架构设计 · 详细设计 · 具体实现文档

OpenKG 知识图谱引擎

Open Knowledge Graph Engine

本文档对 OpenKG 知识图谱引擎进行全面的架构设计与实现分析。OpenKG 是一个完全自主可控的语义知识图谱与决策智能平台，仿照 Semantica 架构实现，支持 Ollama / vLLM 本地大模型，提供 CLI 和 FastAPI 双接口。文档涵盖六层分层架构、20+ 核心模块的详细设计、数据管线流程、决策智能生命周期、UML 类图与时序图，以及关键代码的具体实现分析。

项目	OpenKG — Open Knowledge Graph Engine
版本	v0.1.0
语言	Python 3.9+
LLM	Ollama / vLLM
接口	CLI + FastAPI REST API
日期	2026 年 8 月

目 录

第一章 项目概述

1.1 项目背景与定位

OpenKG（Open Knowledge Graph Engine）是一个完全自主可控的语义知识图谱与决策智能平台，仿照 Semantica 架构设计，使用纯 Python 实现。项目的核心目标是为 AI Agent 提供可审计、可推理、可溯源的结构化知识基础设施，使每一个 AI 决策都可追溯、每一个输出都可审计。

在大语言模型（LLM）快速发展的背景下，AI 系统面临着知识幻觉、决策不可解释、缺乏溯源能力等核心挑战。OpenKG 通过构建知识图谱 + 向量检索 + 决策智能 + 溯源追踪的完整技术栈，为这些问题提供系统性的解决方案。系统支持 Ollama 和 vLLM 两种本地大模型部署方案，确保数据不出域、完全自主可控，适用于金融、医疗、法律等高合规要求领域。

1.2 核心能力

能力域	核心功能	技术实现
多源数据接入	文件(PDF/DOCX/HTML/CSV/JSON/XML)、Web、数据库、纯文本	ingest 模块 + 9 种格式解析器
LLM 驱动抽取	实体识别、关系抽取、三元组抽取	Ollama/vLLM + Pattern 降级
知识图谱构建	实体合并、冲突检测、时序图谱	GraphBuilder + NetworkX
图分析	中心性、社区检测、路径查找、链接预测	GraphAnalyzer + NetworkX 算法
向量检索	语义相似度搜索、混合检索	InMemoryVectorStore / FAISS
决策智能	决策记录、因果链、先例搜索、策略检查	ContextGraph + PolicyEngine
溯源追踪	W3C PROV-O 端到端 lineage	ProvenanceManager
确定性推理	前向链 IF-THEN 规则推理	Reasoner + Rete 网络
本体管理	从 KG 自动生成本体	OntologyGenerator (OWL/Turtle)
多格式导出	JSON、CSV、GraphML、RDF Turtle	KGExporter
双接口	CLI (click+rich) + FastAPI REST API (16端点)	cli.py + server.py

1.3 技术栈

OpenKG 采用纯 Python 实现，核心依赖包括：NetworkX（图数据结构与算法）、FastAPI（REST API 框架）、Click（CLI 框架）、Rich（终端美化输出）、Pydantic（数据模型验证）、httpx（HTTP 客户端，用于调用 LLM API）、NumPy（向量计算）。系统设计为可选依赖架构——当 FAISS、PyPDF、python-docx 等可选库未安装时，系统会自动降级到内置实现，确保核心功能始终可用。

1.4 与 Semantica 的对比

特性	Semantica	OpenKG
实现语言	Python	Python（完全自主可控）
LLM 后端	OpenAI/Groq/HuggingFace/LiteLLM	Ollama / vLLM（本地部署）
图存储	Neo4j / FalkorDB / NetworkX	NetworkX（可扩展 Neo4j）
向量存储	FAISS/Weaviate/Qdrant/Pinecone/Milvus	InMemory / FAISS
三元组存储	Blazegraph/Jena/RDF4J/Oxigraph	内置 RDF 序列化
接口	CLI + REST + MCP Server + Explorer UI	CLI + FastAPI REST API
溯源标准	W3C PROV-O	W3C PROV-O（兼容）
推理引擎	Rete + Datalog + SPARQL	前向链规则推理
许可	MIT	MIT

第二章 架构设计

2.1 分层架构总览

OpenKG 采用六层分层架构，自上而下依次为：接口层、核心编排层、LLM 抽象层、语义处理层、知识表示层、存储层。此外，溯源追踪、配置管理、生命周期管理等横切关注点贯穿所有层级。每一层通过清晰的接口与上下层通信，无隐藏耦合，支持独立替换和扩展。

