零、LLM

LLM（Large Language Model）即大型语言言模型

LLM相关面试题

1、什么是大模型微调？与预训练的核心区别？

大模型微调（Fine-tuning of Large Models） 是指在预训练模型的基础上，使用特定任务的数据对模型进行再训练，以适应特定应用场景的需求。与**预训练（Pre-training）** 相比，微调的核心区别在于目标、数据来源和训练方式：

1）**目标不同**：

- **预训练**：旨在让模型学习通用的语言或知识表示，通常在大规模通用数据集上进行训练。
- **微调**：旨在让模型适应特定任务，如情感分析、问答系统等，通常在特定任务的数据集上进行训练。

2）**数据来源不同**：

- **预训练**：使用大规模的通用数据集，如维基百科、书籍语料等。
- **微调**：使用与特定任务相关的标注数据集。

3）**训练方式不同**：

- **预训练**：通常采用无监督或自监督学习方式。
- **微调**：通常采用监督学习方式，利用标注数据进行训练。

特性	预训练	微调
数据量	巨量通用数据	小量特定任务数据
目标	学习通用语言表示	适应特定任务或风格
参数更新	所有参数	全量或部分参数（LoRA/Adapter）
成本	高	较低
输出能力	通用	专精

举例：假设你要做一个法律问答系统：

1. **预训练阶段**：使用大规模通用语料（维基百科、新闻、书籍）训练出一个 GPT-5 模型
2. **微调阶段**：用法律文档、判例、法规条文对 GPT-5 做 LoRA 微调
3. **效果**：模型从通用语言能力 → 具备专业法律知识 → 可以回答法律问题

2、常见的微调任务有哪些？

3、讲一下Transformer架构

LLM就是基于Transformer架构训练出来的大模型

一句话总结　：

> Transformer 是一种基于多头自注意力和前馈网络的序列模型，通过注意力机制捕捉序列中任意位置的依赖关系，并通过残差连接和位置编码实现高效并行训练。用注意力机制算法公司让每个词能看序列中所有词，结合位置编码和残差结构，高效并行处理序列，实现长距离依赖建模。
>
> **自注意力机制**：自注意力机制让序列中每个词根据与其他词的相关性重新表示自己，实现长距离依赖建模和并行计算。

1️⃣ Transformer 的基本概念

- **用途**：Transformer 是一种深度学习模型，用于处理序列数据（文本、时间序列等），尤其擅长 NLP（自然语言处理）。
- **突破点**：
  - 不像 RNN（循环神经网络）需要按顺序处理序列，Transformer 可以并行处理整个序列，大幅提高训练效率。
  - 使用 **注意力机制（Attention）**，可以捕捉序列中任意位置的依赖关系，而不是局限于前后顺序。

2️⃣ Transformer 的整体结构

Transformer 模型分为两部分：

```
[Encoder] -> [Decoder]
```

- **Encoder**（编码器）：
  - 接收输入序列（如一句话）
  - 输出隐藏向量（每个词的表示）
  - 可以堆叠多个 Encoder 层
- **Decoder**（解码器）：
  - 接收 Encoder 输出 + 已生成的序列
  - 输出下一个词预测
  - 可以堆叠多个 Decoder 层

> NLP 任务中：
>
> - 机器翻译：Encoder 读源语言，Decoder 输出目标语言
> - 文本生成/问答：也可以只用 Decoder（GPT 系列就是这种结构）

3️⃣ Encoder 内部结构（每一层）

每一层 Encoder 有两大模块：

1. **多头自注意力（Multi-Head Self-Attention）**
   - 作用：每个词根据序列中所有词的关系重新计算自己的表示。
   - 核心公式：

     ```
     Attention(Q, K, V) = softmax(QK^T / sqrt(d_k)) V
     ```

     - Q：查询（Query）
     - K：键（Key）

        - V：值（Value）
    - 多头：多个独立的注意力头并行，捕捉不同的关系。
2. **前馈全连接网络（Feed-Forward Network）**
    - 对每个词独立处理
    - 两层线性 + 激活函数（通常用 ReLU 或 GELU）
3. **残差连接 + LayerNorm**
    - 残差连接：保证梯度传播顺畅
    - LayerNorm：归一化，稳定训练

示意图（Encoder 层）：

```
输入序列
    |
[多头自注意力] --残差+Norm
    |
[前馈全连接] --残差+Norm
    |
输出隐藏向量
```

4️⃣ Decoder 内部结构

Decoder 与 Encoder 类似，但有三部分：

1. **Masked Self-Attention**
    - 只看当前位置及之前的位置，防止泄露未来信息
2. **Encoder-Decoder Attention**
    - 用 Decoder 的查询向量 Q 去注意 Encoder 的输出 K, V
    - 作用：将输入序列信息融入生成过程
3. **前馈全连接 + 残差 + LayerNorm**

示意图（Decoder 层）：

```
输入序列
    |
[Masked 自注意力] --残差+Norm
    |
[Encoder-Decoder 注意力] --残差+Norm
    |
[前馈全连接] --残差+Norm
    |
输出预测
```

5️⃣ 位置编码（Positional Encoding）

- Transformer 没有循环或卷积，无法天然捕捉序列顺序。
- 解决方案：加上 **位置编码**，告诉模型每个词的位置。
- 常用公式（正弦/余弦）：

```
PE(pos,2i) = sin(pos / 10000^(2i/d_model))
PE(pos,2i+1) = cos(pos / 10000^(2i/d_model))
```

6️⃣ Transformer 的优点

1. 并行训练快（不像 RNN 必须顺序处理）
2. 可以建模长距离依赖（Long-range）
3. 灵活扩展（堆叠更多层）
4. Multi-head attention 捕捉多维语义信息

一、Agent

Agent相关面试题

1、什么是大模型 Agent？它与传统的 AI 系统有什么不同？

大模型 Agent *是基于大型语言模型并结合*模块化规划、**记忆**和**工具调用**的自主决策系统，它能够根据最终目标把复杂任务拆分成子任务，调用 API、检索数据库或使用插件，再通过内部循环不断优化执行流程，基本不需要人在每一步都监督。

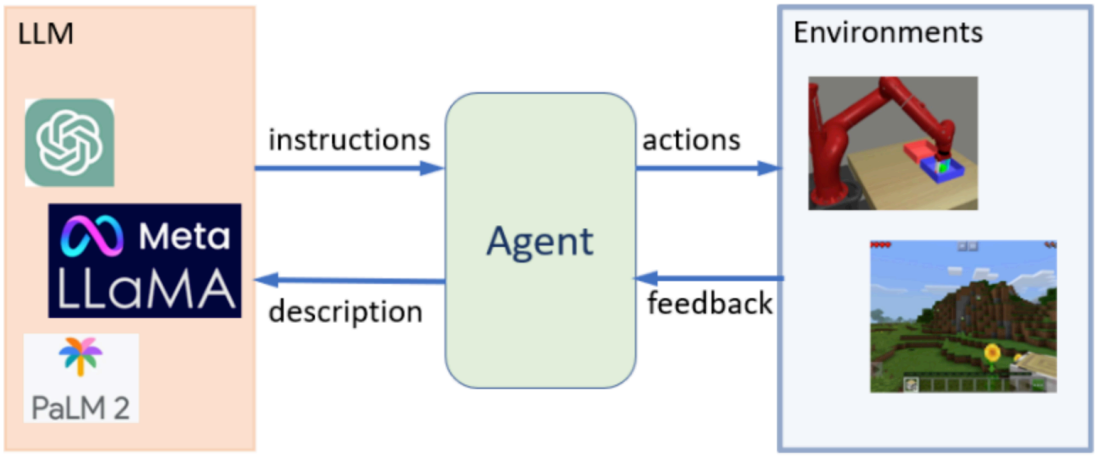
主要区别如下：

1）**目标导向 vs. 被动响应**：传统大模型通常根据用户输入生成文本，缺乏主动性；而 Agent 以明确目标为驱动，能够主动规划并执行任务。

2）**记忆与状态管理**：Agent 具备短期和长期记忆能力，能够维护状态信息并根据历史经验调整行为；而传统大模型通常依赖于上下文窗口进行信息处理。

3）**多任务协同能力**：Agent 能够处理复杂的多步骤任务，协调多个子任务的执行；而传统大模型通常处理单一任务，缺乏任务协同能力。

4）**推理与环境适应能力**：和传统 AI 系统主要依赖预先设定的规则引擎或静态模型不同，大模型 Agent 具备多步推理能力和动态环境适应能力，它能在执行过程中不断评估结果、调整策略，真正实现端到端的目标导向执行。



扩展知识

1）大模型 Agent 的核心组件一般包含**规划器**（Planner）、**执行器**（Executor）、**记忆**（Memory）和**工具调用**（Tool Use）模块，规划器负责制定任务分解方案，执行器负责落地操作，记忆模块保存交互上下文，工具模块负责与外部系统接口。
2）与传统 AI 系统只能进行被动响应相比，Agentic AI 更强调主动性和自治性，它不只是给出建议，还能主动执行、反馈和学习，类似从“图书检索”升级为“智能管家”。
3）传统 AI 通常需要手动编写大量规则或对模型进行单任务微调，Agent 则倾向于通过预训练大模型加上少量指令微调来控制行为，这种方式兼顾了灵活性和可控性。
4）在实际应用中，大模型 Agent 已经在自动化运维、客户服务、财务分析等领域展现出端到端自动化能力，但也因多步执行易出现错误累积和计算资源消耗大而受到挑战。
5）打个比方，传统 AI 系统就像一个图书馆的检索系统，帮我们找到答案；而大模型 Agent 更像一个经验丰富的管家，不仅能帮我们查资料，还能主动安排日程、预订机票、跟进各项事务。

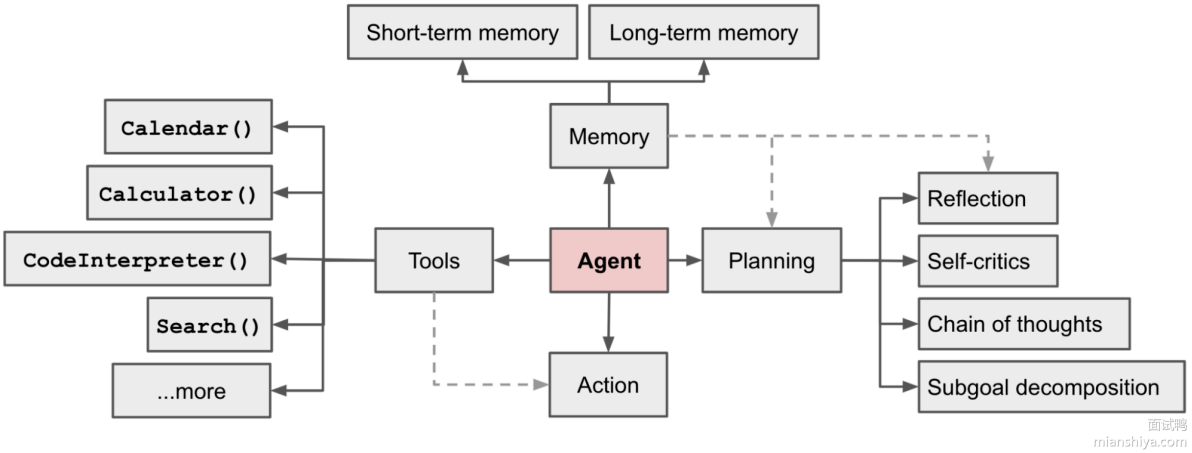
2、LLM Agent 的基本架构有哪些组成部分?

1）**Agent核心（LLM本身）**：作为“大脑”，负责理解输入、生成计划和下发指令，串联其他模块协同工作。

2）**规划模块（***Planning***）**：将复杂目标拆解成有序的子任务，制定多步执行方案，并在必要时动态重规划。

3）**记忆模块（***Memory***）**：短期保存当前对话或任务状态，长期存储跨会话的知识和经验，保障多轮交互的连贯性和上下文保持。

4）**工具使用（***Tool use***）**：接入搜索、计算、数据库、代码执行、第三方插件等多种外部工具，为执行模块提供能力扩展。



扩展知识

在一个基于大模型（LLM）的自主代理系统中，大模型充当代理的“大脑”，并通过几个关键组件来增强其能力：

1）**规划（Planning)**

- **子目标和任务分解**：代理将大任务拆解成更小、更易管理的子目标，这样可以更高效地处理复杂任务。就像我们在做一个大项目时，把每一小步分开执行，目标更清晰，完成更容易。
- **反思和优化**：代理可以在执行任务后，回顾自己的表现，进行自我批评和反思，找出哪里做得不好，然后调整策略。这种自我改进的能力能让代理在接下来的任务中表现得更好，结果也更精准。

2）**记忆（Memory)**

- **短期记忆**：短期记忆就像是模型在当前任务或对话中临时记住的信息，类似于我们在聊天时记住了对方刚说的话，但任务一结束这些信息就会消失。
- **长期记忆**：长期记忆让代理能够长期保存和调用信息，类似于保存了一些重要的笔记或知识，可以随时拿出来用。这通常是通过外部的存储系统来实现，能够快速检索大量信息。

3）**工具使用（Tool Use)**

- 代理可以通过调用外部的API，来获得模型训练时无法学到的额外信息，比如最新的数据、运行代码的能力，或者访问一些专有的信息源。这样，代理可以超越其原本的知识库，获取到实时或特殊的帮助。

3、LLM Agent 常见功能有哪些?

1）**指令理解与上下文处理**：通过自然语言解析用户指令，理解任务目标和上下文信息。

2）**任务规划与分解**：LLM Agent能先拟定解决问题的多个步骤，再按顺序逐步执行，确保思路清晰、有条不紊地推进任务。
3）**外部工具使用**：具备调用搜索引擎、数据库、计算器、各类API等工具的能力，以获取或处理信息，实现“知其然，更知其所以然”的效果。
4）**巡环观测与执行**：在每次工具调用后，观测模块会收集执行结果并反馈给决策模块，形成持续的Action-Observation循环，不断微调策略。
5）**记忆管理**：支持短期对话上下文记忆和长期任务历史记忆，将关键数据存储以备后续参考，提升多轮交互的一致性和深度。
6）**决策与分支**：根据观测结果和已有记忆灵活做出决策，处理条件分支和循环逻辑，使执行路径能够动态调整，避免僵化流程。
7）**检索增强生成**：集成检索模块，必要时调用外部知识库或文档，以检索增强生成（RAG）的方式提高回答的准确性和信息丰富度。
8）**自我反思与纠错**：通过内省模块评估自身输出的正确性和完整性，发现偏差时触发重试或修正策略，实现自我纠错。

9）**多模态能力**：结合图像、音频等多种输入输出方式，扩展应用场景。

扩展知识

LLM Agent 的核心优势在于其高度的灵活性和适应性。通过集成记忆、规划和工具调用等模块，LLM Agent 能够模拟人类的思考和决策过程，实现自主的任务执行。

1）记忆（Memory）

LLM Agent 的记忆模块分为短期记忆和长期记忆。短期记忆用于处理当前会话的上下文信息，而长期记忆则用于存储历史交互和知识，以便在未来的任务中进行调用。

2）规划（Planning）

规划模块使 LLM Agent 能够将复杂任务分解为可管理的子任务，并制定执行计划。常用的技术包括思维链（Chain of Thought, CoT）和思维树（Tree of Thoughts, ToT），这些方法帮助智能体理清思路，逐步推进任务的完成。

3）工具调用（Tool Use）

LLM Agent 通过集成外部工具，扩展其功能范围。例如，调用计算器进行数学运算，使用搜索引擎获取最新信息，或调用 API 访问特定服务。

4）自我反思与改进（Self-Reflection）

自我反思模块使 LLM Agent 能够在任务执行后进行评估，识别不足之处，并在后续任务中进行改进。常见的方法包括 Reflexion 和 Self-Refine，这些方法帮助智能体不断提升性能。

5）多模态能力（Multimodal Capabilities）

随着技术的发展，LLM Agent 不再局限于文本输入输出。通过结合图像、音频等多种输入输出方式，LLM Agent 能够处理更复杂的任务，扩展应用场景。

4、Agent智能体的工作过程是怎样的?

1）**接收与理解输入**：Agent首先"感知"环境或用户输入，将自然语言指令转为内部可处理的表示，以明确目标和约束条件。
2）**规划**：基于输入，LLM生成多步行动计划，拆解为子任务并排序，就像制定详细食谱确保烹饪思路清晰。
3）**工具调用**：Agent根据计划动态选择并调用外部工具或API（如搜索、数据库、计算器等）来完成各子任务。
4）**观测**：在每次工具执行后，Agent将输出作为"Observation"反馈给LLM，用于更新当前状态和后续决策。
5）**记忆**：关键观测和交互细节被存入短期或长期记忆，提高多轮对话或复杂任务中的上下文一致性。
6）**决策**：基于最新的观测结果、记忆内容和任务目标，Agent判断是否继续执行下一步操作或结束流程。
7）**输出**：在所有必要步骤完成后，Agent汇总信息，生成并返回最终结果给用户。
8）**反思与纠错**：可选地启动自我反思模块，评估已执行行动的正确性，必要时重规划或修正策略，避免重复错误。

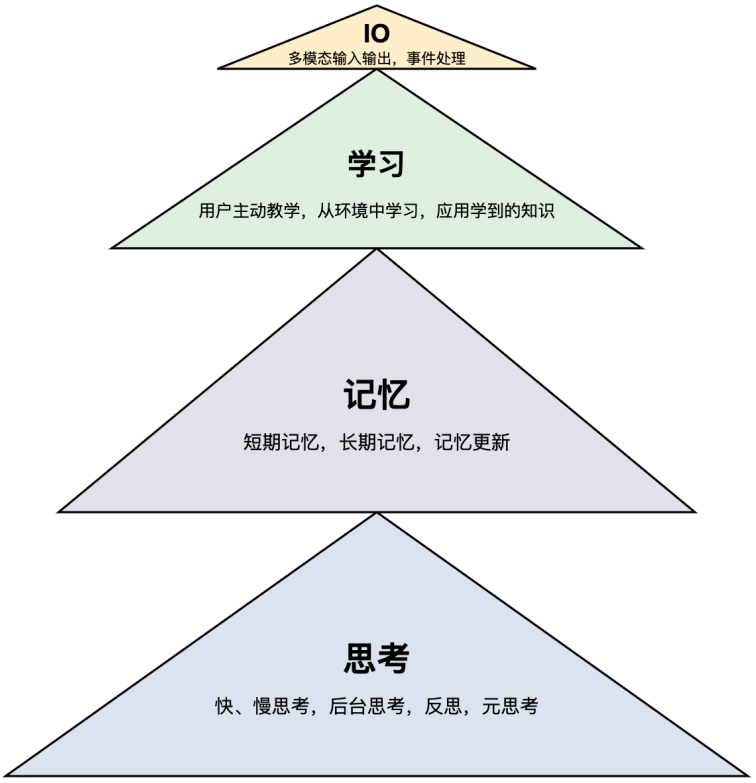
扩展知识

Agent智能体的核心组成模块包括**规划（Planning）**、**工具使用（Tool Use）**、**观测反馈（Observation Feedback）**、**记忆（Memory）**、**决策（Decision Making）****和可选的**自我反思（Reflection）***以及*安全合规（Safety and Compliance）机制。**
在规划阶段，Agent会利用LLM生成一系列子任务步骤，就像厨师在烹饪前先列好详细食谱，确保过程有序可控。
工具使用阶段则是根据各子任务选择相应"厨房用具"（即API或外部工具）来完成操作，例如调用搜索引擎查询信息或执行计算器功能。
在观测反馈循环中，Agent会把工具执行结果作为"尝味"环节，将输出重新输入到LLM，让模型根据最新信息调整下一步动作。
记忆管理帮助Agent在多轮交互中记住关键数据，就像厨师在烹饪笔记里记录经验，以便下次使用时能一目了然。

自我反思模块（Reflection）可让Agent在遇到错误时像厨师试味失败后回炉再制，通过分析前次决策和结果来优化策略。
安全与合规模块则在整个流程中进行内容审查和策略校验，防止生成敏感或违规信息，类似厨房中的卫生和安全标准。

5、如何让 LLM Agent 具备长期记忆能力?

LLM 本身的上下文窗口通常只有几千到数万 tokens，无法直接处理过多的历史交互信息，所以需要借助外部机制来扩展其“记忆”能力。
1）**通过向量数据库 + RAG 机制增强长期记忆**：先将对话和知识转换成向量 embeddings 存入外部数据库（如 FAISS、ChromaDB 或 Pinecone），在新会话发起时根据用户查询检索相关历史内容，再将检索到的结果拼接至模型输入上下文，弥补原生上下文窗口的缺陷。
2）**Memory Transformer / 分层记忆体系**：结合短期记忆（会话上下文）和长期记忆（外部存储的关键摘要或 embeddings），通过 Memory Networks、Neural Turing Machines 或类似机制，将重要信息定期摘要成紧凑表示，存入专用存储，并在需要时根据上下文召回，实现分层记忆管理。



扩展知识

在 RAG 的标准流程中，首先对外部数据进行 **Indexing**（将文本切分并生成 **embeddings** 存入向量库），然后根据用户输入**执行 Retrieval**，选出最相关的文档，接着在** Augmentation 阶段**将检索结果拼接到原始提示中，最后由 LLM 执**行 Generation**，生成最终回答。
可以把向量数据库想象成一个大型图书馆，embeddings 是书籍编号，而 RAG 就像图书管理员，能够在海量“书架”中快速找到最贴近用户需求的“书”，帮助模型借阅并“阅读”之前的对话或知识。
对于 Memory Transformer 或 Hierarchical Memory 思路，则更像是给模型提供了一个分层文件柜：近期对话放在易取的抽屉里（短期记忆），具有重要价值的信息则压缩成摘要或 embeddings 储存在更深层的保密文件柜（长期记忆），需要时再取出翻阅。
目前还可结合持续学习（Continual Learning）或少量参数微调（如 LoRA、P-tuning）将核心知识直接内化到模型参数中，实现更稳定的长期记忆，但需要额外的训练和版本管理成本。
在实际落地时，需权衡存储成本、检索延迟以及用户数据隐私等问题，向量库规模越大搜索速度越可能变慢，同时也要确保数据安全合规。

另外，新兴的长上下文处理技术如 LongRoPE、LLoCO、CEPE 等，也可以在一定程度上提升原生上下文窗口，通过改进位置编码或并行编码等方式，将上下文扩展到数十万甚至百万 tokens，帮助 LLM 在更大范围内保持连贯和记忆。

6、LLM Agent 如何进行动态 API 调用?

在面对让 **LLM Agent 动态调用外部 API** 的需求时，核心思路就是让模型能够像调用本地函数一样，把各种 API 当作"工具"提供给它，然后根据用户的意图自动选择并执行对应的工具。
1）**插件机制**：先在平台（如 OpenAI Plugin 或 LangChain Agents）中注册好各类 API，把它们封装成插件或工具，当模型检测到用户需要某项功能时，就会自动加载对应插件并发起调用。
2）**动态函数调用**：利用 OpenAI GPT-4 Turbo 的 function-calling 能力，事先定义好函数接口（函数名、参数格式、返回值结构等），模型在生成响应时如果判断需要调用，就会输出符合该接口的 JSON，并由后端框架解析后触发真实 API 调用。
3）**代码解释器**：部署一个受限的 Python 运行环境（或沙箱），当模型需要做一些计算、数据处理甚至调用第三方库时，会生成相应的代码片段，在该环境中执行后再将结果反馈给模型和用户。

扩展知识

插件机制就像给模型配备了一个"**工具箱**"，每个 API 都事先打包成一个"**工具**"，只要模型在对话中触发，就能直接拿出来用。这种方式特别灵活，后续要加新的功能，只需再注册一个插件即可，而且各插件相互隔离，不会互相干扰。
动态函数调用更像是在模型和外部世界之间架了一个"**转换桥**"：需要先告诉模型有哪些函数可用，包括名字和入参格式，模型只要在对话里写出调用哪个函数、传什么参数，就相当于在桥上留下了指令，后端一收就执行，最后把结果再用自然语言或者 JSON 形式传回去。
代码解释器则是给模型一个"小实验室"，它需要算点儿东西、读读文件、处理下数据时，就像亲自在实验台上写代码调试一样，把代码提交到沙箱里跑，拿到结果后再用对话形式反馈给用户。这种方式对复杂计算、数据分析、甚至动态生成新工具都很有帮助。

除了上面三种常用手段，还有一些其他的思路：

1）**Responses API**：OpenAI 推出的新接口，用内置的"插件化"能力取代部分外部框架，让模型在一个统一的生态里就能完成更多工具调用，无需额外引入 LangChain 等中间层。

2）**自适应工具生成**（如 ATLASS）：让模型在运行时自己去判断需要什么工具，甚至自动从文档或外部资源生成对应的调用脚本，再加载执行，实现"按需造轮子"。

3）**强化学习训练**：把 API 调用当成一系列动作，通过与环境（API 接口）的交互进行强化训练，模型能在长期交互中学会更准确、稳健地调用外部服务。

4）**本地化高速缓存**（LLM-dCache）：针对频繁相似的数据访问场景，把常用的 API 调用结果缓存下来，让模型先在本地缓存里查一查，减少实际请求量，提高效率。

7、LLM Agent 在多模态任务中如何执行推理?

LLM Agent 在多模态推理中，核心是先把不同模态的数据（如图像、音频、视频、文本）转换成统一的**向量或语义**表示，再将这些表示注入到大模型进行跨模态融合和推理。
1）**视觉-语言模型**：常见做法是用 CLIP、BLIP-2 等视觉编码器将图像转为 embeddings，然后与文本输入一起提供给 LLM；或者直接使用 GPT-4V 这样内置视觉理解能力的模型来处理图像并输出自然语言回答。
2）**语音-文本桥接**：用 Whisper 或者其他 ASR 模型将音频转换成文本，再交给 LLM 去分析和生成响应，前后端串联即可实现多模态推理。
3）**工具链调用**：结合 OCR、物体检测、视频帧提取等工具，LLM Agent 根据任务需求按需调用这些工具，将结果合并进它的上下文，再由模型执行更高级别的推理或决策。

扩展知识

多模态推理不仅仅是“贴图+读字”，还涉及跨**模态对齐**和**注意力融合**。比如 P2 G（Plug-and-Play Grounding）框架中，模型内部会专门插入视觉投影模块，将视觉特征映射到语言空间，与文本特征在 Transformer 每一层并行交互，做到更细粒度的视觉-语言对齐。

在实际工程中，LangChain 这类中间件也开始支持多模态输入和工具调用，让开发者更方便地把 OCR、图像分类、语音识别等“插件”接入到 ChatAgent 流程里，通用化地管理多模态任务。

如果要处理视频，多数会对视频做帧抽取，然后对关键帧依次进行图像编码，同时还可能结合声音信号提取的语音文本信息，最后把所有模态的信息通过时间序列与 LLM 的记忆或长短期上下文融合，实现时序化的多模态推理。

在模型选型上，BLIP-2 提供了一种轻量化思路：它把图像编码器和 LLM 都保持冻结，只在中间加一个小型映射层，就能高效对齐视觉和语言，这在资源受限场景下很有优势。

对于需要实时或在线多模态交互的场景，还可以使用 GPT-4 Vision API，直接调用 OpenAI 的托管模型，让云端完成图像理解，再通过后续的语言模型生成环节来给出最终响应。

最后，如果对多模态推理精度和解释性有更高要求，可以在 LLM 的推理链（Chain of Thought）中插入辅助提示，让模型分步“**看图—识别—推理**”，不仅提升准确度，也更易于审计和调试。

8、市面上有哪些主流的 LLM Agent 框架？各自的特点是什么？

1）**LangChain**：模块化设计，支持多模型接入、链式调用、工具集成和记忆管理，适合快速构建复杂的 LLM 应用。

2）**LlamaIndex**（原 GPT Index）：专注于数据索引和检索，优化 LLM 与外部数据的结合，增强检索能力（RAG）。

3）**AutoGPT**：实现自主 AI 代理，能够生成目标、拆解任务、自主迭代，适用于需要自主决策的场景。

4）**BabyAGI**：最小化的自主 AI Agent，基于 OpenAI + Pinecone 进行任务迭代，适合轻量级的任务调度。

5）**CrewAI**：支持多个 Agent 组成团队协作，不同 Agent 具有不同角色，适用于复杂的工作流程。

6）**LangGraph**：基于有向无环图（DAG）的 LLM 工作流管理，使 Agent 任务更具可控性和可扩展性，适合多步骤推理和任务分解。

扩展知识

1）LangChain

LangChain 是一个开源框架，旨在简化使用大型语言模型的应用程序。它提供了一个标准接口，用于将不同的语言模型（如 GPT-4、BERT、T5）连接在一起，以及与其他工具和数据源的集成。LangChain 的主要特点包括：

- **链式调用（Chains）**：支持多步推理，如 CoT（Chain of Thought）。
- **工具集成（Tools）**：整合数据库、API、搜索引擎等，扩展模型能力。
- **内存管理（Memory）**：支持长期会话记忆，提升对话连续性。
- **代理机制（Agents）**：可以动态选择工具，执行复杂任务。

LangChain 的模块化设计，使得开发者可以根据需求灵活组合各个组件，快速构建出功能丰富的 LLM 应用。

2）LlamaIndex

LlamaIndex（原 GPT Index）专注于优化 LLM 与外部数据的结合，增强检索能力（RAG）。它的主要特点包括：

- **数据索引（Indexing）**：支持不同格式的文档（如 PDF、SQL）进行索引。

- **查询路由（Query Routing）**：智能选择索引，提升查询效率。
- **向量存储集成**：与 FAISS、Weaviate 等向量数据库无缝集成。

通过构建高效的索引和检索机制，LlamaIndex 使得 LLM 能够更准确地从大量外部数据中提取相关信息，提升回答的准确性和上下文相关性。

3）AutoGPT

AutoGPT 是一个开源的自主 AI 代理程序，能够根据用户设定的目标，自动拆解任务并执行。其主要特点包括：

- **自主性**：能够生成目标、拆解任务、自主迭代，无需人工干预。
- **长记忆**：结合本地文件存储与向量数据库，支持长期记忆管理。
- **多工具调用**：支持 API 访问、代码执行等多种工具调用。

AutoGPT 适用于需要自主决策和执行的场景，如自动化研究、项目管理等。

4）BabyAGI

BabyAGI 是一个最小化的自主 AI Agent，基于 OpenAI 和 Pinecone 进行任务迭代。其主要特点包括：

- **任务队列（Task Queue）**：控制任务调度，确保任务按顺序执行。
- **轻量级设计**：适合资源有限的环境，快速部署和运行。

BabyAGI 适用于需要简单任务调度和执行的场景，如自动化数据处理、定期报告生成等。

5）CrewAI

CrewAI 支持多个 Agent 组成团队协作，不同 Agent 具有不同角色，适用于复杂的工作流程。其主要特点包括：

- **多智能体架构**：支持多个 Agent 协同工作，每个 Agent 负责特定任务。
- **角色分工**：不同 Agent 具有不同角色（如 Researcher、Writer），实现专业化协作。
- **LangChain 兼容**：可与 LangChain Agents、Tools、Memory 结合，增强任务流管理能力。

CrewAI 适用于需要多个角色协同工作的场景，如内容创作、市场分析等。

6）LangGraph

LangGraph 提供基于有向无环图（DAG）的 LLM 工作流管理，使 Agent 任务更具可控性和可扩展性。其主要特点包括：

- **图计算架构（Graph-based Execution）**：基于 DAG 结构设计任务流，支持并行执行，提高效率。
- **状态管理（State Management）**：支持持久化存储任务执行状态，确保上下文一致性。

9、如何实现长短期记忆机制?

1. **短期记忆**：利用**上下文窗口**（如Claude的100k tokens）缓存最近的对话或输入，或通过**滑动窗口**（Buffer Window）保留固定轮次的历史。
2. **长期记忆**：结合**摘要压缩**（Summarization）、**向量数据库**（如RAG检索增强）持久化存储关键信息。

从目前的大模型实现来看，短期记忆依赖模型原生能力，长期记忆通过外部存储（如知识图谱、数据库）实现检索。

短期记忆实现方式介绍

- 直接缓存：如GPT-4的32k tokens窗口，将最近输入全部保留在内存中处理。
- 滑动窗口：仅保留最近N轮对话（例如5轮），防止过长历史消耗资源。
- Token限制：按Token数量截断历史（例如仅保留12k tokens）。

长期记忆实现方式介绍

1）摘要压缩与分层管理：对历史内容生成摘要（例如每10轮对话压缩为1段），模仿人类记忆的"模糊记忆"机制。

可以利用 LangChain 的 `SummaryBufferMemory` 结合短期缓存与长期摘要。

这里再简单介绍下LangChain 记忆模块：

- ConversationBufferMemory：存储完整对话历史。
- VectorStoreRetrieverMemory：将对话嵌入向量数据库，检索相关片段。
- ConversationKGMemory：构建知识图谱，支持实体关系查询。

2）检索增强生成（RAG）：将历史信息存入向量数据库（如Milvus），通过语义检索动态召回相关片段。

这个方式可以突破模型原生窗口限制，支持海量数据。

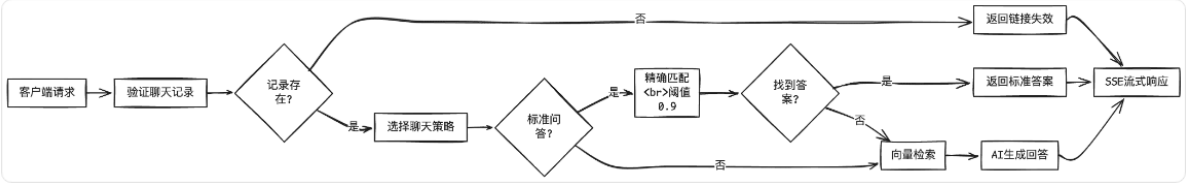
二、RAG

> 1，说一下RAG有哪几个步骤？ 2，实际RAG项目中，用过哪些优化技巧？ 3，说一下RAG一般怎么做效果评估？ 4，为什么RAG中会出现幻觉？如何解决？ 5，实际项目中遇到的各种边界case，你们是怎么解决的？

1、PIGX AI知识库问答源码解析

`/msg/list` 接口是 PigX 知识库系统中用于处理 AI 聊天请求的核心接口，采用 Server-Sent Events (SSE) 技术实现流式响应。该接口通过统一的 RAG（检索增强生成）策略，支持多种聊天模式，特别是标准问答策略（StandardQuestionAnswerStrategy）的精确匹配功能。

整体流程图



接口调用流程

1. 接口入口

```
@GetMapping(value = "/msg/list", produces = MediaType.TEXT_EVENT_STREAM_VALUE)
public Flux<AiMessageResultDTO> msg(@RequestParam Long key) {
    try {
        return chatService.chatList(key);
    } catch (Exception e) {
```

```
        log.error("chat error", e);
        return Flux.just(new AiMessageResultDTO(e.getMessage()))
            .concatWithValues(new AiMessageResultDTO(END_MSG));
    }
}
```

关键特性：

- 使用 `MediaType.TEXT_EVENT_STREAM_VALUE` 支持 SSE 流式响应
- 异常处理机制，确保错误信息也能通过流返回
- 返回 `Flux<AiMessageResultDTO>` 响应式流

2. 聊天服务处理

```
@Override
public Flux<AiMessageResultDTO> chatList(Long key) {
    // 1. 获取聊天记录
    AiChatRecordEntity recordEntity = recordMapper.selectById(key);
    if (Objects.isNull(recordEntity)) {
        return Flux.just(new AiMessageResultDTO("链接已失效"), new
AiMessageResultDTO(END_MSG));
    }

    // 2. 构建聊天消息DTO
    ChatMessageDTO chatMessageDTO = new ChatMessageDTO();
    chatMessageDTO.setMessageKey(key);
    chatMessageDTO.setModelName(recordEntity.getModelName());
    chatMessageDTO.setContent(recordEntity.getQuestionText());
    chatMessageDTO.setConversationId(recordEntity.getConversationId());
    chatMessageDTO.setDatasetId(recordEntity.getDatasetId());

 chatMessageDTO.setWebsearch(YesNoEnum.YES.getCode().equals(recordEntity.getWebse
archFlag()));

 chatMessageDTO.setInner(YesNoEnum.YES.getCode().equals(recordEntity.getInnerFlag
()));

    // 3. 规则引擎风险检测（可选）
    if (flowExecutorOptional.isPresent()) {
        Flux<AiMessageResultDTO> aiMessageResultDTO = flowRisk(chatMessageDTO);
        if (aiMessageResultDTO != null)
            return aiMessageResultDTO.concatWithValues(new
AiMessageResultDTO(END_MSG));
    }

    // 4. 根据聊天类型选择处理规则
    ChatTypeEnums chatTypeEnums =
ChatTypeEnums.fromCode(chatMessageDTO.getDatasetId());
    ChatMessageContextHolder.set(chatMessageDTO);
    return chatRuleMap.get(chatTypeEnums.getType()).process(chatMessageDTO)
        .map(aiMessageResultDTO -> {
            aiMessageResultDTO.setMessageKey(key.toString());
            return aiMessageResultDTO;
        }).concatWithValues(new AiMessageResultDTO(END_MSG));
```

```
}
```

处理步骤：

1. **记录验证**：根据 key 获取聊天记录，验证链接有效性
2. **消息构建**：将数据库记录转换为聊天消息DTO
3. **风险检测**：可选的规则引擎风险控制
4. **策略选择**：根据数据集ID选择对应的聊天规则

3. 聊天类型枚举

```
public enum ChatTypeEnums {
    FUNCTION_CHAT(-1L, "functionChat"),    // 功能聊天
    SIMPLE_CHAT(0L, "simpleChat"),         // 简单聊天
    DATABASE_CHAT(-2L, "databaseChat"),    // 数据库聊天
    IMAGE_CHAT(-3L, "text2ImageChat"),     // 图片生成
    MARKMAP_CHAT(-4L, "text2MarkMapChat"), // 脑图生成
    FLOW_CHAT(-5L, "functionChat"),        // 流程编排
    JSON_CHAT(-6L, "jsonChat"),            // JSON聊天
    REASON_CHAT(-7L, "reasonChat"),        // 推理聊天
    MCP_CHAT(-8L, "mcpChat"),              // MCP聊天
    VECTOR_CHAT(1L, "vectorChat");         // 知识库聊天（默认）
}
```

StandardQuestionAnswerStrategy 检索逻辑深度分析

1. 策略概述

`StandardQuestionAnswerStrategy` 是专门用于标准问答对精确匹配的策略实现，适用于 FAQ、客服等需要精确回答的场景。

```
@Component("standardQuestionAnswerStrategy")
@RequiredArgsConstructor
public class StandardQuestionAnswerStrategy implements UnifiedRagStrategy {

    private static final String HANDLER_TYPE = "Q2Q_STANDARD";
    private static final double HIGH_SIMILARITY_THRESHOLD = 0.9; // 高相似度阈值

    private final EmbeddingStoreService embeddingStoreService;
}
```

