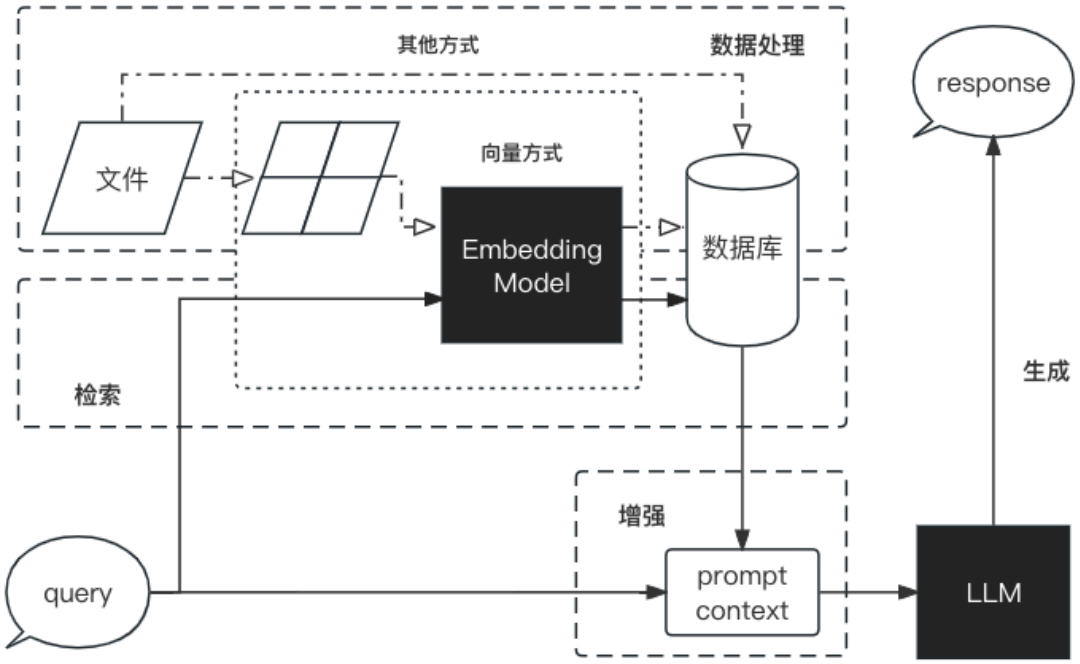
RAG检索增强生成



数据处理阶段

1. 对原始数据进行清洗和处理。
2. 将处理后的数据转化为检索模型可以使用的格式。
3. 将处理后的数据存储在对应的数据库中。

检索阶段

1. 将用户的问题输入到检索系统中，从数据库中检索相关信息。

增强阶段（prompt,context）

1. 对检索到的信息进行处理和增强，以便生成模型可以更好地理解和使用。

生成阶段

1. 将增强后的信息输入到生成模型中，生成模型根据这些信息生成答案。

RAG和微调对比

特征比较	RAG	微调
知识更新	直接更新检索知识库，无需重新训练。信息更新成本低，适合动态变化的数据。	通常需要重新训练来保持知识和数据的更新。更新成本高，适合静态数据。
外部知识	擅长利用外部资源，特别适合处理文档或其他结构化/非结构化数据库。	将外部知识学习到 LLM 内部。