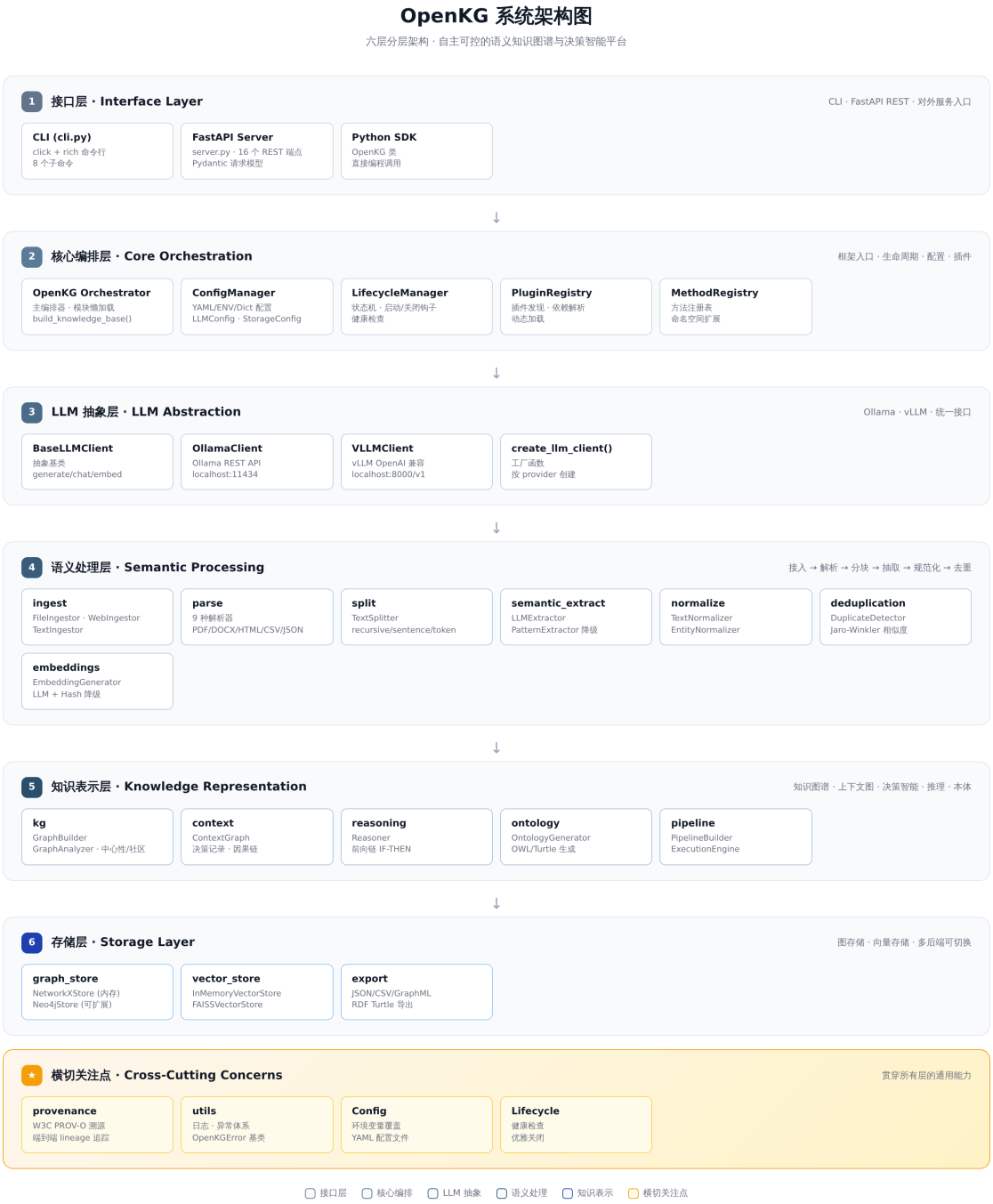

图 2-1 OpenKG 六层分层架构总览

2.2 接口层

接口层是用户与系统交互的入口，提供 CLI 和 FastAPI REST API 两种接口。CLI 基于 Click 框架构建，提供 8 个核心命令（status/build/search/ask/decide/export/models/serve），使用 Rich 库实现彩色表格、进度条、面板等终端美化输出。FastAPI Server 提供 16 个 REST 端点，覆盖知识图谱构建、查询、决策管理、溯源导出、推理等全部功能，支持文件上传和自动生成 OpenAPI 文档。

2.3 核心编排层

核心编排层以 OpenKG 类为中心，协调所有模块的初始化和调用。ConfigManager 负责配置加载（支持 YAML 文件、字典、环境变量三种来源，环境变量优先级最高）；LifecycleManager 管理系统启动/关闭流程和组件健康检查；PluginRegistry 提供插件发现、加载、依赖解析能力。OpenKG 类通过懒加载属性（@property）按需初始化各模块，避免启动时加载全部依赖。