核心特性：

- 高精度匹配：相似度阈值设置为 0.9，确保精确匹配
- 问答对处理：专门处理 QUESTION 类型的文档
- 快速响应：直接返回预定义的标准答案

2. 检索流程详解

2.1 向量化查询

```
@Override
public Flux<AiMessageResultDTO> processChat(Embedding queryEmbedding,
AiDatasetEntity dataset,
        ChatMessageDTO chatMessageDTO) {
    log.debug("使用标准问答策略处理查询: {}", chatMessageDTO.getContent());

    // 1. 构建高精度搜索请求
    EmbeddingSearchRequest searchRequest =
RagHelper.buildSearchRequest(queryEmbedding, dataset,
            DocumentTypeEnums.QUESTION.getType(), HIGH_SIMILARITY_THRESHOLD);
```

关键点：

- 使用预计算的查询向量 `queryEmbedding`
- 指定文档类型为 `QUESTION`（问题类型）
- 设置高相似度阈值 0.9

2.2 搜索请求构建

```
public static EmbeddingSearchRequest buildSearchRequest(Embedding queryEmbedding,
AiDatasetEntity dataset,
        String documentType, Double minScore) {
    double finalMinScore;
    if (Objects.nonNull(minScore)) {
        finalMinScore = minScore;
    } else {
        // 使用数据集配置的默认分数阈值
        finalMinScore =
NumberUtil.div(Double.parseDouble(dataset.getScore().toString()), 100.0, 2);
    }

    return new EmbeddingSearchRequest(queryEmbedding, dataset.getTopK(),
finalMinScore,

 metadataKey(AiDocumentEntity.Fields.datasetId).isEqualTo(dataset.getId().toStrin
g())

.and(metadataKey(DocumentTypeEnums.Fields.type).isEqualTo(documentType)));
}
```

搜索条件：

- 数据集ID匹配：`datasetId = dataset.getId()`
- 文档类型匹配：`type = "1"`（QUESTION类型）
- 相似度阈值：0.9（高精度要求）
- 返回数量：`dataset.getTopK()`

2.3 向量搜索执行

```
// 2. 执行向量搜索
EmbeddingStore<TextSegment> embeddingStore =
embeddingStoreService.embeddingStore(dataset.getCollectionName());
EmbeddingSearchResult<TextSegment> searchResult =
embeddingStore.search(searchRequest);
List<EmbeddingMatch<TextSegment>> embeddingMatches = searchResult.matches();
```

搜索过程：

1. 获取指定集合的向量存储实例
2. 执行向量相似度搜索
3. 获取匹配结果列表

2.4 结果验证与处理

```
// 3. 检查是否找到高度相似的标准问题
if (RagHelper.isEmpty(embeddingMatches)) {
    log.debug("未找到相似度 > {} 的标准问题", HIGH_SIMILARITY_THRESHOLD);
    return Flux.empty();
}

// 4. 提取标准答案
List<String> standardAnswers =
RagHelper.extractStandardAnswers(embeddingMatches);

if (CollUtil.isEmpty(standardAnswers)) {
    log.warn("匹配到标准问题但未找到对应答案");
    return Flux.empty();
}

// 5. 返回第一个匹配的标准答案
String answer = standardAnswers.get(0);
log.debug("标准问答策略：返回标准答案，相似度：{}", embeddingMatches.get(0).score());

return Flux.just(new AiMessageResultDTO(answer));
```

处理逻辑：

1. **空结果检查**：如果没有匹配结果，返回空流
2. **答案提取**：从匹配结果中提取标准答案
3. **答案验证**：确保答案存在
4. **返回结果**：返回第一个（最相似）的标准答案

2.5 标准答案提取

```
public static List<String>
extractStandardAnswers(List<EmbeddingMatch<TextSegment>> embeddingMatches) {
    return embeddingMatches.stream()
        .map(match ->
match.embedded().metadata().getString(AiChatRecordEntity.Fields.answerText))
        .filter(Objects::nonNull)
        .toList();
}
```

提取过程：

- 从每个匹配结果的元数据中提取 `answerText` 字段
- 过滤掉空值
- 返回答案列表

策略选择与降级机制

1. VectorChatRule 策略选择

```
@Override
public Flux<AiMessageResultDTO> process(ChatMessageDTO chatMessageDTO) {
    // 获取数据集配置和向量化模型
    AiDatasetEntity dataset =
aiDatasetService.getById(chatMessageDTO.getDatasetId());
    DimensionAwareEmbeddingModel embeddingModel =
modelProvider.getEmbeddingModel(dataset.getEmbeddingModel());
    Embedding queryEmbedding =
embeddingModel.embed(chatMessageDTO.getContent()).content();

    // 如果启用了标准化数据，先尝试标准问答匹配
    if (YesNoEnum.YES.getCode().equals(dataset.getStandardFlag())) {
        Flux<AiMessageResultDTO> standardResult = standardQuestionAnswerStrategy
            .processChat(queryEmbedding, dataset, chatMessageDTO)
            .cache();

        // 如果标准问答没有结果，降级到向量检索
        return standardResult.switchIfEmpty(

vectorRetrievalAugmentedGenerationStrategy.processChat(queryEmbedding, dataset,
chatMessageDTO));
    }

    // 直接使用向量检索策略
    return vectorRetrievalAugmentedGenerationStrategy.processChat(queryEmbedding,
dataset, chatMessageDTO);
}
```

策略选择逻辑：

1. **标准化数据优先**：如果数据集启用了标准化标志，优先使用标准问答策略
2. **降级机制**：标准问答无结果时，自动降级到向量检索增强生成策略

3. **直接向量检索**：未启用标准化时，直接使用向量检索策略

2. 降级策略对比

策略	相似度阈值	处理方式	适用场景
StandardQuestionAnswerStrategy	0.9	精确匹配，直接返回预定义答案	FAQ、客服问答
VectorRetrievalAugmentedGenerationStrategy	数据集配置	向量检索 + AI生成	文档检索、知识问答

技术架构特点

1. 响应式编程

- 使用 `Flux` 实现流式响应
- 支持 SSE 实时推送
- 异常处理机制完善

2. 策略模式

- 统一的 `UnifiedRagStrategy` 接口
- 多种策略实现
- 动态策略选择

3. 向量检索

- 基于 LangChain4j 框架
- 支持多种向量数据库
- 高精度相似度匹配

4. 元数据管理

- 文档类型区分（QUESTION/ANSWER）
- 数据集隔离
- 命中统计

性能优化

1. 缓存机制

```
.cache()  // 缓存标准问答结果，避免重复计算
```

2. 阈值控制

- 高相似度阈值（0.9）确保精确匹配
- 减少不必要的 AI 生成调用

3. 流式处理

- 实时返回结果
- 减少用户等待时间

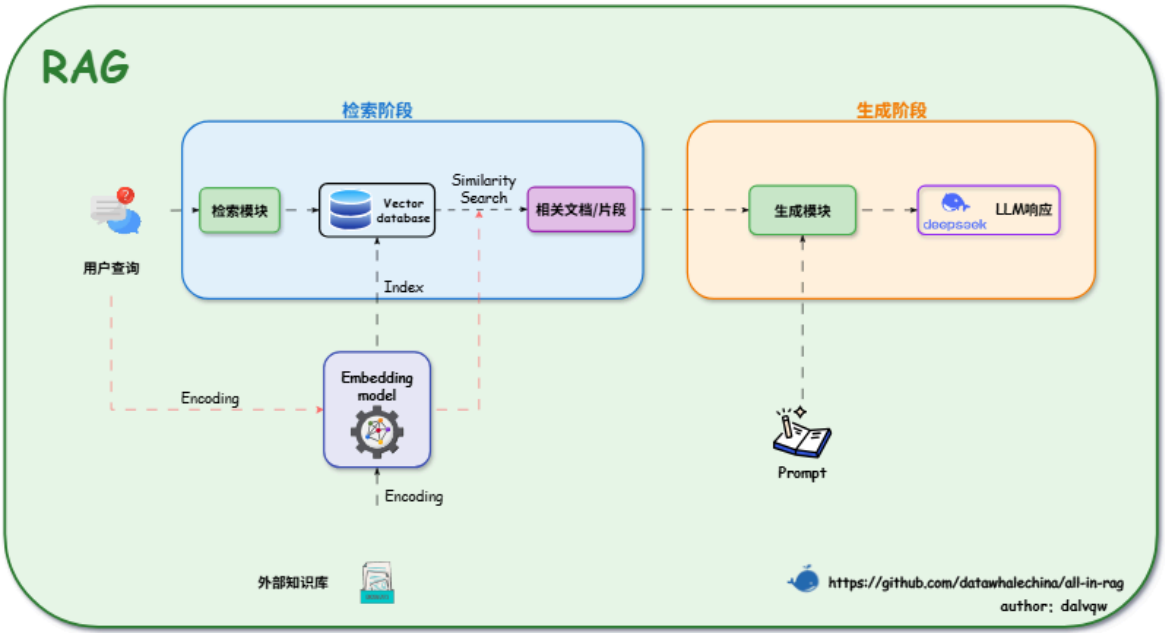
总结

`/msg/list` 接口通过精心设计的策略模式，实现了高效的 AI 聊天功能。`StandardQuestionAnswerStrategy` 作为其中的核心组件，通过高精度向量匹配和预定义答案返回，为 FAQ 和客服场景提供了快速、准确的响应能力。整个系统采用响应式编程和流式处理，确保了良好的用户体验和系统性能。

2、All in RAG

1.技术原理

RAG本质：在LLM生成文本之前，先从**外部知识库**中检索相关信息，作为上下文辅助生成更准确的回答。



- **关键组件**：
  1. **索引（Indexing）** 📑：将非结构化文档（PDF/Word等）分割为片段，通过嵌入模型转换为向量数据。
  2. **检索（Retrieval）** 🔍：基于查询语义，从向量数据库召回最相关的文档片段（Context）。
  3. **生成（Generation）** ✨：将检索结果作为上下文输入LLM，生成自然语言响应。

RAG技术按照复杂度可划分为：

初级RAG	高级RAG	模块化RAG
基础"索引-检索-生成"流程	增加数据清洗流程	灵活集成搜索引擎

初级RAG	高级RAG	模块化RAG
简单文档分块	元数据优化	强化学习优化
基本向量检索机制	多轮检索策略	知识图谱增强
-	提升准确性和效率	支持复杂业务场景

1.1 为什么需要RAG

LLM局限性和问题	RAG的解决方案
静态知识局限	实时检索外部知识库，支持动态更新
幻觉（Hallucination）	基于检索内容生成，错误率降低
领域专业性不足	引入领域特定知识库（如医疗/法律）
数据隐私风险	本地化部署知识库，避免敏感数据泄露

1.2 四步构件最小可行系统（MVP）

1. **数据准备**
    - 格式支持：PDF、Word、网页文本等
    - 分块策略：按语义（如段落）或固定长度切分，避免信息碎片化
2. **索引构建**
    - 嵌入模型：选取开源模型（如text-embedding-ada-002）或微调领域专用模型（金融、医疗）
    - 向量化：将文本分块转换为向量存入数据库
3. **检索优化**
    - 混合检索：需要在数据库中提前建立向量索引和关键词索引，结合关键词（BM25）与语义搜索（向量相似度）提升召回率
    - 重排序（Rerank）：用小模型筛选Top-K相关片段（如Cohere Reranker）
4. **生成集成**
    - 提示工程：设计模板引导LLM融合检索内容，避免编造
    - LLM选型：GPT、Claude、Ollama等（按成本/性能权衡）

> | 技术 | 核心 | 优点 | 缺点 |
> | --- | --- | --- | --- |
> | **向量检索** | 语义相似 | 能理解同义/语义相关 | 可能漏关键字 |
> | **全文检索** | 关键词匹配 | 精确匹配关键词 | 不理解语义 |
> | **混合检索** | 向量+关键词 | 高召回率 | 实现稍复杂 |
> | **重排序** | 候选排序 | 提升准确度 | 需要额外模型计算 |

2.数据准备

> 高质量输入是高质量输出的前提

2.1 文档加载器

1）主要功能：

- **文档格式解析** 将不同格式的文档（如PDF、Word、Markdown等）解析为文本内容。
- **元数据提取** 在解析文档内容的同时，提取相关的元数据信息，如文档来源、页码等。
- **统一数据格式** 将解析后的内容转换为统一的数据格式，便于后续处理。

2）当前主流RAG文档加载器：

工具名称	特点	适用场景	性能表现
PyMuPDF4LLM	PDF→Markdown转换，OCR+表格识别	科研文献、技术手册	开源免费，GPU加速
TextLoader	基础文本文件加载	纯文本处理	轻量高效
DirectoryLoader	批量目录文件处理	混合格式文档库	支持多格式扩展
Unstructured	多格式文档解析	PDF、Word、HTML等	统一接口，智能解析
FireCrawlLoader	网页内容抓取	在线文档、新闻	实时内容获取
LlamaParse	深度PDF结构解析	法律合同、学术论文	解析精度高，商业API
Docling	模块化企业级解析	企业合同、报告	IBM生态兼容
Marker	PDF→Markdown，GPU加速	科研文献、书籍	专注PDF转换
MinerU	多模态集成解析	学术文献、财务报表	集成LayoutLMv3+YOLOv8

3）Unstructured文档处理库：

Unstructured 是一个专业的文档处理库，专门设计用于RAG和AI微调场景的非结构化数据预处理。提供了统一的接口来处理多种文档格式，是目前最受欢迎的文档加载解决方案之一。

格式支持广泛

- 支持多种文档格式：PDF、Word、Excel、HTML、Markdown等
- 统一的API接口，无需为不同格式编写不同代码

智能内容解析

- 自动识别文档结构：标题、段落、表格、列表等

- 保留文档元数据信息

2.2 文本分块

文本分块（Text Chunking）是构建RAG流程的关键步骤。其核心原理是将加载后的长篇文档，切分成更小、更易于处理的单元。这些被切分出的文本块，是后续向量检索和模型处理的**基本单位。**

1）为什么需要分块?

1.嵌入模型Embedding Model：负责将文本块转换为向量，这类模型有严格的输入长度上限
2.大语言模型LLM：负责根据检索到的上下文生成答案，LLM同样有上下文窗口限制

2）为什么分块不是越大越好?

- 嵌入过程信息损失：

  大多数嵌入模型都基于 Transformer 编码器。其工作流程大致如下：

  1.**分词 (Tokenization)**: 将输入的文本块分解成一个个 token。

  2.**向量化 (Vectorization)**: Transformer 为**每个 token** 生成一个高维向量表示。

  3.**池化 (Pooling)**: 通过某种方法（如取`[CLS]`1位的向量、对所有token向量求平均`mean pooling`等），将所有 token 的向量**压缩**成一个**单一的向量**，这个向量代表了整个文本块的语义。

  > 在这个`压缩`过程中，信息损失是不可避免的。一个768维的向量需要概括整个文本块的所有信息。**文本块越长，包含的语义点越多，这个单一向量所承载的信息就越稀释**，导致其表示变得笼统，关键细节被模糊化，从而降低了检索的精度。

- 生成过程大海捞针：

  如果提供给LLM的上下文块又大又杂，充满了与问题无关的噪音，模型就很难从中提取出最关键的信息来形成答案，从而导致回答质量下降或产生幻觉。

- 主题稀释导致检索失败：

  一个好的文本块应该聚焦于一个明确、单一的主题。如果一个块包含太多不相关的主题，它的语义就会被稀释，导致在检索时无法被精确匹配。

3）基础分块策略?

- 固定大小分块
- 递归字符分块
- 语义分块
- 基于文档结构的分块：如Markdown、HTML、LaTex
- 其他优秀开源的分块策略
  - Unstructured：基于文档元素的智能分块
  - LlamaIndex：面向节点的解析和转换

3.构建索引

3.1 向量嵌入

向量嵌入（Embedding）是一种将真实世界中复杂、高维的数据对象（如文本、图像、音频、视频等）转换为数学上易于处理的、低维、稠密的连续数值向量的技术。

- **数据对象**：任何信息，如文本"你好世界"，或一张猫的图片。
- **Embedding 模型**：一个深度学习模型，负责接收数据对象并进行转换。

- **输出向量**：一个固定长度的一维数组，例如 `[0.16, 0.29, -0.88, ...]`。这个向量的维度（长度）通常在几百到几千之间。

1）向量空间语义度量

- **余弦相似度 (Cosine Similarity)**：计算两个向量夹角的余弦值。值越接近 1，代表方向越一致，语义越相似。这是最常用的度量方式。
- **点积 (Dot Product)**：计算两个向量的乘积和。在向量归一化后，点积等价于余弦相似度。
- **欧氏距离 (Euclidean Distance)**：计算两个向量在空间中的直线距离。距离越小，语义越相似。

2）语义检索的通用流程

RAG 的"检索"环节通常以基于 Embedding 的语义搜索为核心。通用流程如下：

1. **离线索引构建**：将知识库内文档切分后，使用 Embedding 模型将每个文档块（Chunk）转换为向量，存入专门的向量数据库中。
2. **在线查询检索**：当用户提出问题时，使用**同一个** Embedding 模型将用户的问题也转换为一个向量。
3. **相似度计算**：在向量数据库中，计算"问题向量"与所有"文档块向量"的相似度。
4. **召回上下文**：选取相似度最高的 Top-K 个文档块，作为补充的上下文信息，与原始问题一同送给大语言模型（LLM）生成最终答案。

> Embedding 的质量直接决定了 RAG 检索召回内容的准确性与相关性。一个优秀的 Embedding 模型能够精准捕捉问题和文档之间的深层语义联系，即使用户的提问和原文的表述不完全一致。反之，一个劣质的 Embedding 模型可能会因为无法理解语义而召回不相关或错误的信息，从而"污染"提供给 LLM 的上下文，导致最终生成的答案质量低下。

3）Embedding技术发展

- 静态词嵌入：上下文无关的表示，无法处理一词多义
- 动态上下文嵌入：`Transformer` 架构的诞生带来了自注意力机制（Self-Attention），它允许模型在生成一个词的向量时，动态地考虑句子中所有其他词的影响。
- RAG对嵌入技术的新要求
    - 领域自适应能力
    - 多粒度和多模态支持
    - 检索效率和混合检索

4）嵌入模型训练原理

现代嵌入模型的核心通常是 Transformer 的编码器（Encoder）部分，`BERT` 就是其中的典型代表。它通过堆叠多个 `Transformer Encoder` 层来构建一个深度的双向表示学习网络。BERT 的成功很大程度上归功于其巧妙的**自监督学习**策略，它允许模型从海量的、无标注的文本数据中学习知识。

1）核心架构BEAR：

- 掩码语言模型MLM：通过这个任务，模型被迫学习每个词元与其上下文之间的关系，从而掌握深层次的语境语义。
- 下一句预测NSP：这个任务让模型学习句子与句子之间的逻辑关系、连贯性和主题相关性。

> 说明：后续的研究（如 RoBERTa）发现2，NSP 任务可能过于简单，甚至会损害模型性能。因此，许多现代的预训练模型（如 RoBERTa、SBERT）已经放弃了 NSP 任务。

2）效果增强：虽然 MLM 和 NSP 赋予了模型强大的基础语义理解能力，但为了在检索任务中表现更佳，现代嵌入模型通常会引入更具针对性的训练策略

- 度量学习：直接以“相似度”作为优化目标。
- 对比学习：向量空间中，将相似的样本“拉近”，将不相似的样本“推远”。

3.2 多模态嵌入

多模态嵌入 (Multimodal Embedding) 的目标正是为了打破这堵墙。其目的是将不同类型的数据（如图像和文本）映射到**同一个共享的向量空间。**在这个统一的空间里，一段描述“一只奔跑的狗”的文字，其向量会非常接近一张真实小狗奔跑的图片向量。

实现这一目标的关键，在于解决 **跨模态对齐 (Cross-modal Alignment)** 的挑战。以对比学习、视觉 Transformer (ViT) 等技术为代表的突破，让模型能够学习到不同模态数据之间的语义关联，最终催生了像 CLIP 这样的模型。

1）CLIP模型

CLIP 的架构清晰简洁。它采用**双编码器架构 (Dual-Encoder Architecture)**，包含一个图像编码器和一个文本编码器，分别将图像和文本映射到同一个共享的向量空间中。

3.3 向量数据库

> 当向量数量从几百个增长到数百万甚至数十亿时，一个核心问题随之而来：**如何快速、准确地从海量向量中找到与用户查询最相似的那几个?**

1）主要功能

向量数据库的核心价值在于其高效处理海量高维向量的能力。其主要功能可以概括为以下几点：

1. **高效的相似性搜索**：这是向量数据库最重要的功能。它利用专门的索引技术（如 HNSW, IVF），能够在数十亿级别的向量中实现毫秒级的近似最近邻（ANN）查询，快速找到与给定查询最相似的数据。
2. **高维数据存储与管理**：专门为存储高维向量（通常维度成百上千）而优化，支持对向量数据进行增、删、改、查等基本操作。
3. **丰富的查询能力**：除了基本的相似性搜索，还支持按标量字段过滤查询（例如，在搜索相似图片的同时，指定`年份 > 2023`）、范围查询和聚类分析等，满足复杂业务需求。
4. **可扩展与高可用**：现代向量数据库通常采用分布式架构，具备良好的水平扩展能力和容错性，能够通过增加节点来应对数据量的增长，并确保服务的稳定可靠。
5. **数据与模型生态集成**：与主流的 AI 框架（如 LangChain, LlamaIndex）和机器学习工作流无缝集成，简化了从模型训练到向量检索的应用开发流程。

2）向量数据库 对比 传统数据库

维度	向量数据库	传统数据库 (RDBMS)
核心数据类型	高维向量 (Embeddings)	结构化数据 (文本、数字、日期)
查询方式	**相似性搜索** (ANN)	**精确匹配**
索引机制	HNSW, IVF, LSH 等 ANN 索引	B-Tree, Hash Index
主要应用场景	AI 应用、RAG、推荐系统、图像/语音识别	业务系统 (ERP, CRM)、金融交易、数据报表

维度	向量数据库	传统数据库 (RDBMS)
数据规模	轻松应对千亿级向量	通常在千万到亿级行数据，更大规模需复杂分库分表
性能特点	高维数据检索性能极高，计算密集型	结构化数据查询快，高维数据查询性能呈指数级下降
一致性	通常为最终一致性	强一致性 (ACID 事务)

向量数据库和传统数据库并非相互替代的关系，而是**互补关系。**利用传统数据库存储业务元数据和结构化信息，而向量数据库则专门负责处理和检索由 AI 模型产生的海量向量数据。

3）工作原理

向量数据库通常采用四层架构，通过以下技术手段实现高效相似性搜索：

1. **存储层**：存储向量数据和元数据，优化存储效率，支持分布式存储
2. **索引层**：维护索引算法（HNSW、LSH、PQ等），创建和优化索引，支持索引调整
3. **查询层**：处理查询请求，支持混合查询，实现查询优化
4. **服务层**：管理客户端连接，提供监控和日志，实现安全管理

主要技术手段包括：

- **基于树的方法**：如 Annoy 使用的随机投影树，通过树形结构实现对数复杂度的搜索
- **基于哈希的方法**：如 LSH（局部敏感哈希），通过哈希函数将相似向量映射到同一“桶”
- **基于图的方法**：如 HNSW（分层可导航小世界图），通过多层邻近图结构实现快速搜索
- **基于量化的方法**：如 Faiss 的 IVF 和 PQ，通过聚类和量化压缩向量

3.4 Milvus实践

与 FAISS、ChromaDB 等轻量级本地存储方案不同，Milvus 从设计之初就瞄准了**生产环境。**其采用云原生架构，具备高可用、高性能、易扩展的特性，能够处理十亿、百亿甚至更大规模的向量数据。

Milvus docker部署：Docker 将会自动拉取所需的镜像并启动三个容器：`milvus-standalone`，`milvus-minio`，和 `milvus-etcd`

1）核心组件

- **Collection（集合）**

  Collection 是 Milvus 中最基本的数据组织单位，类似于关系型数据库中的一张**表 (Table)。**是我们存储、管理和查询向量及相关元数据的容器。所有的数据操作，如插入、删除、查询等，都是围绕 Collection 展开的。

- **Schema（模式）**

  `Schema` 规定了 Collection 的数据结构，定义了其中包含的所有**字段 (Field)** 及其属性

  - 主键字段：每个 Collection 必须有且仅有一个主键字段，用于唯一标识每一条数据（实体）
  - 向量字段：用于存储核心的向量数据
  - 标量字段：用于存储除向量之外的元数据，如字符串、数字、布尔值、JSON 等。这些字段可以用于过滤查询，实现更精确的检索

- **Partition（分区）**

Partition 是 Collection 内部的一个逻辑划分。每个 Collection 在创建时都会有一个名为 `_default` 的默认分区。我们可以根据业务需求创建更多的分区，将数据按特定规则（如类别、日期等）存入不同分区。

分区用于提升查询性能和数据管理，一个 Collection 最多可以有 1024 个分区。合理利用分区是 Milvus 性能优化的重要手段之一。

- **Alias（别名）**

  用于安全更新数据，代码解耦，平滑切换升级

2）索引

Milvus 支持对标量字段和向量字段分别创建索引。
1、**标量字段索引**：主要用于加速元数据过滤，常用的有 `INVERTED`、`BITMAP` 等
2、**向量字段索引**：这是 Milvus 的核心。选择合适的向量索引是在查询性能、召回率和内存占用之间做出权衡

- 主要向量索引算法
  - FLAT（精确查找）
  - IVF（倒排文件索引）
  - HNSW（基于图的索引）
  - DiskANN（基于磁盘的索引）
- 如何选择索引

场景	推荐索引	备注
数据可完全载入内存，追求低延迟	**HNSW**	内存占用较大，但查询性能和召回率都很优秀。
数据可完全载入内存，追求高吞吐	**IVF_FLAT / IVF_SQ8**	性能和资源消耗的平衡之选。
数据量巨大，无法载入内存	**DiskANN**	在 SSD 上性能优异，专为海量数据设计。
追求 100% 准确率，数据量不大	**FLAT**	暴力搜索，确保结果最精确。

3）检索

- 基础向量检索

  这是 Milvus 的核心功能之一，**近似最近邻 (Approximate Nearest Neighbor, ANN) 检索。**
- 增强检索
  - 过滤检索
  - 范围检索
  - 多向量混合检索
  - 分组检索

3.5 索引优化

3、JavaEE源码级流程和原理（基于LangChain4j）

> 以下基于Langchain4j实现RAG知识库

AiChatController.msg 端点获取到前端的消息会建立 SSE（Server-Sent Events 实时数据推送） 双向请求链接

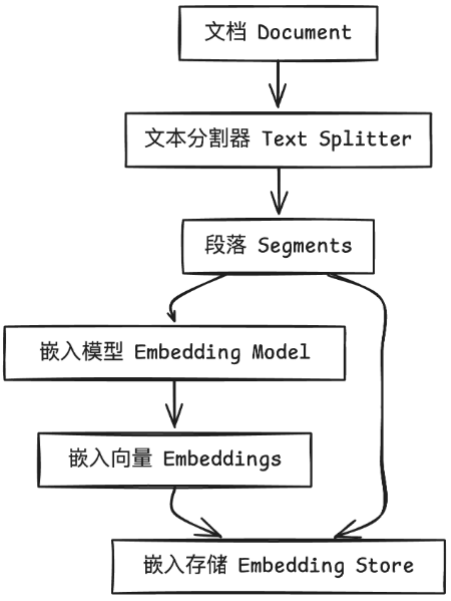
> Server-Sent Events (SSE) 是HTML5引入的一种轻量级的服务器向浏览器客户端单向推送实时数据的技术。在Spring Boot框架中，我们可以很容易地集成并利用SSE来实现实时通信。
>
> 在Spring Boot项目中，无需额外引入特定的依赖，因为Spring Web MVC模块已经内置了对SSE的支持。
>
> SSE主要流程：1、创建SSE端点 2、通过端点发送事件 3、关闭端点连接

AiChatController.msg 是整个聊天流程的入口点，它接收前端传来的消息 key，并建立 Server-Sent Events (SSE) 连接，实现服务器向客户端的实时推送。

RAG主要分为索引阶段、检索阶段、总结阶段

- **索引阶段**

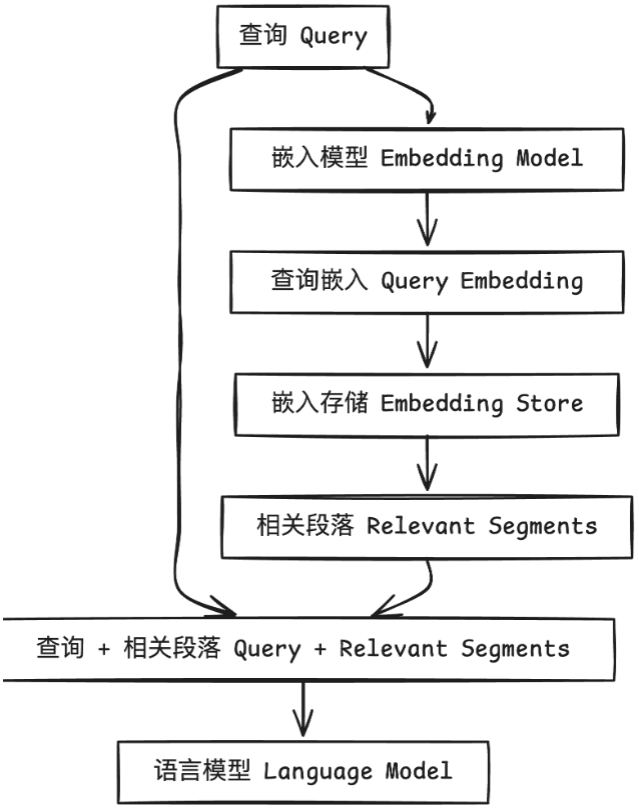
  在索引阶段，文档经过预处理以便在检索阶段进行高效搜索。对于向量搜索，通常包括清理文档、添加额外数据和元数据、将文档分割为小段（分块），将这些段落嵌入为向量，并存储在嵌入存储（向量数据库）中。



  > 索引阶段用到了策略模式和责任链模式，上传文件解析处理器`private final List<UploadFileParseHandler> parseHandlerList`的初始化是通过Spring依赖注入`@RequiredArgsConstructor`注解实现的：
  >
  > 1. Spring 扫描所有 `UploadFileParseHandler` 实现类
  > 2. 根据 `@Order` 或 `Ordered.getOrder()` 排序
  > 3. 通过构造器/字段注入到目标类中
  > 4. 使用时通过 `supports()` 方法选择具体处理器

- **检索阶段**

检索阶段通常是在线进行的，当用户提交问题时，从索引的文档中寻找相关信息。对于向量搜索来说，这通常包括将用户的查询嵌入为向量，并在嵌入存储中执行相似性搜索，找到相关的段落并将它们注入提示中，再发送给 LLM。



0、构建知识库和文档

- **1、配置大模型**

  包含 供应商、模型类型（聊天、推理、向量、排序、图片、视频、解析等）、对应模型标准名称（例text-embedding-v4）、模型别名、apiKey、baseUrl、特殊参数等

- **2、配置向量数据库**

  支持milvus、redis、chroma、qdrant等

- **3、创建知识库**
  - 基本配置：知识库头像、名称、排序、欢迎语等
  - 模型配置：使用的向量库、使用的向量模型、使用的总结嵌入模型（聊天/推理）、重排模型
  - 检索配置：历史上下文会话轮数、匹配条数topK、分片值（500~2000）、向量数据库匹配率（低于这个相似度的片段就不返回）、空查询（查询不到结果时，是否调用大模型查询）、空提示
  - 处理选项：
    - 文档总结：上传文档时通过摘要模型是否提前做总结
    - AI OCR：PDF、图片等文件，自动进行 AI OCR 识别
    - 会话压缩：会话是否提前压缩（历史对话内容是否做摘要存储，节省上下文）
    - 标注数据：是否使用标注数据（比如有人工标注的问答对）
  - 安全配置：是否对外、安全密钥、可见范围、敏感词过滤、敏感词提示、附加信息等
  - 召回测试：召回测试是验证知识库检索效果的功能，开发者可以模拟用户查询并评估系统返回结果。这种测试帮助开发者了解系统的检索能力边界，发现并修复潜在问题，如漏检、误检或相关度不佳的情况，是优化 RAG 系统不可或缺的工具

    - 召回数量：一次检索返回多少条结果（和 topK 相关）
    - 是否全文检索：检索方式是否只用向量，还是加上关键词搜索
    - 召回问题：准备的标准问题集，用来评估系统的召回效果
- **4、知识库关联文档**

  支持文档格式：API、文本录入、文件、图片（视频TODO）、网页等

  分片算法：智能分片、段落分片、句子分片、字符分片

  > 分片值：指每个分片里面字符总数量
  > 重叠值：指分片之间的重叠大小，避免分割丢失上下文

  上传完关联文档后，会自动对该知识库文档进行切片，异步保存文档，核心参考代码如下：

```
/**
 * 处理文档
 * <p>
 * 执行完整的文档处理流程： 1. 保存文档到数据库 2. 创建文档切片 3. 生成文档摘要 4. 向量化切片数据
 * @param aiDocumentDTO AI文档数据传输对象
 */
@Override
public void handle(AiDocumentDTO aiDocumentDTO) {
    log.info("======= AI 正在处理文档: {} =======", aiDocumentDTO.getName());
    // 数据库保存文档
    AiDocumentEntity documentEntity = saveDocument(aiDocumentDTO);

    // 创建文档切片
    try {
        saveSlice(documentEntity, aiDocumentDTO);
    }
    catch (Exception e) {
        // 切片失败，更新状态和失败原因
        documentEntity.setSliceStatus(SliceStatusEnums.FAILED.getStatus());
        documentEntity.setSliceFailReason(e.getMessage());

 SpringUtil.getBean(AiDocumentService.class).updateById(documentEntity);
        return;
    }

    // 生成文档摘要
    try {
        summaryDocument(documentEntity, aiDocumentDTO);
    }
    catch (Exception e) {
        // 摘要生成失败，更新状态和失败原因
        documentEntity.setSummaryStatus(SliceStatusEnums.FAILED.getStatus());
        documentEntity.setSummaryFailReason(e.getMessage());

 SpringUtil.getBean(AiDocumentService.class).updateById(documentEntity);
        return;
    }

    // 向量化切片
    // 使用策略模式处理不同类型向量存储的向量化逻辑 支持带摘要的向量化，提高检索效果 处理失败的切片会标记为失败状态等待重试
```

```
        embedSlice(documentEntity);
        log.info("======= AI 文档：{} 处理结束 =======", aiDocumentDTO.getName());
    }
```

> **用到的设计模式：**
>
> - **模板方法模式（Template Method）**：
>
>   `AbstractFileSourceTypeHandler` 定义了一个完整的处理流程 `handle()`，子类（比如 `UploadSourceTypeHandler`）只需要实现其中的部分步骤（如 `saveDocument()`、`saveSlice()`），而不用重复编写整个流程
>
> - **策略模式（Strategy）**
>
>   `FileSourceTypeHandler` 是接口，不同的文件来源（上传、网页、API...）都可以有不同的实现类。在运行时，可以选择不同的实现类（即不同策略）来处理文档。
>
> - **工厂模式**
>
>   `SpringUtil.getBean(AiDocumentService.class)` 通过 Spring 容器获取依赖，而不是自己 new
>
> - **责任链模式**
>
>   `handle()` 方法像一条流水线：保存 → 切片 → 摘要 → 向量化。一旦失败，就更新状态并中断流程
>
> **用到的软件工程方法：**
>
> - 单一职责原则（SRP）
>
>   每个方法职责清晰，saveDocument()、saveSlice()、summaryDocument()、embedSlice()
>
> - 开闭原则（OCP）
>
>   `AbstractFileSourceTypeHandler` 不需要修改就能支持新类型文件，只要新增一个子类实现即可。👉 系统对扩展开放、对修改关闭。
>
> - 异常处理和回滚
> - 日志追踪

> **注意：**
>
> `@RequiredArgsConstructor` + `private final IOC容器类 xxx`：Spring构造函数注入简写形式，自动将该类实例Bean注册到Spring容器中

0.1 保存文档切片

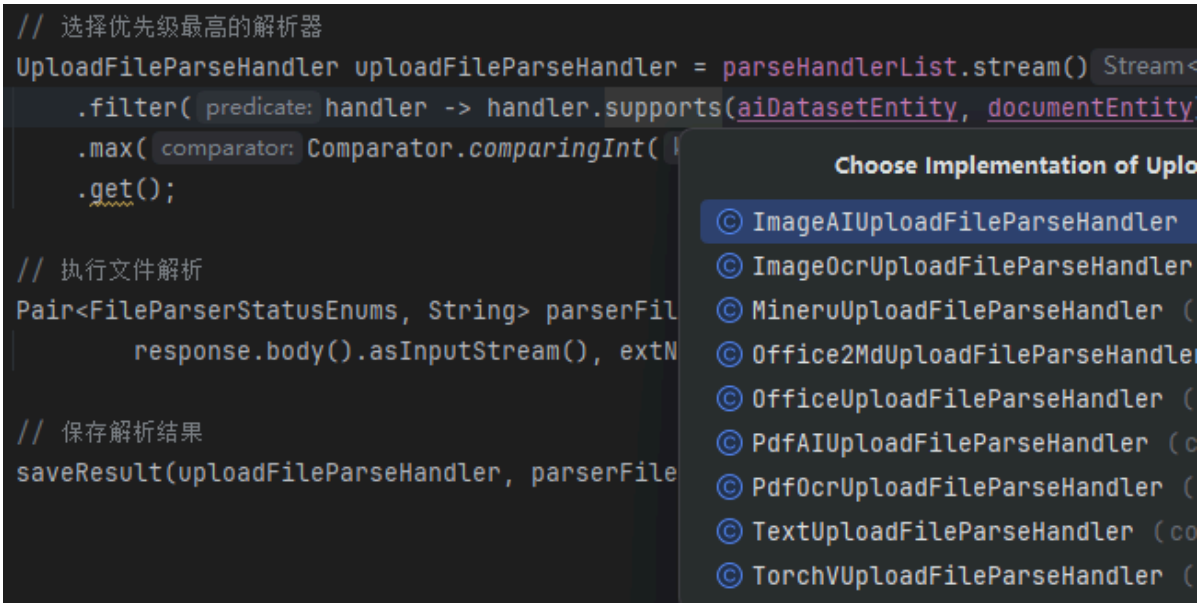
```
/**
 * 保存文档切片
 * <p>
 * 下载文件内容，选择合适的解析器进行解析 将解析结果分割成切片并保存
 * @param documentEntity 文档实体
 * @param documentDTO 文档数据传输对象
 */
@Override
@SneakyThrows
public void saveSlice(AiDocumentEntity documentEntity, AiDocumentDTO documentDTO)
{
    // 从远程服务下载文件
```

```
    Response response = remoteFileService.getFile(documentEntity.getFileUrl());
    String extName = FileUtil.extName(documentEntity.getFileUrl());
    // 查询知识库配置
    AiDatasetEntity aiDatasetEntity =
aiDatasetMapper.selectById(documentEntity.getDatasetId());

    // 选择优先级最高的解析器
    UploadFileParseHandler uploadFileParseHandler = parseHandlerList.stream()
        .filter(handler -> handler.supports(aiDatasetEntity, documentEntity))
        .max(Comparator.comparingInt(Ordered::getOrder))
        .get();

    // 执行文件解析
    Pair<FileParserStatusEnums, String> parserFile =
uploadFileParseHandler.file2String(documentEntity,
            response.body().asInputStream(), extName);

    // 保存解析结果
    saveResult(uploadFileParseHandler, parserFile, documentEntity,
aiDatasetEntity, documentDTO);
}
```