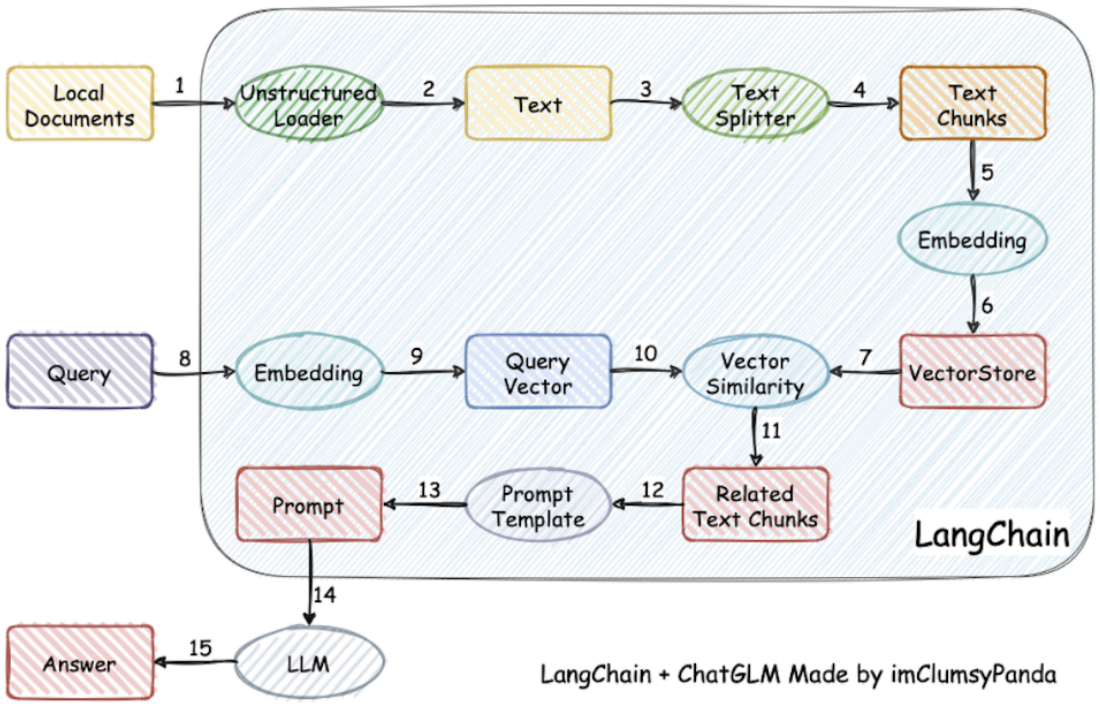
特征比较	RAG	微调
数据处理	对数据的处理和操作要求极低。	依赖于构建高质量的数据集，有限的数据集可能无法显著提高性能。
模型定制	侧重于信息检索和融合外部知识，但可能无法充分定制模型行为或写作风格。	可以根据特定风格或术语调整 LLM 行为、写作风格或特定领域知识。
可解释性	可以追溯到具体的数据来源，有较好的可解释性和可追踪性。	黑盒子，可解释性相对较低。
计算资源	需要额外的资源来支持检索机制和数据库的维护。	依赖高质量的训练数据集和微调目标，对计算资源的要求较高。
推理延迟	增加了检索步骤的耗时	单纯 LLM 生成的耗时
降低幻觉	通过检索到的真实信息生成回答，降低了产生幻觉的概率。	模型学习特定领域的数据有助于减少幻觉，但面对未见过的输入时仍可能出现幻觉。
伦理隐私	检索和使用外部数据可能引发伦理和隐私方面的问题。	训练数据中的敏感信息需要妥善处理，以防泄露。

