date = '2025-06-20T15:24:15+08:00'
draft = false
title = '高级RAG'
series =['学习笔记']
series_weight=2
toc = true
showTableOfContents='article.showTableOfContents'

{{< katex >}}

高级 RAG

高级 RAG是在基本RAG流程基础上添加了很多新步骤（子步骤）。以下是本文将讨论的增强点列表，但总体列表并不仅限于这些。

- **Data Indexing Optimizations（数据索引优化）**：使用滑动窗口进行文本分块和有效利用元数据等技术来创建更易于搜索和更有条理的索引。
- **Query Enhancement（查询增强）**：使用同义词或更广泛的术语修改或扩展初始用户查询，以改进相关文档的检索。
- Hybrid Search：将传统的基于关键字的搜索与使用嵌入向量的语义搜索相结合，以处理各种查询复杂性。
- **Fine Tuning Embedding Model（微调嵌入模型）**：调整预先训练的模型以更好地理解特定领域的细微差别，提高检索到的文档的准确性和相关性。
- **Re-ranking and Filtering（重新排序和过滤）**：根据相关性调整检索到的文档的顺序，并过滤掉不太相关的结果以优化最终输出。

1. Data Indexing Optimizations（数据索引优化）

1.1 用于文本分块的滑动窗口

索引文本的一种简单方法是将文本拆分为 n 个部分，将它们转换为嵌入向量，然后将它们存储在向量数据库中。滑动窗口方法创建重叠的文本块，以确保在块的边界处不会丢失任何上下文信息。

```
import nltk
from nltk.tokenize import sent_tokenize
nltk.download('punkt')  # Ensure the punkt tokenizer is downloaded
nltk.download('punkt_tab')
def sliding_window(text, window_size=3):
    """
    Generate text chunks using a sliding window approach.
    Args:
    text (str): The input text to chunk.
    window_size (int): The number of sentences per chunk.
    Returns:
    list of str: A list of text chunks.
    """
```

```
    sentences = sent_tokenize(text)
    print(sentences)
    return [' '.join(sentences[i:i+window_size]) for i in range(len(sentences) -
window_size + 1)]
# Example usage
text = "This is the first sentence. Here comes the second sentence. And here is the third
one. Finally, the fourth sentence."
chunks = sliding_window(text, window_size=3)
for chunk in chunks:
    print(chunk)
    print("-----")
    # here, you can convert the chunk to embedding vector
    # and, save it to a vector database
```

1.2 元数据利用

元数据可以包含文档创建日期、作者或相关标签等信息，这些信息可用于在检索过程中筛选或确定文档的优先顺序，从而增强搜索过程。

```
import numpy as np
import faiss
documents = [
    "Document 1 content here",
    "Content of the second document",
    "The third one has different content",
]
metadata = [
    {"date": "20230101", "tag": "news"},
    {"date": "20230102", "tag": "update"},
    {"date": "20230103", "tag": "report"},
]
# Dummy function to generate embeddings
def generate_embeddings(texts):
    """Generate dummy embeddings for the sake of example."""
    return np.random.rand(len(texts), 128).astype('float32')  # 128-dimensional embeddings
# Generate embeddings for documents
doc_embeddings = generate_embeddings(documents)
# Create a FAISS index for the embeddings (using FlatL2 for simplicity)
index = faiss.IndexFlatL2(128)  # 128 is the dimensionality of the vectors
index.add(doc_embeddings)  # Add embeddings to the index
# Example search function that uses metadata
def search(query_embedding, metadata_key, metadata_value):
    """定义一个搜索函数，它不仅根据向量相似度查找文档，还会根据这些文档关联的元数据（比如日期、标签等）进
行二次筛选。"""
    k = 2  # Number of nearest neighbors to find
    distances, indices = index.search(np.array([query_embedding]), k)  # Perform the
search
    results = []
    for idx in indices[0]:
        if metadata[idx][metadata_key] == metadata_value:
            results.append((documents[idx], metadata[idx]))
```

```
    return results
# Generate a query embedding (in a real scenario, this would come from a similar process)
query_embedding = generate_embeddings(["Query content here"])[0]
# Search for documents tagged with 'update'
matching_documents = search(query_embedding, 'tag', 'update')
print(matching_documents)
```

