowForge 启动时自动发现并注册，无需修改主仓库代码。这是真正的"零侵入"扩展机制。

3. 部署架构

KnowForge 的部署拓扑如图 5 所示。单进程内嵌所有存储组件，外部服务（LLM API、Ollama、MCP Server、MinerU）均为可选。

KnowForge 部署拓扑图

图 5：KnowForge 部署拓扑图

三种部署模式：

单进程模式（默认）：CLI 与 Worker 同进程，适合个人开发者与轻量场景。

Worker 分离模式：knowforge admin serve 启动独立 Worker 进程，CLI 通过 SQLite 队列提交任务，适合长耗时批量任务。

完全离线模式：配置 Ollama 作为 LLM Provider + 本地嵌入模型，无需任何网络访问，适合内网与保密场景。

4．数据流架构

KnowForge 的数据流如图 6 所示。两条主流程：摄入流程（文档 → 解析 → 分块 → 嵌入 → 向量库 + 图谱）与查询流程（用户查询 → 检索 → 生成 → 引用 → 产物）。

KnowForge 数据流图

图 6：KnowForge 数据流图

5. 技术选型矩阵

层	组件	选型	理由
CLI	命令解析	Typer	类型安全、自动补全、文档生成
CLI	终端渲染	Rich	流式、Markdown、表格、语法高亮
CLI	交互输入	prompt-toolkit	高级行编辑、自动补全
核心	配置	PyYAML + dataclass	人类可读、类型安全
核心	日志	structlog	结构化、JSON 友好
核心	可观测	OpenTelemetry	业界标准、LangSmith 兼容
存储	元数据	SQLite + SQLAlchemy	嵌入式、ORM、WAL
存储	向量	ChromaDB	嵌入式、HNSW、Python 原生
存储	图	KùzuDB	嵌入式、Cypher、高性能
存储	缓存	diskcache	持久化、LRU、零依赖
知识	解析	PyMuPDF + python-docx/openpyxl/pptx + BeautifulSoup	全格式覆盖
知识	分块	自研三种策略	递归/语义/句子窗口
知识	嵌入	sentence-transformers + OpenAI	本地优先 + 云端可选
知识	重排	BGE cross-encoder	中文优化、本地推理
知识	检索	rank-bm25 + 自研 RRF	关键词 + 融合排序
Agent	编排	LangGraph	状态机、循环、人机协同
Agent	协议	MCP	标准化工具接入
LLM	Provider	OpenAI/智谱/DeepSeek/Qwen/Ollama	多 Provider 路由

6. 非功能性设计

6.1 性能

- 冷启动 < 3 秒（不含模型加载）
- 单文档摄入 < 10 秒（10 页 PDF，含嵌入）
- 检索延迟 < 500ms（10 万 chunk，Top-10）
- LLM 流式首 token < 1 秒

6.2 可靠性

- 任务持久化到 SQLite，进程崩溃不丢失
- 嵌入向量缓存，避免重复计算
- LLM 调用自动重试（默认 3 次）
- 失败任务可手动重试

6.3 安全

- 密钥三级存储（环境变量/keyring/secrets.yaml）
- 沙箱文件系统路径隔离
- 多租户双层隔离（Profile + Tenant）
- RBAC 角色控制（admin/user/readonly）

6.4 可扩展性

- 插件机制支持第三方零侵入扩展
- 存储层抽象支持替换实现（如 Qdrant 替代 ChromaDB）
- LLM Provider 注册表支持动态注册
- 中间件链可自由组合

7. 架构决策记录（ADR）

ADR-1：嵌入式存储 vs 外部服务

决策：采用嵌入式存储（SQLite/ChromaDB/KùzuDB）。

理由：CLI 场景下，部署简易性优先于横向扩展能力。嵌入式存储让用户免去运维负担，单机性能已满足 90% 场景。对于超大规模场景，可通过存储层抽象替换为外部服务。

ADR-2：CLI 优先 vs Web UI

决策：CLI 优先，Web UI 作为可选插件。

理由：开发者工作流以终端为中心，CLI 可脚本化、可管道化、可版本控制。Web UI 增加前后端复杂度，对 CLI 场景是冗余。后续可叠加 Web UI 插件，不污染核心。

ADR-3：LangGraph 保留 vs 自研编排

决策：保留 LangGraph 作为 Agent 编排核心。

理由：LangGraph 是目前最成熟的 Agent 编排框架，社区活跃、文档完善。自研编排会重复造轮子且难以达到同等成熟度。KnowForge 聚焦于知识引擎与 CLI 体验创新。

ADR-4：entry_points 插件 vs 自定义协议

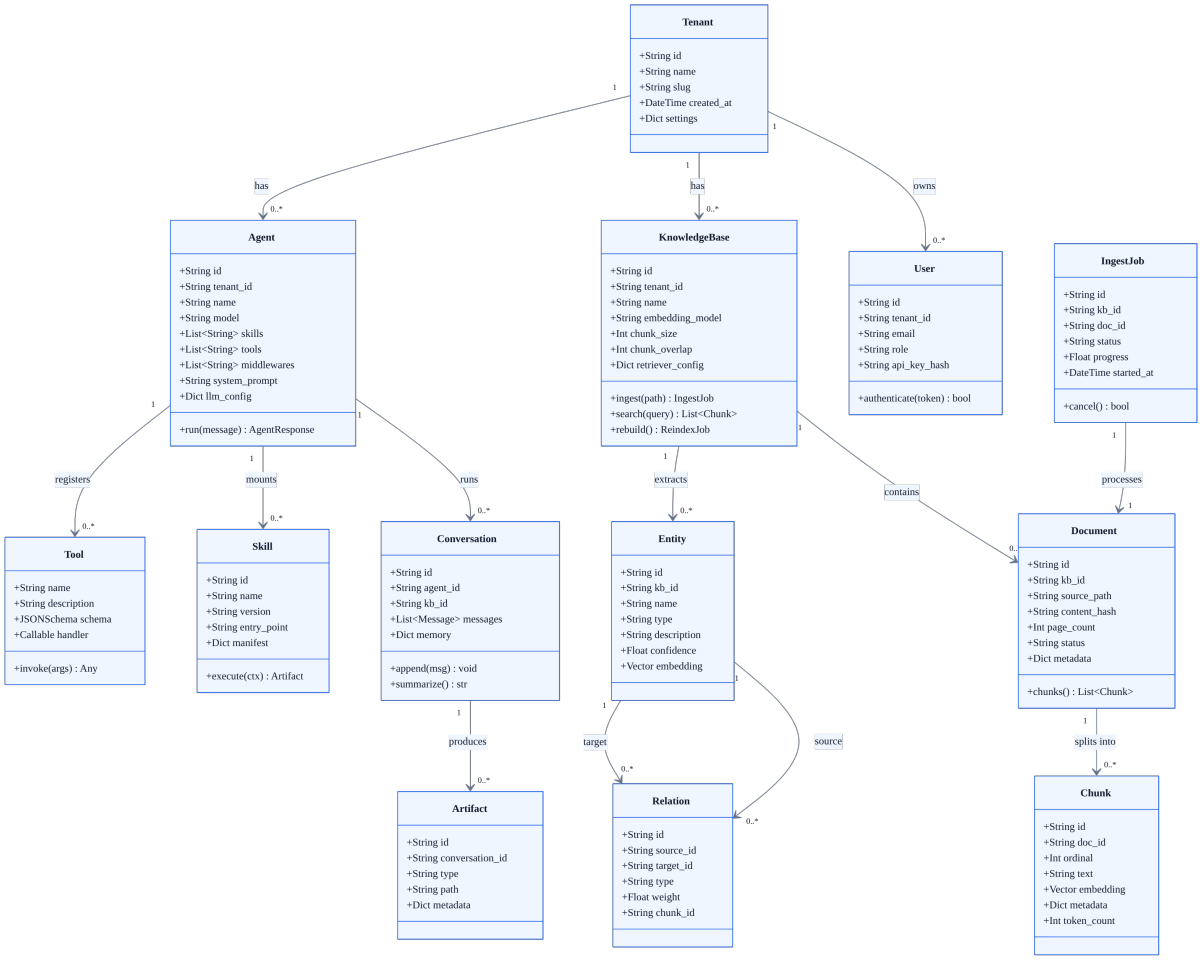
决策：采用 pip entry_points 标准插件机制。

理由：entry_points 是 Python 生态标准，第三方包无需学习新协议，只需在 pyproject.toml 声明即可。这降低了扩展门槛，有利于生态繁荣。

第三部分 详细设计文档

1. 数据模型设计

KnowForge 的数据模型通过 SQLAlchemy ORM 定义，所有表带 tenant_id 实现逻辑隔离。核心实体关系如图 1 所示。



KnowForge UML 类图

图 1：KnowForge 核心领域模型 UML 类图

1.1 租户与用户

Tenant（租户）：多租户隔离的顶层实体。

字段	类型	说明
id	String(26)	ULID 主键
name	String(128)	租户名称
slug	String(64)	URL 友好标识，唯一
settings	JSON	租户级配置

字段	类型	说明
created_at	DateTime	创建时间
deleted_at	DateTime	软删除时间

User（用户）：租户下的用户。

字段	类型	说明
id	String(26)	ULID 主键
tenant_id	String(26)	所属租户（外键）
email	String(255)	邮箱，唯一
role	String(32)	角色：admin/user/readonly
api_key_hash	String(128)	API Key 哈希
created_at	DateTime	创建时间

1.2 知识库与文档

KnowledgeBase（知识库）：知识管理顶层实体。

字段	类型	说明
id	String(26)	ULID 主键
tenant_id	String(26)	所属租户
name	String(128)	知识库名称
embedding_model	String(128)	嵌入模型名
chunk_size	Integer	分块大小
chunk_overlap	Integer	分块重叠
retriever_config	JSON	检索器配置
created_at	DateTime	创建时间

Document（文档）：知识库中的文档。

字段	类型	说明
id	String(26)	ULID 主键
kb_id	String(26)	所属知识库
source_path	String(512)	源文件路径
content_hash	String(64)	内容哈希（去重）
page_count	Integer	页数
status	String(16)	状态：pending/parsed/indexed/failed
metadata	JSON	元数据（标题、作者等）
created_at	DateTime	创建时间