LangChain框架

借助大模型来开发各种下游应用的工具，为各种大型语言模型应用提供通用接口，从而简化应用程序的开发流程

LangChain 框架可以实现数据感知和环境互动，也就是说，它能够让语言模型与其他数据来源连接，并且允许语言模型与其所处的环境进行互动。

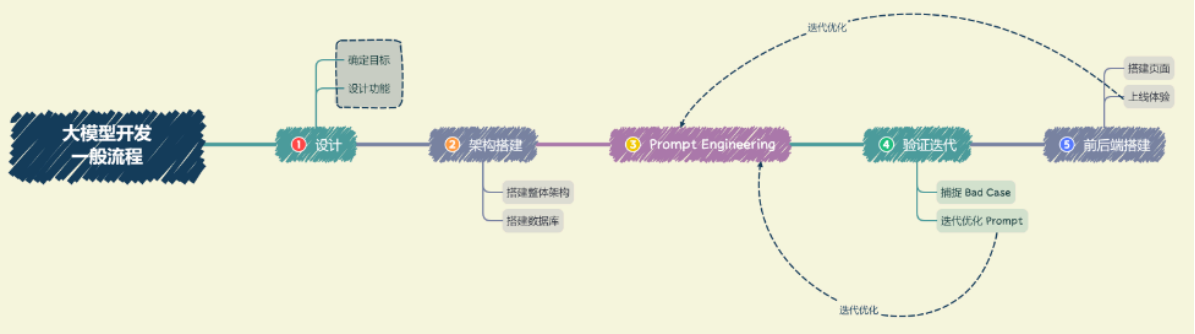
每个椭圆形代表了 LangChain 的一个模块`每个矩形代表了一个数据状态



Langchain核心组件：

- **模型输入/输出（Model I/O）**：与语言模型交互的接口
- **数据连接（Data connection）**：与特定应用程序的数据进行交互的接口
- **链（Chains）**：将组件组合实现端到端应用。比如后续我们会将搭建`检索问答链`来完成检索问答。
- **记忆（Memory）**：用于链的多次运行之间持久化应用程序状态；
- **代理（Agents）**：扩展模型的推理能力。用于复杂的应用的调用序列；
- **回调（Callbacks）**：扩展模型的推理能力。用于复杂的应用的调用序列；

大模型一般开发流程：



Prompt Engineering：

Rule1：编写清晰具体的指令：

1.注意使用分隔符来隔断不同的文本部分，具体符号不重要只要能明确起到隔断作用即可，可以选择用```，"""，< >，等

2.寻求结构化输出，可以要求大模型按照某种格式组织的内容，比如JSON，HTML等

3.可以要求模型检查是否满足条件

4.Few-Shot Prompting,在模型执行实际任务之前，给模型一两个参考样例，让模型了解我们的要求和期望的输出

Rule2:给模型时间去思考：

1.给定一个复杂任务，给出完成该任务的一系列步骤

2.让模型在判断一个问题的回答是否正确之前，先思考出自己的解法，然后再和提供的解答进行对比，判断正确性。

在开发与应用语言模型时，需要注意它们可能生成虚假信息的风险。尽管模型经过大规模预训练，掌握了丰富知识，但它实际上并没有完全记住所见的信息，难以准确判断自己的知识边界，可能做出错误推断。若让语言模型描述一个不存在的产品，它可能会自行构造出似是而非的细节。这被称为`“幻觉”(Hallucination)，是语言模型的一大缺陷。