> 核心流程：
>
> 1. 从远程文件服务获取实际的文件流，取出文件扩展名（比如 `pdf`、`docx`、`jpg`），后续决定用哪个解析器
> 2. `handler.supports(...)`：策略模式调用每个解析器的 `supports` 方法，判断是否能处理这个文档，如果多个解析器都能处理，就选**优先级最高**的那个
> 3. 调用具体解析器，把文件流转为文本字符串，保存解析结果，分片切分存入数据库，用于后续向量化

0.2 MarkItDown 文档解析增强

```
/**
 * 将Office文档转换为Markdown格式文本
 * <p>
 * 该方法执行以下步骤： 1. 将输入流转换为ByteArrayMultipartFile对象 2. 查询默认的视觉模型
(如果有)，用于处理文档中的图片 3.
```

```
     * 如果找到视觉模型，设置模型参数和OCR提示词 4. 调用Markitdown服务进行文档转换
     * @param documentEntity 文档实体，包含文档信息
     * @param inputStream 文件输入流
     * @param extName 文件扩展名
     * @return 包含解析状态和Markdown文本的Pair对象，成功时返回转换后的文本，失败时返回错误信息
     */
    @Override
    public Pair<FileParserStatusEnums, String> file2String(AiDocumentEntity
    documentEntity, InputStream inputStream,
            String extName) {
        try {
            // 把文件流包装成 MultipartFile，MarkItDown 服务通常需要上传文件
            ByteArrayMultipartFile file = new
    ByteArrayMultipartFile(documentEntity.getName(), documentEntity.getName(),
                    null, inputStream.readAllBytes());
            // 构建Markitdown请求DTO
            AiMarkitdownDTO request = new AiMarkitdownDTO();

            // 查询默认的视觉模型
            AiModelEntity aiModelEntity = aiModelMapper.selectOne(Wrappers.
    <AiModelEntity>lambdaQuery()
                .eq(AiModelEntity::getModelType, ModelTypeEnums.VISION.getType())
                .eq(AiModelEntity::getDefaultModel, YesNoEnum.YES.getCode()), false);
            // 设置视觉模型参数以及指定的ocr-image模版提示词
            if (Objects.nonNull(aiModelEntity)) {
                request.setModel(aiModelEntity.getModelName());
                request.setApi_key(aiModelEntity.getApiKey());
                request.setBase_url(aiModelEntity.getBaseUrl());
                request.setPrompt(PromptBuilder.render("ocr-image.st"));
            }
            // 调用Markitdown服务执行解析
            MarkitdownResponseDTO markitdownResponseDTO =
    markitdownAssistantService.upload(file, request);
            // 获取<文件解析的结果-markdown文本>键值对
            return Pair.of(FileParserStatusEnums.PARSE_SUCCESS,
    markitdownResponseDTO.getText());
        }
        catch (Exception e) {
            log.error("文件 {} 解析失败", documentEntity.getName(), e);
            return Pair.of(FileParserStatusEnums.PARSE_FAIL, e.getMessage());
        }
    }
```

> **适配器模式**：输入流 → `MultipartFile` 的转换，相当于做了一个 **接口适配**
>
> **Pair<K, V>**：hutool中简单的键值对类，用来存放两个相关联的对象，`Pair.of(K key, V value)` 是一个 **静态工厂方法**，用来快速创建一个 `Pair` 对象

0.3 MinerU 文档解析增强

```
/**
 * 将文件上传到Mineru服务并开始异步解析
 * <p>
 * 该方法执行以下步骤： 1. 创建文件信息并向Mineru服务申请上传链接 2. 使用获取到的上传链接将文
件内容上传到Mineru服务 3.
```

```
     * 返回批次ID，后续可通过该ID查询解析结果
     *
     * 注意：该方法返回AI_PARSING状态，表示文件正在异步解析中
     * @param documentEntity 文档实体对象，包含文档信息
     * @param inputStream 文件输入流
     * @param extName 文件扩展名
     * @return 包含处理状态和批次ID/错误信息的Pair对象
     * @throws Exception 处理过程中可能抛出异常
     */
    @Override
    public Pair<FileParserStatusEnums, String> file2String(AiDocumentEntity
    documentEntity, InputStream inputStream,
            String extName) {
        try {
            // 创建文件信息并申请上传链接
            MineruBatchUploadDTO request = getMineruBatchUploadDTO(documentEntity);
            MineruBatchResponseDTO uploadUrlsResult = applyUploadUrl(request);
            // batchId查询解析结果的关键凭证
            String batchId = uploadUrlsResult.getData().getBatchId();
            String uploadUrl = uploadUrlsResult.getData().getFileUrls().get(0);

            log.info("Mineru 申请上传链接成功，batchId: {}, uploadUrl: {}", batchId,
    uploadUrl);

            // 上传文件
            uploadFileToUrl(inputStream, uploadUrl);
            // 注意这里文件正在解析中，调用方需要用 batchId 定时去 MinerU 查询结果 (polling)
            log.info("Mineru 文件上传完成，batchId: {}", batchId);
            return Pair.of(FileParserStatusEnums.AI_PARSING, batchId);

        }
        catch (Exception e) {
            log.error("Mineru 服务异常，文件: {}，错误: {}", documentEntity.getName(),
    e.getMessage());
            return Pair.of(FileParserStatusEnums.PARSE_FAIL, e.getMessage());
        }
    }
```

> 需要通过配置类SliceTrainTaskConfiguration来定时任务来负责定期扫描和处理文档，包括生成摘要、文档切片、OCR识别和Mineru解析。
>
> 注解@Configuration(proxyBeanMethods = false)：
>
> - `proxyBeanMethods = true` → 保证 `@Bean` 方法之间的依赖是同一个 Spring 管理的实例（安全，慢一点）。
> - `proxyBeanMethods = false` → 不做代理，方法调用就是 new（快，但可能导致多例）。

MarkItDown对比MinerU：

特性	MarkItDown	MinerU
定位	文档转 Markdown	文档结构化解析
输出	Markdown 为主	Markdown + JSON（结构化）

特性	MarkItDown	MinerU
场景	普通 Office、PDF → LLM 预处理	复杂 PDF、科研论文、表格/公式精细化还原
易用性	轻量、依赖少	较重，需要 OCR/深度模型
精度	普通格式 OK，复杂文档弱	表格/公式/版面识别强

1、创建对外连接

创建聊天对外连接，返回唯一的聊天message key

```
/**
 * 创建对外连接
 *
 * @param chatMessageDTO 消息内容体
 * @return R<uuid>
 */
@Inner(value = false)
@PostMapping("/create")
public R<String> createPublicConnection(@Valid @RequestBody ChatMessageDTO
chatMessageDTO) {
    return chatService.saveConnectionParams(chatMessageDTO);
}
```

2、发起聊天 后端入口

实际创建sse链接调用大模型的入口 依赖于上文的createConnection， 所以即使此接口设置暴露，没有message key也无法调用。

> `@GetMapping` 配合 `produces = MediaType.TEXT_EVENT_STREAM_VALUE` 用于实现 **Server-Sent Events (SSE)** 实时数据推送

```
/**
 * 获取SSE消息流 实际调用大模型并通过SSE方式返回消息流 需要先调用createPublicConnection接口
获取消息键值
 * @param key 消息键值，通过createPublicConnection接口获取
 * @return AI响应消息流
 */
@Inner(value = false)
@GetMapping(value = "/msg/list", produces = MediaType.TEXT_EVENT_STREAM_VALUE)
public Flux<AiMessageResultDTO> msg(@RequestParam Long key) {
    try {
        return chatService.chatList(key);
    }
    catch (Exception e) {
        log.error("chat error", e);
        return Flux.just(new
AiMessageResultDTO(e.getMessage())).concatWithValues(new
AiMessageResultDTO(END_MSG));
    }
}
```

> Mono和Flux核心概念对比

特性	Mono	Flux
数据量	0或1个元素	0~N个元素（无限流可能）
典型场景	单次返回（如HTTP GET请求）	流式返回（如SSE、实时数据流）
类比传统对象	`Optional<T>` 或 `Future<T>`	`List<T>` 或 `Stream<T>`
空值处理	`Mono.empty()` 表示空值	`Flux.empty()` 表示空流
选择时机	1、查询数据库单条记录 2、保存实体到数据库（返回保存结果） 3、HTTP请求的单个响应	1、返回实时数据流 2、批量查询数据库结果 3、Server-Sent Events (SSE)

```
/**
 * 处理聊天请求
 * <p>
 * 根据记录ID获取聊天信息并处理 支持规则引擎进行风险控制 返回响应流式结果
 * @param key 聊天记录ID
 * @return AI响应消息的响应流
 */
@Override
public Flux<AiMessageResultDTO> chatList(Long key) {
    AiChatRecordEntity recordEntity = recordMapper.selectById(key);
    if (Objects.isNull(recordEntity)) {
        return Flux.just(new AiMessageResultDTO("链接已失效"), new
AiMessageResultDTO(END_MSG));
    }

    ChatMessageDTO chatMessageDTO = new ChatMessageDTO();
    chatMessageDTO.setMessageKey(key);
    chatMessageDTO.setModelName(recordEntity.getModelName());
    chatMessageDTO.setContent(recordEntity.getQuestionText());
    chatMessageDTO.setConversationId(recordEntity.getConversationId());
    chatMessageDTO.setDatasetId(recordEntity.getDatasetId());

 chatMessageDTO.setWebsearch(YesNoEnum.YES.getCode().equals(recordEntity.getWebse
archFlag()));

 chatMessageDTO.setInner(YesNoEnum.YES.getCode().equals(recordEntity.getInnerFlag
()));
    if (Objects.nonNull(recordEntity.getExtDetails())) {
        ChatMessageDTO.ExtDetails extDetails =
JSONUtil.toBean(recordEntity.getExtDetails(),
                ChatMessageDTO.ExtDetails.class);
        chatMessageDTO.setExtDetails(extDetails);
    }

    // 如果开启了规则引擎，先进行风险检测
    if (flowExecutorOptional.isPresent()) {
        Flux<AiMessageResultDTO> aiMessageResultDTO = flowRisk(chatMessageDTO);
        if (aiMessageResultDTO != null)
```

```
            return aiMessageResultDTO.concatWithValues(new
AiMessageResultDTO(END_MSG));
    }

    // 根据聊天类型选择对应的处理规则
    ChatTypeEnums chatTypeEnums =
ChatTypeEnums.fromCode(chatMessageDTO.getDatasetId());
    ChatMessageContextHolder.set(chatMessageDTO);
    return
chatRuleMap.get(chatTypeEnums.getType()).process(chatMessageDTO).map(aiMessageRes
ultDTO -> {
        aiMessageResultDTO.setMessageKey(key.toString());
        return aiMessageResultDTO;
    }).concatWithValues(new AiMessageResultDTO(END_MSG));
}
```

- **2.1 首先根据key查找消息聊天记录，第一次发起聊天时会默认创建消息聊天记录**
- **2.2 基于线程局部变量（ThreadLocal）的工具类，用于在多线程环境中传递和存储聊天消息的上下文信息**，使用TransmittableThreadLocal实现跨线程传递聊天消息上下文， 支持在异步任务和线程池场景下保持消息上下文的正确传递
- **2.3 执行风控逻辑flowRisk：**

  ```
  // 如果开启了规则引擎
  if (flowExecutorOptional.isPresent()) {
      Flux<AiMessageResultDTO> aiMessageResultDTO = flowRisk(chatMessageDTO);
      if (aiMessageResultDTO != null)
          return aiMessageResultDTO;
  }
  ```

  风控逻辑主要包括：

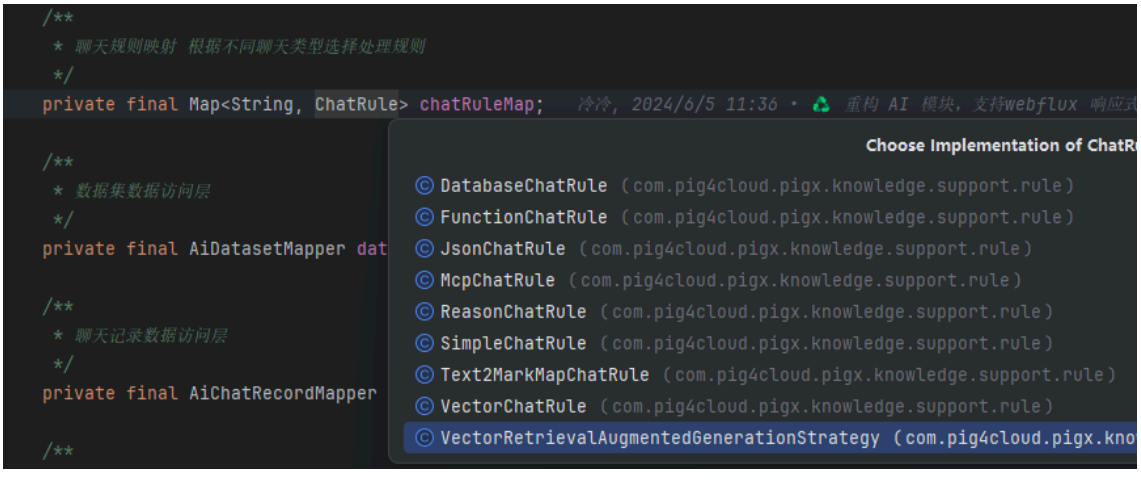
  - 敏感词检测：检测用户输入是否包含敏感词
  - 限流控制：根据调用方式不同采取不同的限流策略
    - 内部调用：按用户名 + 总量控制
    - 外部调用：按 IP+ 总量控制
- **2.4 自动装配聊天规则映射（根据请求类型匹配处理规则）：**
- 基于`@Autowired`或构造函数需要`Map<String, ChatRule>`时，会自动注入private final Map<String, ChatRule> chatRuleMap;　　Spring会自动将所有实现`ChatRule`接口的Bean收集为Map，键为Bean名称，值为实例。

  ```
  ChatTypeEnums chatTypeEnums =
  ChatTypeEnums.fromCode(chatMessageDTO.getDatasetId());
  ChatMessageContextHolder.set(chatMessageDTO);
  return chatRuleMap.get(chatTypeEnums.getType()).process(chatMessageDTO);
  ```

  系统根据 datasetId 确定请求消息类型，主要包括：

Chat Type	Bean Name	Code	Name	Description
FUNCTION_CHAT	functionChat	-1L	功能聊天	支持函数调用和工具使用的聊天模式，可以执行特定功能操作
SIMPLE_CHAT	simpleChat	0L	简单聊天	基础的问答聊天模式，不涉及知识库和函数调用
DATABASE_CHAT	databaseChat	-2L	数据库聊天	支持自然语言转SQL，可以查询和分析数据库数据
IMAGE_CHAT	text2ImageChat	-3L	生成图片	文本转图片功能，根据描述生成对应的图片
MARKMAP_CHAT	text2MarkMapChat	-4L	生成脑图	将文本内容转换为思维导图（MarkMap格式）
FLOW_CHAT	functionChat	-5L	流程编排	支持复杂的AI工作流编排，可以组合多个AI能力
JSON_CHAT	jsonChat	-6L	JSON 聊天	强制输出JSON格式的聊天模式，适合结构化数据交互
REASON_CHAT	reasonChat	-7L	推理聊天	增强推理能力的聊天模式，适合复杂逻辑分析和推理任务
VECTOR_CHAT	vectorChat	1L	知识库聊天	基于向量数据库的知识问答，可以检索相关文档回答问题
MCP_CHAT	mcpChat	-8L	MCP聊天模式	支持模型上下文协议（Model Context Protocol）的聊天模式

- **2.5 多个请求类型的实现类通过策略模式来实现同一接口，运行时动态选择**



```
public interface ChatRule {
    /**
     * 处理聊天信息
     *
     * @param chatMessageDTO 聊天上文
     * @return flux stream
     */
    default Flux<AiMessageResultDTO> process(ChatMessageDTO chatMessageDTO) {
        return Flux.empty();
    }
}
```

3、知识库聊天处理流程（VectorChatRule）

```
/**
 * 处理基于知识库的聊天请求
 * <p>
 * 直接使用RAG策略处理，根据数据集配置选择合适的策略
 * </p>
 * @param chatMessageDTO 聊天消息DTO，必须包含datasetId
 * @return 基于知识库检索的AI响应流
 */
@Override
public Flux<AiMessageResultDTO> process(ChatMessageDTO chatMessageDTO) {
    // 获取数据集配置和向量化模型
    AiDatasetEntity dataset =
aiDatasetService.getById(chatMessageDTO.getDatasetId());
    DimensionAwareEmbeddingModel embeddingModel =
modelProvider.getEmbeddingModel(dataset.getEmbeddingModel());
    Embedding queryEmbedding =
embeddingModel.embed(chatMessageDTO.getContent()).content();

    // 如果启用了标准化数据，先尝试标准问答匹配
    if (YesNoEnum.YES.getCode().equals(dataset.getStandardFlag())) {
        Flux<AiMessageResultDTO> standardResult = standardQuestionAnswerStrategy
            .processChat(queryEmbedding, dataset, chatMessageDTO)
            .cache();

        // 如果标准问答没有结果，降级到向量检索
        return standardResult.switchIfEmpty(

 vectorRetrievalAugmentedGenerationStrategy.processChat(queryEmbedding, dataset,
chatMessageDTO));
    }

    // 直接使用向量检索策略
    return vectorRetrievalAugmentedGenerationStrategy.processChat(queryEmbedding,
dataset, chatMessageDTO);
}
```

> 核心流程：
>
> 1. 向量化查询：把问题转成 embedding。
> 2. 优先走标准问答（精确匹配，类似 FAQ）。

> 3. 没命中时降级到向量检索 RAG（模糊召回 + LLM）。
> 4. 响应流 Flux：支持流式输出。
> 5. 用策略模式：`standardQuestionAnswerStrategy` 和 `vectorRetrievalAugmentedGenerationStrategy` 都是可插拔的策略实现。

3.1 向量检索RAG处理

```
/**
 * 执行端到端RAG处理
 * <p>
 * 根据用户查询和数据集配置，执行完整的RAG处理流程： 检索 -> 重排序 -> 生成答案
 * </p>
 * @param queryEmbedding 用户查询的向量表示
 * @param dataset 数据集配置信息
 * @param chatMessageDTO 聊天消息DTO，包含用户问题和上下文
 * @return 处理结果流，包含生成的回答和相关信息
 */
default Flux<AiMessageResultDTO> processChat(Embedding queryEmbedding,
AiDatasetEntity dataset,
        ChatMessageDTO chatMessageDTO) {
    throw new UnsupportedOperationException("该策略不支持端到端RAG处理");
}

@Override
public Flux<AiMessageResultDTO> processChat(Embedding queryEmbedding,
AiDatasetEntity dataset,
        ChatMessageDTO chatMessageDTO) {
    log.debug("使用向量检索增强生成策略处理查询: {}", chatMessageDTO.getContent());

    // 1. 执行向量检索
    List<EmbeddingMatch<TextSegment>> vectorMatches =
performVectorSearch(queryEmbedding, dataset);

    // 2. 执行全文检索（如果支持）
    List<EmbeddingMatch<TextSegment>> fullTextMatches =
performFullTextSearch(dataset, chatMessageDTO);

    // 3. 合并检索结果
    List<EmbeddingMatch<TextSegment>> allMatches =
RagHelper.mergeSearchResults(vectorMatches, fullTextMatches);

    // 4. 检查是否有匹配结果
    if (RagHelper.isEmpty(allMatches)) {
        log.debug("未找到匹配的文档片段");
        return RagHelper.handleEmptyResult(dataset, chatMessageDTO);
    }

    // 5. 重排序搜索结果
    List<Content> rerankedContent = RagHelper.rerankSearchResults(dataset,
chatMessageDTO.getContent(), allMatches);

    // 6. 更新命中计数
    updateHitCount(allMatches);
```

```
    // 7. 生成答案并附加参考资料
    return generateAnswerWithReferences(dataset, chatMessageDTO, rerankedContent,
allMatches);
}
```

> 核心流程：用户提问 → 向量检索 →（可选）全文检索 → 合并 → 检查空结果 → 重排序 → 更新统计 → 生成最终答案（带参考）

3.2 大模型总结生成答案

```
/**
 * 生成答案并附加参考资料
 */
private Flux<AiMessageResultDTO> generateAnswerWithReferences(AiDatasetEntity
dataset,
        ChatMessageDTO chatMessageDTO, List<Content> rerankedContent,
        List<EmbeddingMatch<TextSegment>> embeddingMatches) {
    // 使用模型根据检索内容生成回答
    List<String> contentTexts =
rerankedContent.stream().map(Content::textSegment).map(TextSegment::text).toList(
);

    Flux<AiMessageResultDTO> answerFlux = RagHelper.summaryResult(dataset,
chatMessageDTO, contentTexts).cache();

    // 创建参考资料链接
    AiMessageResultDTO referenceLinks = new AiMessageResultDTO();
    referenceLinks.setMessage("");
    List<AiMessageResultDTO.ExtLink> extLinks =
RagHelper.buildReferenceLinks(embeddingMatches, aiDocumentService);
    referenceLinks.setExtLinks(extLinks);

    // 在回答后添加参考资料
    return answerFlux.concatWithValues(referenceLinks);
}
```

> **参数含义**
>
> - `dataset` → 当前知识库配置（比如是否需要引用来源）
> - `chatMessageDTO` → 用户输入的原始问题
> - `rerankedContent` → 已经过重排的高相关文档片段
> - `embeddingMatches` → 原始检索命中的文档片段（含文档 ID、相似度等信息）

4、Langchain4j实现RAG核心类整理

用户问题 → 文本嵌入 → 向量检索 → 构造上下文 Prompt → 调用 LLM → 输出答案

模块	核心类	功能说明
1. 文档加载 (Loaders)	`DocumentLoader`, `FileSystemDocumentLoader`, `PdfDocumentLoader`, `WebDocumentLoader`	加载原始文档（PDF、TXT、HTML等）

模块	核心类	功能说明
2. 文本分块 (Splitters)	`DocumentSplitter`, `TokenTextSplitter`, `ParagraphSplitter`	将文档拆成适合嵌入的文本块（chunks）
3. 向量化 (Embeddings)	`EmbeddingModel`, `OpenAiEmbeddingModel`, `BgeSmallZhEmbeddingModel`	将文本块转成向量
4. 向量存储 (Vector Store)	`VectorStore`, `InMemoryEmbeddingStore`, `PgVectorStore`, `MilvusEmbeddingStore`, `QdrantEmbeddingStore`	存储和检索文本向量
5. 检索器 (Retriever)	`Retriever`, `EmbeddingStoreRetriever`	根据 query 嵌入查询最相似的文档片段
6. LLM 模型 (LLM)	`ChatLanguageModel`, `OpenAiChatModel`, `OllamaChatModel`, `BaichuanChatModel`	调用大模型生成回答
7. Prompt 模板 (Prompt)	`PromptTemplate`, `ChatPromptTemplate`	组织上下文和用户输入
8. RAG 逻辑封装	`ConversationalRetrievalChain`, `RetrievalAugmentedGenerator`	将 “检索 + 生成” 组合起来的核心链

典型代码流程

```
// 1 加载文档
DocumentLoader loader = new FileSystemDocumentLoader("docs/");
List<Document> docs = loader.load();

// 2 文本分块
DocumentSplitter splitter = new TokenTextSplitter(500, 100);
List<Document> chunks = splitter.split(docs);

// 3 向量化 (Embedding)
EmbeddingModel embeddingModel = new OpenAiEmbeddingModel("gpt-3.5-turbo");
List<Embedding> embeddings = embeddingModel.embed(chunks);

// 4 存储到向量库
EmbeddingStore store = new InMemoryEmbeddingStore();
store.addAll(embeddings, chunks);

// 5 检索器
Retriever retriever = new EmbeddingStoreRetriever(store, embeddingModel);

// 6 构建LLM
ChatLanguageModel llm = OpenAiChatModel.withApiKey("your-key");

// 7 构建RAG链
ConversationalRetrievalChain chain = ConversationalRetrievalChain.builder()
    .retriever(retriever)
    .chatLanguageModel(llm)
```

```
    .build();

// 8 执行问答
String question = "文档中提到的数据加密方法有哪些？";
String answer = chain.execute(question);
System.out.println(answer);
```

4、RAG相关面试题

1、你有多个知识库，做 RAG 的时候，怎么保证查询效率和准确性兼容，并尽可能减少幻觉?

如果有多个知识库，那么首先需要统一嵌入模型将语义粒度标准化。

然后先利用轻量 LLM 作为知识路由，动态选择子知识库。

接着再分阶段召回机制，第一阶段段粗召回，第二阶段 rerank。

最后再结合生成前验证 + 输出后校对机制，多层兜底，提升整体鲁棒性。

扩展知识

1）统一嵌入模型 & 语义粒度标准化

多个知识库必须用同一个向量模型（比如 `bge-base`、`text-embedding-3-large`）来生成嵌入，否则相似度计算会失真。

同时，统一切块策略（如按段/按句，控制 chunk overlap），避免有库查不到、查到了但语义漂移。

2）多知识库路由策略

根据用户 query 的意图，先用轻量 LLM 判断 query 属于哪个知识域，**动态选择子知识库**，避免乱搜一通。

常见做法包括：关键词分类器、基于 embedding 的领域聚类、甚至训练个小模型做意图路由。

3）分阶段召回机制

- 第一阶段粗召回：快速从向量库中拉出 N 条可能命中高的 chunk（比如 top 20）
- 第二阶段精 rerank：用 CrossEncoder、ColBERT 或 mini LLM 对这 N 条进行重排序

这三个点就是回答面试题中的保证查询效率和准确性兼容。

关于如何减少幻觉

主要有以下三点：

1）来源可信度过滤

在生成前，设定数据可信标签（如高置信度标红，低置信度干脆不入 prompt）。或者干脆只允许模型回答，**与知识库无关的问题时拒答。**

2）生成前约束提示词

在提示词中明确声明：**“请仅基于提供信息回答，不允许编造或联想”。**可加入输出格式限制，例如必须列出参考信息出处（chunk ID 或原始文档）

3）生成后结构化校验

通过规则或轻量模型验证答案是否落在知识库真实存在的实体上。

2、如何优化 RAG 的检索效果?

这个问题必然要从 RAG 的流程入手，看我们能干预的步骤有哪些，总结来说 RAG 检索效果的优化主要可以分为以下几个方面：

1）文档预处理与切片优化

- 高质量文档：原始文档内容不行一切都白搭，所以要保证知识库中的原始文档内容准确、结构清晰、格式规范，尽量减少噪音（水印、不相关图片）。
- 智能切片：采用合适的文档切片策略。切片过小可能导致语义不完整，过大则可能引入过多无关信息。可以考虑基于语义边界或智能分块算法，避免固定长度切分导致语义断裂。Spring AI 的 `TokenTextSplitter` 就提供了基于 Token 的文本分割，也考虑了语义边界。
- 元数据标注：为文档切片添加合适的元数据（来源、日期、类别、标签），后续进行更精确的过滤和检索都能使用到。

2）查询增强

- 查询重写：使用大模型把用户的原始查询改写的更清晰、更详细、更规范，提高后续检索的准确性。
- 查询扩展：将用户的单个查询扩展为多个语义相近的查询，然后合并检索结果，提高召回率。例如，将“鱼皮是谁”扩展为“程序员鱼皮介绍”、“鱼皮的技能”等。
- 查询翻译：如果知识库和用户查询的语言不一致，可以把查询翻译成知识库的语言。

3）检索器配置与策略：

- 相似度阈值：合理设置检索时返回文档的相似度阈值和数量。阈值过高可能漏掉相关文档，过低则可能引入噪音；返回文档过多会增加后续处理成本和模型幻觉风险，这些参数都需要根据具体场景进行调试。
- 元数据过滤：用预先标注的元数据对检索范围进行精确过滤，只在特定子集文档中搜索，提升检索的相关性和效率。
- 混合检索策略：结合不同检索方法的优势，像关键词检索、向量检索、知识图谱检索等。比如先用向量检索召回语义相关的文档，再用关键词过滤精确匹配。

4）嵌入模型的选择与优化：选择高质量的嵌入模型对文本进行向量化，这会直接影响语义相似度计算的准确性。

5）重排：在召回多个文档后，可以用更复杂的排序模型对初步检索到的结果进行重新排序，把更相关、质量更高的文档排在前面。

3、什么是 Spring AI 提出的模块化 RAG 架构？预检索、检索和后检索阶段各自负责什么?

Spring AI 提出的模块化 RAG 架构是将整个检索增强生成过程分解为 **预检索、检索、检索后** 三个核心阶段，每个阶段包含可配置的组件，以提升大模型响应的准确性和灵活性。

1）预检索阶段 (Pre-Retrieval)：

- 职责：接收用户的原始查询，并对其进行优化和转换，生成更适合后续检索的查询版本。
- 组件：包括各种 `QueryTransformer`，如 `RewriteQueryTransformer`（改写查询使其更清晰）、`TranslationQueryTransformer`（翻译查询）、`CompressionQueryTransformer`（在多轮对话中压缩历史和当前问题）、以及 `MultiQueryExpander`（将单查询扩展为多查询，提高召回）。

2）检索阶段 (Retrieval)：

- 职责：使用预检索阶段优化后的查询，从知识库中搜索并召回最相关的文档片段。
- 组件：核心是 `DocumentRetriever`（如 `VectorStoreDocumentRetriever`），它负责执行相似性搜索并根据元数据过滤结果。如果涉及多源检索，还可能用到 `DocumentJoiner` 来合并结果。

3）检索后阶段 (Post-Retrieval)：

- 职责：对检索到的文档集进行进一步处理和优化，筛选出最适合提供给大模型的上下文，可以解决上下文丢失问题、上下文长度限制，并减少冗余内容。
- 组件：可能包括文档重排序、无关文档移除、文档内容压缩或摘要等。Spring AI 提供了 `DocumentPostProcessor` API 来支持自定义的后处理逻辑，但目前并不成熟。

4、上下文查询增强？它有什么作用？如何基于 Spring AI 实现上下文查询增强来处理无关问题？

上下文查询增强是 RAG 流程中的一个核心环节，指的是把用户的原始查询与从知识库中检索到的相关文档进行结合，形成一个信息更丰富的增强提示，然后将这个增强提示提供给 AI，让模型能基于这些特定知识生成回答。主要作用是为大模型提供必要的、实时的外部知识，这样 AI 的回答就不仅仅依赖于其预训练的通用知识，提高答案的准确性、相关性和时效性。

Spring AI 的 `RetrievalAugmentationAdvisor` 内部使用 `ContextualQueryAugmenter` 来实现上下文查询增强。当处理用户提出的无关问题时，`ContextualQueryAugmenter` 提供了空上下文处理机制。

```
public final class ContextualQueryAugmenter implements QueryAugmenter {

    private static final Logger logger = LoggerFactory.getLogger(ContextualQueryAugmenter.class);

    private static final PromptTemplate DEFAULT_PROMPT_TEMPLATE = new PromptTemplate("""
            Context information is below.

            ---------------------
            {context}
            ---------------------

            Given the context information and no prior knowledge, answer the query.

            Follow these rules:

            1. If the answer is not in the context, just say that you don't know.
            2. Avoid statements like "Based on the context..." or "The provided information...".

            Query: {query}

            Answer:
            """);

    private static final PromptTemplate DEFAULT_EMPTY_CONTEXT_PROMPT_TEMPLATE = new PromptTemplate("""
            The user query is outside your knowledge base.
            Politely inform the user that you can't answer it.
            """);
```


我们可以配置 `ContextualQueryAugmenter` 的 `allowEmptyContext(false)`，并提供一个自定义的 `emptyContextPromptTemplate`。 检索不到相关文档时，系统会使用这个自定义模板来指示大模型如何回应。在我们的项目中，这个自定义模板会引导 AI 礼貌地告知用户 “它只能回答指定知识库相关的问题”，并给出联系客服的方式，优雅地处理了超出知识库范围的提问。

5、什么是查询重写？它有什么作用？如何基于 Spring AI 实现查询重写？

查询重写是 RAG 预检索阶段的优化手段。它利用 AI 大模型对用户原始输入的查询进行改写润色，然后生成一个对后续文档检索更有效、更精确的新查询。

查询重写可以提高检索的准确性和相关性，尤其是当用户查询较为模糊、口语化、不完整，或者和知识库语言风格不一致时，通过重写可以将查询变得更规范、更详细，更容易在向量数据库中匹配到最相关的文档。

Spring AI 主要通过 `QueryTransformer` 接口及其实现类来支持查询重写。比如 `RewriteQueryTransformer` 可以将简单查询改写得更具体、更专业；而 `CompressionQueryTransformer` 则专注于处理多轮对话场景，将对话历史和当前问题压缩成一个独立的、包含完整上下文的新查询。使用时，需要为这些转换器配置一个 `ChatClient` 或 `ChatModel`，然后调用其 `transform` 方法传入原始查询即可得到重写后的查询。

```
42  public class RewriteQueryTransformer implements QueryTransformer {
43
44      private static final Logger logger = LoggerFactory.getLogger(RewriteQueryTransformer.class);
45
46      private static final PromptTemplate DEFAULT_PROMPT_TEMPLATE = new PromptTemplate("""
47              Given a user query, rewrite it to provide better results when querying a {target}.
48              Remove any irrelevant information, and ensure the query is concise and specific.
49
50              Original query:
51              {query}
52
53              Rewritten query:
54              """);
55
```


6、Rerank？具体需要怎么做？

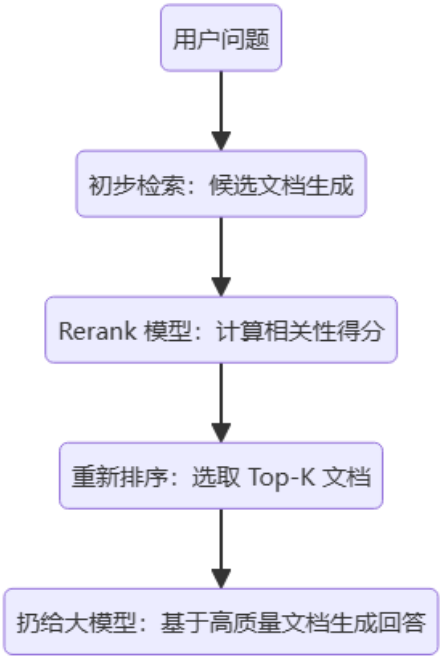
在 RAG 中，Rerank 是一个对初步检索返回的候选文档列表进行再次排序的过程。

因为初步检索需要**快速地在海量的文档中**找出大致相关的文档，其需要考虑效率，所以查找出的文档不会非常准确，这步是**粗排。**

在已经筛选的相关的文档中再进行精筛，找出匹配度更高的文档让其排在前面，选其中的 Top-K 然后扔给大模型，提高答案的准确性，这就是 Rerank，也是**精排。**

Rerank 需要怎么做?

1. **初步检索生成候选文档**：使用速度较快的传统检索方法获得一组候选文档。
2. **根据Rerank模型重新排序**：根据Rerank模型匹配得分对候选文档进行排序，选出最相关的 Top-K 文档。
3. **交给生成模块**：Top-K 候选文档传递给大模型，帮助生成更精准、更富信息量的回答。



为什么需要 Rerank？如果没有 Rerank 会怎么样?