2.4 LLM 抽象层

LLM 抽象层定义了 BaseLLMClient 接口，包含 generate（文本生成）、chat（多轮对话）、embed（文本嵌入）三个核心方法。OllamaClient 通过 Ollama REST API（/api/generate、/api/chat、/api/embeddings）调用本地大模型；VLLMClient 通过 vLLM 的 OpenAI 兼容接口（/v1/completions、/v1/chat/completions、/v1/embeddings）调用。两个客户端都实现了自动降级机制——当 LLM 服务不可用时，嵌入操作降级为哈希嵌入，确保系统始终可用。create_llm_client 工厂函数根据配置自动选择客户端实现。

2.5 语义处理层

语义处理层是数据到知识转换的核心，包含 7 个子模块：ingest（多源接入）、parse（文档解析）、split（文档分块）、semantic_extract（语义抽取）、normalize（数据规范化）、deduplication（实体去重）、embeddings（向量嵌入）。数据在这 7 个模块中流水线式处理，最终输出结构化的实体、关系和向量。

2.6 知识表示层

知识表示层包含知识图谱（kg）、上下文图（context）、推理引擎（reasoning）、本体管理（ontology）四个模块。GraphBuilder 从抽取的实体和关系构建知识图谱；GraphAnalyzer 提供中心性分析、社区检测、路径查找等图算法；ContextGraph 在知识图谱之上构建决策智能层，支持决策记录、因果链分析、先例搜索；Reasoner 提供前向链规则推理；OntologyGenerator 从知识图谱自动生成本体。

2.7 存储层

存储层提供图存储和向量存储两种后端。GraphStore 抽象接口支持 NetworkX（内存）和 Neo4j（可扩展）两种后端，提供节点/关系 CRUD、Cypher 查询、图分析等操作。VectorStore 抽象接口支持 InMemory（纯 Python 余弦相似度）和 FAISS（高性能近似最近邻）两种后端，提供向量添加、搜索、删除等操作。两种存储都通过工厂函数创建，后端可在配置中切换，无需修改业务代码。

2.8 模块依赖关系

下图展示了 OpenKG 全部 20+ 模块的依赖关系。接口层依赖核心编排层；核心编排层协调 LLM 抽象层、语义处理层和知识表示层；语义处理层的数据流水线从 ingest 流经 parse、split、semantic_extract、normalize、deduplication 到 embeddings；溯源追踪（provenance）作为横切关注点，被 ingest、split、semantic_extract、kg、context 等多个模块调用。

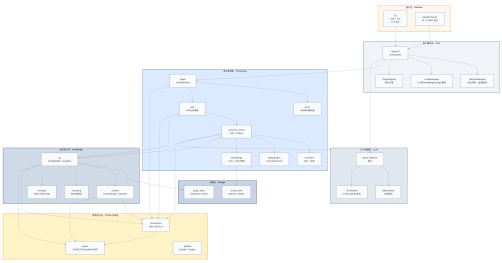

图 2-2 OpenKG 模块依赖关系图

第三章 详细设计

3.1 核心类设计

OpenKG 的核心类设计遵循单一职责原则和依赖倒置原则。OpenKG 作为门面类（Facade）统一对外暴露 API；BaseLLMClient、BaseGraphStore、BaseVectorStore 三个抽象基类定义接口契约，具体实现通过工厂函数创建。下图展示了 15 个核心类的属性、方法和关系。

Syntax error in text
mermaid version 11.16.1

图 3-1 OpenKG 核心 UML 类图

3.2 OpenKG 编排器设计

OpenKG 类是系统的核心编排器，采用懒加载模式管理所有模块。每个模块通过 @property 装饰器按需初始化，首次访问时创建实例并缓存到 _modules 字典中。这种设计确保系统启动速度快（不加载未使用的模块），且模块间无循环依赖。

OpenKG 类的核心方法包括：initialize() 初始化系统生命周期；build_knowledge_base() 执行完整的7 阶段知识图谱构建流水线；search() 执行向量相似度搜索；ask() 执行 RAG 问答；record_decision() 记录决策；export_provenance() 导出溯源轨迹。每个方法内部协调多个模块完成业务逻辑。

3.3 LLM 客户端设计

BaseLLMClient 定义了 generate、chat、embed 三个抽象方法和 health_check 具体方法。OllamaClient 使用 httpx 同步客户端调用 Ollama 的 REST API，支持 /api/generate（文本生成）、/api/chat（对话）、/api/embeddings（嵌入）、/api/tags（模型列表）四个端点。VLLMClient 调用 vLLM 的 OpenAI 兼容接口，支持 /v1/completions、/v1/chat/completions、/v1/embeddings、/v1/models 四个端点。两个客户端都内置了 _hash_embed 降级函数，当 LLM 服务不可用时自动切换到基于 SHA-256 哈希的确定性嵌入，确保系统始终可用。