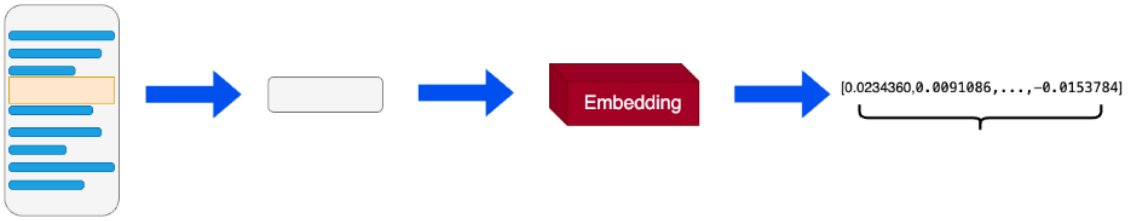
向量知识库：

向量：

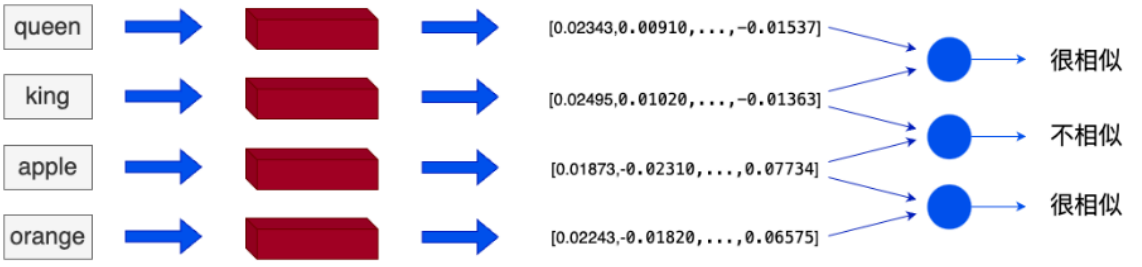
向量中包含了原文本的语义信息，可以通过计算问题与数据库中数据的点积、余弦距离、欧几里得距离等指标，直接获取问题与数据在语义层面上的相似度

向量却可以通过多种向量模型将多种数据映射成统一的向量形式----多模态的基础

词向量：



以单词为单位将每个单词转化为实数向量，词向量背后的主要想法是，相似或相关的对象在向量空间中的距离应该很近。如下图含义相近的单词在向量空间中的位置很接近



通用文本向量：

单词在不同语境中的意思会受到影响，在RAG应用中使用的向量技术一般为通用文本向量(Universal text embedding)，该技术可以对一定范围内任意长度的文本进行向量化，与词向量不同的是向量化的单位不再是单词而是输入的文本，输出的向量会捕捉更多的语义信息

向量数据库：

1.向量数据库是一种专门用于存储和检索向量数据（embedding）的数据库系统。它与传统的基于关系模型的数据库不同，它主要关注的是向量数据的特性和相似性

2.在向量数据库中，数据被表示为向量形式，每个向量代表一个数据项。这些向量可以是数字、文本、图像或其他类型的数据。向量数据库使用高效的索引和查询算法来加速向量数据的存储和检索过程

3.向量数据库通过计算与目标向量的余弦距离、点积等获取与目标向量的相似度。当处理大量甚至海量的向量数据时，向量数据库索引和查询算法的效率明显高于传统数据库

主流的向量数据库:[Chroma],[Weaviate],[Qdrant]

数据处理：

数据读取：

pdf文档：我们可以使用 LangChain 的 PyMuPDFLoader 来读取知识库的 PDF 文件结果会包含 PDF 及其页面的详细元数据，并且每页返回一个文档

md文档：使用LangChain的UnstructuredMarkdownLoader

数据清洗：

我们要删除低质量的、甚至影响理解的文本数据

质量过滤：

分类器过滤：用传统的分类器对文章打分，接近1的就是质量好的文章，可以人为设定一个阈值，比如0.8， 超过阈值的就是高质量文章，分类器的冷启动阶段数据可以从一些权威的网站，比如维基百科、arxiv.org 等，这些站点的文章作为正样本，人工选一些 不入流的N线网站文章作为作为负样本训练分类器

人为设定规则：比如去掉网页标签、 、%20、\u3000等无用字符

Perplexity困惑度：根据之前所有的token预测下一个token生成的概率值，计算公式如下，困惑度越大，说 明语料质量越低

$$
\begin{aligned}
perplexity &= p(s)^{-\frac{1}{n}} \\
&= p(w_1, w_2, \ldots, w_n)^{-\frac{1}{n}} \\
&= \sqrt[n]{\frac{1}{p(w_1, w_2, \ldots, w_n)}} \\
&= \sqrt[n]{\prod_{i=1}^{n} \frac{1}{p(w_i|w_1, w_2, \ldots, w_{i-1})}}
\end{aligned}
$$