初步检索方法通常为了**速度**牺牲了一些**精度**，可能包含许多噪音文档。

而 Rerank 能够利用更高级的模型深入捕捉查询与文档之间的细微语义关系，从而筛除无关文档，确保生成模块获得高质量上下文信息。

如果没有 Rerank，生成模块可能基于噪音或不相关的文档生成回答，这会影响回答的准确性，甚至还可能引发误解。

Rerank 通俗理解

在仓库里，你想要找"坤你太美"主题的周边玩具。仓库管理员先按照模糊标准，比如只要有"坤"字或者颜色鲜艳的都挑出来，给你抱来一堆。这堆里可能有篮球造型的小摆件、印有各种明星的卡片，还有一些卡通玩偶，很杂乱。

这时，Rerank就像一位更专业的助手，它会把这堆东西重新审视，按照和"坤你太美"主题的贴合度重新排序。比如，印有蔡徐坤经典唱跳动作图案的扇子、模仿他舞台服装的小玩偶会被排在前面，而那些关联不大的就往后排。

排序完成后，不会把所有东西都给你，因为太多了看不过来。助手会把排在前几名、和"坤你太美"主题最贴近的周边拿给你，这样你就能快速找到自己真正想要的，不用在那一大堆杂乱物品里费劲翻找了。 :)

常见的 Rerank 模型

BAAI/bge-reranker-base （开源）

这是咱们北京智源人工智能研究院（BAAI）开发的交叉编码器模型，旨在对检索得到的候选文档进行重排序，以提高信息检索的精度。

Rerank 模型列表如下:

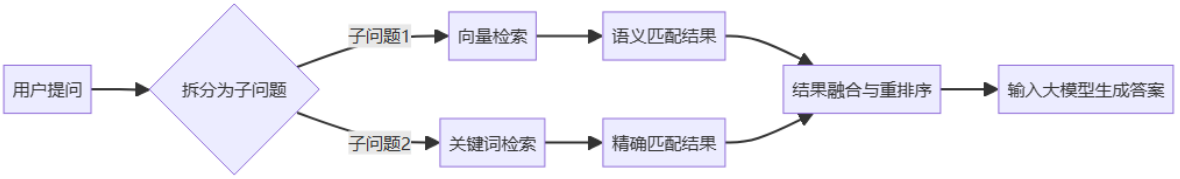
Model	Language	Description
BAAI/bge-reranker-v2-m3	Multilingual	一个轻量级的交叉编码器模型，具有强大的多语言能力，易于部署，具有快速的推理能力。

Model	Language	Description
BAAI/bge-reranker-v2-gemma	Multilingual	一个支持多语言的交叉编码器模型，在英文和多语言能力方面均表现出色。
BAAI/bge-reranker-v2-minicpm-layerwise	Multilingual	一个支持多语言的交叉编码器模型，在英文和中文方面均表现良好，允许自由选择输出层，以便加速推理。
BAAI/bge-reranker-v2.5-gemma2-lightweight	Multilingual	一个支持多语言的跨编码器模型，不仅在英文和中文上表现良好，还允许自由选择输出层、压缩比例和压缩层，从而便于加速推理。
BAAI/bge-reranker-large	Chinese and English	交叉编码器模型，精度比向量模型更高但推理效率较低
BAAI/bge-reranker-base	Chinese and English	交叉编码器模型，精度比向量模型更高但推理效率较低

7、什么是混合检索？在基于大模型的应用开发中，混合检索主要解决什么问题？

混合检索 是指在基于大模型的 RAG（检索增强生成）应用中，结合向量检索和关键词检索等检索技术的互补优势，提升检索结果的全面性和准确性。

它主要为了提升大模型的上下文理解和回答准确性，因为向量检索擅长语义理解（如“猫捕猎老鼠”与“猫追逐老鼠”的关联），但难以精准匹配专有名词（如“iPhone 15”）或缩写（如“RAG”）；关键词检索则反之。



所以在大模型 RAG 应用中，混合检索主要通过**并行**两种检索方式实现：

- **向量检索**：将文本转化为高维向量，计算语义相似度
- **关键词检索**：基于倒排索引、BM25等算法，精确匹配关键词（例如“GPT-4”必须完全命中）。

两种结果通过权重融合或重排序模型（如RRF）合并，最终输出最优答案。

在工程上例如 ElasticSearch 就同时支持关键词检索和向量检索，可以使用工具链例如 LlamaIndex + ElasticSearch 实现混合检索。

扩展知识

混合检索的核心原理

在大模型 rag 应用中，混合检索主要通过**并行**两种检索方式实现：

- **向量检索**：将文本转化为高维向量，计算语义相似度（例如，用户问“猫抓老鼠”，匹配到“猫捕猎啮齿动物”）。
- **关键词检索**：基于倒排索引、BM25等算法，精确匹配关键词（例如“GPT-4”必须完全命中）。

两种结果通过权重融合或重排序模型（如RRF）合并，最终输出最优答案。

所以我们在数据库中需要提前建立好**向量索引**和**关键词索引**。

向量检索与传统关键词搜索对比(来自Dify)

向量检索优势：

- 相近语义理解（如老鼠/捕鼠器/奶酪，谷歌/必应/搜索引擎）
- 多语言理解（跨语言理解，如输入中文匹配英文）
- 多模态理解（支持文本、图像、音视频等的相似匹配）
- 容错性（处理拼写错误、模糊的描述）

虽然向量检索在以上情景中具有明显优势，但有某些情况效果不佳。比如：

- 搜索一个人或物体的名字（例如，伊隆·马斯克，iPhone 15）
- 搜索缩写词或短语（例如，RAG，RLHF）
- 搜索 ID（例如， gpt-3.5-turbo ， titan-xlarge-v1.01 ）

而上面这些的缺点恰恰都是传统关键词搜索的优势所在，传统关键词搜索擅长：

- 精确匹配（如产品名称、姓名、产品编号）
- 少量字符的匹配（通过少量字符进行向量检索时效果非常不好，但很多用户恰恰习惯只输入几个关键词）
- 倾向低频词汇的匹配（低频词汇往往承载了语言中的重要意义，比如“你想跟我去喝咖啡吗？”这句话中的分词，“喝”“咖啡”会比“你”“想”“吗”在句子中承载更重要的含义）

混合检索的补充解释

混合检索也叫多路召回或者融合检索。它并不仅限于向量检索和关键词检索的叠加。

比如在不同数据源上（如文档和数据库）进行同时检索，也叫混合检索。

所以，在检索过程中，同时使用多种检索方式，最后将多种检索结果进行融合，这样的检索都是混合检索。

只不过在大模型的场景下的 RAG 应用中的混合检索常常指的是向量检索+关键词检索。

8、RAG为了优化检索精度，数据清洗和预处理怎么做?

1、首先是数据接入与格式的统一

- **多源数据整合**：文档（PDF/Word）、表格（CSV/Excel）、网页（HTML）、API接口等，用工具（如PyPDF2、BeautifulSoup）提取纯文本。
- **格式标准化**：统一编码（UTF-8）、去除乱码（如“�”替换为空格）、转换大小写（统一小写避免“iPhone”和“iphone”被视为不同词）。

2、开始清洗降噪，过滤无效信息

- 去除重复内容（如文档中的页眉页脚、会议记录的重复发言）。
- 过滤噪声符号（标点、特殊字符、markdown格式），保留核心文本。

> 还要注意**敏感信息处理**！用正则或库（如PII Detection）删除隐私数据（手机号、邮箱），避免泄露。

3、需要进行分块

- 段落、章节标题、标点（句号/分号）切分，避免切断上下文（如“第一段前半+第二段后半”导致语义断裂）。

- 控制长度，比如单个知识块300-500字（约500-800 tokens），适配大模型上下文窗口（如GPT-4默认8k tokens，可放10-15个块）。

具体操作的时候，可以用 LlamaIndex 的`RecursiveCharacterTextSplitter`（优先按标题/段落切分，再按字符数截断）、Hugging Face的`TokenTextSplitter`（按token数切分）。

4、元数据标注

- 用规则或LLM提取结构化信息（如文档中的"时间""地点""关键词标签"等元数据），作为检索时的过滤条件（如用户问"2025年的政策"，优先召回含"2025"元数据的块）。

最后就可以进行索引的构建了，为了用上混合检索，需要进行向量索引和关键词的倒排索引。

拓展知识

```
原始数据（多格式）
|
├─ 格式转换（提取纯文本）
|   ├─ PDF → PyPDF2提取文本
|   ├─ HTML → BeautifulSoup解析正文
|   └─ Excel → Pandas读取表格内容
|
├─ 清洗降噪
|   ├─ 去除重复（相邻重复段落检测）
|   ├─ 过滤噪声（正则表达式去除标点、特殊符号）
|   └─ 敏感信息处理（PII检测库删除隐私数据）
|
├─ 智能分块（核心步骤）
|   ├─ 动态切分：按"标题→段落→句子"优先级切分（如先按"### 第三章"分章节，再按句号分句）
|   ├─ 长度校验：单块不超过模型最大token数的60%（预留空间给用户问题和提示词）
|   └─ 重叠处理：相邻块保留10%重叠（如块1结尾100字作为块2开头，避免切断跨段语义）
|
├─ 语义增强
|   ├─ 元数据标注：
|   |   ├─ 规则提取：用正则匹配"2024年""张三"等实体
|   |   └─ LLM提取：用GPT-4生成"核心关键词""所属部门"等标签
|   └─ 向量化：
|       ├─ 通用模型：Sentence-BERT（快速上线）
|       └─ 领域模型：BGE-M3（多语言）、ColBERTv2（长文本优化）
|
└─ 存储索引
    ├─ 向量库：Milvus（高并发）
    └─ 关键词库：Elasticsearch（BM25算法）+ 元数据索引（支持过滤查询）
```

9、查询扩展?

查询扩展是指对用户原始查询进行优化和补充，通过添加**同义词、相关术语、隐含意图**等信息，让查询更精准、覆盖范围更广，从而提升信息检索的效果。比如用户输入"减肥"，扩展后可能变成"健康减肥方法 饮食运动 避免反弹"。

为什么在RAG中需要查询扩展?

RAG（检索增强生成）的核心是"先检索、后生成"，如果原始查询不够准确或覆盖范围不足，会导致检索到的文档不相关或信息不全，最终生成的回答质量会受影响。查询扩展能解决两个关键问题：

1. **词汇匹配问题**：用户用词和知识库中的术语可能不一致（如"新冠"vs"COVID-19"），扩展后能匹配更多相关内容。

2. **语义补全问题**：用户查询可能简短模糊（如“怎么理财”），扩展后能明确需求（如“新手理财入门 低风险投资 基金股票区别”），让检索更精准。

结合 RAG 简单流程图，查询扩展发生在“用户查询”到“检索文档”之间的预处理阶段：

> 对于<原始查询>，补充其上位概念、下位具体场景及相关关联词（例如“跑步”→上位词“运动”，下位词“慢跑/马拉松”，相关词“跑鞋/运动手环”），用|分隔。

10、自查询?

自查询（Self-Query）是指当用户输入 **模糊表述或隐含需求**时，RAG 系统通过内部处理让模型自动解析用户查询中的隐含条件（如时间、作者、标签等元数据），生成**结构化查询语句**的过程。

比如用户说“2025年鸭鸭的用户报告”，自查询会解析出两个条件：

1. 语义匹配：用户报告
2. 元数据过滤：作者=鸭鸭、时间=2025

为什么在RAG中需要自查询?

因为用户提问往往包含**模糊表述**或**隐含需求**，传统向量检索可能忽略元数据导致结果偏差。自查询通过**解析+过滤**两步走，让检索同时满足语义相关性和元数据条件，解决“检索不准”的核心问题。

> 自查询发生在“用户输入”到“检索文档”之间的 **意图解析阶段**

自查询与查询扩展的区别

维度	自查询	查询扩展
核心目标	精准匹配“内容+元数据”双重条件（如作者、时间、标签）	解决词不匹配问题，通过添加同义词、相关词提升查全率
技术手段	① 用LLM解析用户查询中的隐含元数据 ② 生成结构化查询（如SQL或过滤条件）	① 基于同义词/语义词典扩展关键词 ② 利用LLM生成多样化查询变体
数据依赖	依赖知识库的元数据标注质量（如字段类型、格式）	依赖语料库、用户日志或语义关系库（如WordNet）
典型应用场景	带条件的专业搜索（如“2023年后发布的政府报告”）	问答系统、文档检索（解决用户表述模糊问题）
优点	① 精准过滤噪声数据 ② 支持权限控制（如部门权限过滤）	① 提升语义覆盖范围 ② 减少因用户表述差异导致的漏检
缺点	① 需预定义元数据字段 ② 解析错误可能导致结果偏差	① 可能引入无关词导致噪声 ② 扩展策略需平衡查全率与查准率
典型技术方案	① LLM解析意图生成过滤条件 ② 向量数据库混合检索（如Qdrant、Milvus）	① 基于全局/局部语义分析扩展 ② 伪相关反馈（如Rocchio算法）

简单来说：

- **自查询**：通过“解析+过滤”两步走，例如用户查询“2025年鸭鸭的用户报告”会拆解为语义关键词（用户报告）和元数据（作者=鸭鸭四，时间=2025），再组合为结构化查询条件。它更适合**精确过滤**场景（如法律文档需限定颁布时间、电商商品需价格区间）

- **查询扩展**：通过“生成+合并”策略，例如用户查询“大模型幻觉”可能扩展为“LLM生成内容真实性”“GPT输出偏差”等变体，合并多个检索结果提升覆盖。它更适合**语义泛化**场景（如用户输入“AI生成文本问题”，扩展为“大模型幻觉”“生成内容纠偏”等）

11、提示压缩?

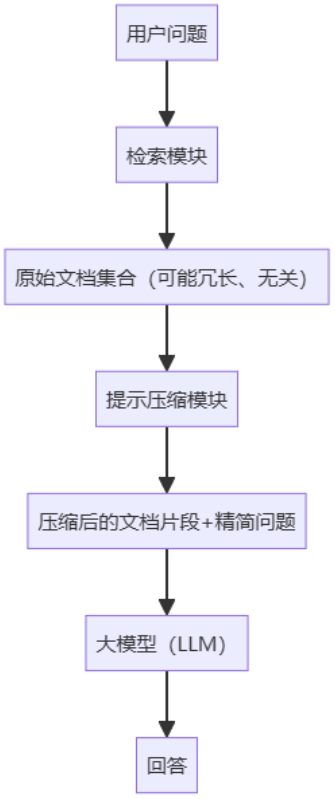
在RAG（检索增强生成）中，**提示压缩**主要指对**检索出的文档内容**进行精简处理，通过提取核心信息、过滤无关文本、压缩冗长内容，使最终输入大模型的“提示”（包含用户问题+文档片段）既完整保留关键信息，又符合模型输入长度限制。

> 例如：检索到一篇10页的技术白皮书，压缩后仅保留与用户问题相关的2个核心章节和关键数据，避免无关段落占用模型上下文空间。

为什么在RAG中需要提示压缩?

RAG的生成效果高度依赖“输入给模型的文档质量”，而检索出的文档通常存在三大问题，必须通过压缩解决：

1. **控制输入长度**： 大语言模型都存在输入长度限制，直接拼接大量检索内容可能会超出模型的上下文窗口，通过压缩可以将关键信息浓缩进有限的 token 内。
2. **提高知识相关性**：检索出的文档中可能有大量无关内容（如重复描述、背景知识、与问题无关的案例）会稀释关键信息，导致模型聚焦困难（例如用户问“如何优化代码”，文档中80%是算法原理，仅20%是优化技巧，未压缩时模型可能错误引用原理部分生成回答），甚至引发“幻觉”。
3. **降低计算资源消耗**： 减少不必要的内容可以降低模型处理和推理的计算负担，如果调用的是商业大模型，还涉及到金额问题（都是通过 token 来计费的）。



不做提示压缩的问题有哪些?

1、上下文溢出导致回答失效:

假设模型最大支持8k tokens，用户问题占 500 tokens，未压缩的文档占 9000 tokens，输入时文档会被截断，模型因缺乏关键信息无法回答（如用户问“文档第3章的结论”，但第3章因长度超限未被输入，尬住了）。

2、生成质量下降：
文档中大量无关内容会误导模型，例如用户问“如何治疗感冒”，检索到一篇包含“感冒”和“癌症”内容的文档，未压缩时模型可能错误关联癌症治疗方法 :)。

3、成本飙升：
处理长文档消耗更多计算资源（如token单价成本增加50%），且生成速度变慢（每1000 tokens处理时间增加约200ms）。

12、如何进行RAG调优后的效果评估?

RAG调优后的效果评估可以围绕这三个维度来看：检索质量、生成质量、系统性能。

1）**检索质量评估**

客观指标：

- **Precision@k**（前k个结果的相关性比例）
- **MRR**（首个相关结果的排名倒数均值）
- **NDCG**（文档相关性等级和排名折扣，区分不同相关度的重要性）
- **Recall@k**（覆盖所有相关文档的比例）

主观评测：通过人工审核检索结果是否满足业务需求。

2）**生成的质量评估**

- **CR 检索相关性（Context Relevancy）**（答案是否基于检索内容）
- **AR 答案相关性（Answer Relevancy）**（回答是否解决用户问题）
- **F 可信度（Faithfulness）**（评估生成的答案中是否存在幻觉）

评测方法：

- **大模型打分**（利用大模型评估答案质量）
- **人工打分**（对CR、AR、F评分）。

3）**系统性能评估**

除了功能性需要外，我们作为应用开发，也要关注下非功能性需求

- **延迟**（响应时间）
- **吞吐量**（单位时间处理请求量）
- **错误率**（生成错误答案的比例）。

在真实的企业场景流程中，我们先：

1. **分层测试**：先测检索质量，再测生成质量，最后压测系统性能。
2. **持续监控**：上线后实时跟踪用户满意度（这也是现在大模型回复都有点评功能）、业务指标（如问题解决率）。

扩展知识

客观指标详解

Precision@k（前k个结果的相关性比例）

衡量前k个检索结果中相关文档的比例，反映检索结果的精准度。

例如前5个结果中有3个相关，则Precision@5=60%。适用于对结果精度敏感的场景（如法律咨询、金融风控）。

MRR（平均倒数排名，Mean Reciprocal Rank）

通过首个相关结果的排名倒数均值，反映系统快速定位相关文档的能力。

若相关结果分别排第1、3位，则MRR=(1/1 +1/3)/2=0.67，反映排名优化效果。

NDCG（归一化折扣累积增益，Normalized Discounted Cumulative Gain）

结合文档相关性等级和排名折扣，评估排序质量的综合指标（区分高/低相关文档的价值）。

假设前3个结果的相关性等级为3（高）、2（中）、1（低），则DCG=3 + 2/1.58 + 1/2=4.58；若理想排序为3、3、2，则NDCG=4.58/5.76≈0.79。适合需要区分相关度优先级的场景（如电商推荐、新闻排序），高度相关文档排前显著提升体验。

Recall@k（覆盖所有相关文档的比例）

衡量前k个结果覆盖所有相关文档的比例，反映检索的全面性。

若知识库有10个相关文档，前5个结果包含4个，则Recall@5=40%。适用于需要全面覆盖的场景（如医疗诊断、科研文献检索），避免漏检关键信息。

注意：若k过大（如k=1000），召回率可能虚高但实际无意义

指标对比与选型建议

指标	核心关注点	适用场景	局限性
Precision@k	前k结果的精度	高精度需求（如法律、金融）	忽略漏检文档
Recall@k	前k结果的覆盖率	全面性需求（如医疗、科研）	k值选择影响结果可信度
MRR	首个相关结果的位置	快速响应需求（如客服、搜索）	忽略后续相关文档排序
NDCG	相关性等级与排序折扣	多级相关性场景（如推荐、排序）	需人工标注文档相关性等级

人工打分

- **CR 检索相关性（Context Relevancy）** 0-3分：判断文档与问题的关联程度（如3=直接解答，1=部分相关）
- **AR 答案相关性（Answer Relevancy）** 0-3分：判断回答是否解决用户问题，是否覆盖问题所有关键点）
- **F 可信度（Faithfulness）** 0/1：判断答案是否存在幻觉

医疗问答场景标注示例

任务背景：优化医疗客服RAG系统，要求标注员对“儿童流感防治”相关问答数据进行标注。

输入数据：

- 用户问题：“5岁孩子高烧39度该吃什么药？”
- 检索文档：儿科诊疗指南、药品说明书等10篇相关文档

- 生成答案："建议服用布洛芬混悬液，剂量为每次10mg/kg"

利用 **LabelStudio** 完成标注。

标注步骤分解

步骤	操作说明	示例结果
1. 检索相关性标注	判断文档是否包含"儿童用药剂量""退烧药种类"等信息	3篇文档评3分，2篇评1分
2. 答案相关性	核对答案中的"布洛芬10mg/kg"是否与《中国药典》一致，检查是否遗漏"物理降温方法""就医指征"（如持续高热）	相关性=2（缺就医建议、药物禁忌）
3. 可信度检测	未发现无关回答	可信度=1

LabelStudio：是一个开源的多模态数据标注平台，支持对文本、图像、音频、视频、时间序列等多元数据类型进行专业标注。

13、为什么需要分块?

分块就是把原始长文本（比如一本书、一篇论文）拆成若干个 **"小块"**（通常几百字到上千字，比如500-1000字），每个小块包含**相对完整的语义单元**，比如一个段落、几个段落或一个小节。

为什么需要分块?

1. **模型处理能力限制**：大语言模型（如GPT）一次能处理的文本长度有限，太长的文本塞进去会"消化不良"，分块后每个小块能塞进模型的"肚子"里。
2. **精准定位信息**：用户提问通常针对局部内容（比如"第三章第二部分的案例是什么"），分块后每个小块像"信息卡片"，检索时能快速找到最相关的卡片，避免在整本大书里"大海捞针"。
3. **平衡上下文与效率**：小块既能保留足够上下文（比如前后句子的逻辑），又能让计算机高效存储和检索（小块的向量计算更快）。

所以分块就是将输入文档或大段文本切分成多个较小的、可控粒度的"块"，以便后续的向量化检索和生成模块高效调用与组合。

分块后的"附加操作"：元数据标注

每个小块除了内容，还要记录"身份信息"，比如：

- 来源：属于原文档的哪一章、哪一节?
- 位置：在原文档中的页码、段落编号?
- 类型：是正文、标题、列表还是图表说明?

14、常见分块策略?

- **自然结构分块**：按文档原有格式拆分，比如遇到标题、空行、章节编号、标点符号（如 ###、\n\n、句号） 就分块。
- **固定大小分块**：将文本按固定字符数、词数或 token 数等均匀切分，简单易实现。
- **滑动窗口分块**：在固定大小基础上为相邻 chunk 保留一定重叠，以减少上下文出现断裂
- **递归分块**：先按段落/章节粗分，超长部分再递归细化（按句子或空行），适合复杂文档。
- **语义分块**:用 NLP 模型（如 BERT、GPT）判断文本语义边界，确保每个块是完整的语义单元（比如一个论点、一个案例）。

- **混合分块**：同时结合固定、滑动、语义等策略，或依据标题、段落层级自适应切分，实现性能与精度平衡。比如在初始阶段使用固定长度快速分块，在后续阶段再通过语义分块进行更精细的分块。

15、什么是Embedding向量嵌入?

简单说，就是把文本内容、图像、音频、视频等形式的信息映射为高维空间中的密集向量（一串数字），这个过程叫“嵌入”（Embedding）。

向量就是语义空间中的坐标，捕捉对象之间的语义关系和隐含的意义。每个向量就像文本的“数字指纹”，包含了文本的语义信息，比如“猫”和“狗”的向量会很接近，“开心”和“悲伤”的向量会远离。即**语义相近的对象在向量空间中彼此邻近，而语义相异的对象则相距较远。**

然后通过在向量空间中进行数学计算（如余弦相似度），判断两段话是否相关（比如用户问题和文档块的匹配）。

因此，分块后的文本块需要先生成 Embedding，存入向量数据库（如 FAISS），用户提问时，系统通过计算提问的 Embedding 与文本块的 Embedding 相似度，找到最相关的内容，再交给大模型生成回答。

16、你知道哪些Embedding向量嵌入模型?

在 RAG 中，常用的 Embedding Model（嵌入模型）主要分为以下几类：

- `text-embedding-ada-002（2022年12月发布）`：OpenAI 的第二代模型，支持多语言，性价比高，适合大多数场景（如文档检索、相似度匹配）。
- `text-embedding-3-small（large）`：OpenAI 推出的第三代高效模型，性能更强，MIRACL（multi-language retrieval ）比上代从 31.4% 提升到了44.4%，MTEB(Massive Text Embedding Benchmark)从 61.0% 提升到了62.3%，且价格更低。
- Sentence-BERT（SBERT）：基于BERT优化，大幅提升句子嵌入速度和相似度计算效果，开源且免费（如`all-mpnet-base-v2`性能最好，而`all-MiniLM-L6-v2`速度最快）。
- Gemini Embedding：在 MTEB 基准测试中表现出色，目前排名第一。
- Cohere Embed：有 embed-english-light-v2, embed-english-v3等模型
- BGE（BAAI General Embedding）：智源研究院研发，专为中文优化（如bge-large-zh），在中文MTEB榜单排名前列。
- M3E（Moka Massive Mixed Embedding）：开源轻量模型，专为中文优化，适合本地部署。

对我们实战而言：**中文选BGE/M3E，英文选OpenAI/Cohere，轻量部署选Sentence-BERT。**

MTEB（Massive Text Embedding Benchmark） 和 C-MTEB （Chinese Massive Text Embedding Benchmark)

目前这个 Embedding 模型相关的榜单被广泛认可，从分类、聚类、语义文本相似性、重排序和检索等多个角度评测排行各种模型，大家可以从上面了解更多的 Embedding 模型。

榜单链接：https://huggingface.co/spaces/mteb/leaderboard

17、如何选择Embedding嵌入模型？考虑因素?

在 RAG 中选择 Embedding Model 时，核心考虑 **7 大因素**，分别是“**准、快、专、广、大、活、省**”：

1. **准（语义准确性）**：模型能否精准捕捉文本语义（比如长句理解、上下文关联、同义词区分），直接影响向量相似度计算的可靠性。
2. **快（模型效率）**：推理速度是否满足业务实时性要求（比如 QPS 高的场景不能用太大的模型），显存/内存占用是否适配硬件资源。

3. **专（领域适配）**：是否针对垂直领域（如法律、医疗、代码）做过预训练或微调，原生支持特定术语和逻辑（比如金融模型懂“PE 估值”，通用模型可能误解为“体育器材”）。
4. **广（多语言支持）**：是否支持业务所需语言（尤其是小语种），以及跨语言对齐能力（比如中英混合文本能否正确嵌入）。
5. **大（数据规模匹配）**：模型参数量和训练数据规模是否匹配你的语料复杂度（小数据用大模型可能过拟合，大数据用小模型可能语义坍缩）。
6. **活（开放性与生态）**：是否开源、是否有活跃社区维护（方便定制化微调），API 调用是否灵活（比如 OpenAI Embedding 适合快速上线，Hugging Face 模型适合自建部署）。
7. **省（成本）**：计算成本（训练/推理的硬件投入）和使用成本（比如第三方 API 的 token 费用，商用模型的授权费）。

扩展知识

1、语义准确性

这个是模型的“理解基本功”，通过 **语义相似度任务**（如 STS-B 数据集）评估，看模型对“同义句/反义句”的向量距离是否合理。

需要注意：

- 部分模型擅长短文本（如 Sentence-BERT），但长文本下会丢失上下文（需选 RoBERTa 变种或 Longformer 类模型）。
- 通用模型（如 text-embedding-ada-002）在专业领域可能“词不达意”（比如“主诉”在医疗文本中是专有名词，通用模型可能理解为“主要诉求”）。

2、模型效率

这块主要关注的是“速度”和“精度”间找平衡！

推理速度：

- 轻量级模型（如 MiniLM、DistilBERT）速度快（毫秒级/句），适合实时问答；
- 重型模型（如 BERT-large、GPT-4 Embedding）速度慢（秒级/句），适合离线批量处理。

3、领域适配

这块主要是为了让模型“懂行话”。

常见有三种策略：

1. 直接选领域专用模型（如 LegalBERT 用于法律文档，PubMedBERT 用于医学文献）；
2. 用通用模型+领域数据微调（适合有私有语料的场景，比如用公司内部客服对话数据微调）；
3. 添加领域适配器（如 LoRA 技术，在不改变原模型的前提下，新增少量参数适配领域）。

比如在某电商场景下用 RAG 用通用模型时，“SKU”“客单价”等词嵌入效果差，切换到零售领域预训练的模型后，召回准确率提升 23%。

4、多语言支持

- **单语言模型**：如 Chinese-BERT（仅中文）、XLM-RoBERTa（支持 100+ 语言但需分别处理）；
- **跨语言模型**：如 mBERT（基于双语对齐训练，中英句子嵌入在同一空间，适合翻译场景）。

需要注意：小语种（如斯瓦希里语）可能没有专用模型，需用“通用多语言模型+数据增强”（比如用谷歌的 multilingual T5 模型，搭配少量目标语言语料微调）。

5、数据规模与模型大小

语料规模	推荐模型类型	示例模型
小（<10万句）	轻量蒸馏模型	MiniLM、DistilBERT
中（10万-1000万句）	中等规模预训练模型	BERT-base、text-embedding-ada-002
大（>1000万句）	大规模模型或定制化训练	BERT-large、GPT-4 Embedding

咱们需要注意，模型参数量并非越大越好，比如 OpenAI 的 ada 模型参数量小但优化了嵌入任务，在通用场景比 GPT-3 嵌入效果更好。

6、开放性与生态

开源 vs 闭源：

- 开源模型（如 Sentence-BERT）：灵活修改代码，适合深度定制（比如加入自定义分词器），但需自行解决部署和优化问题；
- 闭源 API（如 OpenAI Embedding、Cohere）：开箱即用，适合快速验证 MVP，但受限于厂商的更新和费用（比如 OpenAI 按 token 收费，长文本成本较高）。

8、成本

训练成本：

- 开源模型微调：GPU 资源（如用 A100 训练 100 万句数据，成本约 500−1000）；
- 闭源模型：无训练成本，但依赖第三方服务（如 Cohere 嵌入 API 每 1000 tokens 约 $0.01）。

平日使用推理成本：

- 自建部署：硬件折旧（如 8 卡 A100 服务器初期投入 $10 万+，适合高并发场景）；
- 云服务：按调用量付费（适合低频场景，比如每月 10 万次调用，OpenAI 成本约 $20）。

18、索引过程中的文档解析怎么做的?

在 RAG 的索引流程中，文档解析主要分 **5 大核心步骤**，分别是“**读、洗、拆、标、存**”。

1. **读（文档加载）**：支持 **多格式解析**（PDF/Word/Markdown/HTML/图片等），用工具库（如 PyPDF2 读 PDF、Docx2Text 读 Word、Unstructured 处理复杂格式）。
2. **洗（文本清洗）**：去除噪声（如页眉页脚、乱码、重复内容），标准化文本（统一大小写、替换特殊符号），处理多语言混合文本（如中英夹杂时保留语义完整性）。
3. **拆（文本分块）**：按 **语义单元拆分成合适长度的“知识块”**（常见分块策略）。
4. **标（元数据标注）**：给每个知识块附加 **来源信息**（文档标题/URL/作者/上传时间）、 **结构信息**（章节标题/段落位置）、 **领域标签**（如“产品手册”“用户协议”），方便后续检索时过滤和排序。
5. **存（结构化输出）**：将分块后的文本和元数据整合成 **索引系统能识别的格式**（如 JSON/CSV），输出给向量数据库（如 FAISS/Elasticsearch）或传统搜索引擎建立索引。

19、提示工程设计?

在 RAG 应用中，提示工程的核心设计技巧主要有以下几点：

1. **明确角色和任务**：提示中要清楚说明 AI 的身份、能力边界和目标任务。
2. **结构化提示**：用明确的格式指导 AI 输出，比如：“背景 + 问题 + 输出格式”。
3. **加上下文约束**：明确告知 AI 只能基于检索到的资料回答，避免“幻觉”。
4. **模板化设计**：将提示设计成模板，方便大规模复用和动态填充检索内容。

5. **加入冗余兜底机制**：如没检索到相关内容，就让 AI 显式回复 “未找到相关资料”。
6. **给几个示例**：给 1-2 个优质问答范例，让模型模仿输出风格和逻辑。

20、什么是Advanced RAG?

Advanced RAG（高级检索增强生成）是传统 RAG（native 检索增强生成）的升级版，核心是通过**检索前、检索中、以及检索后的优化**解决传统 RAG 的痛点（如检索不精准、上下文断裂、生成质量不稳定等）。

1. **检索前**：通过滑动窗口分块、元数据添加、分层索引、查询重写、查询扩展以及向量化等技术进行优化
2. **检索中**：通过动态嵌入、混合检索、文档嵌入、递归块合并等手段进行优化
3. **检索后的优化**：通过重排序、提示压缩、上下文重构、内容过滤等手段进行优化

扩展知识

1）**滑动窗口分块**

传统 RAG 处理长文档时可能因分块过大或过小丢失关键信息，滑动窗口分块通过设定固定窗口大小（如 500 字），以较小步长（如 100 字）滑动切割文档，确保相邻块间有重叠，避免信息断裂。例如处理长篇法律合同，既能保留条款间逻辑，又能精准匹配用户查询。

2）**元数据添加**
给文档附加标签（如时间、领域、来源）、作者、版本等元数据。比如在企业知识库中，给技术文档标注“2025 年更新”“AI 算法类”，当用户查询时，可通过元数据快速过滤无效内容，提升检索针对性。

3）**分层索引**
构建多级索引结构，先通过粗粒度索引（如文档类别）快速缩小范围，再通过细粒度索引（如关键词、语义向量）精准定位。类似图书馆先按学科分区（粗粒度），再按书名/作者查找（细粒度），大幅提升检索效率。

4）**查询重写与扩展**

- **查询重写**：利用小模型或规则引擎优化用户输入，例如将口语化提问“那个电池技术咋回事”转为“解释特斯拉 4680 电池技术原理”。
- **查询扩展**：分析用户查询后补充相关关键词，如用户问“糖尿病治疗”，自动扩展“2025 最新疗法”“药物副作用”等，扩大检索覆盖面又不偏离主题。

5）**向量化**

将文档和查询转为高维向量，通过余弦相似度等计算匹配度。优化向量化过程（如选择更适配的嵌入模型），能让语义相近但表述不同的内容被准确检索，比如用户查“汽车动力系统”，能召回“引擎技术解析”文档。

6） **动态嵌入**
根据用户场景、历史对话调整嵌入方式。例如电商场景中，用户多次查询“笔记本电脑”，动态嵌入会强化“性能参数”“性价比”等维度，使检索更贴合潜在需求。

7）**混合检索**
结合向量检索（快速匹配相似文档）和语义检索（理解查询真实意图）。如法律场景，先通过向量检索找出包含“合同纠纷”的文档，再用语义检索筛选出与“违约责任认定”最相关的内容。

8）**文档嵌入优化**
对文档内容进行预处理（如去除停用词、强化关键术语）后再向量化。例如技术文档中突出“算法公式”“实验数据”等部分，提升嵌入质量，避免无效信息干扰检索。

9）**递归块合并**
将多个相关文档块递归合并，形成更完整的上下文。比如用户查询“某疾病治疗方案”，系统不仅召回单篇文档中“药物治疗”部分，还能合并另一篇文档的“术后护理”内容，提供全面回答。

10）**重排序**

利用排序模型（如 BERT - based 排序器）给检索结果打分，将最相关的文档提前。例如医疗场景中，优先展示权威指南而非普通科普文章，确保回答专业性。

11）**提示压缩**

去除检索内容中的冗余信息（如重复段落、无关描述），保留核心要点。比如用户问“某手机参数”，压缩后只保留“处理器型号、电池容量、摄像头像素”等关键信息，让提示更简洁高效。

12）**上下文重构**

重新组织检索内容逻辑，使其符合人类表达习惯。如将分散的产品优点整合成“首先……其次……最后”的结构，提升回答可读性。

13）**内容过滤**

剔除错误（如过时数据、明显矛盾内容）或无关信息。比如金融场景中，过滤掉超过 1 年的股票建议，避免误导用户。

21、什么是Modular RAG?

Modular RAG（模块化检索增强生成）是将传统的 RAG 系统拆分成一系列**松耦合、可重组**的功能模块，每个模块各司其职（如检索、检索前处理、检索、检索后处理、生成等环节进行模块化设计），并由一个统一的“**编排器**”负责调度与路由，让整体系统更灵活、可插拔、易维护。

几个核心模块

1. **Indexing**（索引）：对文档进行块优化（如拆分小句、合并大段保留上下文）、结构组织（分层处理 PDF 等文档），并构建知识图谱，让知识存储更有序，便于快速检索。
2. **Pre - Retrieval**（检索前）：通过查询转换（如将口语化提问转为精准指令）、查询扩展（补充相关关键词）等，让检索目标更清晰。
3. **Retrieval**（检索）：采用混合检索（向量 + 关键词）、选择不同检索源（句子、文档块、结构化数据），精准召回相关内容。
4. **Post - Retrieval**（检索后）：对检索结果重排序（筛选最相关内容）、压缩（剔除冗余信息），提升输入质量。
5. **Generation**（生成）：利用大模型生成回答，并通过外部知识验证回答准确性，避免 “幻觉”。

Modular RAG 的 Orchestration 编排

- **Routing**（路由）：分析用户查询语义，判断调用哪些模块。例如，区分是查本地知识库（如 “Python 排序算法”）还是实时数据（如 “上海气温”），再匹配对应流程。
- **Scheduling**（调度）：管理流程节奏，如先检索再生成；若生成结果不佳，调度 “再次检索” 流程，确保输出符合预期。
- **Knowledge Guide**（知识引导）：借助知识图谱规划推理路径。比如回答历史事件时，按时间线、因果关系关联知识，让回答更具逻辑性。

三、 MCP

github MCP学习指南：https://github.com/yzfly/Awesome-MCP-ZH

MCP 能干什么?

MCP 能让 AI 从“嘴炮王”变成“实干家”，以下是几个例子：

1. **连工具**：用 Slack 发消息、用 GitHub 管代码、用 Blender 建 3D 模型。
2. **查数据**：直接看你电脑文件、数据库记录，甚至网上实时信息。
3. **干复杂活儿**：写网页时，AI 能查代码、生成图片、调试页面，一条龙搞定。
4. **人机协作**：AI 干一半问你意见，你点头它再继续。

例子：在 Cursor 里装个 Slack MCP 服务器，AI 能一边写代码一边发消息通知团队，超省事！

MCP服务端支持功能清单：

功能	**描述**
浏览器自动化与网页交互	*(让 AI 能够像人一样浏览网页、提取信息、填写表单等)*
开发与代码执行	*(让 AI 能够运行代码、分析代码库、与开发工具集成等)*
命令行与 Shell 交互	*(让 AI 能够执行命令行指令、与 Shell 交互)*
版本控制 (Git / GitHub / GitLab)	*(让 AI 能够操作代码仓库、管理 Pull Request、处理 Issues 等)*
数据库交互	*(让 AI 能够查询数据库、检查表结构、甚至修改数据)*
云平台与服务集成 (AWS, Cloudflare, Azure, K8s, etc.)	*(让 AI 能够管理云资源、调用云服务 API 等)*
搜索	*(让 AI 能够调用各种搜索引擎或专业搜索服务)*
通讯与协作 (Slack, Email, Calendar, Social, etc.)	*(让 AI 能够收发消息、管理日程、参与团队协作等)*
金融与加密货币	*(让 AI 能够获取金融数据、分析股票、与区块链交互等)*
文件系统与存储	*(让 AI 能够访问本地文件、操作云存储等)*
数据分析、处理与可视化	*(让 AI 能够处理表格数据、生成图表、进行数据探索等)*
效率工具与集成 (Office, Project Management, Notes, etc.)	*(让 AI 能够使用日历、任务管理、项目管理、笔记等工具)*
multimedia 多媒体与内容创作	*(让 AI 能够生成动画、编辑视频、处理图像、语音合成等)*
知识、记忆与 RAG	*(让 AI 拥有长期记忆、能够基于特定知识库回答问题等)*
安全与分析	*(让 AI 能够进行安全扫描、二进制分析、风险评估等)*

功能	描述
地理位置与出行	*(让 AI 能够处理地理位置数据、地图、天气、交通出行信息等)*
体育与游戏	*(让 AI 能够访问体育赛事数据、游戏信息等)*
艺术与文化	*(让 AI 能够访问艺术收藏、文化遗产、博物馆数据库等)*
其他实用工具与集成	*(包括计算器、API 集成、特定平台工具、聚合器、框架辅助等)*

1、MCP客户端接入

什么是MCP：把 AI 想象成一个非常聪明的助手，但它只能在自己的房间里工作。MCP 服务集成就像给它安装了电话、网络和各种工具，让它能够：

- 📞 调用外部服务（如地图、天气）
- 💻 访问你的本地文件
- 🔧 执行各种实用工具

MCP (Model Context Protocol) 让 AI 具备了与外部世界交互的能力。PIG AI 支持两种集成模式：

模式	图标	描述
SSE 模式	☁️ cloud	连接互联网上的各种服务 如：百度地图、天气查询等
本地模式	💻 computer	访问你电脑上的工具和文件 如：读写文件、运行命令等

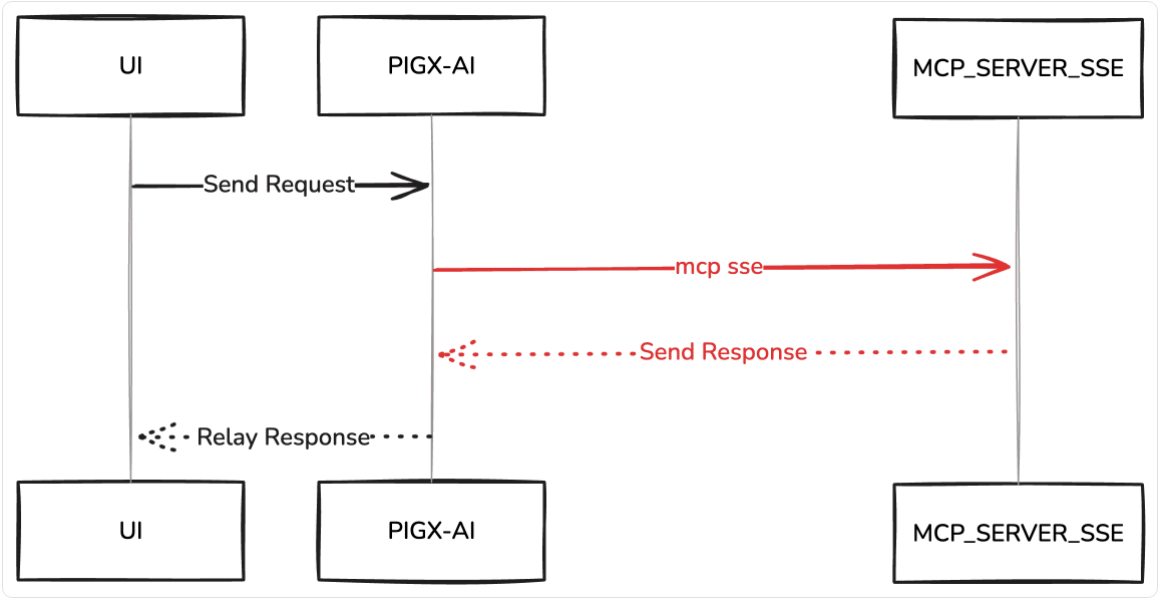
1.SSE模式：连接云端服务

什么是 SSE 模式?