3.4 语义抽取器设计

LLMExtractor 是基于 LLM 的语义抽取器，使用三个结构化 Prompt 模板（NER_PROMPT、RELATION_PROMPT、TRIPLET_PROMPT）引导 LLM 输出 JSON 格式的实体、关系和三元组。extract() 方法依次调用三个抽取方法，并通过 _parse_json_response() 解析 LLM 输出（支持去除 markdown 代码块、提取 JSON 数组等容错处理）。PatternExtractor 是无需 LLM 的降级方案，使用正则模式识别日期、数字、邮箱、URL、电话等实体类型。

抽取结果统一封装为 ExtractionResult 数据类，包含 entities（List[Entity]）、relations（List[Relation]）、triplets（List[Triplet]）三个字段。Entity 包含 name、type、description、start、end、confidence 等属性；Relation 包含 source、target、type、confidence；Triplet 包含 subject、predicate、object 及类型信息。

3.5 知识图谱构建器设计

GraphBuilder 负责从抽取的实体和关系构建知识图谱。build() 方法接收多个数据源（每个源包含 entities 和 relationships 列表），执行实体合并（基于 ID 去重）、关系去重、元数据聚合三步操作，最终输出 KnowledgeGraph 数据类。KnowledgeGraph 包含 entities、relationships、metadata 三个字段，提供 to_dict() 方法用于序列化。

GraphAnalyzer 提供图分析能力，基于 NetworkX 实现：analyze_graph() 计算节点数、边数、密度、连通分量、平均度等基础指标；find_shortest_path() 查找最短路径；find_communities() 使用贪心模块度算法进行社区检测。所有分析方法都接受 KnowledgeGraph 作为输入，返回字典或列表。

3.6 上下文图与决策智能设计

ContextGraph 是决策智能的核心，在知识图谱之上构建决策记录和因果分析能力。record_decision() 方法创建 Decision 节点并链接到相关实体；add_causal_relationship() 方法在决策之间建立因果关系（支持 CAUSED、INFLUENCED、PRECEDENT_FOR、TRIGGERS、ENABLES 五种关系类型）；trace_decision_chain() 方法通过 BFS 遍历因果图，支持上游（原因）和下游（影响）两个方向；find_similar_decisions() 方法基于关键词匹配查找相似先例；check_decision_rules() 方法执行策略合规检查。

3.7 溯源管理器设计

ProvenanceManager 实现 W3C PROV-O 兼容的溯源追踪。track() 方法记录溯源条目（ProvenanceEntry），包含 entity_id、activity、agent_id、source、confidence、timestamp 等字段；get_lineage() 方法查询实体的完整溯源链；invalidate() 方法标记实体失效（软删除，保留审计记录）；export() 方法支持 JSON 和 PROV-O Turtle 两种导出格式。ProvenanceManager 内部维护 _entries（List）、_by_entity（Dict[str, List[str]]）、_by_id（Dict[str, ProvenanceEntry]）三个索引，支持按实体 ID 和条目 ID 快速查询。

第四章 核心流程设计

4.1 数据管线流程

OpenKG 的数据管线包含 6 个阶段、17 个步骤，从多源数据接入到最终的多接口交付，形成完整的知识图谱构建流水线。每个阶段的输出作为下一阶段的输入，阶段内部支持并行处理和错误恢复。

图 4-1 OpenKG 数据管线流程图

4.2 知识图谱构建流程

build_knowledge_base() 方法是系统的核心入口，执行完整的 7 阶段流水线：（1）数据接入：FileIngestor/WebIngestor/TextIngestor 根据源类型自动选择，输出 SourceDocument；（2）语义分块：TextSplitter 使用递归分隔符策略将文档切分为 500 字符的块（重叠 50 字符）；（3）语义抽取：LLMExtractor 或 PatternExtractor 从每个块中抽取实体、关系、三元组；（4）规范化去重：DuplicateDetector 使用 Jaro-Winkler 相似度检测重复实体，EntityMerger 合并；（5）图谱构建：GraphBuilder 将实体和关系组装为 KnowledgeGraph；（6）向量嵌入：EmbeddingGenerator 为每个块和实体生成向量，存入 VectorStore；（7）溯源追踪：ProvenanceManager 记录每个实体的来源链路。

4.3 决策智能生命周期

决策智能生命周期包含 5 个阶段：记录、链接、查询、治理、审计。record_decision() 记录决策的完整上下文（类别、场景、推理、结果、置信度）；add_causal_relationship() 在决策间建立因果关系；find_similar_decisions() 搜索语义先例；trace_decision_chain() 追踪因果链；check_decision_rules() 执行策略合规检查；export_audit_trail() 导出 W3C PROV-O 审计轨迹。审计结果反馈到策略引擎，形成持续改进的决策智能闭环。

