

大语言模型应用开发技术指南

——从 RAG 到 Agent 的实战手册

作者	Modular RAG 项目组
版本	v1.0 — 2026 年 2 月
分类	内部技术文档 / QA 测试用

目录

第一章 大语言模型基础概念

第二章 检索增强生成 (RAG) 架构设计

第三章 文档分块策略详解

第四章 向量检索与混合搜索

第五章 重排序机制与评估方法

第六章 Agent 与工具调用

第七章 生产部署与性能优化

第一章 大语言模型基础概念

大语言模型 (Large Language Model, LLM) 是基于 Transformer 架构，通过海量文本数据进行预训练而得到的深度神经网络。代表性模型包括 GPT-4、Claude、DeepSeek 等。LLM 的核心思想是将自然语言建模为 Token 序列的概率分布，通过自回归方式逐个 Token 生成文本。

与传统 NLP 方法相比，LLM 具有以下关键优势：

- 强大的上下文理解能力，能够处理长文档和复杂指令
- Few-shot 和 Zero-shot 学习能力，无需大量标注数据
- 跨任务泛化能力，同一模型可完成翻译、摘要、QA 等多种任务
- 支持多轮对话和思维链 (Chain-of-Thought) 推理

1.1 Transformer 架构要点

Transformer 架构最初由 Vaswani 等人在 2017 年提出 (Attention Is All You Need)，其核心组件包括多头自注意力机制 (Multi-Head Self-Attention)、前馈神经网络 (FFN)、层归一化 (Layer Normalization) 和残差连接 (Residual Connection)。在 LLM 中，通常只使用 Decoder 部分，通过因果注意力掩码实现自回归生成。

注意力计算的核心公式为 $\text{Attention}(Q, K, V) = \text{softmax}(QK^T / \sqrt{d_k})V$ ，其中 Q、K、V 分别代表查询、键和值矩阵， d_k 为键向量维度。多头注意力将输入投影到 h 个不同的子空间，并行计算注意力后拼接输出。

1.2 Tokenization 与词表

分词 (Tokenization) 是将原始文本转化为模型可处理的整数序列的关键步骤。常见的子词分词算法包括 BPE (Byte Pair Encoding)、WordPiece 和 SentencePiece。中文场景下，BPE 通常将每个汉字作为一个或多个 Token，因此中文文本的 Token 数量往往多于等长的英文文本。这对 RAG 系统的 Chunk 大小设计有直接影响。

第二章 检索增强生成 (RAG) 架构设计

RAG (Retrieval-Augmented Generation) 将信息检索与文本生成相结合，通过在生成时引入外部知识库的相关文档片段，显著减少 LLM 的幻觉问题，同时使模型能够回答超出训练数据范围的实时性问题。

2.1 Naive RAG vs Advanced RAG

Naive RAG 采用最简单的「检索-拼接-生成」流程：将用户查询编码为向量，在向量数据库中检索 Top-K 相似文档，拼接到 Prompt 中交由 LLM 生成回答。其缺点包括：检索质量受限于单一向量相似度、缺乏重排序机制、无法处理多跳推理等。

Advanced RAG 在 Naive RAG 基础上引入多项优化：查询改写 (Query Rewriting)、混合检索 (Hybrid Search combining Dense + Sparse)、重排序 (Reranking via Cross-Encoder 或 LLM)、以及文档分块优化 (Smart Chunking + Metadata Enrichment)。本项目实现了完整的 Advanced RAG 链路。

2.2 Modular RAG 设计理念

Modular RAG 将 RAG 流程分解为可独立替换的模块：Loader → Splitter → Transformer → Embedder → Vector Store → Retriever → Reranker → Generator。每个模块定义抽象接口，通过工厂模式和配置文件驱动实例化，实现「乐高积木式」的灵活组合。

这种设计的核心优势在于：可以在不修改代码的情况下，通过 settings.yaml 一键切换 LLM Provider (Azure / DeepSeek / Ollama)、Embedding 模型、Reranker 策略等，极大地方便了 A/B 测试和技术选型。

第三章 文档分块策略详解

文档分块 (Chunking) 是系统中决定检索质量的关键环节。分块过大会导致检索噪声增加、LLM 上下文利用率下降；分块过小则会丢失语义上下文，影响回答连贯性。

RAG

3.1 常见分块方法

方法	原理	适用场景
固定长度切分	按字符数或 Token 数等分	结构简单的纯文本
递归字符切分	按分隔符优先级递归切分	通用文档 (推荐)
语义分块	基于 Embedding 相似度判断边界	长文档、主题多变
Markdown 切分	按标题层级切分	Markdown / 技术文档

3.2 Chunk 增强：Refiner 与 Metadata Enrichment

原始切分后的 Chunk 可能存在截断不自然、缺乏上下文等问题。Chunk Refiner 利用 LLM 对 Chunk 文本进行改写润色，使其更加自包含、语义完整。Metadata Enricher 则为每个 Chunk 生成 title、summary 和 tags 元信息，在检索阶段可用于辅助过滤和排序。