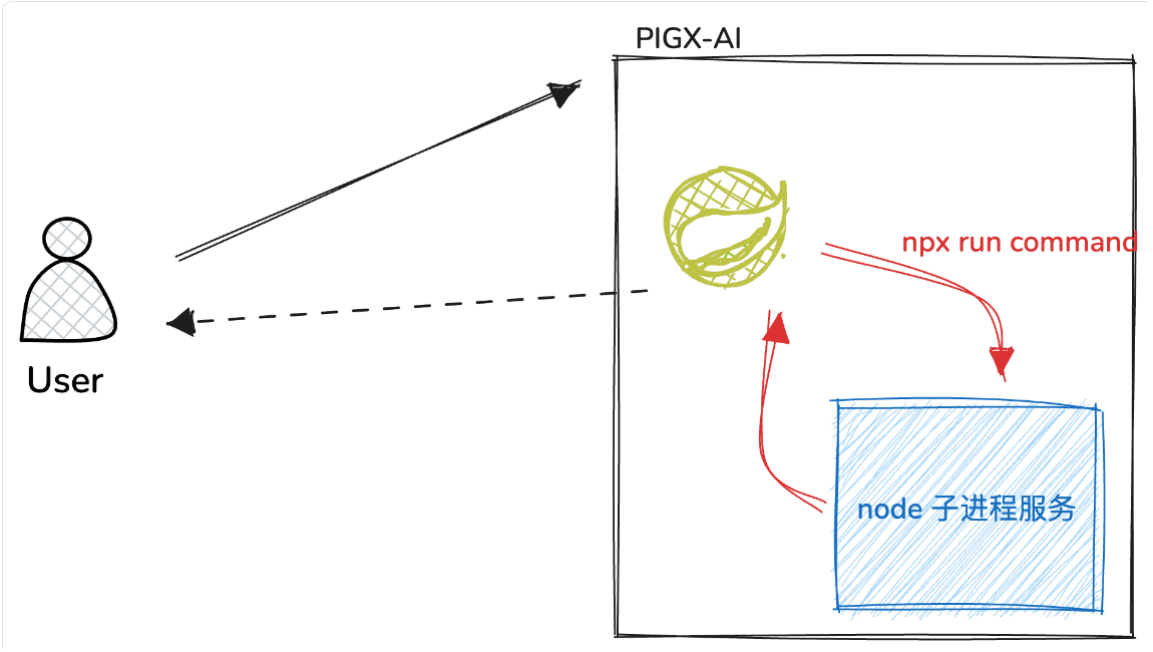
SSE (Server-Sent Events) 模式就像给 AI 接入了互联网。通过一个网址，AI 就能访问各种在线服务，比如：

- 🗺️ 百度地图：查地址、规划路线
- 🌤️ 天气服务：获取实时天气
- 📊 数据服务：查询股票、汇率等

简单理解：你提供一个服务网址，AI 就能调用这个服务的所有功能！



2.本地模式：让AI访问你的电脑



什么是本地模式?

本地模式就像给 AI 装了一个"机器人手臂"，让它能够：

- 📁 读取和编辑你电脑上的文件
- 🔍 搜索本地文档内容
- ⚙️ 运行你电脑上的工具和命令
- 📊 分析本地数据文件

本地 (STDIO) 模式通过标准输入/输出协议，让 AI 运行你电脑上的命令行工具。

环境工具介绍

Node.js 工具

- 像一个"临时工具箱"，需要什么工具就临时下载使用，用完就删除。
- 适用场景：运行各种 Node.js 的 MCP 工具
- 安装方式：

1. 访问 Node.js 官网
2. 下载并安装 Node.js (v18+)
3. 安装完成后就可以使用 `npx` 命令了

Python 工具

- 超快速的 Python 包管理器，比传统的 pip 更快更好用。
- 适用场景：运行各种 Python 的 MCP 工具
- 安装方式：参考 官方安装文档

3.实现原理和流程解析

1）新增MCP客户端服务

包含服务名称、描述、类型（SSE、STEAMABLE、STDIO），是否启用

- SSE

  包含SSE地址、请求头参数配置

- STDIO

  包含使用命令（uvx、npx、java、docker）、执行参数、环境变量

2）对话聊天

聊天实现流程同RAG知识库，只是在处理聊天请求时通过datasetId匹配MCP对应策略模式，采用的是McpChatRule规则

```
/**
 * 处理MCP聊天请求
 * <p>
 * 主要流程： 1. 如果用户未指定MCP，通过向量搜索自动选择 2. 获取MCP配置并创建或获取MCP客户端
3. 使用MCP工具提供者构建助手服务 4.使用模板处理用户输入并返回流式响应
 * @param chatMessageDTO 聊天上下文信息
 * @return AI响应结果流
 */
@Override
public Flux<AiMessageResultDTO> process(ChatMessageDTO chatMessageDTO) {
    List<Long> mcpIdList = new ArrayList<>();
    // 如果用户未提供MCP，则自动选择
    if (Objects.isNull(chatMessageDTO.getExtDetails())
            || StrUtil.isBlank(chatMessageDTO.getExtDetails().getMcpId())) {
        mcpIdList = autoChoice(chatMessageDTO);
        if (CollUtil.isEmpty(mcpIdList)) {
            return Flux.just(new AiMessageResultDTO("未找到相关MCP配置，请点击下方+按
钮选择目标MCP"));
        }
    }
    else {
        // 如果用户提供了MCP ID，则直接使用
        mcpIdList.add(Long.parseLong(chatMessageDTO.getExtDetails().getMcpId()));
    }

    // 获取MCP配置
    List<AiMcpConfigEntity> mcpConfigEntityList = mcpConfigMapper
```

```
        .selectByIds(mcpIdList.stream().distinct().toList());

    // 通过McpClientProvider获取或创建MCP客户端
    List<McpClient> mcpClientList = mcpConfigEntityList.stream()
        .map(mcpClientProvider::getOrCreateMcpClient)
        .filter(Objects::nonNull)
        .toList();

    // 使用MCP客户端设置工具提供者
    ToolProvider toolProvider =
McpToolProvider.builder().mcpClients(mcpClientList).build();

    // 根据请求的模型名称获取AI模型和助手服务
    Pair<StreamingChatModel, AiStreamAssistantService> servicePair =
modelProvider
        .getAiStreamAssistant(chatMessageDTO.getModelName());

    // 使用工具提供者构建助手服务
    AiNoMemoryStreamAssistantService assistant =
AiServices.builder(AiNoMemoryStreamAssistantService.class)
        .streamingChatModel(servicePair.getKey())
        .toolProvider(toolProvider)
        .build();

    // 使用模板处理聊天并返回响应流
    Map<String, Object> promptData = Map.of("mcpName",

 mcpClientList.stream().map(McpClient::key).collect(Collectors.joining(StrUtil.CO
MMA)), "inputMessage",
            chatMessageDTO.getContent(), "accessToken", SecurityUtils.getToken(),
systemTime, DateUtil.now());

    TokenStream tokenStream =
assistant.chatTokenStream(PromptBuilder.render("mcp.st", promptData));

    return Flux.create(fluxSink -> {
        tokenStream.beforeToolExecution(beforeToolExecution -> {
            AiMessageResultDTO.ToolInfo toolInfo = new
AiMessageResultDTO.ToolInfo();
            toolInfo.setName(beforeToolExecution.request().name());
            toolInfo.setParams(beforeToolExecution.request().arguments());
            AiMessageResultDTO aiMessageResultDTO = new AiMessageResultDTO();
            aiMessageResultDTO.setToolInfo(toolInfo);
            fluxSink.next(aiMessageResultDTO);
        })
            .onPartialResponse(message -> fluxSink.next(new
AiMessageResultDTO(message)))
            .onCompleteResponse(chatResponse -> fluxSink.complete())
            .onError(fluxSink::error)
            .start();
    });
}
```

创建mcp客户端实例（MCP 本质是一个 **工具集协议**：客户端通过它可以发现、调用远程工具）：

```
/**
 * 创建MCP客户端实例
 * <p>
 * 根据配置的通信类型（SSE或STDIO）创建对应的传输器和客户端
 * @param mcpConfig MCP配置信息
 * @return MCP客户端实例
 */
private McpClient createMcpClient(AiMcpConfigEntity mcpConfig) {
    // 根据MCP类型创建适当的传输器
    McpTransport transport;
    if (McpTypeEnums.SSE.getType().equals(mcpConfig.getMcpType())) {
        transport = createHttpTransport(mcpConfig);
    }
    else if (McpTypeEnums.STEAMABLE.getType().equals(mcpConfig.getMcpType())) {
        transport = createStreamableTransport(mcpConfig);
    }
    else {
        transport = createStdioTransport(mcpConfig);
    }

    // 配置并构建MCP客户端
    try {
        return new DefaultMcpClient.Builder().transport(transport)
            .logHandler(new DefaultMcpLogMessageHandler())
            .build();
    }
    catch (Exception e) {
        log.error("创建MCP客户端失败: {}, 错误信息: {}", mcpConfig.getName(),
e.getMessage(), e);
        return null;
    }
}
```

核心流程：

1. 从数据库或配置中拿 MCP 配置。
2. 通过封装类 `McpClientProvider` 创建/获取 `McpClient`，MCP 客户端是一个连接器，它知道如何和外部 MCP 服务交互
3. 用 `McpToolProvider` 把 MCP 客户端包装成 `ToolProvider`。`McpToolProvider` 会把多个 `McpClient` 聚合到一起，形成统一的 `ToolProvider`，告诉 LangChain4j：哪些工具可用，怎么调用
4. 结合 LangChain4j 的 `StreamingChatModel` 创建 Assistant。LangChain4j 提供的 `AiServices` 机制，可以把模型和工具挂在一起，选定参数（StreamingChatModel使用的模型、toolProvider服务提供方、AssistantService无记忆聊天助手）
5. 使用 Prompt 模板触发对话，流式处理响应。模板会告诉AI用户的输入、可用mcp工具、系统上下文信息。接着流式输出：

```
tokenStream
    .beforeToolExecution(req -> { ... }) // 调用 MCP 工具前的事件（可记录日志、展示 UI）
    .onPartialResponse(msg -> { ... })  // 流式响应片段（边生成边输出）
    .onCompleteResponse(resp -> { ... })// AI 完成回答
    .onError(err -> { ... })            // 异常处理
    .start();
```

2、MCP服务端开发

MCP服务端开发：将现有的业务接口包装成 MCP 服务，让 AI 能够调用你的接口，通过自然语言让 AI 帮你查询业务数据

> 掌握了这个方法，你可以将任何业务接口都包装成 MCP 服务：
> • 用户管理：让 AI 帮你查询用户信息、重置密码等
> • 订单系统：通过自然语言查询订单状态、统计数据
> • 库存管理：询问库存数量、预警信息等
> • 财务报表：生成各种维度的财务分析报告
> • 日志分析：智能检索和分析系统日志

1.业务接口自动转mcp原理

在 MCP 协议下，**服务端的职责是：提供一组工具 (Tools)**，供客户端调用。
这些工具其实就是通过 `@Tool` 注解方法，MCP 框架会扫描并注册这些方法，让它们能被 AI/客户端动态发现并调用。

1）配置加载

```
@PropertySource(value = "classpath:mcp-config.yaml", factory =
YamlPropertySourceFactory.class)
@Configuration(proxyBeanMethods = false)
@Import(ToolAutoRegister.class)
public class PigxAiMcpAutoConfiguration {
    @Bean
    public PigxSecurityToolAspect toolAspect() {
        return new PigxSecurityToolAspect();
    }
}
```

> - `@PropertySource` + `YamlPropertySourceFactory`
>   加载 `mcp-config.yaml`，用来配置 MCP 服务端相关参数（比如mcp服务端名称、版本号，同步/异步调用、描述、sse-endpoint主通道地址用来建立sse长连接，sse-message-endpoint消息推送的接口地址、能力声明、基础访问路径base-url）。
> - `@Configuration(proxyBeanMethods = false)`
>   声明配置类，不用代理方法（减少 Spring CGLIB 开销）。
> - `@Import(ToolAutoRegister.class)`
>   把 **工具自动注册逻辑**（`ToolAutoRegister`）导入 Spring 容器。

2）工具自动注册器

MCP工具自动配置类：自动扫描并注册带有@Tool注解的方法

```
public class ToolAutoRegister implements BeanPostProcessor,
ApplicationContextAware {
    @Bean("controllerToolCallbackProvider")
    public ToolCallbackProvider controllerToolCallbackProvider() {
        List<Object> toolObjects = new ArrayList<>();
        String[] restBeanNames =
applicationContext.getBeanNamesForAnnotation(RestController.class);

        for (String beanName : restBeanNames) {
            Object bean = applicationContext.getBean(beanName);
            Class<?> clazz = AopUtils.getTargetClass(bean);
            boolean match = Arrays.stream(clazz.getDeclaredMethods())
                .anyMatch(method -> method.isAnnotationPresent(Tool.class));
            if (match) {
                toolObjects.add(bean);
            }
        }

        return
MethodToolCallbackProvider.builder().toolObjects(toolObjects.toArray()).build();
    }
}
```

> - 扫描所有 `@RestController` Bean。
> - 判断里面是否存在带 `@Tool` 注解的方法。
> - 如果有，就把整个 Bean 交给 `MethodToolCallbackProvider`。
> - `MethodToolCallbackProvider` 会基于反射把这些方法注册成 MCP 工具（ToolCallback），暴露给客户端。
>
> 👉 **这样，客户端通过 MCP 协议调用工具时，就能动态路由到这些方法**
>
> `ApplicationContextAware`：获取应用上下文，让一个 bean 拿到 Spring 容器的引用，这里是从容器里找出所有 Controller
> `BeanPostProcessor`：在 Spring 容器创建 bean 的生命周期中，提供两个“钩子”方法，让 bean 初始化前后做增强或替换，这里是在 bean 初始化阶段扫描 `@Tool` 注解，收集工具方法。**最后注册成SpringAI的 ToolCallbackProvider** → 作为提供者让 MCP 框架知道有哪些工具能对外暴露

3）权限切面拦截

```
@Slf4j
@Aspect
@RequiredArgsConstructor
public class PigxSecurityToolAspect implements Ordered {
    @SneakyThrows
    @Around("@annotation(tool)")
    public Object around(ProceedingJoinPoint pjp, Tool tool) {
        MethodSignature signature = (MethodSignature) pjp.getSignature();
        Method method = signature.getMethod();
```

```
        HasPermission annotation = method.getAnnotation(HasPermission.class);
        if (Objects.nonNull(annotation)) {
            McpContextHolder.setAnnotation(annotation);
        }

        try {
            return pjp.proceed(); // 执行工具方法
        } finally {
            McpContextHolder.clear();
        }
    }

    @Override
    public int getOrder() {
        return Ordered.HIGHEST_PRECEDENCE + 1;
    }
}
```

- `@Around("@annotation(tool)")`：拦截所有带 `@Tool` 的方法。
- 如果方法上还加了 `@HasPermission`，就存到 `McpContextHolder`（ThreadLocal 里）。
- 执行完方法后清理上下文，避免线程污染。

👉 **这样，调用 MCP 工具时可以自动做权限校验。**

4）上下文管理器

```
@UtilityClass
public class McpContextHolder {
    private final ThreadLocal<HasPermission> THREAD_LOCAL = new
TransmittableThreadLocal<>();

    public void setAnnotation(HasPermission annotation) {
        THREAD_LOCAL.set(annotation);
    }

    public HasPermission getAnnotation() {
        return THREAD_LOCAL.get();
    }

    public void clear() {
        THREAD_LOCAL.remove();
    }
}
```

- `ThreadLocal` 保存每次工具调用时的权限注解（`HasPermission`）。
- 结合 `PigxSecurityToolAspect`，保证在一个请求上下文内，工具能知道自己需要什么权限。
- 用 `TransmittableThreadLocal`，确保线程池环境下也能传递。

5）给接口@Tool注解

找到你想要让 AI 调用的接口方法，加上 @Tool 注解：

```
// 保持原有的所有注解不变，只需要新增一行
@Tool(description = "查询行政区划分页数据")
public R getSysAreaPage(@ParameterObject Page page, @ParameterObject
SysAreaEntity sysArea) {
    // 原有业务逻辑保持不变
    // ....
}
```

6）调用测试

在 PIG AI 的 MCP 模块中，添加你的 MCP 服务配置：

配置项	配置值	说明
协议类型	**SSE**	选择 Server-Sent Events 模式
单体版本使用此地址	`ip:9999/admin/mcp/sse`	单体版本使用此地址
微服务版本使用	`ip:9999/路由前缀/mcp/sse`	微服务版本使用此地址

3、MCP相关面试题

1、什么是MCP?

MCP 为大型语言模型（LLM）提供了一个统一的通信协议，使得各种数据源和工具只需遵循 MCP 的规范，便可与 LLM 无缝对接。这意味着，无论是数据库、API 还是其他服务，只要通过 MCP 接入，LLM 都能理解并利用这些资源，从而扩展其功能和应用范围。

MCP 的作用主要体现在以下几个方面：

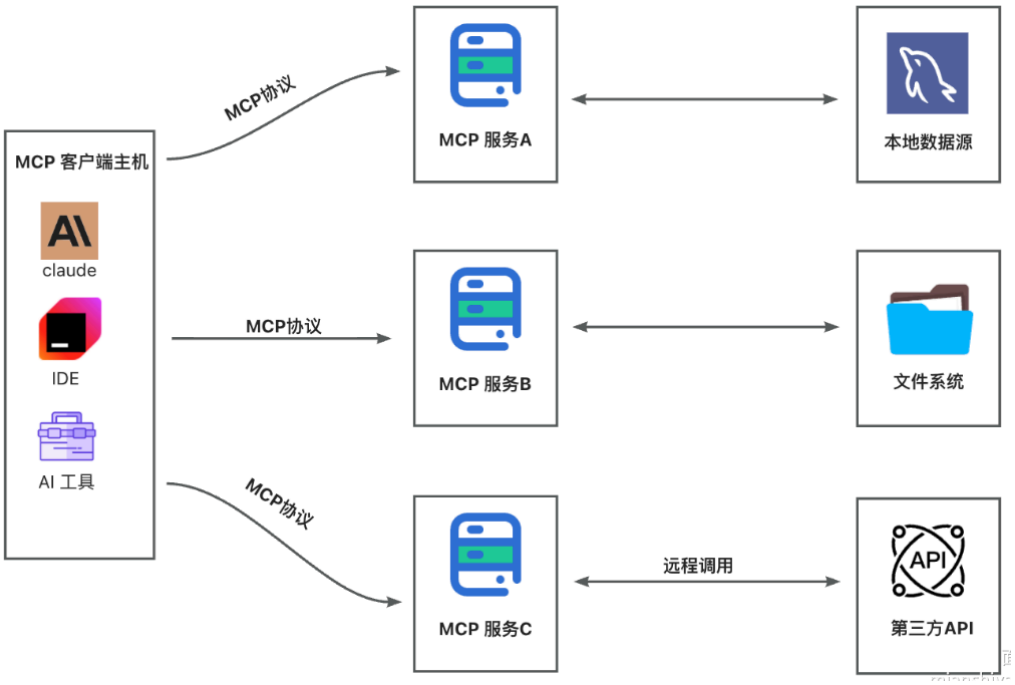
1）**标准化数据接入**：通过 MCP，我们无需为每个模型编写单独的代码，而是通过统一的协议接口，实现一次集成，随处连接。这大大简化了模型与外部系统的集成过程。

2）**增强模型能力**：MCP 使得模型能够实时访问最新的数据和工具，例如直接从 GitHub 获取代码库信息，或从 本地访问文件。这不仅提升了模型的实用性，也拓展了其应用场景。

3）**提升系统可维护性**：通过标准化的协议，系统的各个组件可以更加模块化地协作，降低了维护成本和出错概率。

2、MCP核心组件?

MCP 的核心是**客户端-服务器**架构，其中主机应用程序可以连接到多个服务器：



- **MCP 主机**：希望通过 MCP 访问数据的程序，例如 Claude Desktop、IDE 或 AI 工具
- **MCP 客户端**：与服务器保持 1:1 连接的协议客户端
- **MCP 服务器**：轻量级程序，通过标准化的 MCP 协议向客户端提供特定功能，如数据源、工具和API接口等。
- **本地数据源**：MCP 服务器可以安全访问用户的计算机文件、数据库和服务。
- **远程服务**：MCP 服务器可通过互联网（例如通过 API）连接到的外部系统。

扩展知识

MCP Client

MCP Client 是连接大型语言模型（LLM）与 MCP Server 的桥梁，负责在模型与外部工具之间传递信息和协调操作。它的工作流程大致如下：

1）MCP Client 首先从 MCP Server 获取可用的工具列表。

2）当用户发起请求后，MCP Client 会将工具信息通过 Function Calling 的形式发送给 LLM。

3）LLM 接收到请求后，会根据上下文判断和工具描述是否需要调用工具，以及应当调用哪些工具。

4）如果需要使用工具，MCP Client 会代表模型通过 MCP Server 发起相应的工具调用。

5）工具执行完成后，结果会返回至 LLM，模型据此整合所有上下文信息并生成自然语言响应。

6）最终，MCP Client 将模型的响应返回给用户，完成整个交互闭环。

当前包括 Claude Desktop 和 Cursor 等产品已经支持 MCP Server 接入，它们自身即作为 MCP Client，负责与 MCP Server 建立连接并完成工具调用的上下文感知与执行逻辑。

MCP Server

MCP Server 是 MCP 架构的核心组件，主要负责向 LLM 提供结构化上下文和可调用的操作能力。它定义了三大类基础功能：

1）**Resources（资源）**：类似静态或动态的数据文档，例如 API 返回的 JSON 数据、配置文件、项目结构等，供模型查询和参考。

2）**Tools（工具）**：是模型可调用的函数接口（如发送请求、创建文档、执行命令等），执行前通常需要用户授权，以确保安全性和可控性。

3）**Prompts（提示）**：预定义的提示模板，用于引导模型完成特定任务，如代码审查、生成 commit message、总结会议内容等，提高交互效率和质量。

通过这三类能力，MCP Server 为语言模型提供了更丰富的上下文输入和更强的交互能力，使模型不仅能“看懂”信息，更能“动手”完成任务。

我们可以在社区维护的 [**MCP Servers Repository**] 和 [**Awesome MCP Servers**] 中找到大量开源的 MCP Server 实现示例。其中，使用 TypeScript 编写的 MCP Server 通常通过 `npx` 命令运行；而用 Python 编写的版本则可以通过 `uvx` 命令快速启动，非常方便开发与部署。

3、MCP支持的模式?

MCP 协议支持两种主要的通信模式，即 **标准输入输出（Stdio）** 模式和 **服务器发送事件（SSE）** 模式。

使用场景

标准输入输出(stdio):

1）命令行工具的开发
2）本地系统集成
3）简单的进程通信
4）shell 脚本交互

服务器发送事件(SSE):

1）服务器和客户端需要流式通信
2）在指定的网络环境中
3）MCP 客户端和 MCP 服务器不在同一台机器或者容器里

> 当然，MCP也支持自定义传输层的开发，只需实现Transport接口，可以自定义网络协议，优化传输性能，也可以通过特殊的传输通道进行通信。

消息格式

MCP（模型上下文协议）要求通信格式遵循 `JSON-RPC 2.0` 标准，确保消息在各方之间能够准确传达。这就像为所有参与者统一了一套“规则”——无论是数据服务商、工具提供者，还是AI应用，都需按照这一规范与大模型进行交流。

一共有三种类型的消息使用了`JSON-RPC 2.0`标准：

请求

```
{
  jsonrpc: "2.0",
    id: number | string,
    method: string,
    params?: object
}
```

响应

```
{
  jsonrpc: "2.0",
  id: number | string,
  result?: object,
  error?: {
    code: number,
    message: string,
    data?: unknown
  }
}
```

通知

```
{
  jsonrpc: "2.0",
  method: string,
  params?: object
}
```

4、MCP和Function Calling区别?

MCP是一种对外统一的协议USB接口，让大模型去理解调用；Function Calling偏向应用内部可以自定义函数调用，比较单一固定

- **MCP**

MCP 是一个抽象层面的协议标准。它规定了上下文与请求的结构化传递方式，并要求通信格式符合 JSON-RPC 2.0 标准，它提供了标准化的通信机制，确保不同模型之间的兼容性。我们只需按照协议开发一次接口，即可被多个模型调用，避免了为每个模型单独适配的繁琐工作。

- **Function Calling**

Function Calling 则是某些大模型（如 OpenAI 的 GPT-4）提供的特有接口特性。 它以特定的格式让 LLM 能产出一个函数调用请求，然后应用可以读取这个结构化的请求去执行对应的操作并返回结果。 但这个特性本身并不要求消息一定是 JSON-RPC 格式，也不一定遵守 MCP 的上下文管理方式。 它是由大模型服务提供商定义的一种调用机制，与 MCP 所定义的协议与标准没有内在依赖或直接关联。

两者比较

其实 **MCP** 和 **Function Calling** 两者是完全不同层面的东西，MCP 是一个更底层、更通用的抽象标准，相当于一个基础设施，而 **Function Calling** 则是大模型一个特定的服务，更偏向与具体实现。以支付系统来举例子，以前的 **Function Calling** 相当于请求各个支付系统，微信，支付宝，银联，每个系统都需要单独对接，而 **MCP** 相当于支付网关，只需要对接支付网关，后面对接各种支付系统都是支付网关做，我们不用管。

因此 MCP 协议通过统一通信规范和资源定义标准，实现了“一次开发，全平台通用”的目标。借助 MCP，我们只需按照协议开发一次接口，便可无缝适配 ChatGPT、Deepseek 等不同模型，显著降低了集成成本和复杂性。

5、MCP工作流程?

1）初始化连接：主机应用程序（如 Claude Desktop 或 IDE 插件）启动并初始化 MCP 客户端，每个客户端与一个 MCP 服务器建立连接。

2）获取工具列表

在系统初始化阶段，**MCP Client** 首先从 **MCP Server**获取可用的工具清单和能力描述。这些工具可以是 API、脚本、数据库查询方法等。这个步骤相当于“能力注册”，让模型知道有哪些可以用的“外部技能”。

3）构造 Function Calling 请求

当用户输入一个问题后，MCP Client 会将这些工具描述（包括参数、用途、返回值等）一并传给 LLM。传输方式采用 **Function Calling**，即结构化地把“能用的函数长什么样”告诉模型，让它来决定是否要使用。

4）模型智能判断是否调用工具

LLM 根据当前对话上下文以及工具信息判断是否要使用某个工具，并决定调用哪一个、传入什么参数。这个阶段完全由模型推理完成，比如它可以判断“这题需要查天气，那我就调用 **getWeather 工具”**。

5）工具调用执行阶段

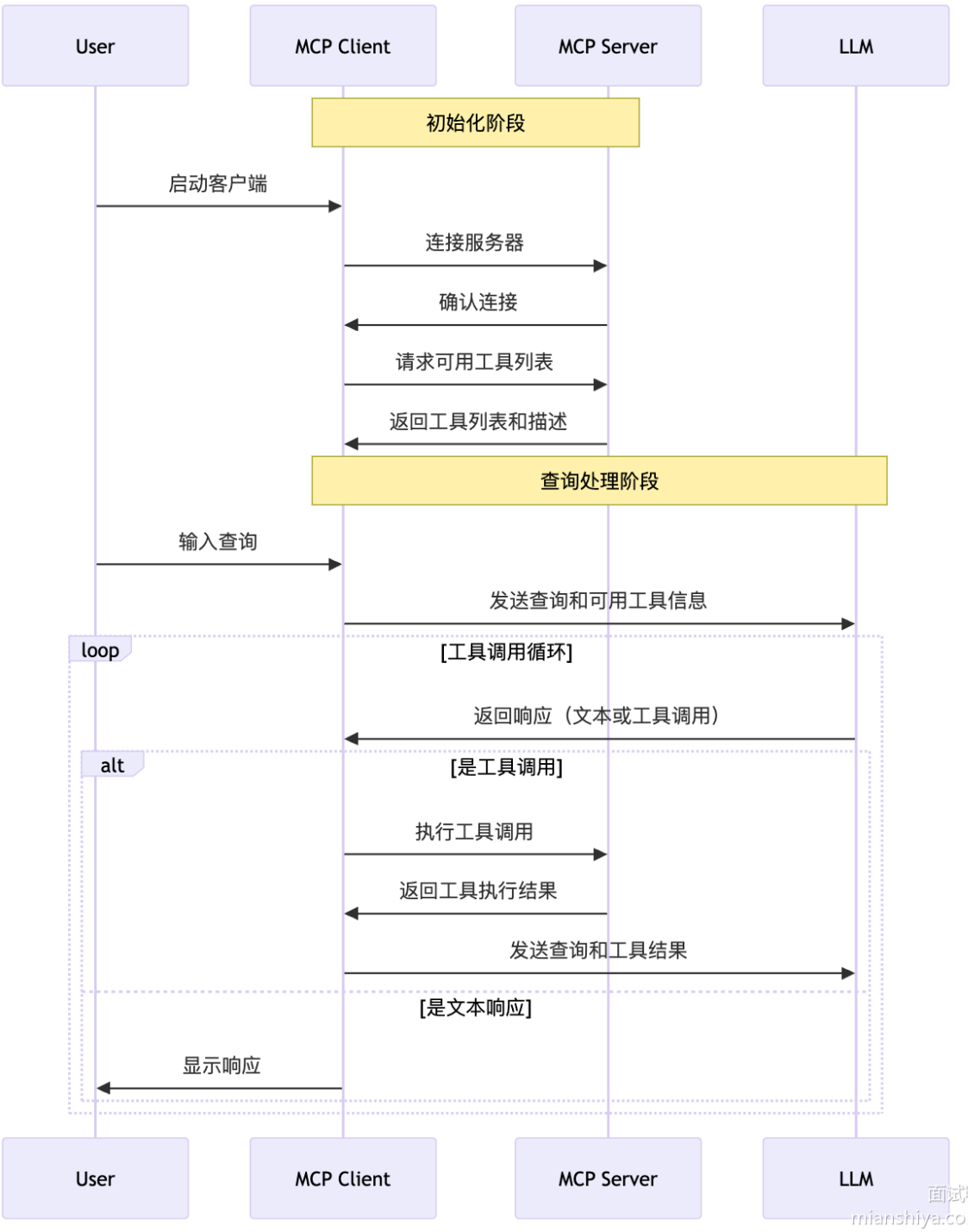
如果模型发起调用请求，**MCP Client** 会根据模型的选择，通过 **MCP Server**发起工具调用。这一步是真正的“执行动作”——MCP Server 去实际跑工具的逻辑，并把结果返回给客户端。

6）结果返回与模型整合

工具执行结果（比如调用 API 得到的数据）被传回 LLM，由模型将这个结果与原始用户问题、已有上下文等信息整合，生成最终的自然语言回应。

7）用户响应输出

最后，MCP Client 将模型生成的回答展示给用户，完成一次完整的“智能+工具”协作流程。



拓展知识

1）**通信机制**：MCP 支持两种通信方式：标准输入输出（Stdio）和服务器发送事件（SSE）。Stdio 适用于本地进程间通信，启动速度快，适合开发调试；SSE 基于 HTTP 长连接，适用于需要实时数据推送的分布式系统。

2）**模块化设计**：MCP 的架构采用客户端-服务器模型，主机应用可以连接多个 MCP 服务器，每个服务器处理不同的资源和工具。这种模块化设计提高了系统的扩展性和可维护性。

3）**安全性**：MCP 通过标准化的数据访问接口，减少了直接接触敏感数据的环节，从而降低了数据泄露的可能性。例如，MCP 服务器自身控制资源，无需将密钥等敏感信息提供给大模型提供商，即使大模型提供商受到攻击，攻击者也无法获取这些敏感信息。

6、SpringAI如何集成MCP?

`FunctionCalling<T>`通过重写该类来定义实现函数业务逻辑。

`@Tool`是 Spring AI MCP 框架中用于快速暴露业务能力为AI工具的核心注解，该注解实现Java方法与MCP协议工具的自动映射，并且可以通过注解的属性`description`，用来标注/声明描述一个MCP工具

`ToolCallback`用于描述一个可被调用的“工具”，封装了函数描述、参数 schema、执行逻辑。Spring AI 框架会通过 `ToolCallback` 执行具体逻辑。

`ToolCallbackProvider`是 Spring AI 中的一个接口，用于定义工具发现机制，主要负责将那些使用 `@Tool` 注解标记的方法转换为工具回调对象，并提供给 ChatClient 或ChatModel 使用，以便 AI 模型能够在对话过程中调用这些工具。`ToolCallbackProvider` 会集中管理所有 `ToolCallback`，统一暴露给模型或 MCP。

7、MCP协议的安全性设计包含哪些层面?

MCP协议的安全性设计涵盖多个关键方面，旨在确保大模型系统在与外部工具和资源交互时的安全性和可靠性。MCP 的安全性设计主要包括以下几个方面：

1）用户同意和控制

所有模型对工具、资源和提示的访问请求都必须经过用户的授权。主机负责管理权限、提示用户批准，并在必要时阻止未经授权的访问。用户必须知道哪些数据是需要提供给大模型的，用户在授权使用之前应该了解每个工具的功能。

2）隔离与沙箱机制

MCP 的设计将实际工具调用封装在 MCP Server 内部，模型本身无法直接访问敏感数据，这种 “中间层”设计有效地降低了直接暴露内部业务系统的风险。同时，沙箱机制可以限制工具调用的执行环境，防止任意代码执行或恶意操作对系统造成破坏。

3）加密传输与来源验证

MCP 内置了安全机制，确保只有经过验证的请求才能访问特定资源，相当于在数据安全又加上了一道防线。同时，MCP协议还支持多种加密算法，以确保数据在传输过程中的安全性。

扩展知识

除了以上这些，我们在开发时应该还要考虑以下几个方面：

1）密钥加密处理

有些的 MCP 服务是需要输入第三方服务的密钥的，比如高德导航MCP，需要输入开放平台申请的key，客服端和服务器端传输的过程可能有泄露的风险，应该使用某种签名或者加密算法来保障数据安全。

2）模型诱导攻击

攻击者可能通过精心设计的提示词诱导模型执行不当的操作，例如泄露敏感信息或调用危险的工具。为防范此类攻击，建议在主机层面实施提示词过滤和上下文审查机制。

3）日志记录与调用链追踪

MCP 客服端和服务端都要记录每一次工具调用的详细日志，包括请求参数、响应结果和执行时间等信息，便于后续的实时监控和问题排查。

8、SpringAI如何将已有应用转为MCP服务?