利用统计特征过滤：可以根据标点符号分布、符号字比(Symbol-to-WordRatio)、文本长度等过滤

冗余去重：

利用向量数据库：每个新向量存入向量数据库时先计算一下库里面有没有相似度大于阈值(比如0.8)的向量，没有再入库；有就说明重复了，直接丢弃

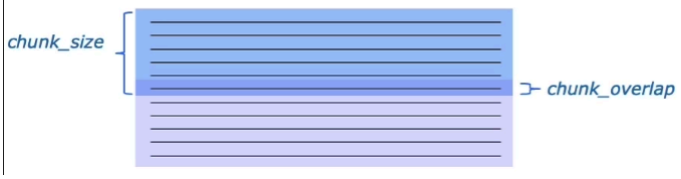
文档分割：

由于单个文档的长度往往会超过模型支持的上下文，导致检索得到的知识太长超出模型的处理能力，因此，在构建向量知识库的过程中，我们往往需要对文档进行分割，将单个文档按长度或者按固定的规则分割成若干个chunk，然后将每个 chunk 转化为词向量，存储到向量数据库中。

在检索时，我们会以 chunk 作为检索的元单位，也就是每一次检索到 k 个 chunk 作为模型可以参考来回答用户问题的知识，这个 k 是我们可以自由设定的

Langchain 中文本分割器都根据 chunk_size (块大小)和 chunk_overlap (块与块之间的重叠大小)进行分割。

langchain.text_splitter.CharacterTextSplitter(
separator: str = "\n\n"
chunk_size=4000,
chunk_overlap=200,
length_function=<builtin function len>,
)
Methods:
create_documents() - Create documents from a list of texts.
split_documents() - Split documents.



如何选择分割方式，往往具有很强的业务相关性——针对不同的业务、不同的源数据，往往需要设定个性化的文档分割方式

构建向量库：

```
from zhipuai_embedding import ZhipuAIEmbeddings
# 定义 Embeddings
embedding = ZhipuAIEmbeddings()
# 定义持久化路径
persist_directory = '../../data_base/vector_db/chroma'
```

```
from langchain.vectorstores.chroma import Chroma
#给定documents，embedding方法，持久化路径，使用Chroma.from_documents创建向量库
vectordb = Chroma.from_documents(
    documents=split_docs,
    embedding=embedding,
    persist_directory=persist_directory  # 允许我们将persist_directory目录保存到磁盘上
)
```

向量检索：

相似度检测

Chroma的相似度搜索使用的是余弦距离：

$$similarity = cos(A, B) = \frac{A \cdot B}{\| A \| \| B \|} = \frac{\sum_1^n a_i b_i}{\sqrt{\sum_1^n a_i^2} \sqrt{\sum_1^n b_i^2}}$$

ai，bi分别是向量A，B的分量

MMR检索

最大边际相关性 (MMR, Maximum marginal relevance) 可以帮助我们在保持相关性的同时，增加内容的丰富度

核心思想是在已经选择了一个相关性高的文档之后，再选择一个与已选文档相关性较低但是信息丰富的文档。这样可以在保持相关性的同时，增加内容的多样性，避免过于单一的结果。

构建检索问答链：

1.加载之前持久化的向量库

此处的embedding层需要跟创建时的embedding层相同

```
vectordb = Chroma(
    persist_directory=persist_directory,  # 允许我们将persist_directory目录保存到磁盘上
    embedding_function=embedding
)
```

2.调用API，创建LLM

```
import os
from dotenv import load_dotenv, find_dotenv
from zhipuai_llm import ZhipuAILLM
_ = load_dotenv(find_dotenv())
zhipuai_api_key = os.environ["ZHIPUAI_API_KEY"]
llm = ZhipuAILLM(model = "glm-4", temperature = 0.1, api_key = zhipuai_api_key)
llm.invoke("请你自我介绍一下自己！")
```