图片处理方面，系统会提取 PDF 中的嵌入图片，调用 Vision LLM (如 GPT-4o) 生成中文图片描述 (Image Caption)，并将描述文本注入对应 Chunk 的 metadata 中，从而实现跨模态检索——用户可以通过文字查询来找到相关的图表和图片内容。

第四章 向量检索与混合搜索

向量检索 (Dense Retrieval) 通过将文本编码为高维稠密向量，利用余弦相似度或内积计算语义相似性，是现代信息检索的核心技术。常用的 Embedding 模型包括 OpenAI text-embedding-ada-002 (1536 维)、BGE 系列 (768/1024 维) 等。

4.1 BM25 稀疏检索

BM25 (Best Matching 25) 是经典的概率检索模型，基于词频 (TF) 和逆文档频率 (IDF) 计算文档与查询的相关性。其优势在于对精确关键词匹配的处理非常有效，特别是专有名词、错别字等场景下表现优于纯语义检索。本项目使用 jieba 分词 + rank_bm25 库实现中文 BM25 检索。

4.2 RRF 融合算法

Reciprocal Rank Fusion (RRF) 是一种简单高效的排名融合算法，公式为： $\text{score}(d) = \sum 1/(k + \text{rank}_i(d))$ ，其中 k 为平滑常数（默认 60）， $\text{rank}_i(d)$ 为文档 d 在第 i 个排名列表中的位置。RRF 的优势在于不依赖原始分数的量纲，可以直接融合不同检索方法的排名结果。

4.3 ChromaDB 向量存储

ChromaDB 是一个轻量级的开源向量数据库，支持 Embedding 存储、元数据过滤和近似最近邻 (ANN) 搜索。本项目使用 ChromaDB 持久化存储模式，数据保存在本地 SQLite 文件中，适合个人项目和中小规模应用场景。

第五章 重排序机制与评估方法

重排序 (Reranking) 是 RAG 的精排阶段，在粗排召回的候选集基础上进行更精细的相关性评估。常见方案包括 Cross-Encoder Reranker 和 LLM Reranker 两种。

5.1 Cross-Encoder vs Bi-Encoder

Bi-Encoder (双塔模型) 将查询和文档分别编码为独立向量，通过余弦相似度计算匹配度，速度快但精度有限。Cross-Encoder

将查询和文档拼接后同时输入模型，通过注意力机制捕捉细粒度交互，精度更高但速度较慢。RAG 中通常使用 Bi-Encoder 做粗排，Cross-Encoder 做精排。

5.2 评估指标

指标名称	类型	说明
Hit Rate	检索指标	Top-K 结果中是否包含正确文档
MRR	检索指标	正确文档首次出现位置的倒数
Faithfulness	生成指标 (Ragas)	生成答案是否忠实于检索上下文
Answer Relevancy	生成指标 (Ragas)	生成答案与用户问题的相关度
Context Precision	生成指标 (Ragas)	检索上下文中相关信息的占比

第六章 Agent 与工具调用

Agent 是指能够自主规划任务、调用外部工具来完成复杂目标的 AI 系统。与简单的 RAG 查询不同，Agent 可以根据用户指令自动分解任务、选择合适的工具（如搜索引擎、计算器、代码执行器），迭代执行直到获得满意结果。

6.1 MCP 协议

Model Context Protocol (MCP) 是 Anthropic 提出的标准化 AI 模型与外部工具交互的协议。它定义了工具发现 (tools/list)、工具调用 (tools/call) 等 JSON-RPC 2.0 接口标准。本项目实现了一个 MCP Server，暴露了 query_knowledge_hub、list_collections 和 get_document_summary 三个工具，可被 VS Code Copilot 或 Claude Desktop 直接调用。

6.2 ReAct 推理模式

ReAct (Reasoning + Acting) 是一种将推理与行动交替进行的 Agent 模式。LLM 先进行推理思考 (Thought)，然后选择一个行动 (Action)，观察行动结果 (Observation)，再继续推理，循环往复直到得出最终答案。这种模式使 Agent 的决策过程更加透明、可调试。

第七章 生产部署与性能优化

将 RAG 系统从开发环境部署到生产环境需要考虑多个维度：服务可用性、响应延迟、数据安全和成本控制。以下是关键优化策略。

7.1 缓存策略

对于重复或相似的查询，可以引入语义缓存（Semantic Cache）：将查询向量与缓存库中的历史查询进行相似度匹配，若超过阈值则直接返回缓存结果，避免重复的 Embedding 计算和 LLM 调用，显著降低延迟和成本。

7.2 批处理与异步

Embedding 计算和 LLM 调用是 RAG 链路中最耗时的环节。通过批量 Embedding（将多个 Chunk 合并为一次 API 调用）和异步并发（asyncio / ThreadPool），可以大幅提升数据摄取和查询的吞吐量。本项目的 BatchProcessor 组件实现了可配置的批量大小和并发度。

7.3 可观测性

生产系统必须具备完善的可观测性能力。本项目通过结构化日志（JSON Lines 格式）、全链路 Trace（记录每个阶段的输入/输出/耗时）和 Streamlit Dashboard 三位一体实现可视化监控。每次摄取和查询操作都会自动生成 Trace 记录，便于问题排查和性能分析。