在 SpringAI 中，**业务接口一键转 MCP 的原理就是：通过 `@Tool` 注解和 `BeanPostProcessor` 扫描，使用MethodToolCallbackProvider把已有的 Controller/Service 方法反射注册为 MCP 的 ToolCallback，框架再通过上下文和切面完成调用路由与权限控制，从而让 LLM 客户端能像调用工具一样调用原有接口。**

> SpringAI 提供 **自动注册工具并暴露 MCP 服务** 的机制，所有实现了 `FunctionCalling<T>`/`ToolCallBack` 的组件会被自动注册到 MCP Server，客户端请求 MCP Server 时，会根据工具 `name` 调用对应的 `call()` 方法。
>
> **重点**：`MethodToolCallbackProvider` 可以把 **已有方法或服务接口** 直接注册到 MCP 服务中。

四、Function Calling

1、函数调用

函数调用（Function Calling）是连接大模型能力与业务系统的关键桥梁。通过将业务功能封装为标准化接口，实现自然语言到结构化操作的智能转换，让AI真正成为业务增长的智能引擎。根据用户输入的意图**自动选择并调用你注册的函数**，并把自然语言转成函数需要的参数。

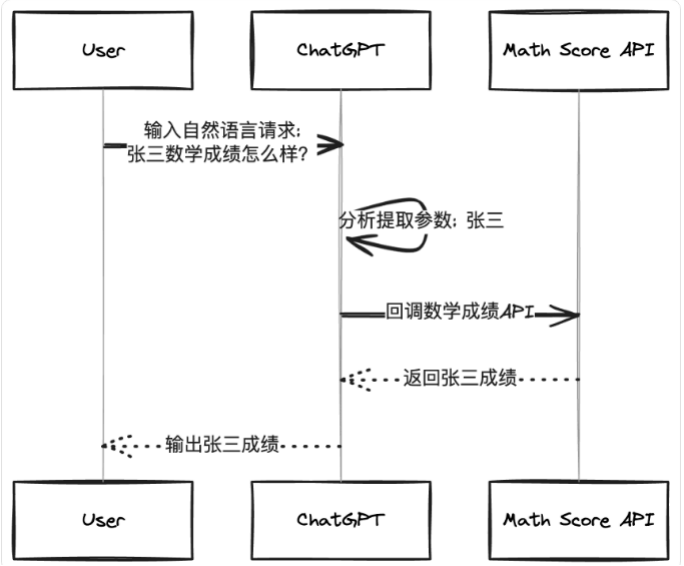
本质：它让 AI 不是只“说话”，而是能“做事”。

- 函数调用（Function Calling）是大模型的核心能力，允许模型**识别用户意图并将其转化为结构化函数调用**。通过这种方式，模型可以与外部系统交互，执行数据查询、API调用、工具使用等操作，并将结果整合到对话中。 函数调用使大模型能够实时获取业务系统数据，弥补知识时效性限制，提供更精准、实时的回答，显著增强模型在专业领域的应用价值。
- 核心优势：**将非结构化自然语言转换为结构化参数**，实现人机交互到系统操作的无缝衔接，大幅提升业务流程自动化水平。

1、业务场景使用

1）业务常见描述

- 通过自然语言告诉大模型：帮我帮我查询小明同学的数据成绩，那么大模型自动回调具体的 PIG AI 代码实现联动查询数据或api 返回具体的成绩



2）开发自己的函数

定义接收字段

当用户在前端输入自然语言查询（如"小明成绩是多少？"）时，我们需要将这种非结构化的查询转换为后端可处理的结构化JSON数据。 为此，我们需要定义一个接收实体类，用于接收大模型解析后的结构化参数：

```
@Data
@JsonInclude(JsonInclude.Include.NON_NULL)
@JsonClassDescription("学生信息")
public class StudentRequest extends BaseAiRequest {

    @FieldPrompt("学生姓名")
    private String studentName;
}
```

定义函数

```
@Component
@RequiredArgsConstructor
public class MathScoreFunctionCalling implements FunctionCalling<StudentRequest>
{

    /**
     * 获取功能描述信息,非常重要，写准确一点，多个函数的时候，AI可以根据描述来选择调用哪个函数
     *
     * @return 功能描述字符串
     */
    @Override
    public String functionDesc() {
        return "学生成绩查询助手: 能根据您的描述帮您查询学生的数学成绩\n";
    }


    /**
     * 检查参数方法 ， 大模型会把用户输入的自然语言 回调到这个 StudentRequest 参数，我们要校验参数是不是正确，这是业务校验
     */
    @Override
    public R checkParams(StudentRequest studentRequest, PigxUser userDetails,
ChatMessageDTO.ExtDetails extDetails) {
        return R.ok();
    }

    /**
     * 业务处理逻辑
     */
    @Override
    public R<String> handle(StudentRequest studentRequest, PigxUser userDetails,
ChatMessageDTO.ExtDetails extDetails) {
        // 模拟根据学生姓名查询数学成绩
        if ("小明".equals(studentRequest.getStudentName())) {
            return R.ok("98");
        }
```

```
        return R.failed("查询失败，学生信息不存在");
    }

}
```

> **作用**：
>
> 1. **参数结构化**：用户说“帮我查一下小明的数学成绩”，AI 会把 `小明` 转换成 `StudentRequest(studentName="小明")`。
> 2. **自动函数选择**：如果你注册了多个函数（查成绩、查天气、订机票），AI 会根据函数的描述选择调用哪个。
> 3. **结果返回**：函数执行返回结果后，可以再交给 AI 组织成自然语言返回给用户。

👉 Function Calling 就是让 AI 自动帮你把用户的自然语言转成函数调用参数，并调用你注册好的业务方法，返回结果再交给 AI 处理。

3）测试使用

在前端的 AI 联动界面中，您可以使用自然语言进行各种提问，例如"查询小明的数学成绩"、"小明这次考试考得怎么样"或"告诉我小明的成绩"等。系统会自动理解您的意图，调用相应的函数，并准确返回小明的数学成绩。这展示了大模型如何通过函数调用与业务系统无缝集成，将自然语言转化为结构化操作，提升用户交互体验。

4）行业应用案例

教育场景：智能成绩分析

- **业务场景**：家长询问"小明最近三次数学考试的平均分是多少？"
- **函数角色**：成绩分析函数自动解析时间范围、学科类型、统计维度
- **实现方式**：`ScoreAnalysisFunction` 对接学校考试系统，动态计算指定维度的学习趋势
- **业务价值**：减少教师30%的重复查询工作，提升家长服务响应速度至秒级

电商场景：订单状态追踪

- **业务场景**：用户提问"我昨天买的手机发货了吗？订单号23456"
- **函数角色**：`OrderQueryFunction` 自动提取订单号，对接ERP系统实时查询
- **实现方式**：通过JWT token验证用户身份，确保订单数据隔离
- **业务价值**：降低客服中心50%的常规查询量，提升用户自助服务率

医疗场景：检查报告解读

- **业务场景**：患者询问"上周三做的血常规报告有什么异常指标？"
- **函数角色**：`MedicalReportFunction` 对接HIS系统，智能标记异常数据
- **实现方式**：结合医学知识图谱进行指标关联分析
- **业务价值**：缩短患者报告解读等待时间从2天到实时响应

5）最佳实践

- **精准功能定义**：每个函数应聚焦单一业务场景，如`PaymentStatusFunction`只处理支付状态查询
- **安全隔离**：通过Spring Security实现函数级权限控制，敏感操作强制二次验证
- **性能优化**：对高频函数（如订单查询）采用Redis缓存+批量查询策略
- **监控体系**：通过Prometheus监控函数调用耗时、成功率等关键指标

> **重要安全规范**：所有业务函数必须实现权限校验方法`checkParams()`，因为大模型返回的数据不一定稳定！

📌 **核心要义**：函数调用不是简单的API封装，而是构建以业务目标为导向的智能决策中枢，通过持续的场景挖掘和效果反馈，实现AI能力的螺旋式进化

工作流

1、Dify工作流

1.工作流

基本介绍

工作流通过将复杂的任务分解成较小的步骤（节点）降低系统复杂度，减少了对提示词技术和模型推理能力的依赖，提高了 LLM 应用面向复杂任务的性能，提升了系统的可解释性、稳定性和容错性。Dify 工作流分为两种类型：

- **Chatflow**：面向对话类情景，包括客户服务、语义搜索、以及其他需要在构建响应时进行多步逻辑的对话式应用程序。
- **Workflow**：面向自动化和批处理情景，适合高质量翻译、数据分析、内容生成、电子邮件自动化等应用程序。

为解决自然语言输入中用户意图识别的复杂性，Chatflow 提供了问题理解类节点。相对于 Workflow 增加了 Chatbot 特性的支持，如：对话历史（Memory）、标注回复、Answer 节点等。为解决自动化和批处理情景中复杂业务逻辑，工作流提供了丰富的逻辑节点，如代码节点、IF/ELSE 节点、模板转换、迭代节点等，除此之外也将提供定时和事件触发的能力，方便构建自动化流程。

常见案例

- 客户服务

通过将 LLM 集成到你的客户服务系统中，你可以自动化回答常见问题，减轻支持团队的工作负担。 LLM 可以理解客户查询的上下文和意图，并实时生成有帮助且准确的回答。

- 内容生成

无论你需要创建博客文章、产品描述还是营销材料，LLM 都可以通过生成高质量内容来帮助你。只需提供一个大纲或主题，LLM将利用其广泛的知识库来制作引人入胜、信息丰富且结构良好的内容。

- 任务自动化

可以与各种任务管理系统集成，如 Trello、Slack、Lark、以自动化项目和任务管理。通过使用自然语言处理，LLM 可以理解和解释用户输入，创建任务，更新状态和分配优先级，无需手动干预。

- 数据分析和报告

可以用于分析大型知识库并生成报告或摘要。通过提供相关信息给 LLM，它可以识别趋势、模式和洞察力，将原始数据转化为可操作的智能。对于希望做出数据驱动决策的企业来说，这尤其有价值。

- 邮件自动化处理

LLM 可以用于起草电子邮件、社交媒体更新和其他形式的沟通。通过提供简要的大纲或关键要点，LLM 可以生成一个结构良好、连贯且与上下文相关的信息。这样可以节省大量时间，并确保你的回复清晰和专业。

2.关键概念

- 节点：**节点是工作流的关键构成**，通过连接不同功能的节点，执行工作流的一系列操作。
- 变量：**变量用于串联工作流内前后节点的输入与输出**，实现流程中的复杂处理逻辑，包含系统变量、环境变量和会话变量。
- 对话流Chatflow：面向对话类情景，包括客户服务、语义搜索、以及其他需要在构建响应时进行多步逻辑的对话式应用程序。该类型应用的特点在于支持对生成的结果进行多轮对话交互，调整生成的结果。常见的交互路径：给出指令 → 生成内容 → 就内容进行多次讨论 → 重新生成结果 → 结束
- 工作流Workflow：面向自动化和批处理情景，适合高质量翻译、数据分析、内容生成、电子邮件自动化等应用程序。该类型应用无法对生成的结果进行多轮对话交互。常见的交互路径：给出指令 → 生成内容 → 结束

应用类型差异

1. End 节点属于 Workflow 的结束节点，仅可在流程结束时选择。
2. Answer 节点属于 Chatflow ，用于流式输出文本内容，并支持在流程中间步骤输出。
3. Chatflow 内置聊天记忆（Memory），用于存储和传递多轮对话的历史消息，可在 LLM 、问题分类等节点内开启，Workflow 无 Memory 相关配置，无法开启。
4. Chatflow 的开始节点内置变量包括：`sys.query`，`sys.files`，`sys.conversation_id`，`sys.user_id`。Workflow 的开始节点内置变量包括：`sys.files`，`sys.user_id`，详见变量。

2、Coze工作流

工作流是一系列可执行指令的集合，用于实现业务逻辑或完成特定任务。它为应用/智能体的数据流动和任务处理提供了一个结构化框架。工作流的核心在于将大模型的强大能力与特定的业务逻辑相结合，通过系统化、流程化的方法来实现高效、可扩展的 AI 应用开发。

扣子提供了一个可视化画布，你可以通过拖拽节点迅速搭建工作流。同时，支持在画布实时调试工作流。在工作流画布中，你可以清晰地看到数据的流转过程和任务的执行顺序。

工作流与对话流

扣子提供以下两种类型的工作流：

- 工作流（Workflow）：用于处理功能类的请求，可通过顺序执行一系列节点实现某个功能。适合数据的自动化处理场景，例如生成行业调研报告、生成一张海报、制作绘本等。
- 对话流（Chatflow）：是基于对话场景的特殊工作流，更适合处理对话类请求。对话流通过对话的方式和用户交互，并完成复杂的业务逻辑。对话流适用于 Chatbot 等需要在响应请求时进行复杂逻辑处理的对话式应用程序，例如个人助手、智能客服、虚拟伴侣等。

节点

在使用节点编排工作流时，灵活性和扩展性是实现高效编排的关键。工作流的开始节点、结束节点、输出节点、插件节点、子工作流节点、代码节点、SQL 自定义节点、新增数据节点、查询数据节点、更新数据节点、删除数据节点、问答节点、批处理节点、循环节点、变量聚合节点、变量节点、选择器节点均支持多种变量类型，包括 String、Integer、Number、Boolean、Object、File 和 Array等。你可以根

据实际需求灵活选择合适的数据类型，而无需额外的数据转换，从而提升工作流编排的灵活性和扩展性。

工作流和对话流差异区别

差异	工作流	对话流
开始节点	预置一个非必填参数，格式为 String，默认命名为 input。	预置了以下必选参数： **USER_INPUT**：用户在对话流中输入的原始内容。 **CONVERSATION_NAME**：对话流绑定的会话。
大模型节点	不支持对话历史。	支持读取对话历史，会话中的上下文会和用户问题一起传递给大模型。
意图识别节点	不支持对话历史。	支持读取对话历史，会话中的上下文会和用户问题一起传递给大模型。

1.节点使用

1）基础节点

- 开始节点：开始节点中默认有一个输入参数 input，表示用户在本轮对话中输入的原始内容，支持多种输入数据类型
- 结束节点：结束节点是工作流的最终节点，用于返回工作流运行后的结果。结束节点支持两种返回方式，即返回变量和返回文本
    - 返回变量：工作流运行结束后会以 JSON 格式输出所有返回参数
    - 返回文本：工作流运行结束后，智能体中的模型将直接使用指定的内容回复对话，回答内容中支持引用输出参数，也可以设置流式输出
- 大模型节点：大模型节点可以调用大型语言模型，根据输入参数和提示词生成回复，通常用于执行文本生成任务，例如文案制作、文本总结、文章扩写等
- 插件节点：插件节点用于在工作流中调用插件运行指定工具，插件是一系列工具的集合，每个工具都是一个可调用的 API
- 工作流节点：实现工作流嵌套工作流的效果

2）业务逻辑节点

- 代码节点：支持通过编写代码来生成返回值，支持 JavaScript 和 Python 两种语言
- 选择器节点：是一个 if-else 节点，用于设计工作流内的分支流程
- 意图识别节点：意图识别节点能够让智能体识别用户输入的意图，并将不同的意图流转至工作流不同的分支处理，提高用户体验，增强智能体的落地效果
- 循环节点：用于重复执行一系列任务，直到满足某个条件为止
- 批处理节点：批处理节点适用于大量数据并行处理的场景。相对于添加多个相同的节点执行任务，批处理节点的效率更高。
- 变量聚合节点：将多路分支的输出变量整合为一个，方便下游节点统一配置。

3）输入输出节点

- 输入节点：在比较复杂的工作流场景中，某些节点的执行往往需要额外的用户输入。如果上游节点中没有获取到这些信息，你可以添加一个输入节点来主动收集信息。工作流执行到输入节点时会暂时中断，直到此节点收集到必要的用户输入
- 输出节点：通常情况下，工作流会在执行完毕后通过结束节点输出最终的执行结果。当工作流处理流程较长、运行时间较久时，开发者可以在工作流中添加输出节点，临时输出一段消息，避免用户等待时间过长、放弃对话。例如提示用户任务正在执行中，建议用户耐心等待。

4）数据库节点

- SQL自定义节点：用于对指定数据库进行常见的 SQL 操作

5）知识和数据节点

- 变量赋值节点
- 知识库写入节点
- 知识库检索节点
- 长期记忆节点

6）图像处理节点

- 图像生成节点
- 画板节点

7）图像处理插件节点

8）音视频处理节点

- 视频生成节点
- 视频提取音频节点
- 视频抽帧节点

9）组件节点

- HTTP请求节点
- 问答节点
- 文本处理节点
- JSON序列化节点
- JSON反序列化节点

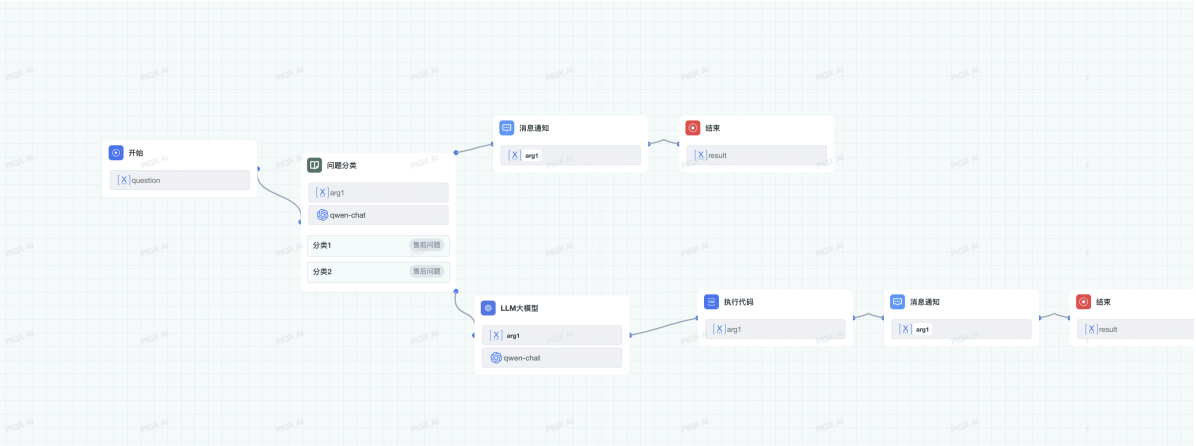
10）触发器节点

11）会话管理节点

12）会话历史节点

13）消息节点

3、PIGX自定义工作流节点



本手册以 MCP 节点为例，详细说明如何在 PigX 工作流系统中新增一个自定义节点类型。工作流节点是系统的核心组件，每个节点负责执行特定的业务逻辑。

AI Flow 架构设计

组件	说明
AiNodeProcessor 接口	定义节点处理器的基本接口，所有节点处理器必须实现此接口
AbstractNodeProcessor 抽象基类	提供通用的节点处理逻辑和工具方法，减少重复代码
MCPNodeProcessor	MCP节点处理器实现
LLMNodeProcessor	LLM节点处理器实现
其他节点处理器	实现特定类型节点的业务逻辑

节点开发快速指南

步骤	操作	说明
1	定义节点类型常量	在 `NodeTypeConstants.java` 中添加新的节点类型标识符
2	创建节点配置类	定义节点所需的配置参数结构
3	扩展节点定义	在 `AiNodeDefinition.java` 中添加新节点的配置字段
4	实现节点处理器	创建具体的节点处理器类，继承 `AbstractNodeProcessor`
5	注册节点处理器	通过 Spring 注解完成自动注册

🔧 详细实现步骤

第一步：定义节点类型常量

在 `NodeTypeConstants.java` 中添加新的节点类型：

```
// 文件位置: pigx-knowledge/src/main/java/com/pig4cloud/pigx/knowledge/support/flow/constants/NodeTypeConstants.java

/**
 * MCP节点
 * 用于调用MCP (Model Context Protocol) 服务的节点类型
 */
String MCP = "mcp";
```

第二步：创建节点配置类

为了接收前端节点配置参数，我们需要创建相应的配置类来定义节点的参数结构。以MCP节点为例：

```
// 文件位置: pigx-knowledge/src/main/java/com/pig4cloud/pigx/knowledge/support/flow/model/nodes/AiMCPNode.java

package com.pig4cloud.pigx.knowledge.support.flow.model.nodes;

import lombok.Data;

/**
 * MCP节点配置
 * 用于定义MCP (Model Context Protocol) 节点的配置参数
 */
@Data
public class AiMCPNode {
    private String mcpId;
    private String mcpName;
    private String prompt;
}
```

第三步：扩展节点定义

在 `AiNodeDefinition.java` 中添加新节点的配置字段，使系统能够识别和处理新的节点类型。

```
// 文件位置: pigx-knowledge/src/main/java/com/pig4cloud/pigx/knowledge/support/flow/model/AiNodeDefinition.java

/**
 * MCP节点参数配置
 */
private AiMCPNode mcpParams;
```

第四步：实现节点处理器

```
// 文件位置: pigx-knowledge/src/main/java/com/pig4cloud/pigx/knowledge/support/flow/model/processor/MCPNodeProcessor.java

/**
 * MCP节点处理器
 * 负责处理流程中的MCP (Model Context Protocol) 节点，直接调用McpChatRule避免代码重复
 *
 * @author lengleng
 * @date 2025/03/22
 */
@Slf4j
@Component(NodeTypeConstants.MCP)  // 重要：使用节点类型常量作为Spring Bean名称
@RequiredArgsConstructor
public class MCPNodeProcessor extends AbstractNodeProcessor {

    /**
     * 模板引擎，用于变量替换
     */
    public static final TemplateEngine engine = TemplateUtil.createEngine(new TemplateConfig());

    private final McpChatRule mcpChatRule;

    @Override
    protected Dict doExecute(AiNodeDefinition node, FlowContextHolder context) {
        try {
            // 验证节点配置
            AiMCPNode config = validateNodeConfig(node);

            // 获取输入参数并处理消息模板
            Dict variables = getInputVariables(node, context);
            String inputMessage = engine.getTemplate(config.getPrompt()).render(variables);

            // 构建ChatMessageDTO
            ChatMessageDTO chatMessageDTO = buildChatMessageDTO(config, inputMessage, context);

            // 调用McpChatRule处理MCP调用
            return processMcpCallViaRule(chatMessageDTO, context);

        } catch (Exception e) {
            throw FlowException.nodeError(node.getId(), "[MCP节点] -> " + e.getMessage());
        }
    }

    /**
     * 验证节点配置
     */
    private AiMCPNode validateNodeConfig(AiNodeDefinition node) {
        AiMCPNode config = node.getMcpParams();
```

```
        if (config == null) {
            throw FlowException.invalidParam("MCP节点配置无效");
        }

        if (StrUtil.isBlank(config.getMcpId())) {
            throw FlowException.invalidParam("MCP配置ID不能为空");
        }

        if (StrUtil.isBlank(config.getPrompt())) {
            throw FlowException.invalidParam("提示模板不能为空");
        }

        return config;
    }

    /**
     * 构建聊天消息DTO
     */
    private ChatMessageDTO buildChatMessageDTO(AiMCPNode config, String
inputMessage, FlowContextHolder context) {
        return ChatMessageDTO.builder()
            .mcpId(config.getMcpId())
            .message(inputMessage)
            .conversationId(context.getParameters().getStr("conversationId"))
            .userId(context.getParameters().getStr("userId"))
            .build();
    }

    /**
     * 通过规则处理MCP调用
     */
    private Dict processMcpCallViaRule(ChatMessageDTO chatMessageDTO,
FlowContextHolder context) {
        // 调用现有的MCP聊天规则
        String result = mcpChatRule.execute(chatMessageDTO);

        // 返回标准格式的结果
        return Dict.create()
            .set(FlowConstant.CONTENT, result)
            .set(FlowConstant.ROLE, "assistant")
            .set(FlowConstant.TIMESTAMP, System.currentTimeMillis());
    }
}
```

第五步：自动注册机制

节点处理器通过 Spring 的组件扫描机制自动注册，关键要点：

要点	说明
Bean 名称规范	必须使用节点类型常量 `@Component(NodeTypeConstants.MCP)`
自动发现机制	`AiNodeProcessorFactory` 会自动发现所有实现

```java
@Component(NodeTypeConstants.MCP)  // Bean名称 = "mcp"
public class MCPNodeProcessor extends AbstractNodeProcessor {
    // 实现内容...
}
```

📋 开发规范指南

1. 类结构规范

标准类结构模板：

```java
@Slf4j
@Component(NodeTypeConstants.XXX)  // 使用节点类型常量
@RequiredArgsConstructor
public class XXXNodeProcessor extends AbstractNodeProcessor {

    // 依赖注入的服务
    private final SomeService someService;

    // 核心执行方法
    @Override
    protected Dict doExecute(AiNodeDefinition node, FlowContextHolder context) {
        // 1. 参数验证
        // 2. 业务逻辑执行
        // 3. 结果处理
        // 4. 返回标准格式
    }

    // 私有辅助方法
    private void validateConfig() { }
    private void processData() { }
}
```

2. 配置验证规范

每个节点处理器都应该包含严格的配置验证逻辑，确保运行时的稳定性。

```java
private AiXXXNode validateNodeConfig(AiNodeDefinition node) {
    AiXXXNode config = node.getXxxParams();
    if (config == null) {
        throw FlowException.invalidParam("XXX节点配置无效");
    }

    // 具体参数验证
    if (StrUtil.isBlank(config.getRequiredParam())) {
        throw FlowException.invalidParam("必要参数不能为空");
    }

    return config;
}
```

3. 返回结果规范

所有节点处理器必须返回统一格式的 Dict 对象，确保系统的一致性。

```
return Dict.create()
    .set(FlowConstant.CONTENT, resultContent)      // 主要内容
    .set(FlowConstant.ROLE, "assistant")           // 角色标识
    .set(FlowConstant.TOKENS, tokenCount)          // 令牌使用量
    .set(FlowConstant.TIMESTAMP, System.currentTimeMillis());  // 时间戳
```

🛠️ 工具方法说明

AbstractNodeProcessor 提供的工具方法

方法	功能	用途	返回值
getInputVariables(node, context)	获取节点输入参数	从上下文中提取当前节点需要的输入变量	Dict 对象，包含所有输入变量的键值对
getOutputVariables(node, result)	处理节点输出参数	将节点执行结果按照配置输出到上下文	无返回值，直接更新上下文
addQueryParam(url, name, value)	添加URL查询参数	构建带参数的URL请求	拼接好参数的完整URL

FlowContextHolder 上下文方法

方法类型	方法名	说明
变量管理	`getParameters()`	获取流程执行参数
变量管理	`getVariables()`	获取变量映射表
变量管理	`setVariable(key, value)`	设置变量值
变量管理	`getVariable(key)`	获取指定变量值
执行配置	`getAiFlowExecuteDTO()`	获取执行配置对象
执行配置	`getUserId()`	获取当前用户ID
执行配置	`getConversationId()`	获取会话ID

🌊 流式处理支持

对于需要实时响应的节点（如AI对话、长时间计算等），系统支持流式处理机制。

流式数据发送

```
// 检查是否支持流式回调
if (context.getAiFlowExecuteDTO() != null &&
    context.getAiFlowExecuteDTO().getCallback() != null) {

    // 发送流式数据
```

```
    context.getAiFlowExecuteDTO()
        .getCallback()
        .execute(AiFlowExecuteDTO.FlowCallbackResult.builder()
            .data(AiFlowExecuteDTO.FlowCallbackData.builder()
                .content(content)              // 内容片段
                .nodeId(node.getId())          // 节点ID
                .timestamp(System.currentTimeMillis())  // 时间戳
                .build())
            .build());
}
```

向量数据库

相关面试题

1、说一下向量数据库?

向量数据库是一种专门设计用来存储和管理向量嵌入(vector embeddings)的数据库系统。它可以将非结构化数据(如文本、图片、音频等)转换成高维向量的形式进行存储，并提供高效的相似性搜索功能。

在基于大模型的应用开发中，向量数据库主要解决以下核心问题：

1）高效的相似性搜索

通过将用户查询转换为向量，可以快速找到语义相似的内容，这对于实现智能问答、推荐系统等功能至关重要。

2）海量数据处理

能够高效处理大模型生成的海量数据，传统数据库难以处理百万甚至数十亿的数据点，而向量数据库专门针对这种场景进行了优化。

3）实时交互支持

在需要实时用户交互的应用中(如聊天机器人)，向量数据库可以确保快速检索相关上下文信息，提供实时响应。

拓展知识

1）向量数据库工作原理

2）与传统数据库的区别

传统数据库采用行列结构存储数据，主要用于精确匹配查询。而向量数据库针对高维向量数据优化，支持近似最近邻(ANN)搜索算法，更适合语义相似性搜索。

3）在大模型应用中的具体应用场景

文本理解：将文档内容向量化，实现语义搜索和文档相似度分析。

图像处理：存储图像特征向量，支持以图搜图等功能。

个性化推荐：基于用户行为向量，实现精准的内容推荐。

4）主流向量数据库解决方案

FAISS：Facebook开发的向量检索库 Milvus：开源的向量数据库系统 Annoy：Spotify开发的近似最近邻搜索库

5）性能优化考虑

索引技术：使用高效的索引方法提升搜索性能 查询优化：采用低延迟的查询处理机制 扩展性：支持分布式部署和水平扩展

2、了解哪些向量数据库？如何选型？

常见的向量数据库有 **Milvus、Pinecone、Weaviate、Qdrant、Chroma、Faiss、Annoy** 等。

在选型时需综合考虑 **功能特性、性能表现、成本预算、扩展性、团队技术栈** 等因素。例如，Milvus 开源且支持大规模分布式部署，适合企业级应用；Pinecone 是全托管服务，便于快速部署但成本较高；Chroma 轻量级，适合小规模数据场景快速验证。

主流向量数据库详解

- **Milvus**：开源国产，支持 TB 级向量的增删改和近实时查询，采用分布式架构，适合推荐系统、图像检索、NLP 等场景。例如电商平台中处理海量商品描述的向量检索。
- **Pinecone**：全托管式商业服务，开箱即用，高并发性能好，适合实时搜索、快速原型开发，但按使用量付费，成本较高。
- **Weaviate**：支持文本、图像等多模态数据，内置语义搜索，适合知识图谱、智能问答，但复杂查询时延迟较高。
- **Qdrant**：支持向量与元数据联合搜索，过滤和排序灵活，轻量级部署，适合推荐系统中元数据（如价格、品类）与向量结合的复杂查询。
- **Chroma**：专注嵌入式向量存储，支持本地化部署，Python SDK 集成简单，适合中小规模数据（如小型知识库项目）。
- **Faiss**：高效的相似性搜索库，支持 CPU/GPU 计算，适合研究型场景或对速度要求极高的任务，但安装依赖复杂，不支持元数据存储。
- **Annoy**：基于近似最近邻（ANN）的搜索库，适合大规模数据集的快速检索，常用于搜索引擎底层优化。

其他支持向量搜索的数据库

上面是专用的向量数据库，除此之外还有一些**支持向量搜索的开源数据库**。

例如：OpenSearch、PostgreSQL、ClickHouse 和 Cassandra。

3、向量数据库原理？

向量数据库的核心原理是通过将高维数据（如图像、文本）转换为多维向量，并基于相似性度量（如余弦相似度、欧氏距离），利用高效的索引结构和近似最近邻（ANN）算法，快速检索与目标最相似的向量结果。

关键三步：

1. 向量化：将非结构化数据转化为数值向量，保留语义或特征信息
2. 索引构建：通过HNSW图、产品量化（PQ）、位置敏感哈希（LSH）等结构预处理向量，加速搜索
3. 近似搜索：允许一定误差，用ANN算法在速度与准确性间平衡，返回Top-K相似结果

流程图：

```
▼text

复制代码原始数据 → 向量化 → 索引构建（HNSW/PQ/LSH） → 输入查询向量 → ANN近似搜索 → 返回Top-K结果
```

4、HNSW、LSH、PQ 分别是什么意思?

它们是向量数据库中三种核心索引与压缩技术，用于加速高维向量的相似性搜索。

HNSW（Hierarchical Navigable Small World）图（分层图结构）

- 在高维空间中，将所有向量组织成分层“小世界”图。
- 查询时，从上层稀疏图贪心跳转到最相似邻居，逐层下探到密集的底层图，快速找到近似最邻近点。

LSH（Locality-Sensitive Hashing）哈希（哈希分桶）

- 利用一组专门设计的哈希函数，把相似向量映射到同一个或相邻的哈希桶。
- 查询时只需查对应桶内的候选项，再做精确排序，极大缩小检索范围。

PQ（Product Quantization）量化（向量量化）

- 将高维向量切分成若干子块，对每块分别做聚类，用“码字”来近似原始子向量。
- 存储时只保存每段的码字索引，检索时通过查预先计算好的子块距离表来快速估算向量间距离。

技术	核心目的	适用场景	优势	代价
HNSW	高精度快速搜索	十亿级数据实时检索	精度高、速度快	内存占用高
LSH	极速粗筛	去重、过滤重复内容	速度极快（O(1)）	精度损失
PQ	压缩存储+加速计算	移动端、内存受限场景	内存/计算量大幅降低	轻微精度损失

扩展知识

- HNSW（Hierarchical Navigable Small World）

  构建多层图结构，每一层都是一个“小世界”网络。上层节点稀疏，像“高速公路”，能快速跳跃式定位大致范围；下层节点密集，像“小路”，用于精细搜索。例如在1000万向量中找相似向量，HNSW可能先通过上层快速跳到某个区域，再通过下层在该区域内精准查找。

  在大规模向量数据（如亿级向量）中，查询速度与精度的平衡表现优秀，是目前许多向量数据库（如Milvus）的常用索引技术。

- LSH（Locality Sensitive Hashing）

  设计特殊的哈希函数，使相似向量以较高概率映射到同一个哈希桶，不相似向量尽量分散到不同桶。查询时，只需搜索查询向量所在桶及相邻桶，而非全部数据。比如在图像去重中，相似图片的向量会被“扔”到同一桶，直接在桶内找重复项。

  适合处理高维数据（如文本、图像特征向量），在推荐系统、图像检索等海量数据近似查询场景中应用广泛。

- PQ（Product Quantization）

  将高维向量（如1024维）拆分成多个低维子向量（如16个64维），对每个子向量集合进行聚类，生成聚类中心。存储时，用聚类中心的编号（而非原始数值）表示向量。例如一个1024维向量，拆成16块后，每块用8位编号表示，大大减少存储空间。

压缩向量数据，降低内存/磁盘占用，同时加速向量间距离计算（只需计算编号对应聚类中心的距离），常用于工业级向量检索系统。

5、ANN是什么?

ANN 是“近似最近邻（Approximate Nearest Neighbor）”的缩写，它不是某一种具体算法，而是一类**通过牺牲一定精确性来换取搜索速度的算法框架**或者技术。

其核心是在海量高维向量中，快速找到与目标向量“近似相似”的结果，而非耗费大量时间寻找绝对精确的最近邻。

因为当数据量达到百万甚至十亿级时，暴力遍历所有向量计算距离（精确搜索）会慢到无法接受，而ANN通过智能索引和近似策略，实现**毫秒级响应**，满足实时搜索需求。

拓展知识

实际上，HNSW、LSH、PQ都是实现 ANN 的具体技术/算法

- HNSW：图结构（Graph-based ANN）
- LSH：哈希分桶（Hash-based ANN）
- PQ：向量量化（Quantization-based ANN）

可以总结以下一些实现都是 ANN 家族：

```
▼text复制代码ANN
├─ Graph-based              (例如：HNSW、NSG)
├─ Hash-based               (例如：LSH、Multi-Probe LSH)
├─ Tree-based               (例如：KD-Tree 变种、RP-Tree)
├─ Quantization-based       (例如：PQ、OPQ、IVF+PQ)
└─ 其他混合型/自研方案      (例如：ScaNN、ANNOY、NMSLIB…)
```

换句话说，ANN→【HNSW、LSH、PQ、KD-Tree变种、IVF+PQ...】→最终实现近似最近邻搜索。

6、常见向量搜索算法?

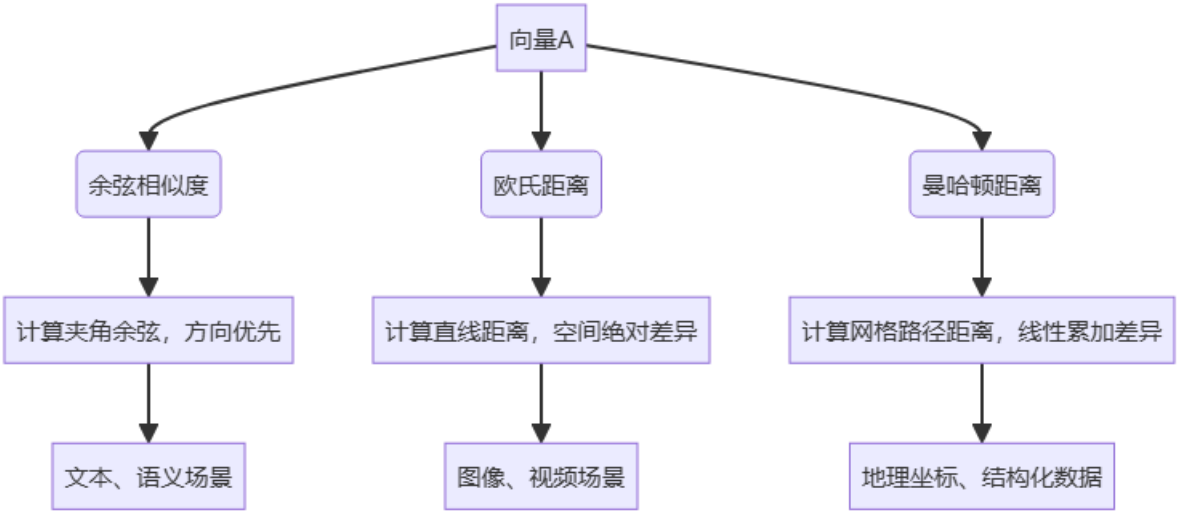
1. **余弦相似度**：衡量两个向量的“方向相似性”，不关心向量长度，取值范围[-1, 1]，值越大方向越接近（比如“猫”和“狗”的文本向量）。
2. **欧几里得距离**（欧氏距离）：计算两个向量在空间中的“直线距离”，取值≥0，距离越小向量越相似（比如两张图片的像素特征向量）。
3. **曼哈顿距离**：计算两个向量在各维度上的差值绝对值之和，类似“在城市街区中沿道路行走的距离”，取值≥0，适用于网格状数据（如地图坐标）。

简单来看：余弦比“方向”，欧氏比“绝对距离”，曼哈顿比“线性累加距离”。

因此：

- 文本、推荐系统常用**余弦相似度**（不关心文本长度，只看语义方向）
- 图像、视频检索常用**欧氏距离**（直接比较像素特征的空间差异）
- 网格数据（如城市坐标、表格数据）常用**曼哈顿距离**（符合实际移动逻辑，稀疏数据的处理）

三种方法的几何直观对比：



7、向量数据库工作流程?

向量数据库的工作流程可拆解为五步，核心是将非结构化数据转化为可计算、可检索的向量形式：

1. 数据处理：清洗数据（去噪、归一化）、标注元数据（如标签、时间）。例如，处理电商评论时，需过滤乱码、特殊符号，保留有效文本内容，为向量化做准备。
2. 向量化：用AI模型（如BERT、ResNet）提取特征，生成高维向量，向量维度的每个数值反映词语间的语义关系。
3. 向量存储：将向量与原始数据关联，存入分布式存储（如分块存储）
4. 索引构建：用HNSW、LSH等技术组织向量，建立高效检索结构
5. 相似性检索：输入目标向量，计算余弦相似度，通过索引快速返回Top-K近似结果

框架

1、A2A

A2A相关面试题

1、什么是 A2A 协议，它的核心架构及主要组件有哪些?