1.3 MultiVectorRetriever

多向量检索器（MultiVectorRetriever）允许每个文档存储多个向量，这在多种情况下非常有用。LangChain提供了一个基础的MultiVectorRetriever，使得查询这类设置变得简单。这种设置的主要复杂性在于如何为每个文档创建多个向量。这篇笔记涵盖了一些常见的创建向量的方法，并展示了如何使用MultiVectorRetriever。

创建多个向量的方法：

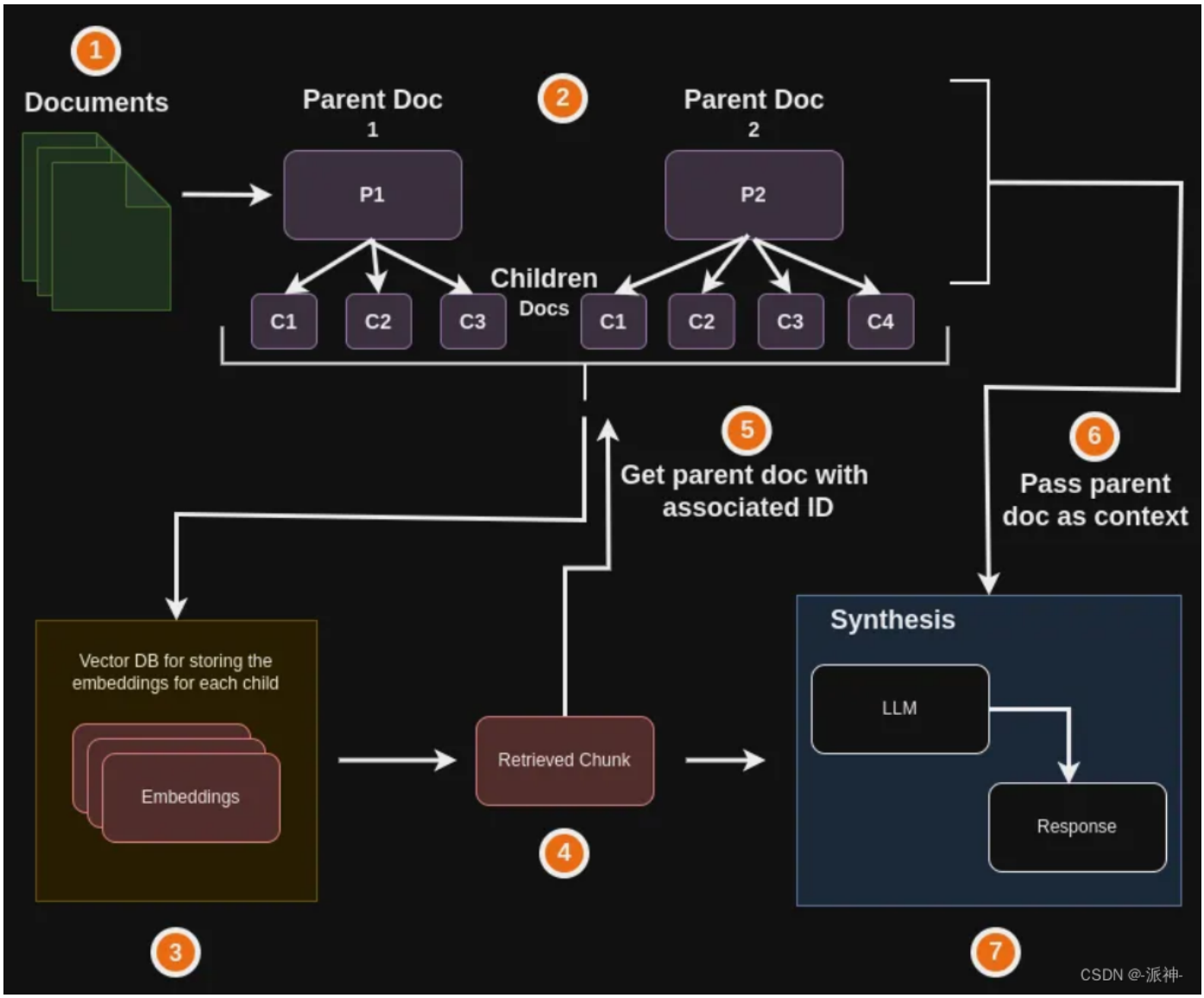
1. **较小的分块**：将文档分割成较小的部分，并对这些部分进行嵌入（例如ParentDocumentRetriever）。
2. **摘要**：为每个文档创建摘要，并将摘要（或代替整个文档）嵌入。
3. **假设性问题**：创建每个文档可能回答的假设性问题，并将这些问题（或代替文档）进行嵌入。