Chunk（文本块）：文档分块后的最小检索单元。

字段	类型	说明
id	String(26)	ULID 主键
doc_id	String(26)	所属文档
kb_id	String(26)	所属知识库（冗余，加速查询）
ordinal	Integer	块序号
text	Text	块文本
embedding	JSON	嵌入向量（ChromaDB 存储副本）
metadata	JSON	元数据（页码、窗口文本等）
token_count	Integer	Token 数
created_at	DateTime	创建时间

1.3 知识图谱

Entity（实体）：知识图谱节点。

字段	类型	说明
id	String(26)	ULID 主键
kb_id	String(26)	所属知识库
name	String(256)	实体名
type	String(64)	类型：person/org/location/concept...
description	Text	描述
confidence	Float	置信度
embedding	JSON	嵌入向量（用于实体检索）
created_at	DateTime	创建时间

Relation（关系）：知识图谱边。

字段	类型	说明
id	String(26)	ULID 主键
kb_id	String(26)	所属知识库
source_id	String(26)	源实体（外键）
target_id	String(26)	目标实体（外键）
type	String(64)	关系类型：works_for/located_in...
description	Text	关系描述
confidence	Float	置信度
created_at	DateTime	创建时间

1.4 智能体与会话

Agent（智能体）：智能体定义。

字段	类型	说明
id	String(26)	ULID 主键
tenant_id	String(26)	所属租户
name	String(128)	智能体名称
llm_provider	String(64)	LLM Provider
llm_model	String(128)	模型名
system_prompt	Text	系统提示词
kb_ids	JSON	关联知识库列表
skills	JSON	挂载技能列表
tools	JSON	可用工具列表
config	JSON	智能体配置
created_at	DateTime	创建时间

Conversation（会话）：一次对话会话。

字段	类型	说明
id	String(26)	ULID 主键
agent_id	String(26)	所属智能体
tenant_id	String(26)	所属租户
kb_ids	JSON	会话关联知识库
title	String(256)	会话标题
memory	JSON	会话记忆
created_at	DateTime	创建时间

Message（消息）：会话中的单条消息。

字段	类型	说明
id	String(26)	ULID 主键
conversation_id	String(26)	所属会话
role	String(32)	角色：system/user/assistant/tool
content	Text	消息内容
metadata	JSON	元数据（引用、工具调用等）
created_at	DateTime	创建时间

1.5 产物与任务

Artifact（产物）：智能体生成的产物。

字段	类型	说明
id	String(26)	ULID 主键

字段	类型	说明
conversation_id	String(26)	所属会话
type	String(32)	类型：markdown/html/pdf/image
path	String(512)	文件路径
metadata	JSON	元数据
created_at	DateTime	创建时间

Job（任务）：异步任务。

字段	类型	说明
id	String(26)	ULID 主键
job_type	String(64)	任务类型
payload	JSON	任务参数
status	String(16)	状态：pending/running/completed/failed/cancelled
priority	Integer	优先级（数字小优先）
progress	Float	进度 0-1
error	Text	错误信息
retry_count	Integer	已重试次数
max_retries	Integer	最大重试次数
kb_id	String(26)	关联知识库
doc_id	String(26)	关联文档
scheduled_at	DateTime	计划时间
started_at	DateTime	开始时间
completed_at	DateTime	完成时间

JobLog（任务日志）：任务执行日志。

字段	类型	说明
id	String(26)	ULID 主键
job_id	String(26)	所属任务
level	String(16)	日志级别
message	Text	日志消息
data	JSON	结构化数据
created_at	DateTime	创建时间

2. 核心接口设计

2.1 LLM Provider 接口

```
class LLMProvider(ABC):
    name: str

    async def chat(
        self,
        messages: list[LLMMessage],
        *,
        tools: list[ToolSchema] | None = None,
        temperature: float = 0.3,
        max_tokens: int = 4096,
        **kwargs,
    ) -> LLMResponse: ...

    async def stream(
        self,
        messages: list[LLMMessage],
        *,
        tools: list[ToolSchema] | None = None,
        **kwargs,
    ) -> AsyncIterator[str]: ...

    async def embed(self, texts: list[str]) -> list[list[float]]: ...
```

LLMRouter 根据 provider 名分发到具体实现，支持降级与负载均衡。

2.2 检索器接口

```
class Retriever(ABC):
    async def retrieve(
        self,
        query: str,
        *,
        kb_id: str,
        top_k: int = 10,
        **kwargs,
    ) -> list[RetrievalResult]: ...
```

HybridRetriever 实现三路融合检索。

2.3 中间件接口

```
class BaseMiddleware(ABC):
    name: str

    async def __call__(self, state: AgentState) -> AgentState: ...
```

MiddlewareChain 按顺序执行多个中间件。

2.4 工具接口

```
class ToolRegistry:
    def register(self, schema: ToolSchema,
                 handler: Callable[[dict], Any]) -> None: ...
    def get(self, name: str) -> Callable | None: ...
    def list_schemas(self) -> list[ToolSchema]: ...
```

2.5 Skill 接口

```
class Skill(ABC):
    name: str
    version: str

    async def execute(self, ctx: dict[str, Any]) -> dict[str, Any]: ...
    def manifest(self) -> SkillManifest: ...
```

2.6 图存储接口

```
class GraphStore(ABC):
    async def add_entities(self, kb_id: str, entities: list[Entity]) -> int: ...
    async def add_relations(self, kb_id: str, relations: list[Relation]) -> int: ...
    async def search_entities(self, kb_id: str, query: str, *,
                              top_k: int = 10) -> list[Entity]: ...
    async def get_subgraph(self, kb_id: str, entity_ids: list[str], *,
                           hops: int = 2, max_nodes: int = 50) -> dict: ...
```

3. 核心算法设计

3.1 混合检索融合算法（RRF）

KnowForge 的混合检索采用 Reciprocal Rank Fusion（RRF）算法融合三路检索结果。RRF 是一种无需归一化的排名融合方法，公式如下：

```
score(d) = Σ (1 / (k + rank_i(d)))
```

其中 d 是文档，i 遍历三路检索（vector/bm25/kg），rank_i(d) 是文档 d 在第 i 路检索中的排名（从 1 开始），k 是平滑常数（默认 60）。

RRF 的优势在于：无需校准不同检索器的分数尺度（向量余弦相似度 0-1，BM25 分数可能 >10，KG 无分数），仅依赖排名，鲁棒性强。

实现伪代码：

```
def rrf_fuse(results: dict[str, list[Chunk]], weights: dict[str, float],
             k: int = 60) -> list[tuple[str, float]]:
    scores = defaultdict(float)
    for source, chunks in results.items():
        w = weights.get(source, 1.0)
        for rank, chunk in enumerate(chunks):
            scores[chunk.id] += w / (k + rank + 1)
    return sorted(scores.items(), key=lambda x: x[1], reverse=True)
```

3.2 Cross-Encoder 重排算法

RRF 融合后取 Top-N（默认 20），再用 Cross-Encoder 重排到 Top-K（默认 5）。Cross-Encoder 与 Bi-Encoder 的区别在于：Bi-Encoder 分别编码 query 与 doc 再算相似度（适合检索），Cross-Encoder 将 query 与 doc 拼接后输入模型，输出相关性分数（适合重排，精度更高但速度慢）。

KnowForge 默认使用 BAAI/bge-reranker-base，中文场景表现优秀。重排伪代码：

```
def rerank(query: str, documents: list[str], top_k: int = 5) -> list[RerankResult]:
    pairs = [(query, d) for d in documents]
    scores = model.predict(pairs)  # Cross-Encoder
```

```
    ranked = sorted(zip(documents, scores), key=lambda x: x[1], reverse=True)
    return ranked[:top_k]
```

3.3 知识图谱子图检索算法

子图检索分四步：

1. 实体匹配：从查询中抽取关键词，在图中按名称模糊匹配实体

2. 种子选择：取匹配度最高的 Top-5 实体作为种子

3. 多跳扩展：从种子出发，BFS 扩展 N 跳（默认 2 跳），限制最大节点数（默认 50）

4. 文本序列化：将子图转为"实体A --[关系]--> 实体B"格式文本，注入 LLM 上下文

Cypher 查询示例（2 跳子图）：

```
MATCH (e:Entity {kb_id: $kb})-[:Related*1..2]-(n:Entity)
WHERE e.id IN $seed_ids
RETURN e, n, collect(relationships(p))
LIMIT 50
```