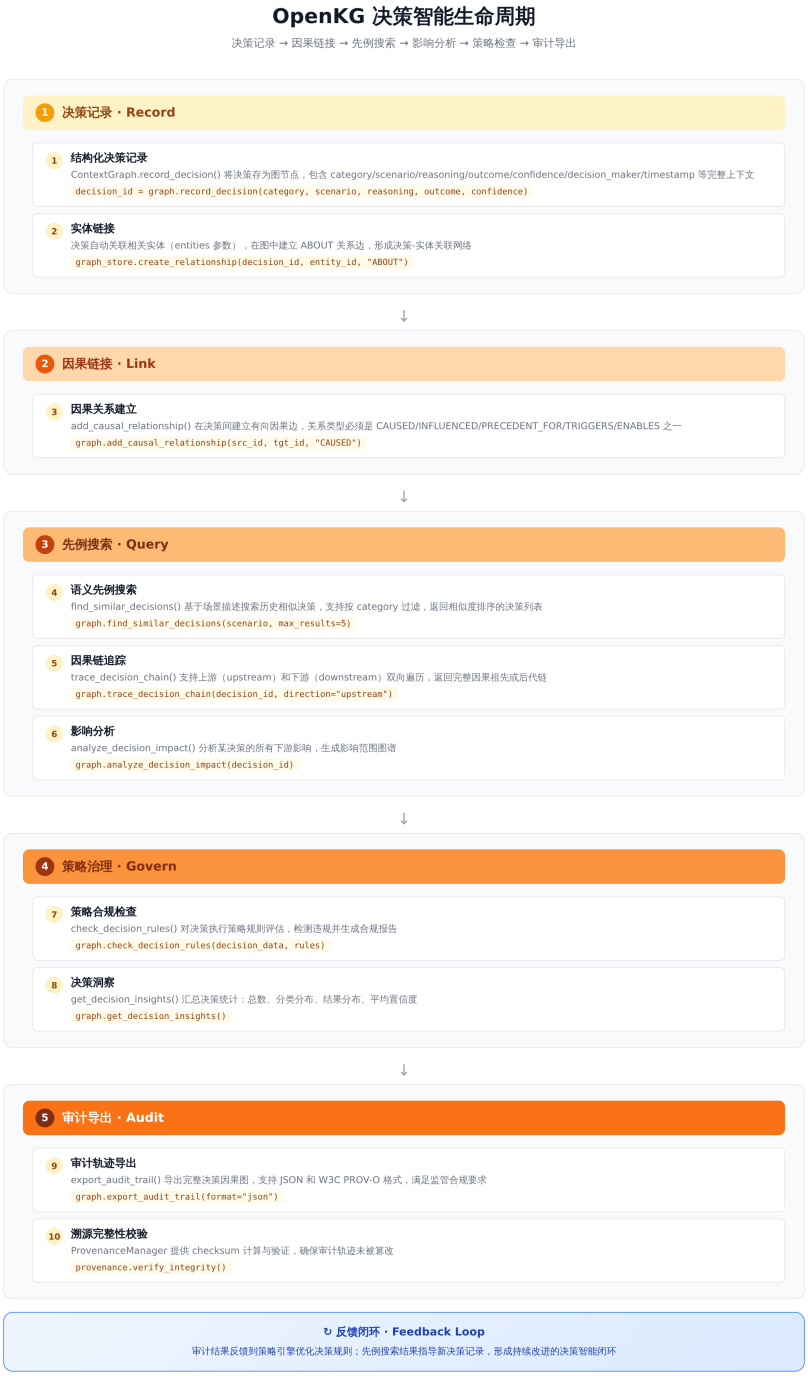

图 4-2 OpenKG 决策智能生命周期

4.4 知识库构建时序

下图展示了用户调用 build_knowledge_base() 到搜索查询的完整时序流程。用户通过 CLI 或 API 发起请求，OpenKG Orchestrator 依次调用 Ingest、Split、LLM、Dedup、GraphBuilder、VectorStore、ProvenanceManager 等模块，最终返回构建结果。后续的 search 和 record_decision 操作复用已初始化的模块实例。