实现示例：

- **向量存储**：使用Chroma进行向量的存储。
- **文档存储**：使用InMemoryByteStore进行文档的存储。
- **检索器初始化**：使用MultiVectorRetriever进行初始化，设置好文档和向量存储。

```
from langchain.retrievers.multi_vector import MultiVectorRetriever
from langchain.storage import InMemoryByteStore
from langchain_openai import OpenAIEmbeddings

vectorstore = Chroma(collection_name="full_documents",
embedding_function=OpenAIEmbeddings())
store = InMemoryByteStore()
retriever = MultiVectorRetriever(vectorstore=vectorstore, byte_store=store,
id_key="doc_id")
```

1.4 ParentDocumentRetriever



在进行文档分割以便检索时，我们通常会遇到几种需求的冲突：

1. **精确的嵌入表示**：我们希望拥有较小的文档，这样它们的嵌入能够更准确地反映其含义。如果文档过长，那么嵌入可能会失去其意义。
2. **保留上下文**：我们需要保证文档足够长，以保留每个文档块的上下文。

父文档检索器（ParentDocumentRetriever） 通过分割和存储小数据块来实现上述平衡。在检索时，它首先获取这些小数据块，然后查找这些块的父ID，并返回这些较大的文档。

> **父文档**指的是小数据块源自的文档。这可以是整个原始文档或者一个更大的数据块。

主要流程如下

1. **索引阶段**：
    - 分割层级：
        - **子文档（Child Documents）**：对原始文档进行**细粒度分块**（例如：小段句子或小段文本），用于向量嵌入和精准检索。
        - **父文档（Parent Documents）**：对原始文档进行**粗粒度分块**（例如：整节、整页），用于提供完整上下文。
    - 存储关系：

        - **向量库（VectorStore）**：存储**子文档**的嵌入向量。其元数据中记录对应的**父文档ID**。
        - **文档存储（DocStore）**：存储**父文档**的原始文本，并通过 ID 与子文档关联。
        - *（可选）字节存储（ByteStore）*：缓存子文档原文，加速返回结果（避免反复切分）。
2. **检索阶段**：
    - **Step 1 - 召回子文档**：用查询向量在 `VectorStore` 中检索最相似的 **K 个子文档**。
    - **Step 2 - 关联父文档**：通过子文档的元数据 `parent_doc_id` 找到对应的 **父文档原文**。
    - **最终返回**：**父文档内容**（而非原子文档），提供完整上下文。

代码示例：

- 数据准备

```
from langchain.embeddings import HuggingFaceBgeEmbeddings
from langchain.document_loaders import TextLoader

#创建BAAI的embedding
bge_embeddings = HuggingFaceBgeEmbeddings(model_name="BAAI/bge-small-zh-v1.5")

#创建loaders
loaders = [
    TextLoader("./docs/华为智驾遥遥领先.txt",encoding='utf8'),
    TextLoader("./docs/小米SU7遥遥领先.txt",encoding='utf8'),
]
docs = []
for loader in loaders:
    docs.extend(loader.load())
```

- 创建父文档检索器

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.retrievers import ParentDocumentRetriever
from langchain.storage import InMemoryStore
from langchain.vectorstores import Chroma