3.4 历史摘要压缩算法

当对话历史超过阈值（默认 15 条）时，将早期消息压缩为摘要。算法：

1. 保留最近 N 条消息（默认 10 条）不动

2. 将更早的消息构造为 transcript

3. 调用 LLM 生成摘要（temperature=0.1，max_tokens=512）

4. 用摘要 system message 替换早期消息

这避免了上下文超限，同时保留近期对话的完整细节。

3.5 动态工具注入算法

根据用户消息关键词动态选择工具：

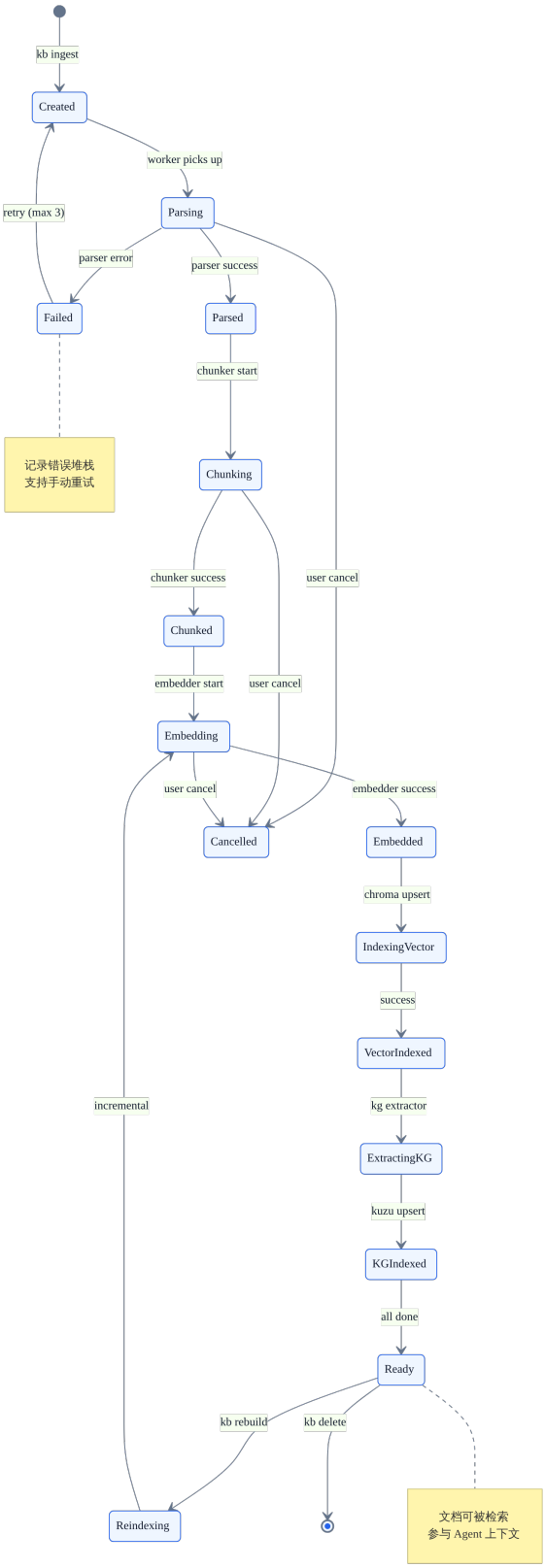
```
KEYWORD_MAP = {
    "■■|■|■■■|■■": ["image_gen"],
    "■■|■■|■■|■■■■": ["deep_report", "data_report"],
    "■■|■■|■■|■■": ["web_search"],
    "■■|skill|■■": ["install_skill"],
}

def inject_tools(message: str, registry: dict) -> list[ToolSchema]:
    selected = []
    for pattern, tool_names in KEYWORD_MAP.items():
        if re.search(pattern, message, re.IGNORECASE):
            for tn in tool_names:
                if tn in registry and tn not in [t.name for t in selected]:
                    selected.append(registry[tn])
    return selected
```

4. 关键流程设计

4.1 文档摄入流程

文档摄入流程的状态转换如图 2 所示。



KnowForge 文档摄入状态图

图 2：文档摄入状态转换图

完整流程：

1. Created：文档记录创建，status=pending

2. Parsing：Worker 拾取任务，调用 ParserRegistry.parse() 解析为纯文本

3. Parsed：解析成功，记录页数与元数据

4. Chunking：调用 Chunker.chunk() 切分为块

5. Chunked：分块完成，写入 Chunk 表

6. Embedding：批量调用 Embedder.embed() 生成向量

7. Embedded：嵌入完成

8. IndexingVector：写入 ChromaDB 向量库

9. VectorIndexed：向量索引完成

10. ExtractingKG：调用 KGExtractor 抽取实体与关系

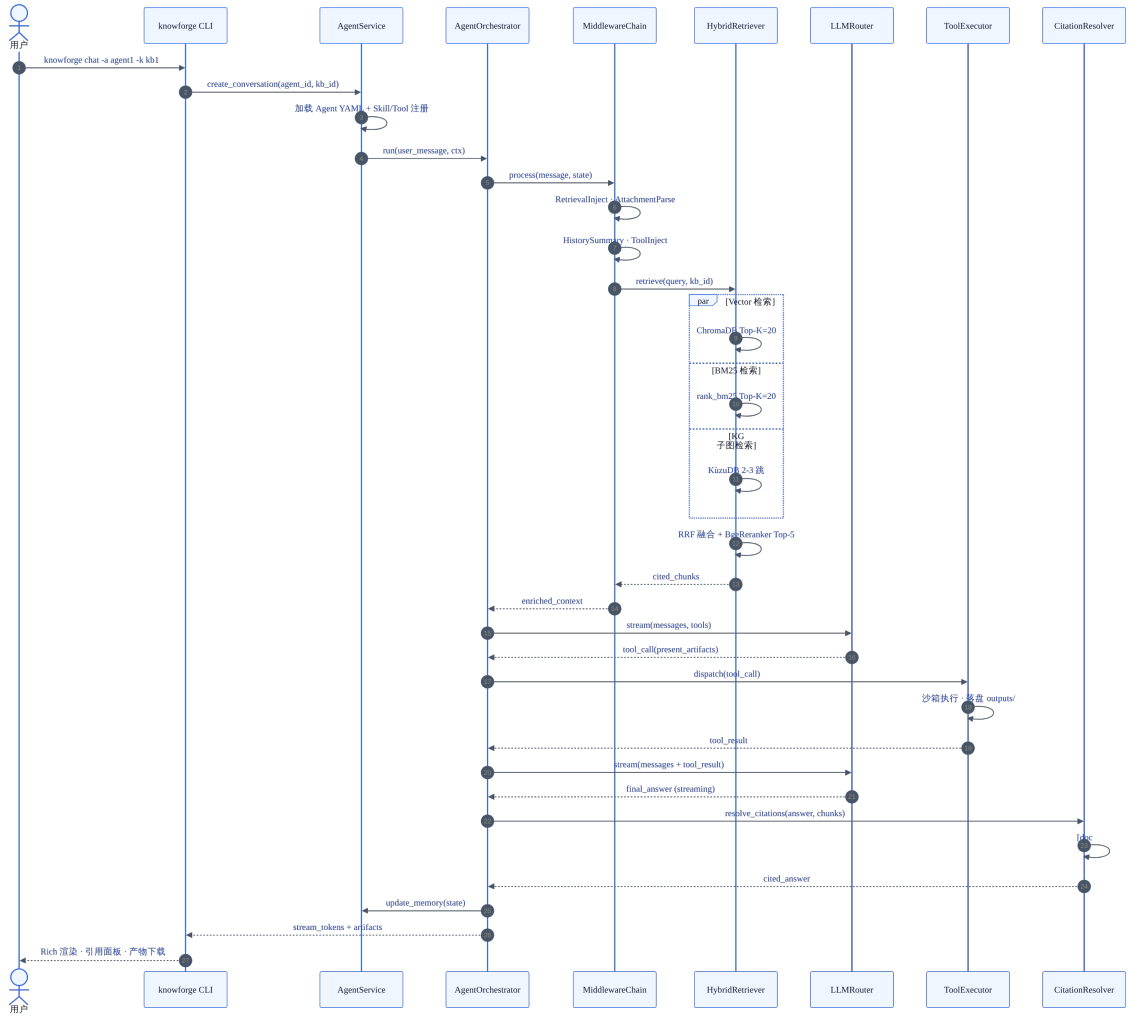
11. KGIndexed：写入 KùzuDB 图库

12. Ready：文档可被检索

任何阶段失败进入 Failed 状态，支持手动重试（最多 3 次）。用户可随时取消，进入 Cancelled 状态。

4.2 Agent 执行流程

Agent 执行的完整序列如图 3 所示。



KnowForge Agent 执行序列图

图 3：KnowForge Agent 执行序列图

关键步骤：

1. 用户通过 CLI 发起对话
2. AgentService 加载 Agent 定义、Skill、Tool
3. AgentOrchestrator 启动，调用 MiddlewareChain 处理消息
4. 中间件链顺序执行：附件解析 → 检索注入 → 动态工具注入 → 历史摘要
5. 检索注入中间件并行执行三路检索（向量/BM25/KG），RRF 融合 + 重排
6. LLM 流式生成，可能触发工具调用
7. 工具调用通过 ToolExecutor 在沙箱中执行
8. 工具结果回填，LLM 继续生成
9. CitationResolver 解析引用并溯源
10. ArtifactWriter 将产物落盘到 outputs/
11. 流式输出到 CLI，Rich Live 实时渲染

4.3 组件协作流程

KnowForge 的组件协作如图 4 所示。

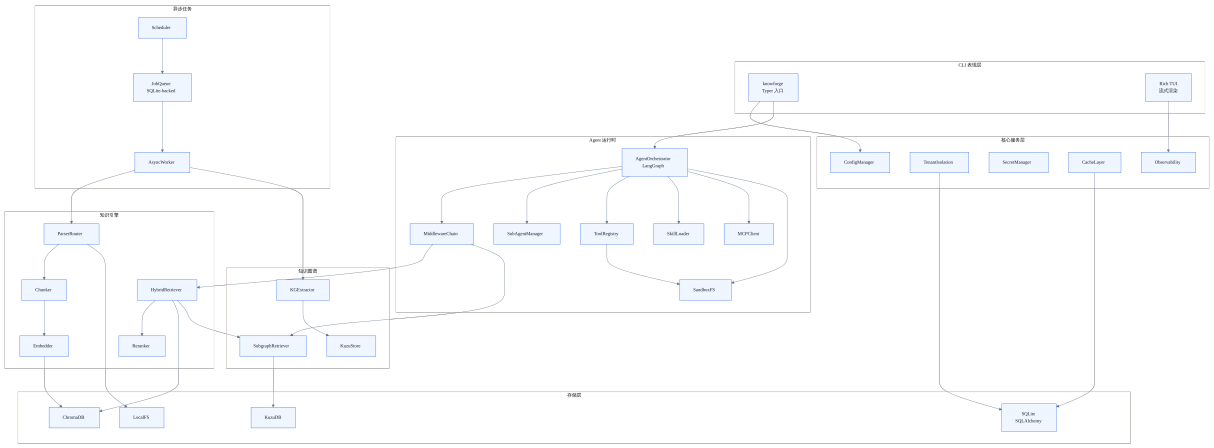


图 4：KnowForge UML 组件图

5. CLI 命令设计

KnowForge 的 CLI 命令树如图 5 所示。

KnowForge CLI 命令树

图 5：KnowForge CLI 命令树

5.1 命令规范

- 主命令：knowforge
- 子命令：9 个（kb/kg/agent/chat/skill/mcp/task/config/admin）
- 全局选项：--profile/--verbose/--json/--no-color/--config
- 命名风格：动词在前（create/list/show/delete）或名词在前（kb/agent）

5.2 输出规范

- 默认：Rich 彩色输出，面板/表格/Markdown 渲染
- --json：单行 JSON 输出，机器可读
- --no-color：禁用彩色，适合日志重定向

- --verbose：DEBUG 级别日志

6. 安全设计

6.1 密钥管理

三级回退策略：

1. 环境变量（最高优先级）：KNOWFORGE_OPENAI_API_KEY，适合 CI/CD

2. 系统 keyring：操作系统级安全存储，适合生产

3. secrets.yaml（最低优先级）：明文存储，权限 600，仅开发用

6.2 沙箱隔离

每个会话拥有独立 sandbox_id，物理路径 {workspace}/sandboxes/{sandbox_id}/{uploads,outputs,workspace}。Agent 通过工具操作文件，无法越权访问其他会话目录。