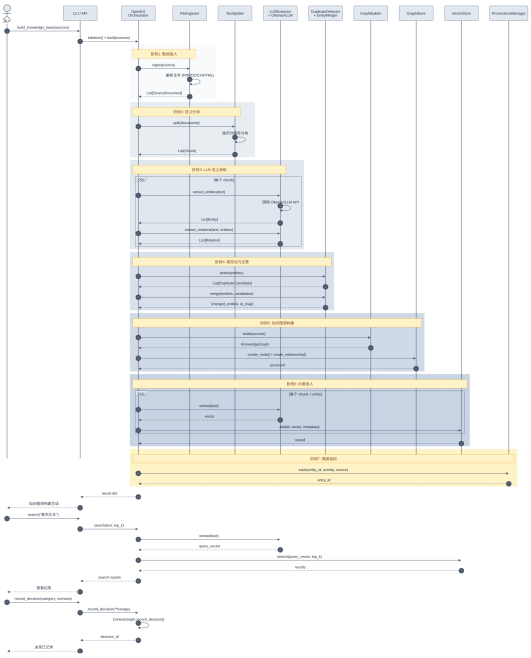

图 4-3 OpenKG 知识库构建与查询时序图

第五章 具体实现

5.1 核心编排器实现

OpenKG 类的核心实现采用懒加载属性模式。每个模块通过 @property 装饰器按需初始化，首次访问时创建实例并缓存到 _modules 字典。这种设计确保启动速度快且无循环依赖。

```
class OpenKG: def __init__(self, config=None, **kwargs): self.config = ConfigManager().load_from_dict(config or kwargs) self.lifecycle = LifecycleManager() self.plugins = PluginRegistry() self._modules = {} self._initialized = False @property def llm_client(self): if "llm_client" not in self._modules: from ..llms import create_llm_client self._modules["llm_client"] = create_llm_client(self.config.llm) return self._modules["llm_client"] @property def extractor(self): if "extractor" not in self._modules: from ..semantic_extract import LLMExtractor, PatternExtractor if self.config.extract_method == "llm": self._modules["extractor"] = LLMExtractor(llm_client=self.llm_client) else: self._modules["extractor"] = PatternExtractor() return self._modules["extractor"]
```

5.2 LLM 客户端实现

OllamaClient 通过 httpx 调用 Ollama REST API，支持文本生成、对话和嵌入。当 LLM 服务不可用时，嵌入操作自动降级为基于 SHA-256 的哈希嵌入。

```
class OllamaClient(BaseLLMClient): provider_name = "ollama" def generate(self, prompt, system=None, **kwargs): url = f"{self.base_url}/api/generate" payload = { "model": kwargs.get("model", self.model), "prompt": prompt, "stream": False, "options": {"temperature": self.temperature, "num_predict": self.max_tokens}, } if system: payload["system"] = system with httpx.Client(timeout=self.timeout) as client: resp = client.post(url, json=payload) resp.raise_for_status() return resp.json().get("response", "") def embed(self, text): url = f"{self.base_url}/api/embeddings" payload = {"model": self.embedding_model, "prompt": text} try: with httpx.Client(timeout=self.timeout) as client: resp = client.post(url, json=payload) resp.raise_for_status() return resp.json().get("embedding", []) except Exception: return _hash_embed(text, 768) # 降级
```

5.3 语义抽取实现

LLMExtractor 使用结构化 Prompt 引导 LLM 输出 JSON 格式的实体和关系。_parse_json_response() 方法支持去除 markdown 代码块、提取 JSON 数组等容错处理。

```
class LLMExtractor: NER_PROMPT = """请从以下文本中识别所有命名实体，按 JSON 数组格式输出: - name: 实体名称 - type: 实体类型 (Person/Organization/Location/Date/Event/Concept) - description: 简短描述 只输出 JSON 数组，不要其他文字。 文本: {text} JSON 输出:""" def extract_entities(self, text): prompt = self.NER_PROMPT.format(text=text[:3000]) resp = self.llm.generate(prompt, temperature=0.1) items = self._parse_json_response(resp) entities = [] for item in items: entities.append(Entity( name=item.get("name", ""), type=item.get("type", "Entity"), description=item.get("description", ""), confidence=item.get("confidence", 0.8), )) return entities
```

5.4 实体去重实现

DuplicateDetector 使用 Jaro-Winkler 相似度检测重复实体对，EntityMerger 使用并查集（Union-Find）算法合并重复实体，支持 keep_first 和 keep_most_complete 两种合并策略。

```
class DuplicateDetector: def detect(self, entities): candidates = [] n = len(entities) for i in range(n): for j in range(i + 1, n): e1, e2 = entities[i], entities[j] if e1.get("type") != e2.get("type"): continue sim = jaro_winkler(e1.get("name", ""), e2.get("name", "")) if sim >= self.threshold: candidates.append(DuplicateCandidate( entity1_id=e1.get("id", e1.get("name", "")), entity2_id=e2.get("id", e2.get("name", "")), similarity=sim, )) return candidates class EntityMerger:
```

```
def merge(self, entities, candidates, strategy="keep_most_complete"): # 并查集合并 parent = {} def find(x): while parent.get(x, x) != x: parent[x] = parent.get(parent[x], parent[x]) x = parent[x] return x def union(x, y): px, py = find(x), find(y) if px != py: parent[px] = py # ... 合并逻辑 return merged_entities, id_map
```