3.构建检索问答链

构建prompt模板

```
from langchain.prompts import PromptTemplate

template = """使用以下上下文来回答最后的问题。如果你不知道答案，就说你不知道，不要试图编造答案。最多使用三句话。尽量使答案简明扼要。总是在回答的最后说“谢谢你的提问！”。
{context}
问题: {question}
"""

QA_CHAIN_PROMPT = PromptTemplate(input_variables=["context","question"],
                                 template=template)
```

基于刚创建的prompt模板创建一个检索链

```
from langchain.chains import RetrievalQA

qa_chain = RetrievalQA.from_chain_type(llm,
                                       retriever=vectordb.as_retriever(),
                                       return_source_documents=True,
                                       chain_type_kwargs=
{"prompt":QA_CHAIN_PROMPT})
```

效果：

LM 对于一些近几年的知识以及非常识性的专业问题，回答的并不是很好。而加上我们的本地知识，就可以帮助 LLM 做出更好的回答。另外，也有助于缓解大模型的“幻觉”问题。

未使用检索问答链：

```
from zhipuai_llm import ZhipuAILLM
llm = ZhipuAILLM(model = "glm-4", temperature = 0.1, api_key = zhipuai_api_key)

llm.invoke("什么是南瓜书？")
```

✓ 6.5s

```
'南瓜书通常指的是《南瓜头的故事》，这是一本儿童文学书籍，作者是美国作家雷蒙德·布里格斯。这本
```

```
prompt_template = """请回答下列问题:
                    {}""".format(question_2)

### 基于大模型的问答
llm.predict(prompt_template)
```

✓ 2.9s

```
'"Prompt Engineering for Developer" 这本书是由多位作者共同编写的，其
```

使用了检索问答链，大模型结合向量库的内容返回：

```
result = qa_chain({"query": "什么是南瓜书？"})
print("大模型+知识库后回答 question_1 的结果：")
print(result["result"])
```

✓ 4.4s

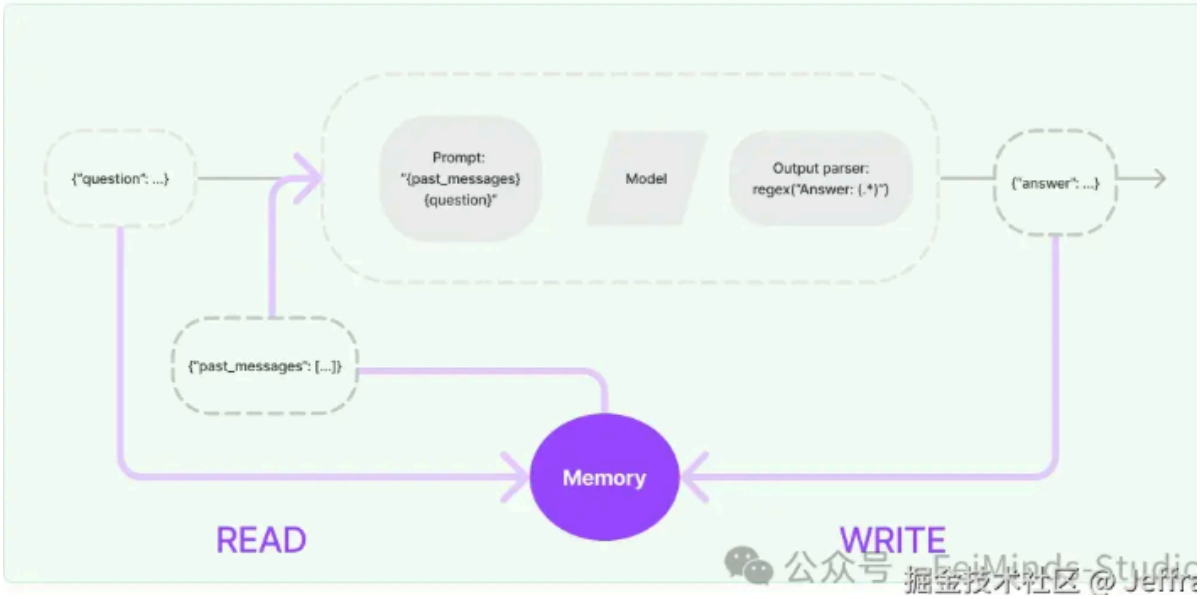
```
大模型+知识库后回答 question_1 的结果：
南瓜书是一部针对周志华教授的《机器学习》（西瓜书）中的公式和概
谢谢你的提问！
```

```
result = qa_chain({"query": "Prompt Engineering for Developer是谁写的？"})
print("大模型+知识库后回答 question_2 的结果：")
print(result["result"])
```