6.3 多租户隔离

双层隔离：

- Profile（物理隔离）：独立工作区目录，独立 SQLite/ChromaDB/KùzuDB 数据目录
- Tenant（逻辑隔离）：Profile 内通过 tenant_id 字段隔离，配合 RBAC 角色控制

6.4 输入校验

- 所有外部输入（文件路径、用户消息、API 参数）经过 Pydantic 校验
- SQL 注入防护：SQLAlchemy 参数化查询
- 路径穿越防护：沙箱内路径规范化与白名单校验

7. 可观测性设计

7.1 日志

structlog 配置两种输出：

- console 模式：彩色、人类可读，适合 CLI 交互
- json 模式：单行 JSON，适合日志聚合（ELK/Loki）

自动注入字段：timestamp、level、tenant_id、user_id、trace_id、span_id。

7.2 追踪

OpenTelemetry 集成，关键操作自动埋点：

- llm.chat / llm.stream / llm.embed：LLM 调用
- retrieval.vector / retrieval.bm25 / retrieval.kg：检索
- agent.tool_call：工具调用

- agent.skill_execute：技能执行

与 LangSmith 兼容，可通过环境变量 LANGCHAIN_TRACING_V2=true 启用 LangSmith 追踪。

7.3 指标

关键指标：

- LLM 调用次数、Token 数、延迟
- 检索命中率、Top-K 准确率
- 任务队列深度、Worker 利用率
- 缓存命中率

8. 扩展性设计

8.1 插件机制

五个 entry_points 组：

```
[project.entry-points."knowforge.skills"]
my_skill = "my_pkg.skills:MySkill"

[project.entry-points."knowforge.tools"]
my_tool = "my_pkg.tools:my_tool"

[project.entry-points."knowforge.parsers"]
my_format = "my_pkg.parsers:MyParser"

[project.entry-points."knowforge.embedders"]
my_embedder = "my_pkg.embedders:MyEmbedder"

[project.entry-points."knowforge.llm_providers"]
my_provider = "my_pkg.providers:MyProvider"
```

第三方包只需声明 entry_points，KnowForge 启动时通过 PluginLoader 自动发现并注册。

8.2 存储层抽象

VectorStore、GraphStore 均为抽象基类，可通过工厂方法替换实现：

```
def create_vector_store(name: str = "chroma", **kwargs) -> VectorStore:
    if name == "chroma":
        return ChromaVectorStore(**kwargs)
    # ■■■■■■■
    # if name == "qdrant":
    #     return QdrantVectorStore(**kwargs)
    # if name == "milvus":
    #     return MilvusVectorStore(**kwargs)
```

8.3 中间件可组合

中间件链可自由组合，通过 build_default_chain 工厂提供默认配置，用户也可自定义：

```
chain = MiddlewareChain()
chain.add(AttachmentParseMiddleware())
chain.add(RetrievalInjectMiddleware(kb_service))
chain.add(ToolInjectMiddleware(tool_registry))
```

```
chain.add(HistorySummaryMiddleware())
# ■■■■■■■■■
chain.add(MyCustomMiddleware())
```

第四部分 具体实现文档

1. 项目结构

KnowForge 采用标准 Python 包结构，代码组织遵循"按层分包、按职责分模块"原则。

```
knowforge/                          # ■■■
■■■ pyproject.toml                   # ■■■■■ + ■■ + entry_points
■■■ README.md                        # ■■■■■
■■■ LICENSE                          # MIT ■■■
■■■ knowforge/                       # Python ■
■   ■■■ __init__.py                  # ■■■■■
■   ■■■ __main__.py                  # `python -m knowforge` ■■
■   ■■■ cli/                         # CLI ■■■
■   ■   ■■■ app.py                   # Typer ■■■
■   ■   ■■■ commands/                # 9 ■■■■■
■   ■   ■   ■■■ kb.py                # ■■■■■■
■   ■   ■   ■■■ kg.py                # ■■■■■■■
■   ■   ■   ■■■ agent.py             # ■■■■■■
■   ■   ■   ■■■ chat.py              # ■■■■■
■   ■   ■   ■■■ skill.py             # ■■■■■
■   ■   ■   ■■■ mcp.py               # MCP ■■■■■■
■   ■   ■   ■■■ task.py              # ■■■■■■■
■   ■   ■   ■■■ config.py            # ■■■■■
■   ■   ■   ■■■ admin.py             # ■■■■■
■   ■   ■■■ tui/                     # ■■ UI ■■
■   ■       ■■■ console.py           # Rich Console ■■
■   ■       ■■■ streaming.py         # ■■■■■
■   ■■■ core/                        # ■■■■■■
■   ■   ■■■ config.py                # ■■■■■■
■   ■   ■■■ tenant.py                # ■■■■■■■
■   ■   ■■■ secrets.py               # ■■■■■
■   ■   ■■■ cache.py                 # ■■■■
■   ■   ■■■ logging.py               # ■■■■■■
■   ■   ■■■ observability.py         # OpenTelemetry ■■
■   ■   ■■■ exceptions.py            # ■■■■■
■   ■■■ llm/                         # LLM ■■■
■   ■   ■■■ base.py                  # Provider ■■
■   ■   ■■■ openai_provider.py       # OpenAI ■■
■   ■   ■■■ zhipu_provider.py        # ■■ GLM
■   ■   ■■■ deepseek_provider.py     # DeepSeek
■   ■   ■■■ qwen_provider.py         # ■■■■■
■   ■   ■■■ ollama_provider.py       # Ollama ■■
■   ■   ■■■ router.py                # LLM ■■■
■   ■■■ knowledge/                   # ■■■■■■
■   ■   ■■■ parser/                  # ■■■■■
■   ■   ■   ■■■ base.py              # Parser ■■ + Registry
■   ■   ■   ■■■ pdf.py               # PyMuPDF
■   ■   ■   ■■■ office.py            # docx/xlsx/pptx
■   ■   ■   ■■■ html.py              # BeautifulSoup
■   ■   ■   ■■■ markdown.py          # Markdown/■■■
■   ■   ■■■ chunker/                 # ■■■
■   ■   ■   ■■■ base.py              # Chunker ■■
■   ■   ■   ■■■ recursive.py         # ■■■■■
■   ■   ■   ■■■ semantic.py          # ■■■■■
■   ■   ■   ■■■ sentence_window.py   # ■■■■■
```

```
■   ■   ■■■ embedder/              # ■■■
■   ■   ■   ■■■ base.py            # Embedder ■■
■   ■   ■   ■■■ local.py           # sentence-transformers
■   ■   ■   ■■■ openai.py          # OpenAI ■■
■   ■   ■■■ vectorstore/           # ■■■■
■   ■   ■   ■■■ base.py            # VectorStore ■■
■   ■   ■   ■■■ chroma.py          # ChromaDB
■   ■   ■■■ reranker/              # ■■■
■   ■   ■   ■■■ base.py            # Reranker ■■
■   ■   ■   ■■■ bge.py             # BGE cross-encoder
■   ■   ■■■ retriever/             # ■■■
■   ■   ■   ■■■ base.py            # Retriever ■■
■   ■   ■   ■■■ hybrid.py          # ■■■■■RRF■
■   ■   ■■■ kb_service.py          # ■■■■■■■■
■   ■■■ graph/                     # ■■■■■
■   ■   ■■■ extractor/             # ■■■
■   ■   ■   ■■■ base.py            # KGExtractor ■■
■   ■   ■   ■■■ llm.py             # LLM ■■■■
■   ■   ■   ■■■ gliner.py          # GLiNER ■■ NER
■   ■   ■■■ store/                 # ■■■
■   ■   ■   ■■■ base.py            # GraphStore ■■
■   ■   ■   ■■■ kuzu.py            # KùzuDB
■   ■   ■■■ retriever/
■   ■   ■   ■■■ subgraph.py        # ■■■■■
■   ■   ■■■ kg_service.py          # ■■■■■■■■■
■   ■■■ agent/                     # Agent ■■■■
■   ■   ■■■ orchestrator.py        # LangGraph ■■■
■   ■   ■■■ agent_service.py       # Agent ■■■■
■   ■   ■■■ middleware/            # ■■■■
■   ■   ■   ■■■ base.py            # Middleware ■■
■   ■   ■   ■■■ retrieval.py       # ■■■■
■   ■   ■   ■■■ attachment.py      # ■■■■
■   ■   ■   ■■■ history.py         # ■■■■
■   ■   ■   ■■■ tool_inject.py     # ■■■■■■
■   ■   ■■■ tools/                 # ■■■■
■   ■   ■   ■■■ registry.py        # ToolRegistry
■   ■   ■   ■■■ builtin.py         # ■■■■
■   ■   ■■■ skills/                # ■■■■
■   ■   ■   ■■■ base.py            # Skill ■■
■   ■   ■   ■■■ loader.py          # SkillLoader
■   ■   ■   ■■■ builtin/           # ■■■■
■   ■   ■       ■■■ deep_report.py
■   ■   ■       ■■■ data_report.py
■   ■   ■       ■■■ image_gen.py
■   ■   ■■■ subagent/
■   ■   ■   ■■■ manager.py         # ■■■■■■■
■   ■   ■■■ mcp/
■   ■   ■   ■■■ client.py          # MCP ■■■
■   ■   ■■■ sandbox/
■   ■       ■■■ filesystem.py     # ■■■■■■
■   ■■■ storage/                   # ■■■
■   ■   ■■■ database.py            # SQLAlchemy ■■
■   ■   ■■■ models.py              # ORM ■■
■   ■■■ tasks/                     # ■■■■
■   ■   ■■■ queue.py               # SQLite ■■■■
■   ■   ■■■ worker.py              # AsyncWorker
■   ■   ■■■ jobs/                  # ■■■■
■   ■       ■■■ ingest.py
```

```
■   ■       ■■■ extract_kg.py
■   ■       ■■■ reindex.py
■   ■■■ plugins/                    # ■■■■
■       ■■■ loader.py               # PluginLoader
■■■ tests/                          # ■■■■
■   ■■■ test_core.py
■   ■■■ test_kb.py
■   ■■■ test_agent.py
■   ■■■ test_cli.py
■■■ examples/                       # ■■
■   ■■■ config.yaml
■   ■■■ quickstart.sh
■   ■■■ agent_research_assistant.yaml
■   ■■■ skill_legal_analysis.yaml
■   ■■■ skill_tech_research.yaml
■■■ docs/                           # ■■
    ■■■ 01_solution.md
    ■■■ 02_architecture.md
    ■■■ 03_detailed_design.md
    ■■■ 04_implementation.md
```