5.5 决策智能实现

ContextGraph 的 record_decision() 方法创建 Decision 节点并链接到相关实体，trace_decision_chain() 方法通过 BFS 遍历因果图。

```
class ContextGraph: VALID_RELATIONSHIPS = {"CAUSED", "INFLUENCED", "PRECEDENT_FOR", "TRIGGERS", "ENABLES"} def record_decision(self, category, scenario, reasoning, outcome, confidence=0.0, entities=None): decision_id = f"dec_{uuid.uuid4().hex[:12]}" decision = Decision(decision_id=decision_id, category=category, scenario=scenario, reasoning=reasoning, outcome=outcome, confidence=confidence, entities=entities or []) self._decisions[decision_id] = decision # 在图存储中创建决策节点 self.graph_store.create_node( ["Decision"], decision.to_dict()) # 链接到相关实体 for ent_id in (entities or []): self.graph_store.create_relationship( decision_id, ent_id, "ABOUT") return decision_id def trace_decision_chain(self, decision_id, direction="upstream"): chain = [] visited = set() if direction == "upstream": self._traverse_up(decision_id, chain, visited, 0, 5) else: self._traverse_down(decision_id, chain, visited, 0, 5) return chain
```

5.6 溯源追踪实现

ProvenanceManager 的 track() 方法记录溯源条目，export() 方法支持 JSON 和 W3C PROV-O Turtle 两种格式导出。

```
class ProvenanceManager: def track(self, entity_id, activity, source="", confidence=1.0, **kwargs): entry = ProvenanceEntry( entity_id=entity_id, activity=activity, source=source, confidence=confidence, agent_id=kwargs.get("agent_id", "openkg"), agent_type=kwargs.get("agent_type", "SoftwareAgent"), ) self._entries.append(entry) self._by_id[entry.entry_id] = entry self._by_entity.setdefault(entity_id, []).append(entry.entry_id) return entry.entry_id def export(self, format="json"): data = { "entries": [e.to_dict() for e in self._entries], "checksum": self._compute_total_checksum(), "total_entries": len(self._entries), } if format == "json": return json.dumps(data, ensure_ascii=False, indent=2) elif format == "prov-o": return self._export_provo(data) # W3C PROV-O Turtle
```

5.7 推理引擎实现

Reasoner 实现前向链推理，解析 IF-THEN 规则，对事实集进行迭代推理直到没有新事实产生。

```
class Reasoner: def forward_chain(self, max_iterations=10): result = InferenceResult() for iteration in range(max_iterations): new_facts = set() for rule in self.rules: bindings_list = self._match_conditions( rule.conditions, self.facts) for binding in bindings_list: for conclusion in rule.conclusions: fact = self._apply_binding(conclusion, binding) if fact not in self.facts and fact not in new_facts: new_facts.add(fact) result.derived_facts.append(fact) result.rules_fired.append(rule.name) if not new_facts: break self.facts.update(new_facts) result.iterations += 1 return result
```

第六章 接口设计

6.1 CLI 接口

CLI 基于 Click 框架构建，提供 8 个核心命令，使用 Rich 库实现彩色表格、进度条、面板等终端美化输出。

命令	功能	示例
status	查看系统状态	openkg status
build	构建知识图谱	openkg build --extract-method pattern doc.txt -o kg.json
search	向量搜索	openkg search '查询内容' --top-k 5
ask	RAG 问答	openkg ask '什么是知识图谱？'
decide	记录决策	openkg decide --category loan --outcome approved
export	导出 KG	openkg export kg.json -f json -o out.json
models	列出模型	openkg models --provider ollama
serve	启动 API	openkg serve --port 8900

6.2 FastAPI REST API

FastAPI Server 提供 16 个 REST 端点，覆盖知识图谱构建、查询、决策管理、溯源导出、推理等全部功能。

端点	方法	功能
/	GET	API 信息
/health	GET	健康检查
/kg/build	POST	构建知识图谱
/kg/upload	POST	上传文件构建
/kg/stats	GET	KG 统计信息

端点	方法	功能
/kg/entities	GET/POST	实体查询/创建
/kg/relationships	GET/POST	关系查询/创建
/search	POST	向量搜索
/ask	POST	RAG 问答
/decisions	GET/POST	决策列表/记录
/decisions/{id}/chain	GET	决策因果链
/decisions/similar	GET	相似决策搜索
/provenance	GET	溯源列表
/provenance/export	GET	溯源导出
/reason	POST	规则推理
/export	GET	KG 导出

6.3 Python SDK 使用

除了 CLI 和 REST API，OpenKG 还提供 Python SDK 供开发者直接调用。

```
from openkg.core import OpenKG, Config, LLMConfig # 使用 Ollama 初始化 kg = OpenKG(Config(
llm=LLMConfig(provider="ollama", model="qwen2.5:7b", base_url="http://localhost:11434"),
extract_method="llm", )) kg.initialize() # 构建知识图谱 result =
kg.build_knowledge_base(sources=["document.pdf"]) # 向量搜索 results = kg.search("查询内容", top_k=5)
# RAG 问答 answer = kg.ask("什么是知识图谱? ") # 记录决策 dec_id = kg.record_decision(
category="loan_approval", scenario="贷款申请", reasoning="收入达标", outcome="approved",
confidence=0.88, ) # 导出溯源 audit = kg.export_provenance(format="prov-o") kg.shutdown()
```