✓ 3.7s

```
大模型+知识库后回答 question_2 的结果：
我不知道《Prompt Engineering for Developer》是由谁写的，因为上下文中没有提供这方面的信息。谢谢你的提问
```

历史对话功能



LangChain 中的储存模块，它保存聊天消息历史记录的列表，这些历史记录将在回答问题时与问题一起传递给聊天机器人，从而将它们添加到上下文中。

Memory

1.ConversationBufferMemory--非常简单的内存形式，它只是将聊天消息列表保存在缓冲区中，并将它们传递到提示模板中

```
from langchain.memory import ConversationBufferMemory

memory = ConversationBufferMemory(
    memory_key="chat_history",  # 默认为history也可以自己设置key，用于区分不同历史对话
    return_messages=True  # 将以消息列表的形式返回聊天记录，而不是单个字符串
)
```

2.ConversationBufferWindowMemory--它只使用最后的 K 个交互。这对于保持最新交互的滑动窗口非常有用，因此缓冲区不会变得太大

```
from langchain.memory import ConversationBufferWindowMemory
# 创建记忆实例
memory1 = ConversationBufferWindowMemory(
k=5,memory_key="chat_history",return_messages=True) # 设置窗口大小为 5
```

对话检索链（ConversationalRetrievalChain）--结合Memory来使用

在检索 QA 链的基础上，增加了处理对话历史的能力。

它的工作流程是:

1. 将之前的对话与新问题合并生成一个完整的查询语句。
2. 在向量数据库中搜索该查询的相关文档。
3. 获取结果后,存储所有答案到对话记忆区。

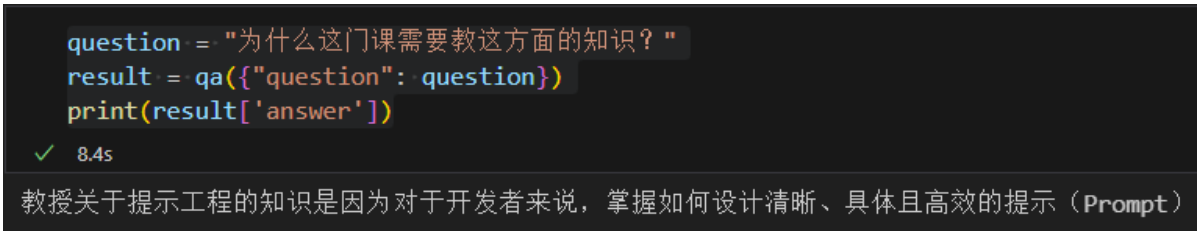
4. 用户可在 UI 中查看完整的对话流程。

使用ConversationBufferMemory的情况：

```
from langchain.chains import ConversationalRetrievalChain
#设置检索
retriever=vectordb.as_retriever()
#实例化对话检索连
qa = ConversationalRetrievalChain.from_llm(
    llm,
    retriever=retriever,
    memory=ConversationBufferMemory(
        memory_key="chat_history",
        return_messages=True  # 将以消息列表的形式返回聊天记录，而不是单个字符串
)
)

question = "我可以学习到关于提示工程的知识吗？"
result = qa({"question": question})
print(result['answer'])
```

可以看到模型知道“这方面的知识”是指的提示工程



查看缓存内容