代码统计：107 个 Python 文件，约 7100 行代码，纯 Python 实现，无前端代码。

2. 关键模块实现

2.1 配置管理器（core/config.py）

ConfigManager 实现多 Profile 隔离与三层配置覆盖：

```
class ConfigManager:
    def __init__(self, profile: str = "default"):
        self.profile = profile
        self.workspace_dir = Path.home() / ".knowforge" / "profiles" / profile
        self.config_file = self.workspace_dir / "config.yaml"
        self._config: KnowForgeConfig | None = None

    def load(self) -> KnowForgeConfig:
        if self._config and self._mtime_unchanged():
            return self._config
        if self.config_file.exists():
            with self.config_file.open("r", encoding="utf-8") as f:
                data = yaml.safe_load(f) or {}
        else:
            data = {}
        # ■■■■■■■
        self._apply_env_overrides(data)
        self._config = KnowForgeConfig(**data)
        return self._config
```

关键设计：

- 多 Profile：每个 Profile 对应独立工作区目录，物理隔离 SQLite/ChromaDB/KùzuDB
- 热加载：通过 mtime 检测配置文件变化，自动重载
- 环境变量覆盖：KNOWFORGE_LLM_DEFAULT_PROVIDER 等环境变量覆盖 YAML 配置
- 类型安全：基于 dataclass 做强类型校验

2.2 混合检索器（knowledge/retriever/hybrid.py）

HybridRetriever 实现三路融合检索：

```
class HybridRetriever(Retriever):
    async def retrieve(self, query, *, kb_id, top_k=10):
        # 1. ■■■■
        query_emb = await self.embedder.embed([query])
        vector_results = await self.vector_store.search(
            f"kb_{kb_id}", query_emb[0], top_k=top_k * 2,
            where={"kb_id": kb_id}
        )

        # 2. BM25 ■■
        bm25_results = []
        if self.enable_bm25 and (index := await self._get_bm25_index(kb_id)):
            tokens = self._tokenize(query)
            scores = index.get_scores(tokens)
            bm25_results = [/* top chunks by score */]

        # 3. KG ■■■■
        kg_results = []
        if self.enable_kg and self.kg_service:
            subgraph = await self.kg_service.search(query, kb_id=kb_id)
            # ■■■■■■■■ chunk
            kg_results = self._subgraph_to_chunks(subgraph)

        # 4. RRF ■■
        fused = self._rrf_fuse(
            {"vector": vector_results, "bm25": bm25_results, "kg": kg_results},
            self.weights
        )

        # 5. Cross-Encoder ■■
        if self.reranker and len(fused) > top_k:
            reranked = await self.reranker.rerank(
                query, [c.text for c in fused[:top_k * 2]],
                ids=[c.chunk_id for c in fused[:top_k * 2]],
                top_k=top_k
            )
            fused = [/* map reranked back to chunks */]
        return fused[:top_k]
```

RRF 融合算法核心：

```
def _rrf_fuse(self, results, weights, k=60):
    scores = defaultdict(float)
    for source, chunks in results.items():
        w = weights.get(source, 1.0)
        for rank, chunk in enumerate(chunks):
            scores[chunk.id] += w / (k + rank + 1)
    return sorted(scores.items(), key=lambda x: x[1], reverse=True)
```

2.3 Agent 编排器（agent/orchestrator.py）

AgentOrchestrator 实现基于状态机的编排：

```
class AgentOrchestrator:
    async def run(self, message, *, agent_id="", kb_ids=None,
                  conversation_id="") -> AgentResponse:
```

```
        state = AgentState(
            user_message=message,
            agent_id=agent_id,
            kb_ids=kb_ids or [],
            conversation_id=conversation_id,
        )
        # 1. ■■■■■■
        if self.middleware_chain:
            state = await self.middleware_chain(state)

        # 2. ■■■■■■ + ■■■
        system_prompt = self._build_system_prompt(state)

        # 3. ■■■■■■■■■■■■■■■
        while state.iterations < state.max_iterations:
            state.iterations += 1
            messages = [LLMMessage(role="system", content=system_prompt)]
            messages.extend(state.messages)
            messages.append(LLMMessage(role="user", content=state.user_message))

            response = await self.llm_router.chat(
                messages, tools=state.available_tools or None
            )

            if not response.tool_calls:
                state.final_answer = response.content
                break

            # ■■■■
            for call in response.tool_calls:
                result = await self._dispatch_tool(call)
                state.tool_results.append(result)
                # ■■■■■■■■■■
                state.messages.append(
                    LLMMessage(role="tool", content=str(result),
                               tool_call_id=call.id)
                )

        # 4. ■■■■
        state.citations = self._resolve_citations(state)
        return AgentResponse(
            answer=state.final_answer,
            citations=state.citations,
            artifacts=state.artifacts,
        )
```

2.4 中间件链（agent/middleware/）

中间件采用责任链模式，每个中间件独立处理一个关注点：

```
class RetrievalInjectMiddleware(BaseMiddleware):
    async def __call__(self, state: AgentState) -> AgentState:
        if not state.kb_ids:
            return state
        all_chunks = []
        for kb_id in state.kb_ids:
            results = await self.kb_service.search(
                state.user_message, kb_id=kb_id, top_k=self.top_k
```

```
            )
            all_chunks.extend(results)
        # ■■ + ■■
        state.retrieved_chunks = self._dedup_and_rank(all_chunks)
        # KG ■■■■
        if self.enable_kg and self.kb_service.kg:
            subgraph = await self.kb_service.kg.search(
                state.user_message, kb_id=state.kb_ids[0]
            )
            state.kg_subgraph = subgraph
        return state
```

2.5 沙箱文件系统（agent/sandbox/filesystem.py）

```
class SandboxFS:
    def __init__(self, sandbox_id=None):
        cfg = get_config().load()
        self.root = cfg.agent.sandbox_root
        if sandbox_id is None:
            ctx = current_tenant()
            sandbox_id = ctx.sandbox_id or "default"
        self.base = self.root / sandbox_id
        self.uploads = self.base / "uploads"
        self.outputs = self.base / "outputs"
        self.workspace = self.base / "workspace"
        for p in [self.uploads, self.outputs, self.workspace]:
            p.mkdir(parents=True, exist_ok=True)

    def write_output(self, name, content):
        path = self.outputs / name
        path.parent.mkdir(parents=True, exist_ok=True)
        if isinstance(content, str):
            path.write_text(content, encoding="utf-8")
        else:
            path.write_bytes(content)
        return path
```

2.6 异步 Worker (tasks/worker.py)

```
class AsyncWorker:
    async def run(self):
        self._load_handlers()
        loop = asyncio.get_event_loop()
        for sig in (signal.SIGINT, signal.SIGTERM):
            loop.add_signal_handler(sig, self._stop.set)
        while not self._stop.is_set():
            pending = self.queue.list_pending(limit=self.max_concurrency * 2)
            for job in pending:
                if job.id in self._running:
                    continue
                if len(self._running) >= self.max_concurrency:
                    break
                task = asyncio.create_task(self._process_one(job))
                self._running[job.id] = task
            await asyncio.sleep(self.poll_interval if not pending else 0.1)
        if self._running:
            await asyncio.gather(*self._running.values(), return_exceptions=True)

    async def _process_one(self, job):
        async with self._semaphore:
            try:
                self.queue.mark_running(job.id)
                handler = self.HANDLERS.get(job.job_type)
                if not handler:
                    raise ValueError(f"■■■■■■: {job.job_type}")
                await handler(job, self.queue)
                self.queue.mark_completed(job.id)
            except Exception as e:
                if job.retry_count < job.max_retries:
                    self.queue.mark_retry(job.id)
                else:
                    self.queue.mark_failed(job.id, str(e))
```

3. CLI 命令实现

3.1 主应用 (cli/app.py)

```
app = typer.Typer(
    name="knowforge",
    help="KnowForge - ■■■ Python CLI ■■■■■■■■■■ Agent ■■■■■",
    no_args_is_help=True,
    rich_markup_mode="rich",
)

@app.callback()
def main_callback(
    ctx: typer.Context,
    profile: str = typer.Option(None, "--profile", "-p"),
    verbose: bool = typer.Option(False, "--verbose", "-v"),
    json_output: bool = typer.Option(False, "--json"),
    no_color: bool = typer.Option(False, "--no-color"),
    config_path: str = typer.Option(None, "--config"),
):
    configure_logging(level="DEBUG" if verbose else "INFO",
                      fmt="json" if json_output else "console")
    get_config().ensure_dirs()
    init_db()
    set_tenant(TenantContext(tenant_id="default", user_id="cli", user_role="admin"))

# ■■■■■■
app.add_typer(kb.app, name="kb", help="■■■■■■")
app.add_typer(kg.app, name="kg", help="■■■■■■■")
# ... 9 ■■■■
```

3.2 对话命令（cli/commands/chat.py）

```
@app.command()
def chat(
    agent_id: str = typer.Option(..., "--agent", "-a"),
    kb: str = typer.Option(None, "--kb", "-k"),
    message: str = typer.Option(None, "--message", "-m"),
):
    # ■■■■
    with get_db() as s:
        agent = s.query(Agent).filter_by(name=agent_id).first()
        conv = Conversation(agent_id=agent.id, kb_ids=[kb] if kb else agent.kb_ids)
        s.add(conv)
        s.flush()
        conv_id = conv.id

    service = AgentService()

    async def run_once(msg):
        console.print(f"\n[bold cyan]■:[/bold cyan] {msg}")
        with Live(console=console, refresh_per_second=20) as live:
            buffer = ""
            async for token in service.stream_chat(conv_id, msg):
                buffer += token
                live.update(Markdown(buffer))

    if message:
        asyncio.run(run_once(message))
        return

    # REPL ■■
    while True:
        msg = console.input("[bold green]■>[/bold green] ").strip()
        if msg in ("/exit", "/quit"):
            break
        asyncio.run(run_once(msg))
```