# 创建文档分割器,设置块大小为200
child_splitter = RecursiveCharacterTextSplitter(chunk_size=200)
# 创建向量数据库对象
vectorstore = Chroma(
    collection_name="full_documents", embedding_function=bge_embeddings
)
# 创建内存存储对象
store = InMemoryStore()
#创建父文档检索器
retriever = ParentDocumentRetriever(
    vectorstore=vectorstore, #指定所使用的向量数据库
    docstore=store, #原始文档存储器
    child_splitter=child_splitter,#子文档分割器
```

```
)
#添加文档集
retriever.add_documents(docs, ids=None)
```

> 一旦完成添加原始文档的工作以后，所有的原始文档就会被child_splitter切割成一个个小的文档块，并且为小文档块与原始文档建立了索引关系，即通过小文档块的Id便能找到其对于的原始文档。

- 检索

```
#搜索与用户问题相似度较高的子文档块
sub_docs = vectorstore.similarity_search("小米SU7智能驾驶系统？")
print(sub_docs[0].page_content)
#检索原始文档的全部内容
retrieved_docs = retriever.get_relevant_documents("小米SU7智能驾驶系统？")
print(retrieved_docs[0].page_content)
```

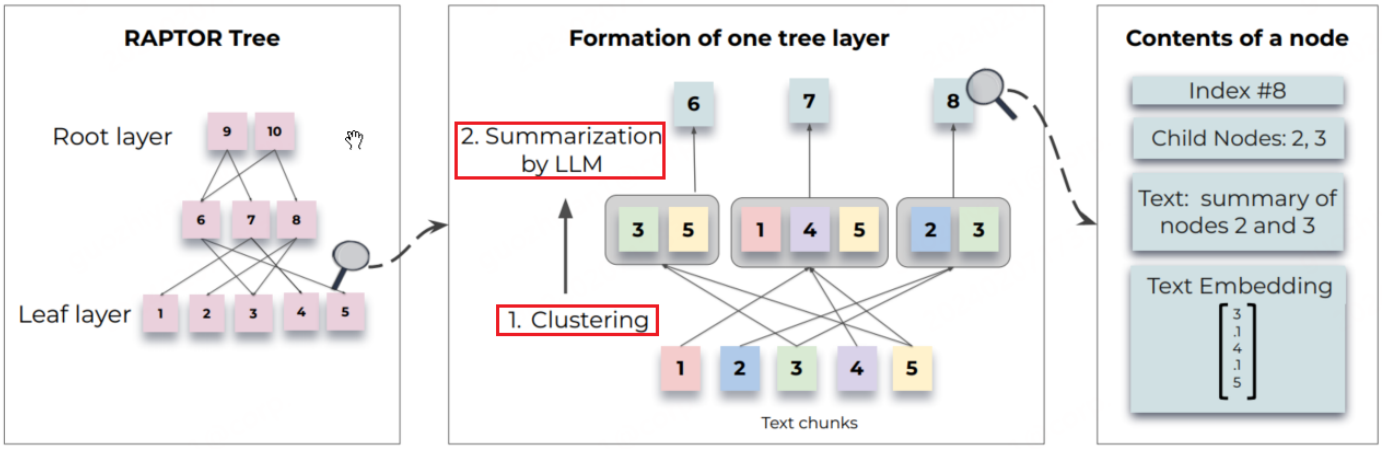
1.5 RAPTOR

RAPTOR旨在解决**超长文档的语义理解和精准检索**难题。它通过递归构建树状文档结构，在复杂内容中实现**多层级语义聚合**，显著优于传统分块检索方法。

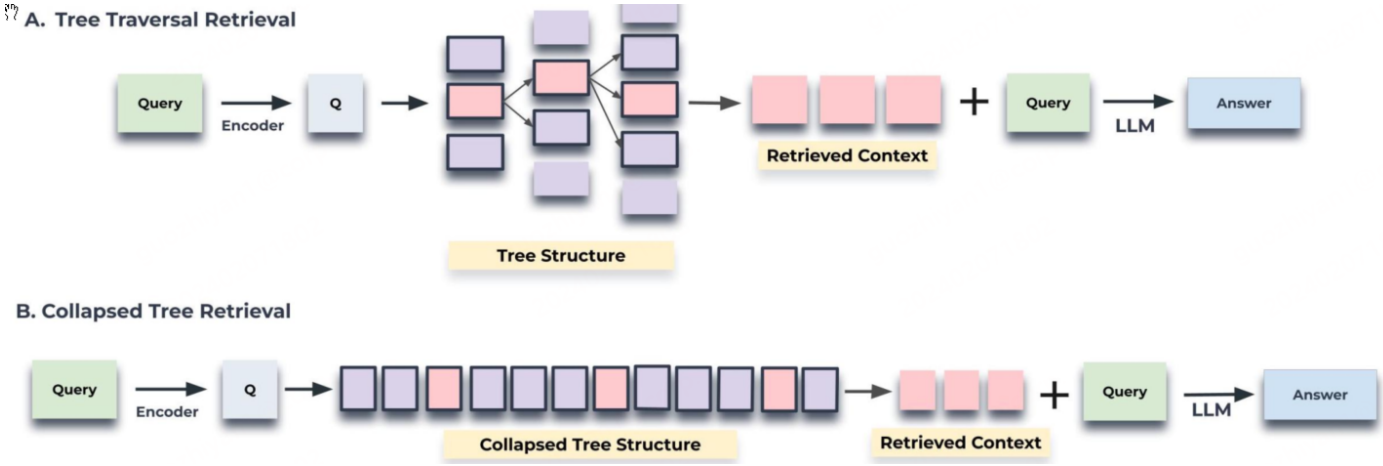
基本流程：

- 分块与嵌入。
  - 按 100 的大小对文本进行分块，如果一个句子长度超过 100，则直接将句子作为一个文本块，保证块内语义的一致性。（如何断句也很重要！）
  - 对文本块进行 embedding。
- 递归的构建 RAPTOR 树。文本块及其 embedding 作为树的叶子节点。
  - 通过聚类把相似的块聚在一起。
  - 利用语言模型为簇内的文本生成总结，并为总结生成 embedding，也作为树的一个节点。