A2A（Agent-to-Agent）协议是由 Google 主导开发的开放协议，旨在实现不同 AI 智能体之间的互操作性，使它们能够跨平台、跨框架地协作。其核心架构包括以下主要组件：

大模型的 Agent2Agent（A2A）协议是一个开放的标准协议，旨在实现不同 AI 代理之间的互联互通。其核心架构包括以下主要组件：

1）**Agent Card（代理卡片）**：一个公开的 JSON 文件，描述代理的能力、技能、端点 URL 和身份验证要求，用于客户端发现代理。

2）**A2A Server（A2A 服务器）**：实现 A2A 协议方法的 HTTP 端点，接收请求并管理任务执行。

3）**A2A Client（A2A 客户端）**：发送请求（如 `tasks/send`）到 A2A 服务器的应用程序或其他代理。

4）**Task（任务）**：客户端发起的工作单元，具有唯一的 ID，并通过状态（如 `submitted`、`working`）进行跟踪。

5）**Message（消息）**：客户端与代理之间的通信回合，包含 `Parts`。

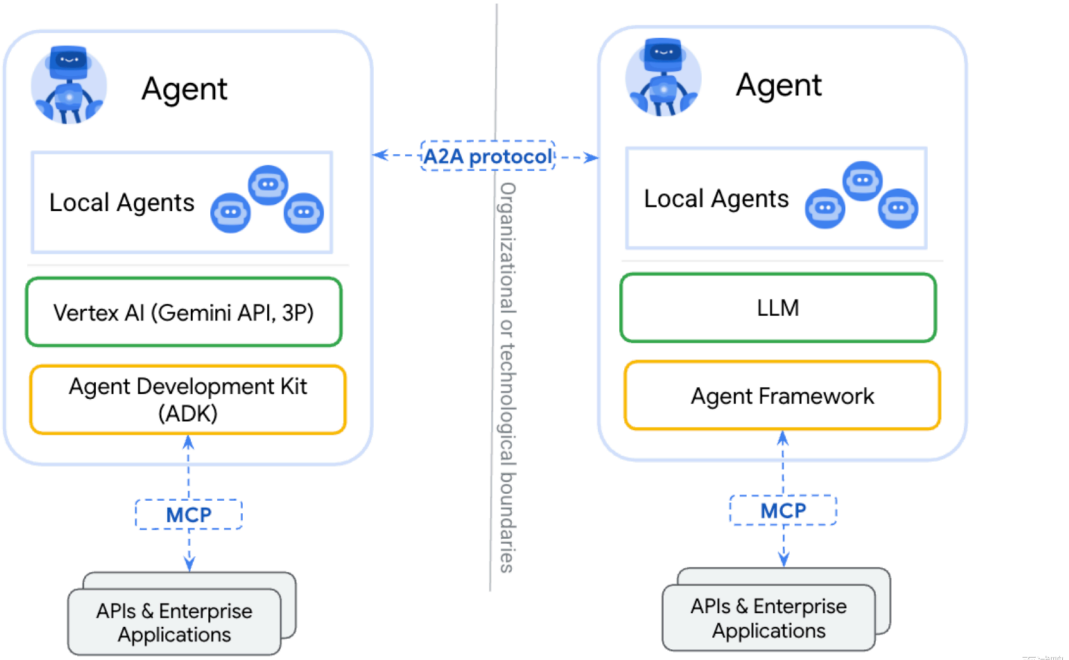
6）**Part（部分）**：消息或工件中的基本内容单元，可以是文本、文件或结构化 JSON。

7）**Artifact（工件）**：代理在任务中生成的输出（如生成的文件、最终的结构化数据）。

8）**Streaming（流式传输）**：对于长时间运行的任务，支持使用 `tasks/sendSubscribe`，客户端通过服务器发送事件（SSE）接收任务状态更新。

9）**Push Notifications（推送通知）**：支持的服务器可以主动将任务更新发送到客户端提供的 webhook URL。

A2A 协议采用 JSON-RPC 2.0 通过 HTTP 进行消息交换，对于流式传输，使用 SSE 协议。



扩展知识

A2A 协议的设计旨在解决企业中不同框架和供应商构建的代理之间的协作问题。通过标准化的通信方式，A2A 使得不同的 AI 代理能够跨平台、跨供应商地协作，提升了多代理系统的效率和可维护性。

与 Anthropic 的 Model Context Protocol（MCP）相比，A2A 更侧重于代理之间的直接通信和任务协调，而 MCP 提供上下文信息和工具支持。这两者可以互补，共同推动企业级 AI 代理的协作能力。

在实际应用中，A2A 协议可以用于构建模块化的 AI 系统，使得不同功能的代理能够独立开发和部署，同时又能通过标准化的协议进行协作。例如，在一个企业的客户服务系统中，查询代理、推荐代理和处理代理可以通过 A2A 协议协同工作，共同完成复杂的客户服务任务。

2、A2A 协议有哪五大设计原则?

A2A 是一种开放协议，为代理之间的协作提供了一种标准方式，与底层框架或供应商无关。协议遵循以下几个核心原则：

- **拥抱 Agentic 能力。**A2A 专注于使 agent 能够以自然、非结构化的方式进行协作，即使它们不共享内存、工具和上下文。我们正在实现真正的 multi-agent 场景，而不会将 agent 限制为 “工具”。谷歌正在启用真正的多 Agent 场景，而不是限制 Agent 成为一个工具。
- **建立在现有标准之上。**该协议建立在现有的流行标准之上，包括 HTTP、SSE、JSON-RPC，这意味着它更容易与企业日常使用的现有 IT 栈集成。
- **默认安全。**A2A 旨在支持企业级身份验证和授权，在发布时与 OpenAPI 的身份验证方案具有同等效力。

- **支持长时间运行的任务。**我们设计了 A2A，使其具有灵活性，并支持从快速任务到深度研究的各种场景，当人类处于循环中时，这些场景可能需要数小时甚至数天才能完成。在整个过程中，A2A 可以向用户提供实时反馈、通知和状态更新。
- **模态无关。**代理世界不仅限于文本，这就是为什么我们设计了 A2A 来支持各种模态，包括音频和视频流。

在多智能体生态系统中，不同厂商和框架的 Agent 常常形成“信息孤岛”，彼此难以协作。Google 发布的 A2A（Agent2Agent）协议正是为了解决这个问题，它得到包括 Salesforce、SAP、MongoDB、PayPal 在内的 50 多家技术公司和咨询公司的支持与贡献，目标是打造一个跨平台、跨供应商的智能体“自由贸易区”，让各路 Agent 能像使用同一种语言一样互通有无。

拥抱智能体原生能力这条原则就好比在不同国家之间制定统一的外交礼仪：每个 Agent 就像一个国家的外交官，自带名牌（能力清单），在没有共享后端的情况下，也能通过统一协议介绍自己、协商任务、交换结果。这样做不仅保证了灵活性，也让系统更具弹性，不再被某个中心化组件拖累。

基于现有标准构建，则是为了让工程落地时不再成为“烧脑”的大迁移项目。A2A 利用 HTTP 的通用性、SSE 的实时推送和 JSON-RPC 的调用规范，让后端开发和运维都能在现有架构上无缝接入，成本最低也最可靠。

默认安全则将企业级的认证与授权直接内嵌进协议，就像银行级别的安全门槛从一开始就设好了，不需要后续再另外打补丁。无论是身份验证、令牌管理，还是访问控制，都与 OpenAPI 的最佳实践保持同步，确保各环节受保护。

支持长时任务是考虑到很多复杂流程并非一锤子买卖，例如科研建模、供应链优化，这类任务可能持续数小时至数天。A2A 在协议层面定义了任务生命周期和状态更新机制，让用户能够随时查看进度、接收中间产出，而非盲目等待。

模态无关的设计理念则是顺应智能世界日益多元的趋势。除了最基础的文本消息，A2A 还原生支持音频和视频流传输，未来甚至可以扩展到图像、传感器数据等更多形式，让虚拟助理、客服、物联网设备等都能用同一套协议协作。

3、A2A协议的工作原理?

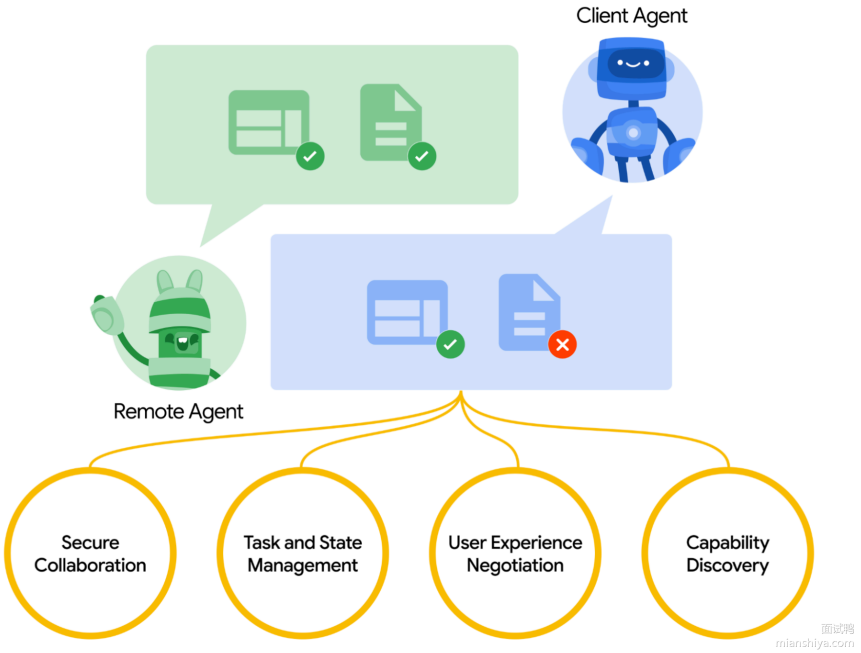
A2A 协议的核心作用是让“客户端代理”和“远程代理”之间顺利沟通。客户端代理负责创建并下发任务，而远程代理则根据这些任务提供信息或执行操作。整个过程中，A2A 提供了几个关键功能：

能力发现：每个代理都有一张“代理卡”，以 JSON 格式描述它能做什么。客户端代理可以通过这些卡片，找到最适合执行当前任务的远程代理，并通过 A2A 协议与之建立通信。

任务管理：客户端与远程代理的交互围绕任务展开。每个任务都有一个由协议定义的“任务对象”，并且具有生命周期。有些任务可以立刻完成，而对于运行时间较长的任务，代理之间可以持续沟通，保持状态同步，确保任务按预期推进。任务完成后会产生一个“工件”，也就是最终的执行结果。

协作能力：代理之间可以直接通信，发送包含上下文、回复、工件或用户指令的消息，便于协同完成更复杂的任务。

用户体验协商：每条消息可以包含多个“部分”，每部分代表一个完整的内容片段，比如生成的图像。每个部分都有明确的内容类型，客户端和远程代理可以据此协商最合适的展示格式，同时也可以决定是否支持如 iframe、视频、网页表单等用户界面功能，从而根据用户需求和设备能力，提供更好的使用体验。



扩展知识

A2A 的工作原理，其实就是帮助客户端 Agent 和远程 Agent 之间顺畅沟通、协同完成任务。客户端 Agent 负责创建任务并把任务传达出去，远程 Agent 接收到任务后，按照指令提供信息或执行操作。整个过程中，A2A 协议提供了几项关键能力来支撑这套机制。

首先，每个 Agent 都会通过一张叫做“Agent 卡”的信息卡来展示自己的能力。这张卡是 JSON 格式的，里面详细说明了这个 Agent 擅长做什么。客户端 Agent 就可以根据这些卡片，挑选最适合当前任务的远程 Agent。

选好远程 Agent 后，客户端 Agent 会用 A2A 协议把任务分配过去。任务的处理过程也是协议中非常重要的一部分。A2A 规定了任务的整个生命周期：有的任务很简单，收到就能立刻完成；而有些任务比较复杂、耗时较长，双方可以不断通信，保持进度同步。任务完成后，会产生一个“工件”，也就是任务的最终输出结果。

除了分配任务，A2A 还支持多个 Agent 之间的协作。它们可以互相发送消息，这些消息可能包含上下文信息、回复内容、工件，或者新的用户指令。通过这种方式，多个 Agent 能配合得更好，处理一些更复杂的需求。

最后，A2A 还考虑到了用户体验。每条消息可以由多个“部分”组成，每个部分都是一个完整的内容片段，比如一张图片或一段文本。每部分还带有明确的内容类型，这样客户端和远程 Agent 就可以协商使用什么格式展示内容，甚至支持 iframe、视频或网页表单等不同的展示方式。通过这种机制，A2A 能根据不同设备和用户的需求，提供更加合适、流畅的体验。

4、A2A协议的工作流程?

1）发现：客户端向 `/.well-known/agent.json` 发起 HTTP GET 请求，获取远端智能体的 Agent Card，其中包含智能体的唯一标识、能力清单、回调 URL、认证方式等元数据，帮助客户端快速了解并筛选合适的智能体来执行任务。
2）启动：客户端根据业务需求生成唯一的 Task ID，然后通过 JSON-RPC 调用 `tasks/send`（用于一次性请求，同步返回最终 Task 对象）或 `tasks/sendSubscribe`（用于订阅式请求，服务端通过 Server-Sent Events 推送状态更新）向目标智能体发送任务请求。
3）处理：远端智能体接收请求后，将任务状态从 `submitted` 切换到 `working`，在内部执行模型推理或外部工具调用；对于订阅式任务，会持续推送 `TaskStatusUpdateEvent` 和可选的 `TaskArtifactUpdateEvent`，让客户端实时获取进度或中间成果。
4）交互：当智能体在处理过程中需要额外输入时，会发出 `input-required` 状态更新，并携带一条

`Message` 请求；客户端收到后可使用相同 Task ID 通过 `tasks/send` 或 `tasks/sendSubscribe` 补充用户输入，保持会话的连续性和上下文一致。
5）完成：任务进入终态（例如 `completed`、`failed` 或 `canceled`），客户端可以选择主动拉取最终的 Task 对象，也可以继续通过 SSE 或 Webhook 机制订阅 `TaskStatusUpdateEvent`，并获取以 JSON Artifact 形式封装的最终结果。

扩展知识

1）底层技术：A2A 协议基于 HTTP(S) 和 JSON-RPC 2.0 构建，不依赖 gRPC 或专有传输协议，便于穿透常见防火墙，最大化兼容现有 Web 基础设施。
2）Agent Card 规范：Agent Card 遵循 JSON Schema，常见字段包括 `id`、`url`、`description`、`version`、`capabilities`（能力列表）、`auth`（认证需求）等，客户端可根据这些字段动态过滤和排序候选智能体
3）安全机制：协议支持多种认证方式，如 OAuth 2.0 Bearer Token、mTLS、API Key 或签名 JWT，消息层面也可选用 HTTPS/TLS 加密和消息签名，确保整个任务流转过程的机密性和可审计性。
4）错误与重试：A2A 定义了标准化的错误事件和状态码（ErrorEvent），并内置重试策略——客户端可依据错误类型自动重试或回退，保证在网络抖动或服务不稳定时仍能平滑恢复。
5）多模态支持：除了文本，协议中的 `parts` 概念允许传输图片、音频、表单、文件等多种内容类型，为多模态协同（比如医疗影像诊断、供应链可视化）提供原生支持。
6）开源生态：Google 在 GitHub 上维护 A2A 规范及示例仓库，提供多语言 SDK（Python、TypeScript/JavaScript）、示例 Agent 和 Orchestrator，以及在线文档；截止目前已有 1.2 万以上星标，社区贡献活跃。
7）未来演进：后续版本计划引入双向音视频流支持、更细粒度的超时与 QoS 控制、多方协作（超过两方的智能体网络交互）等特性，进一步推动异构智能体生态的落地与演化。

5、A2A协议和MCP协议关系?

在智能体与外部系统之间实现互操作，标准化协议至关重要。它主要聚焦两个密切关联的领域：**工具**与**智能体**。

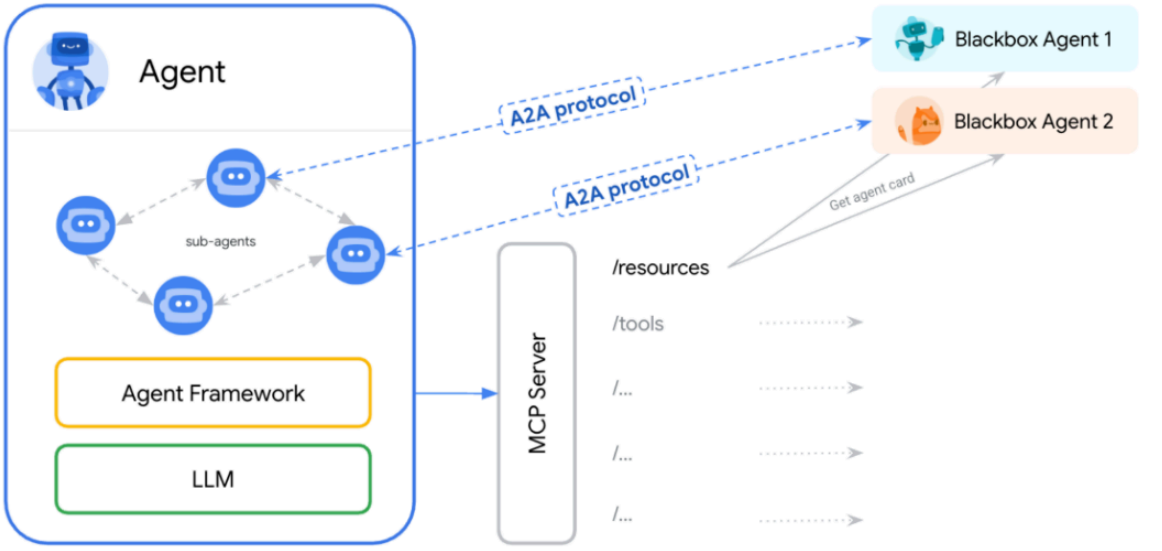
工具是具有结构化输入输出、预定义行为的基本单元；

智能体则是能够调用工具、进行逻辑推理并与用户互动，从而完成新任务的自主应用。

要真正满足用户需求，必须让智能体与工具协同工作，既发挥工具的专长，又利用智能体的灵活性。

互补性协议

在这种背景下，**A2A** 和 Anthropic 发布的 **Model Context Protocol（MCP）** 正好形成互补。可以把 A2A 想象成智能体之间的“电话簿”，负责发现、呼叫和协作；而 MCP 则像工具的“使用说明书”，为智能体接入外部数据源和服务提供统一接口。两者结合，才能构建既能高效联动工具，又能自由组织多方智能体的完整生态。下面具体介绍下：



A2A 协议与 MCP 协议是两个在 AI 生态系统中扮演不同角色但又互补的标准协议。

1）**MCP（Model Context Protocol）**：由 Anthropic 于 2024 年发布，旨在为 AI 模型提供与外部数据源和工具的标准化连接方式。它允许 AI 模型通过统一的接口访问文件、数据库、API 等资源，实现“函数调用”功能的标准化。MCP 采用 JSON-RPC 2.0 协议，支持多种传输方式，如 STDIO 和 HTTP+SSE。它的核心在于提供一个“工具说明书”，让 AI 模型能够安全、高效地与外部系统交互。

2）**A2A（Agent-to-Agent）协议**：这是一个应用层协议，旨在实现 AI 智能体之间的自然语言协作。A2A 允许不同的 AI 智能体以“智能体”或“用户”的身份进行交流，而非仅仅作为工具被调用。它更关注智能体之间的沟通和协作，促进多智能体系统的协同工作。

两者的关系可以通过以下比喻来理解：

- MCP 就像是一个“工具说明书”，告诉 AI 模型如何使用外部工具和数据。
- A2A 就像是一个“电话簿”，让不同的 AI 智能体能够相互联系和协作。

因此，A2A 和 MCP 是互补的协议，共同推动了智能体生态的发展。

2、SpringAI

1.SpringAI相关面试题

1、SpringAI 是什么？它和 LangChain、LangChain4j 有什么区别?

标准答法：
SpringAI 是 Spring 官方推出的 AI 应用开发框架，基于 Spring Boot 生态，提供统一的 LLM 客户端抽象（如 `ChatClient`、`EmbeddingClient`），支持 OpenAI、Azure OpenAI、Ollama、HuggingFace 等主流大模型。
它强调“约定优于配置”，开发者可以像使用 Spring Data 一样轻松调用 AI。

扩展答法：
LangChain / LangChain4j 更偏向于构建复杂的 Agent、RAG pipeline；而 SpringAI 更像是 **AI 能力在 Spring 生态中的“标准 SDK”**。两者可以结合：比如 RAG 用 LangChain4j，服务封装和集成用 SpringAI。

2、SpringAI 的核心组件有哪些?

- **标准答法:**

1. `ChatClient`：对话接口，封装了大模型的 ChatCompletion 调用。
2. `EmbeddingClient`：向量化接口，用于文本向量生成，支持 RAG。
3. `Tool` / `ToolCallbackProvider`：工具接口，支持函数调用与外部 API 集成。
4. `PromptTemplate`：模板化 Prompt 构建。
5. `VectorStore`：知识库接口，支持多种向量数据库。

- **扩展答法:**
  SpringAI 的设计理念是类似 Spring Data、Spring Security 的"统一抽象层"，让底层 LLM 厂商差异对上层透明。

3、如何实现模型切换?

标准答法:
通过配置切换，比如从 OpenAI 切换到 Ollama，只需改 `application.yml` 的配置，不改业务代码。

扩展答法:
可以通过 **Spring Profiles** 在不同环境使用不同模型，比如开发用本地 Ollama，生产用 OpenAI。

4、SpringAI 中 Tool 的概念是什么?

标准答法:
Tool 是模型可以调用的外部能力，比如调用天气 API、数据库查询。
在 SpringAI 中通过 `@Tool` 注解或者 `ToolCallbackProvider` 来注册工具。

扩展答法:
和 LangChain function calling 类似，本质是 **结构化函数调用。**常用于让 LLM 获取实时信息或操作业务系统。

5、如何用 SpringAI 实现 RAG?

- **标准答法:**

1. 用 `EmbeddingClient` 将文档分片向量化。
2. 存储到 `VectorStore`（如 PGVector、Milvus）。
3. 用户提问时，检索相关向量 → 拼接到 Prompt → 调用 `ChatClient`。

- **扩展答法:**
  可以在检索阶段加入 **Hybrid Search（BM25 + 向量检索）**，在生成阶段用 **Re-ranking** 提升准确度。

6、Re-Reading（重读）及基于 Spring AI 的 Re-Reading Advisor 实现?

一、什么是 Re-Reading?

Re-Reading（重读，也称 Re2）是一种通过让大语言模型（LLM）**重新阅读问题或上下文**来提高其推理能力的技术。核心思想是：

- 对于复杂问题，模型在初次阅读后可能理解不完整或生成的回答不够精确。
- 通过**重复阅读**问题和相关上下文，模型可以更好地理解题意，生成更准确、更深入的回答。

- 学术研究表明，Re-Reading 对于提升模型推理和回答质量有一定效果，但会增加计算成本，因此在高并发或资源受限场景下需要谨慎使用。

典型场景：

1. 文档问答（Document QA）：模型初次生成答案后，再回到文档中复读相关段落，纠正或补充答案。
2. 信息抽取或总结：初步提取后，通过复读原文提高准确率。
3. RAG 系统增强生成：初次检索 + 初次生成答案 → Re-Reading → 最终高质量答案。

二、Spring AI 中实现 Re-Reading 的思路

在 **Spring AI** 中，可以通过自定义 **Advisor** 实现 Re-Reading 功能。Advisor 用于在请求到达模型前对用户输入或上下文进行加工处理，从而引导模型重新阅读问题。

1. **创建自定义 Advisor 类**

在 Spring AI 中，需要实现以下接口：

- **CallAroundAdvisor**：用于普通请求
- **StreamAroundAdvisor**：用于流式请求

> 在 Spring AI 1.0 版本中，这两个接口对应的是 **CallAdvisor** 和 **StreamAdvisor**。

自定义 Advisor 主要用于拦截和修改用户输入，将原始问题改写为“重读”的格式。

2. **修改用户提示词**

在 Advisor 的前置处理逻辑中（例如 `aroundCall` 或 `aroundStream` 方法调用之前）：

- 对用户原始输入进行改写，将其指令化为“重读”格式。
- 常用格式示例：

```
{Input_Query}
Read the question again: {Input_Query}
```

其中 `{Input_Query}` 是用户原始的提问内容。

3. **传递给模型处理**

- 将改写后的提示词传给大语言模型进行处理。
- 模型在接收到“重新阅读指令”后，会对问题进行复读和深入理解，从而生成更准确的回答。

三、完整实现示例（伪代码）

```
public class ReReadingAdvisor implements CallAroundAdvisor {

    @Override
    public Object aroundCall(AdvisorContext context, CallChain chain) throws
Exception {
        // 获取用户原始输入
        String originalQuery = context.getInputQuery();

        // 改写输入，指示模型重读
        String reReadQuery = originalQuery + "\nRead the question again: " +
originalQuery;
```

```
        // 将改写后的输入放回上下文
        context.setInputQuery(reReadQuery);

        // 调用下一步链条
        return chain.proceed(context);
    }
}
```

- 对于流式请求，只需实现 `StreamAroundAdvisor`，逻辑类似。

7、什么是结构化输出？SpringAI怎么实现？

结构化输出是指将大语言模型返回的自由文本输出转换为预定义的数据格式，像 JSON、XML 或特定的 Java 类（POJO），对于需要可靠解析 AI 输出值并进行后续处理的需求来说非常重要。

Spring AI 通过 `StructuredOutputConverter` 机制实现结构化输出，其工作流程可以分为调用前和调用后两个阶段：

1. 调用前：`StructuredOutputConverter` 实现了 `FormatProvider` 接口。`FormatProvider` 的作用是提供特定的格式指令给 AI 模型，这些指令会附加到用户的提示词后面，明确告诉模型应该生成何种结构的输出。举个例子，它可能会包含类似 "Your response should be in JSON format. The data structure for the JSON should match this Java class: com.example.MyBean..." 这样的描述，并可能附带一个 JSON Schema 定义，引导模型生成符合指定格式的响应。
2. 调用后：`StructuredOutputConverter` 同时也实现了 Spring 的 `Converter<String, T>` 接口。这个 Converter 负责将大模型返回的文本输出（通常是 JSON 字符串）转换为开发者指定的目标类型 `T`（比如一个 Java Bean 对象、Map 或 List）。Spring AI 提供了多种内置的转换器实现，如 `BeanOutputConverter`（用于转换为 Java Bean，内部基于 ObjectMapper）、`MapOutputConverter` 和 `ListOutputConverter`。

也就是说，Spring AI 的结构化输出转换器首先通过修改提示词来规定模型按特定格式生成文本，然后将该文本转换为 Java 对象。使用 `ChatClient` 的 `.entity(MyClass.class)` 方法时，框架会自动处理这个过程，将模型的 JSON 输出映射到 `MyClass` 的实例。但是，这只是 “尽最大努力” 的转换，模型并不保证一定能严格按要求返回结构化数据！

3、LangChain

LangChain相关面试题

1、什么是LangChain?

LangChain 是一个开源框架，专为快速构建复杂的大语言模型**应用**而设计。它通过模块化组件（Agents、Memory、Tools 等）和预制工具链，解决了传统LLM开发中的三大痛点：

1. **上下文管理**：通过`Memory`组件（如对话历史缓存、实体关系跟踪）实现长对话连贯性。
2. **多工具协同**：支持动态调用外部API、数据库、搜索引擎等工具（如`GoogleSearchTool`），例如在回答“2025年全球GDP排名”时自动触发实时数据查询。
3. **复杂任务编排**：通过`Chains`（链）和`Agents`（代理）将多个LLM调用和工具操作组合成工作流，例如“分析财报→提取关键指标→生成可视化建议”的端到端流程。

部分技术模块的介绍

Chains（链）：

- **LLMChain**：基础链，直接调用LLM（如生成摘要）。
- **RetrievalQA**：结合向量数据库实现"先检索后生成"，例如从企业文档中提取答案。
- **RouterChain**：动态路由请求到不同链（如"中文问题→中文链，技术问题→技术知识库链"）。

Agents（代理）：

- **ReAct**：通过"思考-行动-观察"循环解决问题（如"用户问'杭州今天天气如何？'→调用天气API→返回结果"）。
- **OpenAI Function Calling**：直接调用函数（如`get_current_time()`），无需手动解析JSON。

Memory（记忆）：

- **ConversationBufferMemory**：存储对话历史（如"用户之前询问过订单状态，需保持上下文"）。
- **VectorStoreRetrieverMemory**：将记忆存储在向量数据库中，支持语义检索（如"用户提到'上周会议记录'，自动关联相关文档"）。

2、核心组件?

LangChain 是一个专为大语言模型（LLM）应用开发而设计的框架，其核心组件包括：

1）**Models（模型）**：支持多种语言模型，如 OpenAI、Anthropic、Mistral、Llama 等，提供统一的接口，便于在不同模型之间切换。

2）**Prompt Templates（提示词模板）**：允许用户创建动态提示词，提高模型的泛化能力。通过模板化的方式，可以根据不同的输入生成相应的提示词，从而引导模型生成更准确的输出。

3）**Memory（记忆）**：用于存储对话的上下文信息，支持短期记忆和长期记忆。短期记忆通常用于当前会话的上下文，而长期记忆则结合向量数据库，持久化存储重要信息，便于在未来的对话中调用。

4）**Chains（链式调用）**：将多个处理步骤串联起来，形成一个处理流程。支持 Simple Chains（单步任务）和 Sequential Chains（多步任务），使得复杂任务的处理更加模块化和可复用。

5）**Agents（智能体）**：通过 ReAct 框架，Agent 可以根据用户的输入动态选择合适的工具来完成任务，实现更灵活的任务处理。

6）**Tools（工具）**：提供访问外部资源的能力，如 API、Google 搜索、SQL 数据库等，扩展了模型的功能，使其能够处理更复杂的任务。

3、核心架构?

LangChain 的核心架构由四大关键模块组成：**LangChain Libraries、LangChain Templates**、**LangServe** 和 **LangSmith**。它们各自承担不同的角色，共同构建了一个完整的 LLM 应用开发、部署与监控的闭环体系。

1）**LangChain Libraries**：这是整个框架的基础，包含多个子模块：

- **langchain-core**：提供核心抽象，如模型接口、工具、向量存储等，设计轻量，便于扩展。
- **langchain**：构建链（Chains）和代理（Agents）的主要模块，处理复杂的业务逻辑和外部 API 交互。
- **langchain-community**：整合社区贡献的第三方工具和集成，如模型操作、提示词模板、文件解析、向量化等。

2）**LangChain Templates**：提供一系列易于部署的参考架构，适用于各种任务。这些模板预配置了常用的集成，便于快速上手和定制。

3）**LangServe**：用于将 LangChain 构建的链部署为 REST API 的库，集成了 FastAPI，支持流式、批量处理等功能，方便将应用推向生产环境。

4）**LangSmith**：开发者平台，提供调试、测试、评估和监控功能，帮助开发者优化和部署基于 LangChain 构建的应用。

4、什么是LangGraph?

LangGraph 是 LangChain 生态下的一个基于图结构的开源框架，专为构建**状态化、多代理协同的复杂AI应用**而设计。它通过将任务流程建模为有向无环图（DAG）结构对各节点（如Agent、工具、状态）及其交互进行精细控制。它可以：

- 通过拖拽界面设计工作流，非技术人员也能快速搭建复杂逻辑（如企业级客服系统）。
- 可以在关键节点（如财务审批、合规审核）插入人工确认环节，避免AI决策风险。
- 实时展示模型生成过程（如逐句输出报告内容），提升用户信任度。

架构与核心组件

LangGraph 将整个 Agent 工作流抽象为有向无环图（DAG），节点代表各类执行单元（LLM Agent、工具调用、决策函数），边则承载数据和状态传递，支持流式（streaming）执行和中途插入审批或回退操作。

另外，这个框架自带状态管理（State）模块，可在图中任意节点读写上下文，实现长期记忆与多线程任务隔离

1. **图定义**：用节点（如“数据检索”“LLM生成”）和边（如“成功→下一步”“失败→重试”）构建任务流。
2. **状态管理**：全局状态（如用户ID、历史对话）在节点间传递，支持持久化存储。
3. **动态执行**：根据条件（如“金额>1000元→人工审核”）自动切换执行路径。
4. **人工介入**：关键节点暂停流程，等待人工审批（如“高风险操作需主管确认”）。

5、LangGraph的编排原理?

LangGraph 的编排原理是通过图结构将复杂的 AI 任务分解为可编排的节点（如代理、工具、流程终点），并通过状态流转和条件边实现动态流程控制。

核心包含三个要素：

1. 节点（Node）：代表独立处理单元（如 Agent 调用 LLM、Tool 执行工具函数），每个节点接收状态并返回更新后的状态。
2. 边（Edge）：定义节点间的流转路径，支持条件分支（如根据用户输入选择不同处理逻辑）和循环（如需要多次修正结果）。
3. 状态（State）：贯穿整个流程的上下文数据（如对话历史、中间结果），驱动节点间的动态交互。

简单来说 LangGraph 像“流程图引擎”，我们通过画图（定义节点和边）描述任务逻辑，框架自动根据状态流转执行节点，支持复杂的多 Agent 协作和动态决策。

6、LangChain和LangGraph区别?

LangChain 是基于**链式结构**（Chain），适合线性任务（如文档问答、简单客服），通过预定义步骤顺序执行，类似工厂流水线。

LangGraph：基于**图结构**（Graph），支持循环、分支和动态决策，适合需要多角色协作、状态跟踪的复杂任务（如临床试验审批、多智能体投资分析）。

简单来说 angChain 更像是一个“模块化 AI 应用框架”，用于拼接模型、工具、记忆等组件。而 LangGraph 是专注于流程控制和任务编排的“有状态执行图框架”。

在我们实际开发中，可根据任务复杂度选择：简单任务用 LangChain，复杂任务用 LangGraph，超复杂场景结合两者，如用 LangChain 处理基础链，LangGraph 管理全局流程（**要注意哈，两者并非替代关系，而是互补**！LangGraph 可作为 LangChain 的扩展，在需要动态控制流和状态管理的场景中提升应用的灵活性和可靠性）。

表格对比

场景	LangChain 示例	LangGraph 示例
简单问答	用户提问 → 检索工具 → LLM 生成回答。	用户提问 → 分析意图 → 若需要搜索则调用工具 → 生成回答 → 若用户不满意，自动重试。
多智能体协作	不支持（需手动协调多个链）。	客服代理 → 技术代理 → 经理代理（根据问题复杂度自动路由）。
人机协作	不支持（需手动插入人工步骤）。	用户输入 → 代理生成方案 → 人工审核节点 → 若通过则执行，否则返回修改。
复杂数据分析	数据清洗 → 模型预测 → 结果汇总（固定顺序）。	数据清洗 → 若质量不达标，返回重新清洗 → 分析结果 → 若异常，触发人工审核。

4、其他

1.其他相关面试题

1、如果一个GPU集群的LLM处理能力为1000tokens/s，那1000个用户同时并发访问，响应给每个用户的性能只有1 token/s吗？怎么分析性能瓶颈

不会平均变成每人 1 token/s。

因为 LLM 并非简单线性分配（直接用除法）资源，而是通过**批处理**与**并发调度**来提升吞吐。

若每次批处理包含 100 个用户的请求（每用户 10tokens），则 1000 用户可以分 10 批处理完。单用户性能是 10tokens/s。

实际响应速度取决于**Token长度、批处理策略和资源排队机制等。**

举个实际例子：

假设有一个聊天机器人平台，有 1000 个用户并发请求：

- 请求平均长度为 20 tokens，每秒新进 200 个请求；
- GPU 一次推理最大支持 batch 128，吞吐为 1000 tokens/s。

那服务器可能这样调度：

- 每 10ms 打一批，聚合 50~100 个请求；
- 所有请求每生成一个 token，就进入下一轮调度；
- 整个 pipeline 中持续运行着多个 batch，每个 batch 中都是不同用户的不同 token。

所以实际**每个用户看到的响应速度，可能是几十 token/s，而不是 1 token/s。**

2、如何实现 AI 多轮对话功能？如何解决对话记忆持久化问题？

多轮对话功能的关键在于让 AI 具备 “记忆能力”，即能够记住与用户之前的对话内容并保持上下文连贯。在本项目中，我使用了 Spring AI 框架提供的对话记忆 (Chat Memory) 和 Advisor (顾问) 特性来实现这个功能。

具体实现操作上来说，我主要通过构造 `ChatClient` 来实现功能更丰富、更灵活的 AI 对话。`ChatClient` 支持使用 `Advisors`，可以理解为一系列可插拔的拦截器，在调用 AI 前后执行额外操作。其中`MessageChatMemoryAdvisor` 就是实现多轮对话的关键 `Advisor`，它的作用是从对话记忆中检索历史对话，并将对话历史作为消息集合添加到当前的提示词中，实现让 AI 模型能够 “记住” 之前的交流。

`MessageChatMemoryAdvisor` 依赖于 `ChatMemory` 接口的实现来存取对话历史。`ChatMemory` 接口中定义了保存消息、查询消息和清空历史的方法。

默认情况下会使用`InMemoryChatMemory`实现，从这个类名可以看出对话记忆仅存在于内存中，一旦服务重启，记忆就会丢失。为了解决这个问题，需要将对话记忆持久化。

Spring AI 提供了多种持久化方案，例如 `JdbcChatMemory` 可以将对话保存在关系型数据库中。但是在本项目中，考虑到 `spring-ai-starter-model-chat-memory-jdbc` 依赖版本较少且缺乏相关介绍，我选择了自定义实现 `ChatMemory` 接口的方式：

1. 开发一个 `FileBasedChatMemory` 类，它实现了 `ChatMemory` 接口。
2. 使用高性能的 Kryo 序列化库将对话消息（`Message` 对象及其子类）序列化后保存到本地文件中，读取时再进行反序列化。选择 Kryo 是因为 `Message` 接口有多种实现，结构不一，且没有无参构造和 `Serializable` 接口，普通的 JSON 序列化难以处理。

通过这种方式，我将对话的上下文信息持久化到了指定的文件目录，解决了内存记忆丢失的问题。

3、Agent死循环问题？如何解决？

有的，当时在构建基于大模型的 Agent 时，遇到这样一个现象：调用链是“先查订单号 → 得到物流单号 → 查物流单号 → 返回物流信息”。

但一定的概率会是：订单号给到大模型，拿到了物流单号，再把订单号和物流单号一起给到大模型，大模型给你返回的还是物流单号，并且没返回 endFlag。导致 Java 侧死循环调用。

主要原因是输入提示词设计不明确、Agent 没有添加明确的状态流转判断，且还欠缺调用次数/语义差异的兜底保护策略。

因此我们做了改造：

1、强制声明 “若已获取物流单号，直接调用查物流接口，无需重复生成”。 2、状态变量（如 step=1/2/3）跟踪当前步骤，输入时附带状态标识，模型按状态执行对应动作。 3、设置调用兜底机制：最多调用 3 次，并检查返回字段是否变化，检查是否包含 endFlag。

4、程序和 AI 大模型的集成有哪些方式?

接入方式	实现方式	具体做法	优点	缺点	适用场景
SDK 接入	通过大模型提供商的官方软件开发工具包直接调用 AI 服务。	在项目中引入阿里云灵积的 SDK 依赖，配置 API Key，然后创建调用实例发送请求。	类型安全、错误处理完善、性能优化好。	与特定厂商绑定、依赖特定版本、可能增加项目体积。	需要深度集成特定模型提供商服务、对性能要求高的场景。
HTTP 接入	直接通过 REST API 发送 HTTP 请求调用模型。	使用 Hutool 的 HttpUtil 构造请求体和头部，发送 POST 请求到大模型 API 端点。	无语言限制、不增加额外依赖、灵活性高。	需要手动处理错误、序列化/反序列化复杂、代码冗长。	SDK 不支持的语言、简单原型验证、临时性集成。
Spring AI 框架	通过 Spring AI 的统一抽象接口调用不同的模型服务。	引入 Spring AI 依赖，配置模型参数，注入 ChatClient 或 ChatModel 后调用。	统一的抽象接口、易于切换模型提供商、与 Spring 生态完美融合、提供高级功能。	增加抽象层、可能不支持特定模型的特性、版本更新快。	Spring 应用、需要支持多种模型、需要高级 AI 功能的场景。
LangChain4j 框架	使用 LangChain4j 框架的组件构建 AI 调用链。	引入 LangChain4j 及相应模型的集成包，创建模型实例并调用。	提供完整的 AI 应用工具链、支持复杂工作流、丰富的组件和工具。	学习曲线较陡、文档相对较少、抽象可能引入性能开销。	构建复杂 AI 应用、需要链式操作、RAG 应用开发。

5、如何用 Spring AI 开发应用？用到哪些特性?