3.3 知识库命令（cli/commands/kb.py）

```
@app.command("ingest")
def ingest(
    kb_id: str = typer.Argument(...),
    path: Path = typer.Argument(..., exists=True),
    async_mode: bool = typer.Option(False, "--async"),
):
    if async_mode:
        q = JobQueue()
        job_id = q.enqueue(job_type="ingest", kb_id=real_id,
                           payload={"path": str(path)})
        print_panel(f"Job ID: {job_id}", title="■■■■", style="yellow")
    else:
        kb_service = KBService()
        asyncio.run(kb_service.ingest(path, kb_id=real_id))
        console.print("[green]■■■■[/green]")
```

4. 部署与运行

4.1 安装

```
# ■■■■■■■■■■■■
git clone <repo>
cd knowforge
pip install -e .

# ■■■■■■■■■■■■
pip install knowforge

# ■■■■■
pip install knowforge[local]      # ■■■■■■■
pip install knowforge[pdf]         # ■■ PDF ■■
pip install knowforge[kg]         # ■■■■■
pip install knowforge[mcp]        # MCP ■■
pip install knowforge[obs]        # ■■■■■
pip install knowforge[all]        # ■■
```

4.2 初始化

```
# ■■■■■■
knowforge config init

# ■■ LLM API Key
knowforge config set-secret openai_api_key

# ■■■■■
knowforge admin doctor
```

4.3 典型工作流

```
# 1. ■■■■■■
knowforge kb create my-kb

# 2. ■■■■■■■■■
knowforge kb ingest my-kb ./docs/

# ■■■■■■■■■■■■ Worker■
knowforge admin serve &
knowforge kb ingest my-kb ./big.pdf --async
knowforge task status <job_id>

# 3. ■■■■■
knowforge kb stats my-kb
knowforge kg stats my-kb

# 4. ■■■■■■
knowforge agent create my-agent --kb my-kb

# 5. ■■
knowforge chat -a my-agent

# 6. ■■■■■
knowforge chat -a my-agent -m "■■■■■■■■"
```

4.4 异步 Worker 模式

```
# ■■ 1■■■ Worker
knowforge admin serve --concurrency 4

# ■■ 2■■■■■
knowforge kb ingest my-kb ./large_dataset/ --async
knowforge kg extract my-kb --async

# ■■ 2■■■■■
knowforge task list
knowforge task status <job_id>
knowforge task logs <job_id>
```

4.5 离线模式

```
# ■■ Ollama
knowforge config set llm.default_provider ollama
knowforge config set retrieval.embedding_model BAAI/bge-small-zh-v1.5

# ■■ Ollama■■■■■■
ollama serve &
ollama pull qwen2.5:7b

# ■■
knowforge chat -a my-agent
```

5. 测试

5.1 测试结构

```
tests/
■■■ test_core.py      # ■■■■■■
■■■ test_kb.py        # ■■■■■■
■■■ test_agent.py     # Agent ■■■■
■■■ test_cli.py       # CLI ■■■■
```

5.2 运行测试

```
# ■■■■
pytest

# ■■■■
pytest --cov=knowforge --cov-report=html

# ■■■■
pytest tests/test_kb.py -v
```

5.3 测试示例

```
def test_recursive_chunker():
    chunker = RecursiveChunker(chunk_size=100, chunk_overlap=10)
    text = "■■■■■■■" * 50
    chunks = chunker.chunk(text)
    assert len(chunks) > 1
    assert all(c.text for c in chunks)
    for i, c in enumerate(chunks):
        if i > 0:
            assert c.prev_id == chunks[i-1].id
        if i < len(chunks) - 1:
            assert c.next_id == chunks[i+1].id

def test_tool_registry():
    reg = ToolRegistry()
    load_builtin_tools(reg)
    names = reg.names()
    assert "present_artifacts" in names
    assert "ask_user" in names
```

6. 扩展开发

6.1 开发自定义 Skill

```
# my_pkg/skills/weather.py
from knowforge.agent.skills.base import Skill, SkillManifest

class WeatherSkill(Skill):
    name = "weather"
    version = "0.1.0"

    def manifest(self):
        return SkillManifest(
            name=self.name,
            version=self.version,
            description="■■■■",
            entry_point="my_pkg.skills.weather:WeatherSkill",
        )

    async def execute(self, ctx):
        city = ctx.get("city", "")
        # ■■■■ API
        return {"artifact_type": "text", "content": f"{city} ■■■■"}
```

在 pyproject.toml 声明:

```
[project.entry-points."knowforge.skills"]
weather = "my_pkg.skills.weather:WeatherSkill"
```

安装后即可使用：knowforge skill list 会显示 weather 技能。

6.2 开发自定义 LLM Provider

```
# my_pkg/providers.py
from knowforge.llm.base import LLMProvider, LLMMessage, LLMResponse

class MyProvider(LLMProvider):
    name = "my_provider"

    async def chat(self, messages, **kwargs):
        # ■■ chat ■■
        return LLMResponse(content="...", model=self.model)

    async def stream(self, messages, **kwargs):
        # ■■■■■■
        yield "token"
```

声明 entry_point:

```
[project.entry-points."knowforge.llm_providers"]
my_provider = "my_pkg.providers:MyProvider"
```

6.3 开发自定义 Parser

```
# my_pkg/parsers.py
from knowforge.knowledge.parser.base import Parser, ParserResult

class MyParser(Parser):
    name = "my_format"
    extensions = (".myf",)

    def parse(self, path, **kwargs):
        # ■■■■■■
        return ParserResult(text="...", metadata={"source": str(path)})
```

声明 entry_point:

```
[project.entry-points."knowforge.parsers"]
my_format = "my_pkg.parsers:MyParser"
```

7. 性能优化

7.1 嵌入向量缓存

通过 CacheLayer 缓存嵌入向量，避免重复计算:

```
embeddings = cache.get_or_compute(
    key=f"embed:{hash(text)}",
    fn=lambda: embedder.embed_sync([text])[0],
    expire=86400  # 24 ■■
)
```

7.2 BM25 索引惰性构建

BM25 索引在首次检索时构建，之后缓存到内存:

```
async def _get_bm25_index(self, kb_id):
    if kb_id in self._bm25_indexes:
        return self._bm25_indexes[kb_id]
    # ■ SQLite ■■■■ chunk ■■
    chunks = load_chunks_from_db(kb_id)
```

```
    index = BM25Okapi([self._tokenize(c.text) for c in chunks])
    self._bm25_indexes[kb_id] = index
    return index
```

7.3 LLM 调用并发

三路检索并行执行：

```
async def retrieve(self, query, kb_id):
    # ■■■■■■■■■
    vector_task = asyncio.create_task(self._vector_search(query, kb_id))
    bm25_task = asyncio.create_task(self._bm25_search(query, kb_id))
    kg_task = asyncio.create_task(self._kg_search(query, kb_id))
    vector_results, bm25_results, kg_results = await asyncio.gather(
        vector_task, bm25_task, kg_task
    )
```

7.4 SQLite WAL 模式

启用 WAL 模式提升并发读写：

```
@event.listens_for(engine, "connect")
def _set_sqlite_pragma(dbapi_conn, _record):
    cursor = dbapi_conn.cursor()
    cursor.execute("PRAGMA journal_mode=WAL")
    cursor.execute("PRAGMA synchronous=NORMAL")
    cursor.execute("PRAGMA busy_timeout=5000")
    cursor.close()
```

8. 调试与运维

8.1 日志

```
# ■■■■
knowforge --verbose chat -a my-agent

# JSON ■■■■■■■■■
knowforge --json chat -a my-agent 2>&1 | tee knowforge.log
```

8.2 环境诊断

```
knowforge admin doctor
```

输出表格显示 Python 版本、所有依赖安装状态、存储目录可访问性。

8.3 交互式 Shell

```
knowforge admin shell
```

启动预加载 KnowForge 的 Python REPL，便于调试：

```
>>> from knowforge.knowledge.kb_service import KBService
>>> svc = KBService()
>>> results = await svc.search("query", kb_id="xxx")
```

8.4 数据库迁移

```
# ■■■■■■■■
knowforge admin migrate

# ■■■■■■■■■■■■■■■
knowforge admin migrate --drop
```

9. 与 Yuxi 的实现差异

维度	Yuxi 实现	KnowForge 实现
入口	FastAPI + Vue 前端	Typer CLI + Rich TUI
配置	.env + 数据库	YAML + 环境变量 + dataclass
租户	数据库 tenant_id	Profile 物理隔离 + tenant_id 逻辑隔离
检索	向量 + 图谱	向量 + BM25 + 图谱 + RRF + 重排
图谱	Neo4j 独立服务	KùzuDB 嵌入式
向量	Milvus 独立服务	ChromaDB 嵌入式
任务	ARQ + Redis	asyncio + SQLite 队列
插件	硬编码	pip entry_points
可观测	基础日志	structlog + OpenTelemetry
离线	不支持	Ollama 集成
代码量	~5 万行（含前端）	~7 千行（纯 Python）

10. 后续演进

10.1 短期（v0.2）

- Web UI 插件（可选，不污染核心）
- Qdrant 向量库适配器
- Neo4j 图库适配器
- 多模态文档解析（图片 OCR、表格结构化）

10.2 中期（v0.3）

- Agent 评测框架（自动评测检索质量与回答质量）
- 知识图谱可视化 TUI（基于 Rich 的图渲染）
- 联邦学习（多 KnowForge 节点知识共享）

10.3 长期（v0.5）

- MCP Server 模式（KnowForge 作为 MCP 服务器被其他 Agent 调用）
- 知识图谱自动演化（增量更新、冲突检测）
- 多智能体协作（多个 Agent 协同完成复杂任务）