  - 递归执行上述过程。
- 查询。即如何检索相关的块。文中提供了两种策略：
  - Tree Traversal Retrieval。遍历树的每一层，剪枝并选择最相关的节点。
  - Collapsed Tree Retrieval。评估整个树中每个节点的相关性，找到最相关的几个。

RAPTOR 树构建流程：



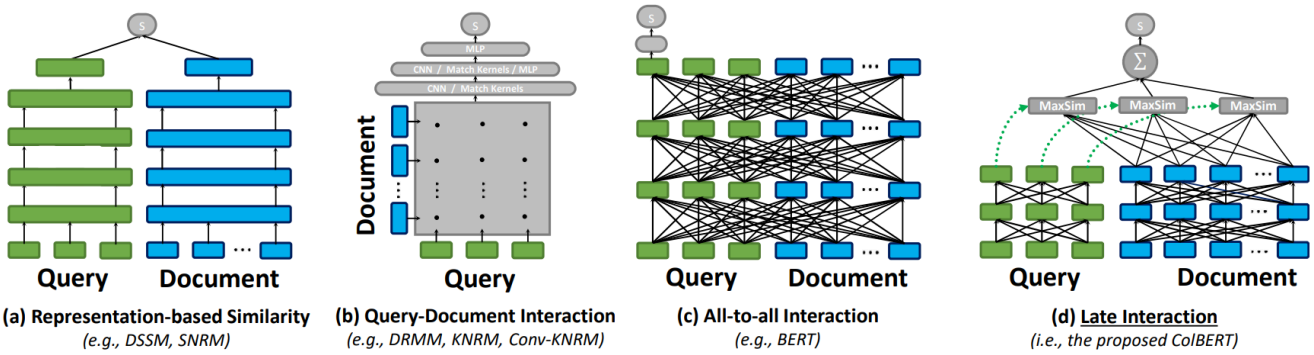
RAPTOR 树的两种查询策略：



- **树遍历**：树遍历方法首先根据其与查询嵌入的余弦相似度选择前 k 个最相关的根节点。这些选定节点的子节点在下一层被考虑，并且根据其与查询向量的余弦相似性再次从此池中选择前 k 个节点。重复此过程，直到我们到达叶节点。最后，将所有选定节点的文本连接起来，形成检索到的上下文。
- **折叠树方法**：提供了一种更简单的方法来搜索相关信息，方法是同时考虑树中的所有节点，如图所示。这种方法不是逐层进行，而是将多层树展平为一层，实质上是将所有节点带到同一级别进行比较。

代码示例：[代码]

1.6 ColBERT



> (a) 基于表示的相似性 (Representation-based Similarity): 在这种方法中,文档片段通过某些深度神经网络进行离线编码 (即预先处理),而查询片段则以类似的方式进行在线编码 (即实时处理)。然后计算查询与所有文档之间的成对相似性（通常是cos相似度）得分,并返回得分最高的几个文档。

> (b) 查询-文档交互 (Query-Document Interaction): 在这种方法中,通过使用 n-gram (即连续的 n 个词组成的序列) 计算查询和文档中所有单词之间的词语和短语级关系,作为一种特征工程的形式。这些关系被转换为交互矩阵,然后作为输入提供给卷积网络。
>
> (c) 全对全交互 (All-to-all Interaction): 这可以被视为方法 (b) 的泛化,它考虑了查询和文档内部以及彼此之间的所有成对交互。这是通过自注意力机制 (self-attention) 实现的。在实践中,通常使用 BERT (Bidirectional Encoder Representations from Transformers) 来实现,因为它是双向的,因此能够真正模拟所有成对交互,而不仅仅是因果关系。

ColBERT 建立在 BERT 模型之上，但它的设计与众不同。传统方法将句子编码为单一向量，而 ColBERT 为每个 token 生成独立的上下文嵌入向量。ColBERT 的核心思想是**延迟交互**，即在查询和文档编码过程中保持独立，但在计算相似度时进行细粒度的交互。