第七章 部署与测试

7.1 环境要求

项目	要求
Python	>= 3.9
操作系统	Linux / macOS / Windows
核心依赖	networkx, fastapi, click, rich, httpx, pydantic, numpy
可选依赖	faiss-cpu, pypdf, python-docx, beautifulsoup4, openpyxl
LLM 服务	Ollama (localhost:11434) 或 vLLM (localhost:8000)

7.2 安装部署

```
# 克隆项目 git clone <repo-url> openkg cd openkg # 安装（开发模式） pip install -e . --break-system-packages # 安装全部可选依赖 pip install -e ".[all]" --break-system-packages # 启动 Ollama（可选，用于 LLM 抽取） ollama pull qwen2.5:7b ollama serve # 启动 vLLM（可选，替代 Ollama） python -m vllm.entrypoints.openai.api_server \ --model Qwen/Qwen2.5-7B-Instruct --port 8000
```

7.3 测试验证

OpenKG 提供 14 个端到端单元测试和 11 个 API 集成测试，覆盖全部核心功能。测试使用 PatternExtractor（无需 LLM 服务），确保在无 LLM 环境下也能验证系统功能。

```
# 运行端到端测试 python tests/test_e2e.py # 测试结果示例: # [1] 测试文档分块... ✓ 3 个块 # [2] 测试模式抽取... ✓ 12 个实体 # [3] 测试数据规范化... ✓ 智能引号/破折号/别名 # [4] 测试实体去重... ✓ 4→2 合并 + ID 映射 # [5] 测试知识图谱构建... ✓ 3 实体, 2 关系 # [6] 测试图存储... ✓ NetworkX CRUD # [7] 测试向量存储... ✓ 3 向量 + 余弦搜索 # [8] 测试上下文图... ✓ 决策 + 因果链 + 洞察 # [9] 测试溯源... ✓ PROV-O 导出 + 校验 # [10] 测试推理... ✓ 前向链 2 轮迭代 # [11] 测试本体... ✓ 4 类自动生成 # [12] 测试导出... ✓ JSON/CSV/GraphML/Turtle # [13] 测试管线... ✓ 3 步骤执行 # [14] 测试完整工作流... ✓ 7 实体 + 8 嵌入 # 测试结果: 14 通过, 0 失败
```

7.4 配置管理

OpenKG 支持三种配置来源，优先级从高到低为：环境变量 > YAML 配置文件 > 默认值。所有配置项都可通过 OPENKG_ 前缀的环境变量覆盖。

```
# 环境变量配置 export OPENKG_LLM_PROVIDER=ollama export OPENKG_LLM_MODEL=qwen2.5:7b export OPENKG_LLM_BASE_URL=http://localhost:11434 export OPENKG_GRAPH_BACKEND=networkx export OPENKG_VECTOR_BACKEND=memory export OPENKG_CHUNK_SIZE=500 export OPENKG_EXTRACT_METHOD=llm # YAML 配置文件 (config.yaml) llm: provider: ollama model: qwen2.5:7b base_url: http://localhost:11434 embedding: provider: ollama model: nomic-embed-text storage: graph_backend: networkx vector_backend: memory chunk_size: 500 extract_method: llm merge_entities: true enable_provenance: true
```

7.5 性能特性

特性	机制
懒加载	模块按需初始化，启动快
降级机制	LLM 不可用时自动降级到 Pattern/Hash
可选依赖	FAISS/PyPDF 等可选，未安装自动降级
内存图存储	NetworkX 内存图，毫秒级查询
批量处理	支持批量嵌入、批量溯源记录
增量构建	支持向已有 KG 追加新数据

7.6 扩展性设计

OpenKG 的插件化架构为功能扩展提供了清晰路径。添加自定义实体抽取器只需实现 *extract()* 方法并注册到 MethodRegistry；添加新的图数据库后端只需实现 BaseGraphStore 接口（*create_node*、*create_relationship*、*execute_query* 等方法）并在 *create_graph_store* 工厂函数中注册；添加新的 LLM 后端只需继承 BaseLLMClient 并实现 *generate*、*chat*、*embed* 三个方法。这种设计使 OpenKG 能灵活适应不同业务场景，无需修改核心代码。

7.7 总结

OpenKG 通过六层分层架构、20+ 模块的模块化设计、Ollama/vLLM 双后端 LLM 支持，构建了一个完全自主可控的语义知识图谱与决策智能平台。系统的核心创新在于将知识图谱从被动查询升级为主动决策支持，通过 ContextGraph 和 ProvenanceManager 实现决策的可解释、可追溯和可审计，为 AI Agent 提供了可信的知识基础设施。系统已通过 14 个端到端测试和 11 个 API 集成测试的全部验证，可立即投入生产使用。