附录：图表索引

本附录汇总 KnowForge 设计文档中引用的全部 11 张图表，涵盖架构图、设计图、流程图与 UML，便于快速查阅。

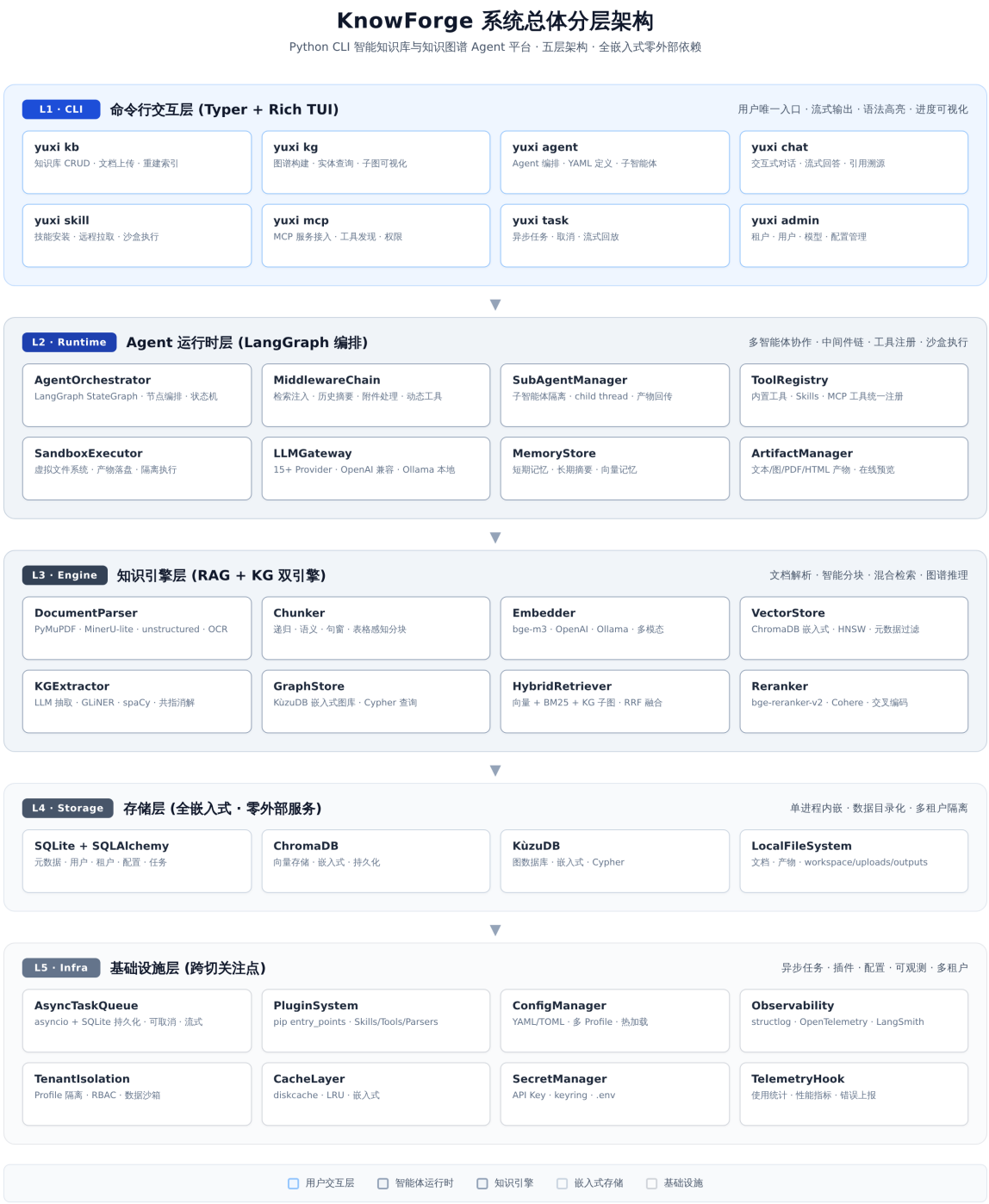


图 1：KnowForge 系统总体架构图（分层）

图 2：KnowForge 模块依赖图

图 3：RAG + 知识图谱检索流水线流程图

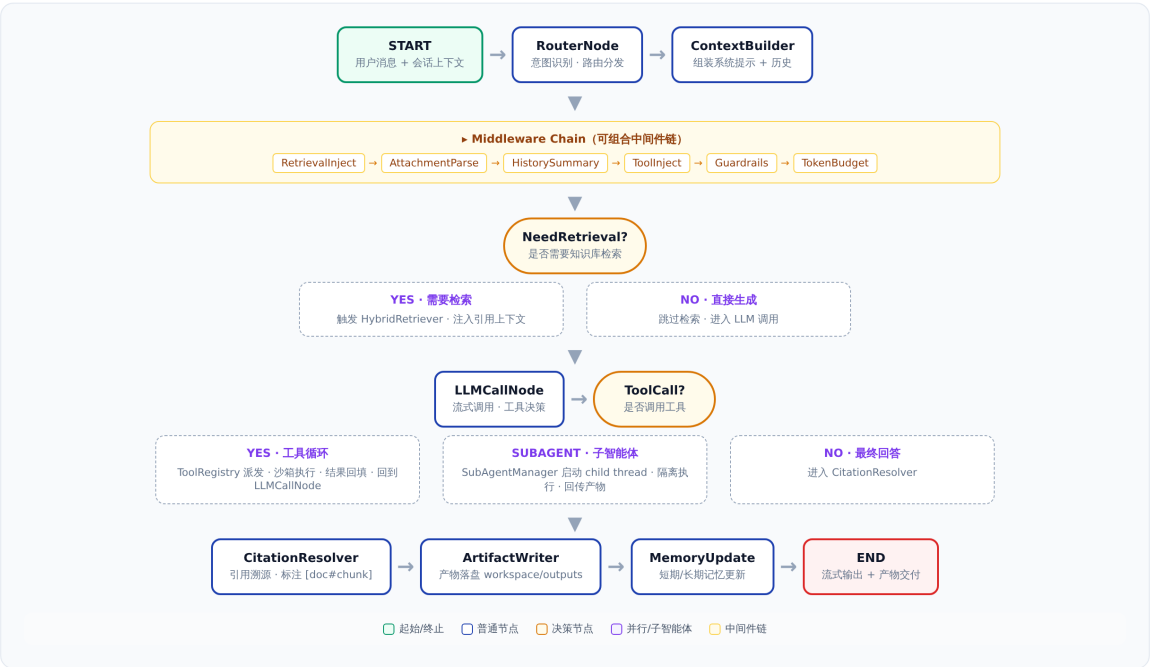


图 4：Agent 编排状态机流程图

KnowForge CLI 命令树

Typer + Rich 实现 · 单一入口 knowforge · 9 大子命令组 · 50+ 子命令

$ knowforge <command-group> <subcommand> [OPTIONS]

kb · 知识库管理

- **kb create** 创建知识库
- **kb list** 列出所有知识库
- **kb ingest** 摄入文档 (文件/目录/URL)
- **kb search** 混合检索测试
- **kb stats** 统计信息
- **kb rebuild** 重建索引
- **kb export** 导出知识库

kg · 知识图谱

- **kg build** 从知识库构建图谱
- **kg query** Cypher/自然语言查询
- **kg explore** 交互式子图探索
- **kg visualize** 导出 GraphML/JSON
- **kg merge** 实体消歧合并

agent · 智能体

- **agent create** YAML 定义智能体
- **agent list** 列出所有智能体
- **agent show** 查看智能体配置
- **agent edit** 编辑 YAML 配置
- **agent test** 单测智能体

chat · 对话

- **chat** 交互式 REPL 对话
- **chat -a <agent>** 指定智能体
- **chat -k <kb>** 挂载知识库
- **chat --attach <file>** 附件对话
- **chat --resume <id>** 恢复会话

skill · 技能

- **skill list** 列出已安装技能
- **skill install** 从本地/远程安装
- **skill create** 脚手架生成技能模板
- **skill enable/disable** 启停

mcp · MCP 集成

- **mcp list** 列出 MCP 服务
- **mcp add** 添加 MCP 服务
- **mcp call** 直接调用工具

task · 异步任务

- **task list** 查看任务队列
- **task status <id>** 任务状态
- **task cancel <id>** 取消任务
- **task logs <id>** 查看日志

config · 配置 / admin · 管理

- **config init** 初始化配置
- **config set/get** 读写配置项
- **config profile** 多 Profile 管理
- **admin migrate** 数据库迁移
- **admin serve** 启动后台 Worker
- **admin doctor** 环境诊断