1）大模型接入与调用：

- 使用 `spring-ai-alibaba-starter` 和 `spring-ai-ollama-spring-boot-starter` 简化大模型的配置和集成。
- 通过 Spring AI 的 `ChatModel` 和更高级的 `ChatClient API` 与 AI 大模型交互，实现对话功能。

2）Advisors：

- 利用`MessageChatMemoryAdvisor`实现了多轮对话中的上下文记忆功能。
- 为了增强应用的可观测性，我还自定义了 Advisor，例如用于记录AI请求和响应日志的`MyLoggerAdvisor`，以及用于尝试提高模型推理能力的`ReReadingAdvisor`。

3）对话记忆：

- 项目初期使用了基于内存的`InMemoryChatMemory`。
- 为了实现对话记忆的持久化，我自定义了`FileBasedChatMemory`，并使用 Kryo 序列化库解决了`Message`对象层级复杂难以直接序列化的问题。

4）结构化输出：通过 Spring AI 的结构化输出转换器，我把 AI 模型的文本输出直接转换为Java 对象，方便后端处理和使用。

5）Prompt 模板：利用`PromptTemplate`管理和动态生成提示词，比如使用占位符替换变量、从外部文件加载复杂的提示词模板，提高了提示词的可维护性和灵活性。

6）RAG (检索增强生成)：

- 文档 ETL：使用了 Spring AI 的 ETL Pipeline 组件，如`MarkdownDocumentReader`加载本地Markdown知识库文档、`TokenTextSplitter`进行文本分割、`KeywordMetadataEnricher`自动为文档添加关键词元数据。
- 向量存储：集成`SimpleVectorStore`和`PgVectorStore`用于存储和检索文档向量。在集成`PgVectorStore`时，我还解决了多个`EmbeddingModel Bean`冲突的问题。
- 检索与增强：使用`QuestionAnswerAdvisor`和更灵活的`RetrievalAugmentationAdvisor`实现 RAG 流程，后者还结合了`VectorStoreDocumentRetriever`和`ContextualQueryAugmenter`等组件进行查询优化和空上下文处理。还实践了查询重写（`RewriteQueryTransformer`）等预检索优化技术。

7）工具调用 (Tool Calling)：

- 通过`@Tool`和`@ToolParam`注解定义了多种外部工具，如文件操作、联网搜索、PDF 生成等，并使用`ChatClient`的`tools()`方法将这些工具注册给 AI 模型，让 AI 能够调用这些工具完成特定任务。
- 在构建 YuManus 智能体时，我手动控制了工具的执行流程（ReAct模式），通过`DashScopeChatOptions`禁用 Spring AI 内部的自动工具执行，更方便、更精细地管理思考-行动循环。

8）MCP (模型上下文协议) 集成：

- MCP客户端：使用`spring-ai-mcp-client-spring-boot-starter`，配置 stdio 和 SSE 连接方式来调用外部 MCP 服务。通过`ToolCallbackProvider`将 MCP 服务提供的工具无缝集成到`ChatClient`的工具调用机制中。
- MCP 服务端：基于`spring-ai-mcp-server-webmvc-spring-boot-starter`开发了自定义的图片搜索 MCP 服务，使用`@Tool`注解暴露工具，然后通过`ToolCallbackProvider Bean`进行注册。

6、提示词优化考虑哪些维度？你们提示词模板有哪些字段？

关于提示词优化的核心维度主要有以下几个点

1. **目标明确性**：让模型清楚知道要解决什么问题。
2. **结构清晰性**：用分点、格式（如JSON/Markdown）或分隔符（###）拆解任务，避免信息混乱
3. **少量样本**：通过输入输出样例告诉模型期望的结果长什么样，减少理解偏差。

4. **角色设定**：给模型定义身份（如“你是资深程序员”），让输出风格更符合场景，比如专业、口语化等。
5. **增加约束条件**：限制输出范围，如字数、格式、禁止内容，避免模型幻觉或偏离主题。

还有一个也很关键就是**持续反馈迭代**，我们需要根据模型输出调整提示词，比如补充细节、优化示例，通过A/B测试找到最优解。

提示词模板的常见字段：

1. **角色定义**：指定模型身份（如你是程序员，擅长代码生成）。
2. **任务描述**：用具体指令拆解目标（如请根据以下数据生成趋势分析，需包含3个核心结论）。
3. **输入内容**：提供原始数据或问题案例。
4. **输出格式**：规定结果结构（如输出JSON格式）。
5. **约束规则**：说明限制条件（如回答不超过200字）。
6. **评估标准**：引导模型自检（如语言需口语化）。

7、如何进行 AI 应用的测试和效果评估?

AI 应用的测试和传统软件测试有相似之处，当然也有其独特性。

相似之处在于，我们仍然需要关注功能性、兼容性、安全性等方面。但其独特性在于，AI 系统的核心是模型和数据，因此测试的重点要向这两个方面倾斜。比如：

- 数据质量：模型是数据驱动的，输入数据的质量、分布以及是否能覆盖各种真实场景，对模型的表现影响很大。我们要测试模型在不同数据集、边缘案例以及对抗性样本下的表现。
- 模型性能：这涉及到模型的准确率、召回率、F1-score 等一系列评估指标。同时，我们也要关注模型在不同环境、不同输入扰动下的稳定性，即鲁棒性。例如，应用是否会对微小的、人眼难以察觉的输入变化产生截然不同的输出。
- 部署：应用最终部署到生产环境上。测试也需要覆盖模型与系统的交互、API 的稳定性以及在真实部署环境下的性能表现，比如延迟、吞吐量等。

效果评估更侧重于衡量 AI 应用在实际业务场景中产生的价值和影响。通常是一个持续的过程，而不仅仅是一次性的测试。

在效果评估方面，我们需要注意以下几个点：

- 明确评估指标：这些指标需要紧密围绕业务目标。例如，一个推荐系统可能关注点击率、转化率、用户停留时长等；一个风控模型更关注误报率、漏报率以及由此带来的经济损失减少。
- A/B 测试：将新功能与现有版本进行对比测试，观察其在真实用户和真实环境中的表现差异，这是评估效果最直接有效的方法之一。
- 成本效益分析：除了技术指标，还要从商业角度评估 AI 应用的投入产出比，考虑其带来的实际经济效益或社会效益。

AI 应用的测试和效果评估是一个系统性的工程，不只是要关注应用本身的技术指标，还要关注我们上面提到的内容，以及最终在真实业务场景中产生的价值，这是一个迭代和持续优化的过程。

8、什么是 CoT 思维链和 ReAct 模式？它们如何提高 AI 推理能力?

CoT 思维链 (Chain of Thought) 和 ReAct 模式 (Reasoning + Acting) 都是增强大型语言模型推理和解决复杂问题能力的技术。

- CoT 思维链：生成最终答案之前，先引导 AI 模型输出一系列中间的、连贯的推理步骤。引导模型“思考过程化”，模拟人类解决问题时逐步分析和推导的过程。它通过让模型显式地写出思考步骤，可以帮助模型分解复杂问题，减少在复杂逻辑链条中出错的概率。这些中间步骤也为我们理解模型的“思路”提供了便利，调试和优化都会更简单方便。
- ReAct 模式：一种将LLM的推理 (Reasoning) 能力与行动 (Acting) 能力相结合的框架。它让模型不仅能思考，还能决定采取何种行动（通常是调用外部工具，例如文件操作、联网搜索），然后根据行动的观察结果执行下一轮的思考和行动，形成一个“思考-行动-观察”的循环。

CoT 主要关注提升模型内部的逻辑推理连贯性和深度，通过显化思考过程实现。ReAct 更侧重于让模型与外部世界互动，通过“推理驱动行动，行动反馈观察，观察指导推理”的闭环来解决那些需要外部信息或操作才能完成的复杂任务，更考研模型动态调整推理路径的能力。

9、有哪些设计和优化 Prompt 的技巧？请举例说明

设计和优化 Prompt 的核心目标是引导 AI 模型生成符合预期的高质量输出。分享一些技巧：

1）明确指定任务和角色：清晰地告诉 AI 它需要扮演什么角色以及具体执行什么任务。

```
▼plain复制代码系统：你是一位经验丰富的Python教师，擅长向初学者解释编程概念。
用户：请解释 Python 中的列表推导式，包括基本语法和 2-3 个实用示例。
```

2）提供详细说明和具体示例：给出足够的上下文信息、期望的输出格式、风格或长度。最好再提供 1-2 个输入输出的范例，帮助模型理解任务模式。

```
▼plain复制代码我将给你一些情感分析的例子，然后请你按照同样的方式分析新句子的情感倾向。

输入："这家餐厅的服务太差了，等了一个小时才上菜"
输出：负面，因为描述了长时间等待和差评服务

输入："新买的手机屏幕清晰，电池也很耐用"
输出：正面，因为赞扬了产品的多个方面

现在分析这个句子：
"这本书内容还行，但是价格有点贵"
```

3）使用结构化格式引导思维：通过列表、表格、JSON Schema 或特定分隔符（如 XML 标签）来组织输入和期望的输出，这些指令更容易被大模型理解，输出也更有条理。

```
▼plain复制代码分析以下公司的优势和劣势：
公司：Tesla

请使用表格格式回答，包含以下列：
- 优势(最少3项)
- 每项优势的简要分析
- 劣势(最少3项)
- 每项劣势的简要分析
- 应对建议
```

4）思维链提示法：引导模型展示其推理过程，逐步思考问题，尤其适用于复杂问题，能提高准确性。比如，在解决一个数学应用题时，可以要求 AI：“请一步步思考解决这个问题：首先...然后...最后...”。

```
▼plain复制代码问题：一个商店售卖T恤，每件15元。如果购买5件以上可以享受8折优惠。小明买了7件T恤，他需要支付多少钱？

请一步步思考解决这个问题：
1. 首先计算7件T恤的原价
2. 确定是否符合折扣条件
3. 如果符合，计算折扣后的价格
4. 得出最终支付金额
```

5）分解任务：把复杂的任务分解为一系列更小、更易于管理的步骤，并指导模型按顺序完成每个步骤。例如，要求 AI“第一步：分析用户需求。第二步：草拟解决方案。第三步：评估方案风险。”

```
▼plain复制代码请帮我创建一个简单的网站落地页设计方案，按照以下步骤：

步骤1：分析目标受众(考虑年龄、职业、需求等因素)
步骤2：确定页面核心信息(主标题、副标题、价值主张)
步骤3：设计页面结构(至少包含哪些区块)
步骤4：制定视觉引导策略(颜色、图像建议)
步骤5：设计行动召唤(CTA)按钮和文案
```

6）迭代式提示优化和错误分析：很少有人能一次就写出完美的 Prompt。而是根据模型的输出进行分析，如果结果不理想，就逐步修改和完善 Prompt。

```
▼plain复制代码初始提示：谈谈人工智能的影响。

[收到笼统回答后]
改进提示：分析人工智能对医疗行业的三大积极影响和两大潜在风险，提供具体应用案例。

[如果回答仍然不够具体]
进一步改进：详细分析AI在医学影像诊断领域的具体应用，包括：
1. 现有的2-3个成功商业化AI诊断系统及其准确率
2. 这些系统如何辅助放射科医生工作
3. 实施过程中遇到的主要挑战
4. 未来3-5年可能的技术发展方向
```

7）控制输出长度和风格：明确要求输出的字数范围、文本风格（正式、友好、专业）等。

```
▼plain复制代码撰写一篇关于气候变化的科普文章，要求：
- 使用通俗易懂的语言，适合高中生阅读
- 包含5个小标题，每个标题下2-3段文字
- 总字数控制在800字左右
- 结尾提供3个可行的个人行动建议
```

8）利用系统提示词：设定 AI 的整体行为、个性和能力边界，这对构建特定领域的 AI 应用非常关键。比如：“你是一位专业的恋爱顾问，请以温暖友善的语气回答用户的恋爱困惑”。

10、如何保证 AI 应用的性能和稳定性?

为了保证 AI 应用的性能和稳定性，需要综合考虑很多方面，从应用设计、外部依赖管理到部署策略都需要考虑。一般来说有以下措施：

1）异步处理：对于耗时的 AI 操作，要采用异步处理，避免阻塞服务器主工作线程，提高并发处理能力和系统响应速度。还可以利用 SSE (Server-Sent Events) 和 `SseEmitter` 实现流式输出，提高用户体验。

2）合理的模型选择：选择跟场景、需求相匹配的 AI 模型。不是所有场景都需要最大、最先进的模型；有时候针对特定任务优化的轻量级模型在成本和延迟上更有优势。比如计算向量 Embedding 时，就不必选择深度推理能力强的模型。

3）RAG 系统优化：选择高质量的原始文档和合理的切片策略。以及选择合适的向量数据库，比如使用 PGVector 方案，可以对索引和查询参数调优。

4）外部依赖管理：对工具调用、MCP 服务及其他外部 API 的调用，实现错误处理和重试机制（比如 Guava Retrying 库）。并且针对所有的外部调用设置合理的超时时间。

5）对于流量不确定或需要快速迭代的服务，可以考虑 Serverless 部署，按需分配资源、自动伸缩。

6）API 设计：避免在每个数据块中包含过多冗余信息，只传输核心内容，减少网络负载。

7）可观测性与监控：集成更全面的监控方案，比如使用 Prometheus + Grafana 追踪关键性能指标、延迟、错误率、资源消耗等。

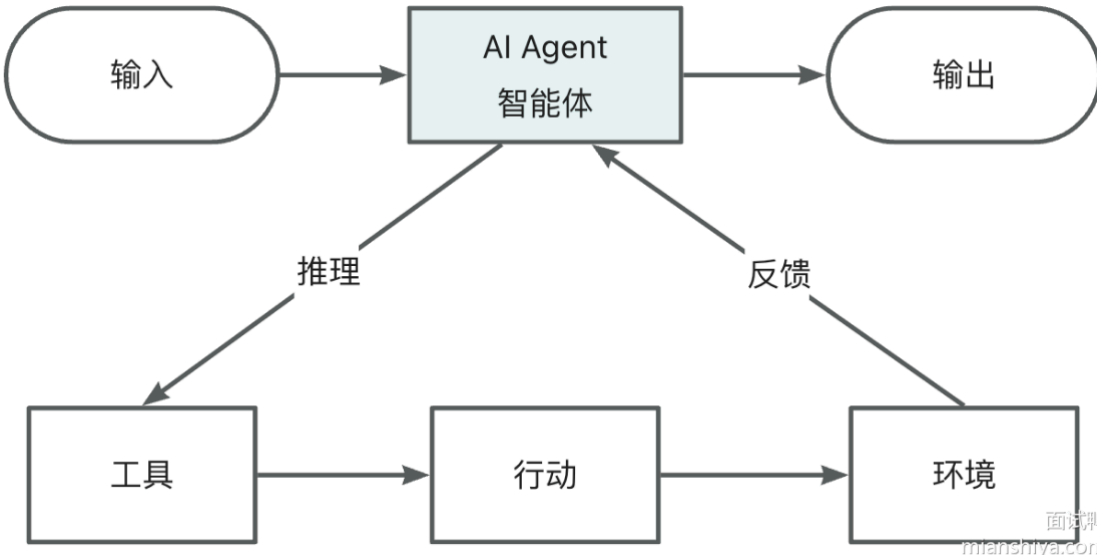
11、什么是 OpenManus？它的实现原理是什么？

OpenManus 是一个开源的自主规划智能体项目。它的核心亮点在于其 “自主执行” 能力，能够自主规划任务，并且在虚拟机中调用各种工具（如编写代码、爬取数据）来完成用户指定的复杂任务。它的实现原理主要基于以下几点：

1. 分层代理架构：OpenManus 采用了分层的代理架构，不同层次的代理负责不同的功能，方便系统的扩展和维护。主要代理层次包括 `BaseAgent`（基础代理，管理状态和执行循环）、`ReActAgent`（实现 ReAct 模式）、`ToolCallAgent`（扩展了工具调用能力）以及具体的智能体实例（如 `Manus`）。
2. ReAct 模式：OpenManus 的工作流程遵循“思考-行动-观察”的循环，`ToolCallAgent` 的 `think` 方法负责与大模型交互、选择工具，`act` 方法负责执行工具，并将结果反馈给大模型进行下一步决策。
3. 工具系统：OpenManus 拥有一个灵活的工具系统。所有工具都继承自 `BaseTool` 抽象基类，提供统一的接口和规范化的参数描述，方便大模型理解和调用。`ToolCollection` 类可以管理多个工具实例，实现了工具系统的可插拔性。
4. 核心组件：包括记忆系统（`Memory` 类存储对话历史和中间状态）、大模型（`LLM` 类提供思考和决策能力）以及流程控制（`AgentState` 和执行循环管理状态转换和任务流程）。
5. 特殊工具设计：比如，`Terminate` 工具是让智能体自主决定何时结束任务，`AskHuman` 工具则是让智能体在遇到困难时向人类寻求帮助。
6. MCP 协议：最新版本的 OpenManus 已经支持，通过 `MCPClients` 类将 MCP 服务集成到其工具系统中，让远程工具能够像本地工具一样被调用。

12、什么是 ReAct？如何基于 ReAct 模式构建具备自主规划能力的 AI 智能体？

ReAct，即 Reasoning + Acting（推理与行动），是一种结合推理和行动的智能体架构。它模仿人类解决问题时“思考 - 行动 - 观察”的循环。AI 首先对问题进行推理（Reason），将原始问题拆分为多步骤任务，明确当前要执行的步骤。然后，它会调用外部工具执行行动（Act），比如调用搜索引擎或访问网页。最后，它会观察（Observe）工具返回的结果，并将这些结果反馈给智能体，用于下一步的决策。这个过程会不断循环迭代，直到任务完成或达到预设的终止条件。



基于 ReAct 模式构建具备自主规划能力的 AI 智能体，核心在于实现这个 “思考-行动-观察” 的循环。这意味着智能体要实现：

1. 接收用户指令后，能分析并拆解成可执行的子任务。
2. 根据子任务的需要，从可用的工具集中选择合适的工具并执行。
3. 分析工具执行的结果，判断任务进展，决定下一步是继续调用工具、向用户澄清还是结束任务。

在本项目中，我实现的拥有自主规划能力的超级智能体就是基于 ReAct 模式，通过 ReActAgent 定义先 think 思考再 act 行动的流程，然后通过 ToolCallAgent 具体实现这两个方法。在 `think()` 方法中决定使用哪个工具（推理），在 `act()` 方法中执行工具（行动），并将执行结果作为下一步思考的输入（观察）。

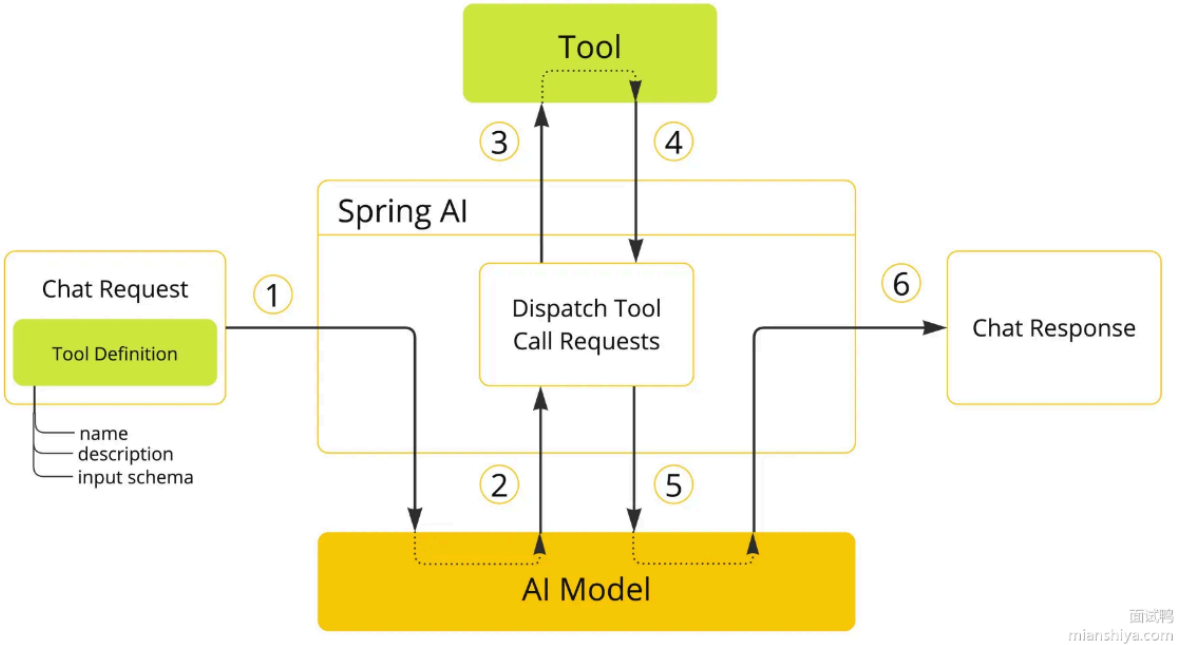
13、什么是工具调用 Tool Calling？如何利用 Spring AI 实现工具调用？

工具调用 (Tool Calling)，也常被称为 Function Calling，是一种允许 AI 大模型在对话过程中，根据需要请求执行外部工具来完成特定任务的机制。

AI 模型本身可能不具备实时查询天气、操作数据库或访问特定 API 的能力，通过工具调用，它可以识别出什么时候需要这些外部能力，并生成一个包含工具名称和所需参数的请求。 应用程序接收到这个请求后，实际执行相应的工具，并将执行结果返回给 AI 模型，模型再基于这个结果继续对话或生成最终答案。

注意，真正执行工具的是我们的应用程序，而非 AI 服务器本身。

Spring AI 极大地简化了工具调用的实现：



1）定义工具：通常使用注解方式，在一个 Java 类中，将希望作为工具的方法标记上 `@Tool` 注解。方法的参数可以使用 `@ToolParam` 注解来提供描述和指定是否必需。Spring AI 支持多种 Java 类型作为参数和返回值，但返回值需要可序列化。

2）注册与使用工具：

- 按需使用：在构建 `ChatClient` 请求时，通过 `.tools()` 方法直接传入工具类的实例。
- 全局使用：在构建 `ChatClient` 时，通过 `.defaultTools()` 注册默认工具，这些工具对该 `ChatClient` 发起的所有对话都可用。
- 集中注册：可以创建一个配置类，使用 `@Bean` 方法将所有工具实例化并通过 `ToolCallbacks.from()` 统一注册为一个 `ToolCallback[]` 数组，方便管理和注入。

3）调用流程：当使用配置了工具的 `ChatClient` 进行对话时，如果 AI 模型判断需要使用某个工具，Spring AI 框架会自动执行以下步骤，无需开发者关心：

- 解析 AI 模型返回的工具调用请求（包含工具名和参数）。
- 根据工具名找到对应的 Java 方法并执行。
- 将工具执行的结果转换并返回给 AI 模型。
- AI 模型根据工具结果生成最终回复。

14、AutoGPT 如何实现自主决策?

AutoGPT 之所以能够实现自主决策，主要得益于其引入了一套完整的**反馈循环机制**。这套机制包括以下几个关键步骤：

1）**目标设定**：用户只需提供一个高层次的目标，AutoGPT 会将其细化为多个子任务。

2）**任务规划**：系统会对目标进行分析，制定出实现目标的详细计划。

3）**任务执行**：AutoGPT 会按照计划调用相应的工具或执行代码，以完成各个子任务。

4）**结果评估**：在每个任务执行后，系统会对结果进行评估，判断是否达到预期目标。

5）**策略调整**：根据评估结果，AutoGPT 会对原有计划进行调整，优化后续的任务执行策略。

通过这样的循环，AutoGPT 能够在无需人工干预的情况下，自主地完成复杂的任务。

扩展知识

为了更深入地理解 AutoGPT 的自主决策机制，我们可以将其与人类的工作方式进行类比。

设想你是一位项目经理，接到一个模糊的项目目标。你会先将这个目标细化为多个具体的子任务，然后制定执行计划，分配资源，监督执行过程，并在每个阶段进行评估和调整，直到项目完成。

AutoGPT 的工作流程与此类似。它接收一个高层次的目标，然后通过其内置的反馈循环机制，不断地规划、执行、评估和调整，直到达到最终目标。

此外，AutoGPT 还具备以下特点：

1）**自主性**：一旦设定目标，AutoGPT 可以在无需人工干预的情况下，自主地完成任务。

2）**适应性**：在任务执行过程中，AutoGPT 能够根据实际情况调整策略，以应对各种变化。

3）**多工具集成**：AutoGPT 可以调用多种工具，如 API、数据库、搜索引擎等，以完成不同类型的任务。

通过这些机制，AutoGPT 实现了从被动响应到主动执行的转变，成为一个真正意义上的自主智能体。

15、什么是Google ADK?

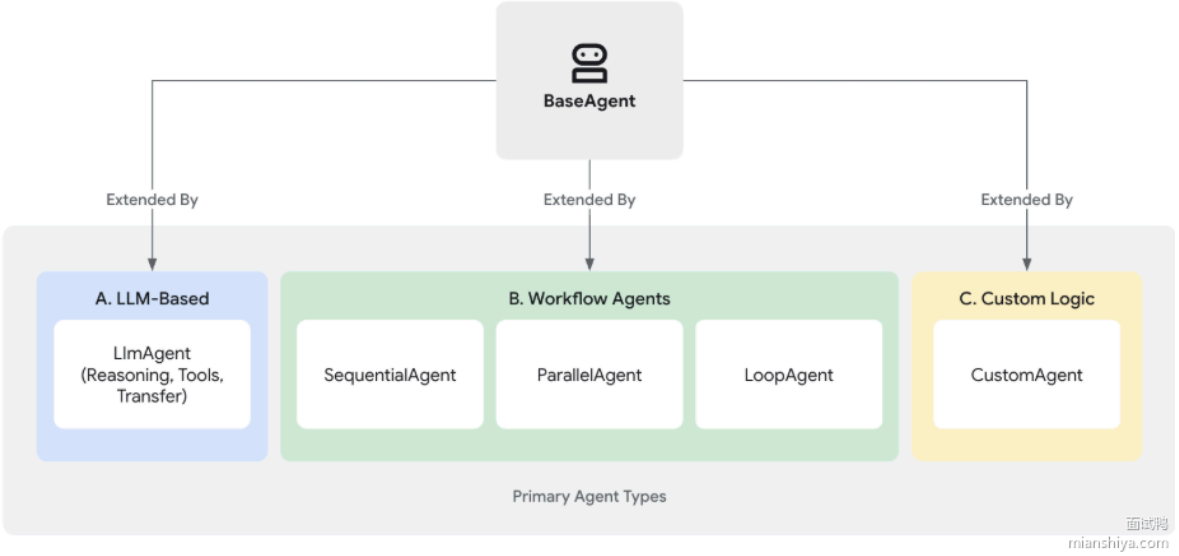
> 前置理解：多智能体（Agent）协作，比如现在要做一个法律咨询系统，可能需要 法律条款分析 Agent、案例匹配Agent、文档生成Agent 三个 Agent 来协同完成。

ADK（Agent Development Kit，智能体开发工具包）是 Google 出的一套专为构建“智能体（Agent）”设计的开发工具集合。它提供模块化功能组件和智能协调机制，帮助我们快速构建能自主决策、多模块协作执行任务的智能体。

按照官网的说法，它的设计目标是**让智能体开发回归软件工程本质**，使得开发者能用熟悉的编码范式，快速构建从简单任务到复杂工作流的智能体架构，并轻松管理其全生命周期。

在 ADK 中，Agent 是一个独立的执行单元，是一个独立的复用“模块”。开发者可以像搭积木一样选择所需模块组合成一个 AI智能体，非常灵活便捷。还能降低维护成本，因为模块可单独升级，不影响整体功能，**这是 ADK 的模块化能力。**

除此之外，它还提供了**协调能力**，负责多 Agent 之间的分工协调。



从 ADK 官网提供的图可以看到，一共有三种协调方式：

- LLM Agents：利用 LLM（大语言模型）来对用户的输入进行分析，结合当前组合内的 Agents 自身描述信息，分析理解自然语言，动态决定下一步操作或工具调用，特别适合需要灵活语言处理能力的任务。

- 工作流：这种方式以预定义的确定性模式（顺序、并行或循环）控制 agent 的执行流，其流程逻辑不依赖LLM，适用于需要稳定、可预测执行路径的结构化流程。
- 自定义 agent：通过直接扩展 BaseAgent 基类创建，开发者可自由实现特殊业务逻辑、定制控制流或集成非标系统，满足高度个性化的需求。

16、什么是护栏技术?

护栏技术是AI系统开发中用于确保模型输出安全、合规、符合伦理的一系列防御性技术手段，**核心作用是防止AI产生有害、错误或违背人类价值观的结果。**

比如：当AI生成内容时，护栏技术会实时检测是否包含歧视、虚假信息、暴力内容等，并阻止或修正这类输出。它就像给AI系统加了一层“安全围栏”，让AI在既定的规则和目标范围内运行，避免“失控”。

常见的护栏技术包括：

1. **内容安全过滤**（如识别有害文本/图像）
2. **伦理规则引擎**（预设禁止行为的逻辑判断）
3. **错误处理机制**（当AI无法处理问题时，拒绝回答或引导人工介入）

护栏技术的核心实现方式

大致流程如下：

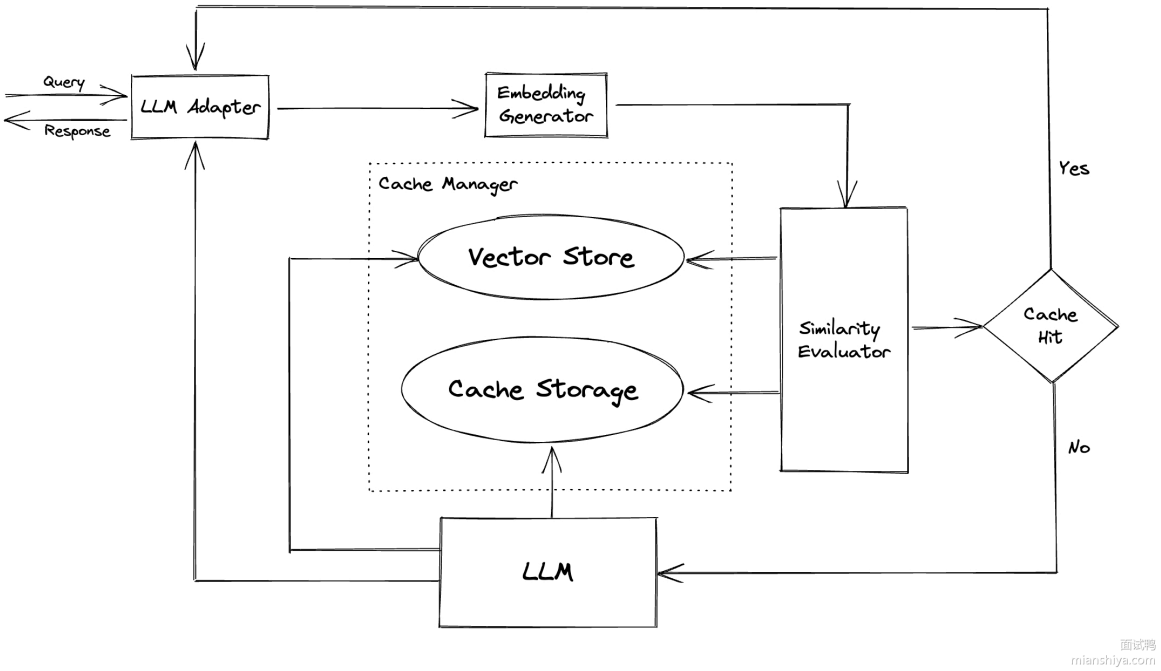
- **1. 内容审核**（用NLP模型检测关键词、语义风险）
- **2. 伦理规则校验**（按预设规则库判断，如“禁止性别歧视”）
- **3. 安全边界检查**（确保输出不超出模型训练范围，如拒绝回答专业医学问题）
- **4. 应急处理**（触发风险时，返回安全提示/禁用功能/人工审核）→ 最终输出）

主要的技术细节是：

- **规则引擎**：通过人工编写或机器学习生成“禁止条件”，比如“包含‘攻击’‘杀死’等词且指向具体人物时，阻断输出”。
- **对抗样本检测**：识别恶意输入（如故意诱导AI违规的prompt），防止被“越狱攻击”。
- **输出校准**：对低风险但有争议的内容进行优化，比如将“男性不适合当程序员”改写为“职业选择与性别无关”。

17、什么是GPTCache?

GPTCache 是专为大语言模型设计的**语义缓存工具**，通过存储和复用模型响应，从而降低 API 调用成本并提升响应速度



它的核心价值在于：

1. **降本增效**：通过缓存相似查询的结果，减少重复调用LLM的次数，例如将ChatGPT的API成本降低10倍，响应速度提升100倍。
2. **语义匹配**：区别于传统精确匹配的缓存，GPTCache使用向量嵌入技术（如OpenAI Embeddings、SentenceTransformers）将用户问题转换为向量，通过向量数据库（如Milvus、FAISS）进行相似性搜索，实现"语义级"缓存命中。
3. **灵活扩展**：支持模块化设计，用户可自定义嵌入模型、缓存存储（如SQLite、MySQL）、逐出策略（LRU/FIFO）等组件，适配不同场景需求。

我们在大模型应用开发的很多场景都能用上它，例如

- 聊天机器人、客服系统中重复问题的快速响应。
- 测试场景，模拟LLM响应，减少对真实API的依赖，加速开发迭代。

大致核心工作流程

1. **预处理**：提取用户问题中的核心信息（如对话中的历史上下文）。
2. **向量生成**：将问题转换为高维向量（如使用BERT模型）。
3. **相似性搜索**：在向量数据库中查找最相似的历史查询。
4. **缓存命中**：若相似度超过阈值，直接返回缓存结果；否则调用LLM并存储新结果。
5. **后处理**：根据温度参数（temperature）调整响应的随机性，平衡准确性和多样性。

18、Manus?

Manus 是咱们由中国团队 **Monica.im** 于 **2025年3月** 推出的 **全球首款通用型AI智能体**，其核心突破在于能够 **独立思考、规划并执行复杂任务**，直接交付完整成果（如股票分析报告、简历筛选结果），而非仅提供对话式建议。

它通过一个中央 模块，将用户的高层指令拆解为多个子任务，再由不同的内部智能体（Agent）或工具执行，形成端到端的自动化执行流程。底层还是调用大模型例如 Claude 、Qwen等来实现规划与决策。

> 可以直接访问这个网址，感受下 https://manus.im/share/brWKUSp51ItvVMBpcXNCZ1?replay=1

扩展知识

Manus 的执行架构可简化为以下四个主要阶段：

- **规划器（Planner）**：基于 LLM 生成总体执行计划，并拆分子任务。
- **智能体/工具执行（Agents/Tools）**：每个子任务可由不同模型或外部 API 完成，如 Claude 进行文字编写、Qwen 负责多语言处理、专用脚本处理数据抓取等
- **状态驱动 & 条件分支**：执行过程中维护统一状态对象，依据中间结果动态决定是否重试、切换方案或提前结束
- **可回溯 & 异步执行**：全过程记录执行“Trace”，可在 Web 控制台回放、Debug，也支持后台长期运行，无需持续人工监控。

19、Computer Use?

Computer Use 是 **Anthropic** 在 **Claude 3.5 Sonnet** 中推出的 **AI 操作计算机的能力**，允许 AI 直接通过 **模拟鼠标点击、键盘输入** 等方式与操作系统和软件交互，实现从“文字对话”到“实际操作”的跨越。

核心原理如下：

1）**API 驱动的自动化交互**：

通过 **操作系统级 API**（如 Windows API、macOS 系统调用），将自然语言指令转化为计算机可执行的操作（如“打开 Chrome 搜索面试鸭”→ 启动浏览器并输入关键词）。

2）**多智能体协作**：

内置 **任务规划代理**（分解任务）、**工具调用代理**（执行操作）、**验证代理**（校验结果），形成流水线式处理（如“生成报告”→ 拆分为“数据获取”“图表绘制”“格式校验”）。

3）**视觉与语义结合**：

利用 **OCR 技术** 识别屏幕内容，结合 **语义理解** 定位目标（如“点击页面右上角的‘登录’按钮”→ 分析页面结构并模拟点击）。

比如 anthropic 官方演示的，下图左边是 AI 相关的思考是指令的展示，中间就是操作浏览器，进行相关搜索，将对应的信息填到右边的表单，来实现**从“文字对话”到“实际操作”的跨越。**

20、ReAct 是什么？说说它的原理？

ReAct（Reasoning and Acting）是一种基于大语言模型的智能体框架，它让模型在生成回答时**交替输出“思考”和“行动”** 步骤，从而在内部推理与外部交互之间形成闭环，能够边“想”边“做”来完成复杂任务。

核心原理

1. **推理（Reasoning）**：模型先通过自然语言生成 “思考过程”，明确是否需要工具、需要什么工具、输入什么参数。
2. **行动（Action）**：根据当前思路选择合适的操作（如调用搜索、工具 API、环境交互等），并输出具体的指令。
3. **观察（Observation）**：执行 Action 后，系统将返回的结果（网页片段、环境反馈等）提供给模型。

4. **循环迭代**：将 Observation 附加到上下文，模型在新的上下文中继续“思考→行动→观察”循环，直到输出最终答案或结束指令

与 Chain-of-Thought（CoT）的区别

- **CoT** 只关注在“Thought”阶段进行长链推理，不支持任何外部交互，容易因缺乏新信息而“凭空想象”或推理死循环。
- **ReAct** 在推理中融入“行动”，能调用搜索、数据库、模拟环境等工具，实时获取新证据，减少幻觉与错误传播

21、Copilot模式Agent模式区别?

1）**Agent（代理智能体模式）**：主要可以自主驱动，大模型独立拆解任务（如规划旅行行程）、调用工具（如API、数据库）完成端到端操作。

比如，用户仅需设定目标（如“安排下周会议”），Agent自主规划、执行并反馈结果。

2）**Copilot（副驾驶协助模式）**：大模型作为“助手”提供实时建议（如代码补全、文案润色），但用户保留最终决策权。

比如，用户明确输入需求（如“写一段 Java 排序代码”），然后多轮交互优化结果

其实还有个 **Embedding(嵌入模式)**，主要是后台辅助，将大模型作为“隐藏组件”集成到现有系统中（如推荐算法、搜索优化），用户无感知。

一句话总结：Embedding是“隐形的数据助手”，Copilot是“协作的智能搭档”，Agent是“全能的AI管家”。

表格对比：

维度	Embedding	Copilot	Agent
交互方式	后台运行，用户无感知	需用户持续输入指令	仅需初始目标，自动闭环执行
自主性	无自主性，被动响应请求	协作型，依赖用户反馈	高度自主，独立拆解任务链
核心技术	微调模型、API集成	提示工程、多轮交互优化	规划/记忆/工具调用/强化学习
典型场景	智能搜索、推荐系统	代码生成、文档辅助	自动化测试、智能客服