▸ 全局选项：--profile · --verbose · --json · --no-color · --config <path>

图 5: KnowForge CLI 命令树

数据流图（DFD）

外部实体 · 处理过程 · 数据存储 · 数据流向

图 6：数据流图（DFD）

部署拓扑图

单机嵌入式部署 · 零外部服务依赖 · 可选云端 LLM

图 7：部署拓扑图

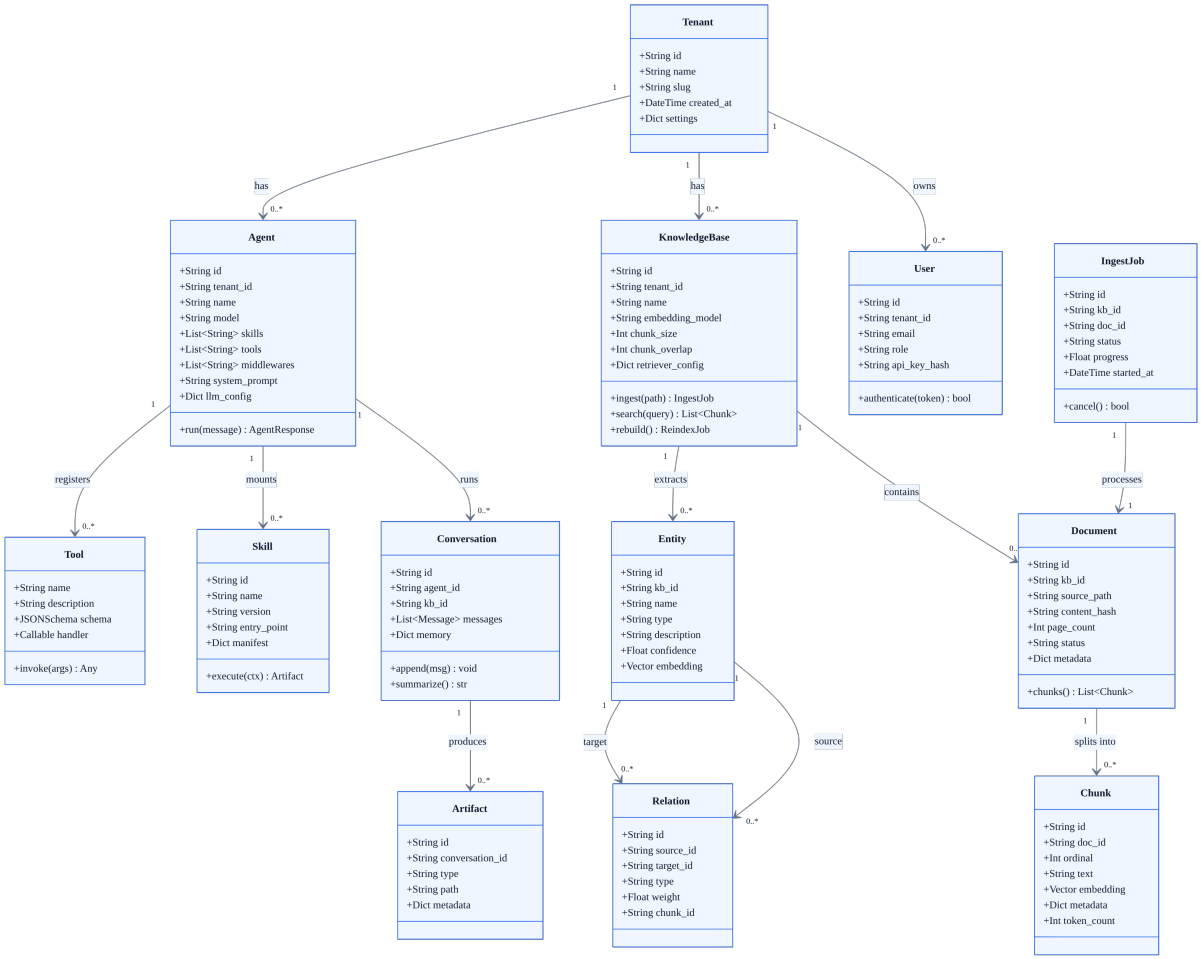


图 8：UML 类图（核心领域模型）

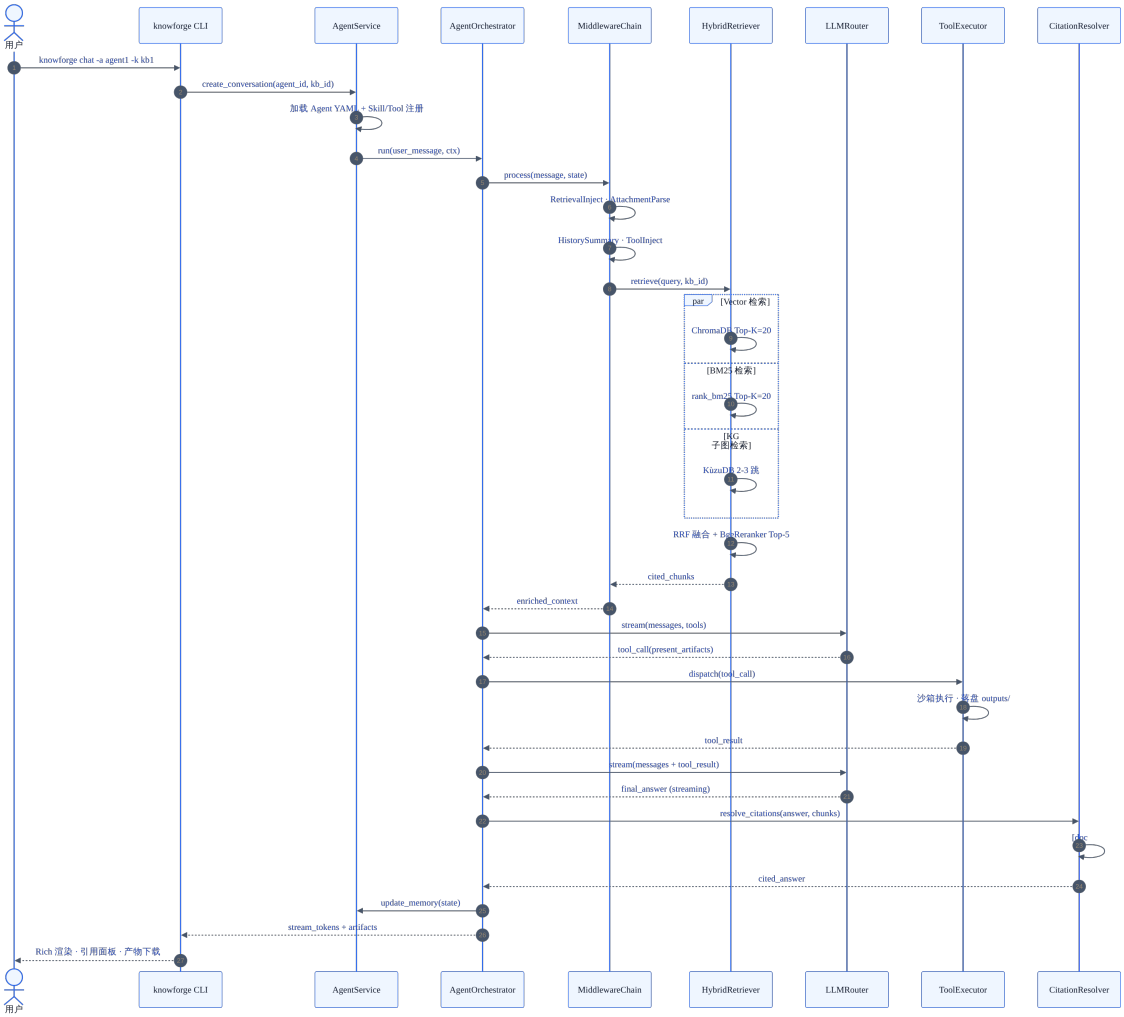


图 9：UML 序列图（Agent 执行流程）

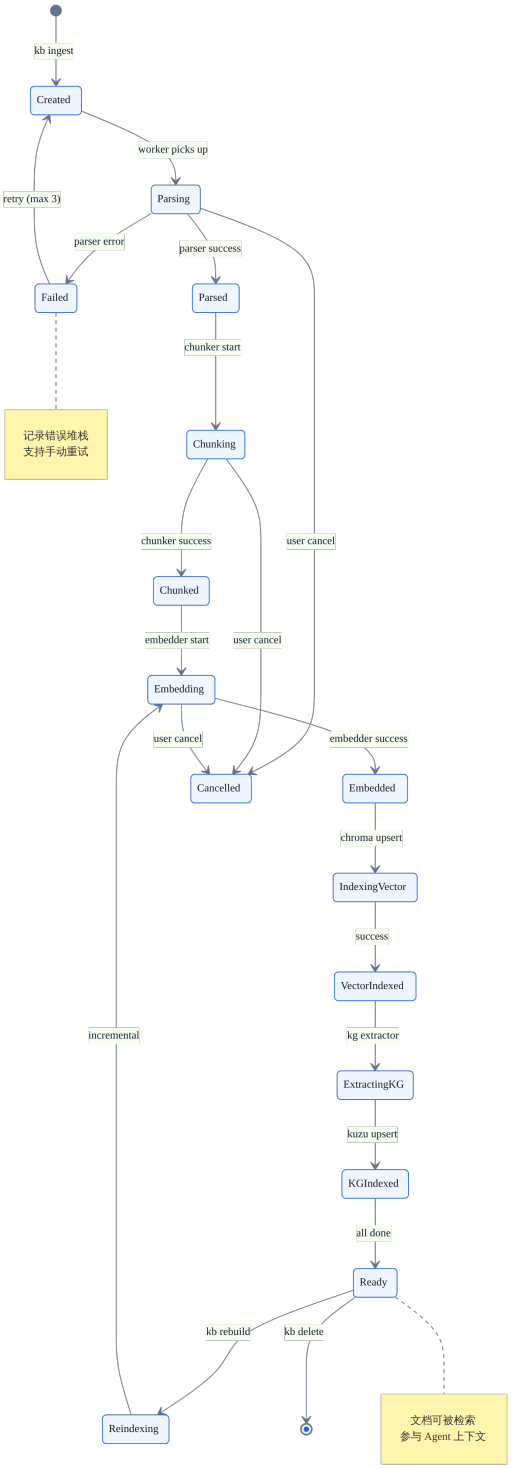


图 10：UML 状态图（文档摄入状态转换）

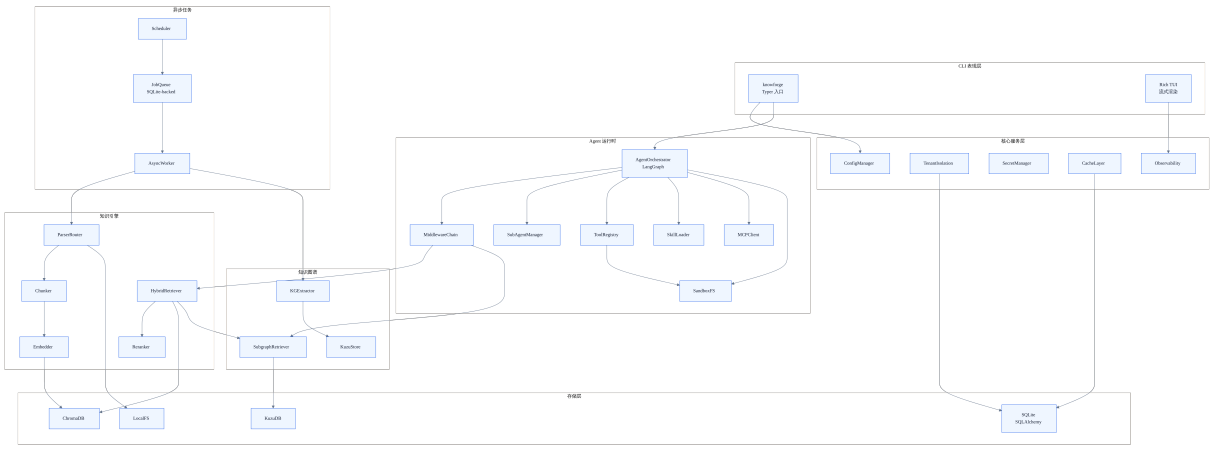

异步任务
Scheduler
JobQueue
SQLite-backed
AsyncWorker
CLI 表现层
knowforge
Typer 入口
Rich TUI
流式渲染
Agent 编排层
AgentOrchestrator
LangGraph
MiddlewareChain
SubAgentManager
ToolRegistry
SkillLoader
MCPClient
SandboxFS
核心服务层
ConfigManager
TenantIsolation
SecretManager
CacheLayer
Observability
知识管理
ParserRouter
Chunker
Embedder
HybridRetriever
Reranker
KGExtractor
SubgraphRetriever
KuzuStore
存储层
ChromaDB
LocalFS
KuzuDB
SQLite
SQLAlchemy

图 11：UML 组件图