> 考虑查询“What is BGE?”，单一向量可能因句子整体平均而削弱“BGE”的特征。而 ColBERT 赋予“BGE”独立向量，保留了细粒度信息，非常适合需要局部匹配的任务，如信息检索。

- **延迟交互机制机制流程**：
  - 分别生成查询 Q 和文档 D 的 token 向量集合，保持独立性。
  - 对每个查询向量 qi，在文档向量中寻找最匹配的 dj，通过点积 qi·dj 计算相似性。
  - 将所有查询 token 的最大匹配得分求和，得到最终得分。
- **为何延迟**：
  提前融合可能导致语义损失，尤其在长句子中。延迟交互则保留了每个 token 的独立性，直到需要匹配时才进行计算。
- **简单示例**：
  查询“What is BGE?”与文档“BGE is a model”对比。ColBERT 能直接匹配“BGE”与“BGE”，避免整体平均带来的模糊。换句话说，可以理解成他是匹配局部与局部之间的相关性。

使用 ColBERT 查找最相关的前 K 个文档计算过程包括:

- 批量点积计算:用于计算词语级别的相似度。每一个词都和整个文档进行计算
- 最大池化 (max-pooling):在文档词语上进行操作,找出每个查询词语的最高相似度。
- 求和:对查询词语的相似度分数进行累加,得出文档的总体相关性分数。
- 排序:根据总分对文档进行排序。

代码示例：

```
from ragatouille import RAGPretrainedModel

RAG = RAGPretrainedModel.from_pretrained("jinaai/jina-colbert-v2")
docs = [
    "ColBERT is a novel ranking model that adapts deep LMs for efficient retrieval.",
    "Jina-ColBERT is a ColBERT-style model but based on JinaBERT so it can support both 8k
context length, fast and accurate retrieval.",
]
RAG.index(docs, index_name="demo")
query = "What does ColBERT do?"
results = RAG.search(query)
```

2. Query Enhancement（查询增强）

2.1 查询重写

目的:使查询更加具体和详细，提高检索相关信息的可能性。

方案:重写的确认样不仅与原始查询相似，而且还**提供不同的角度或透视图**，从而提高最终生成的质量和深度。

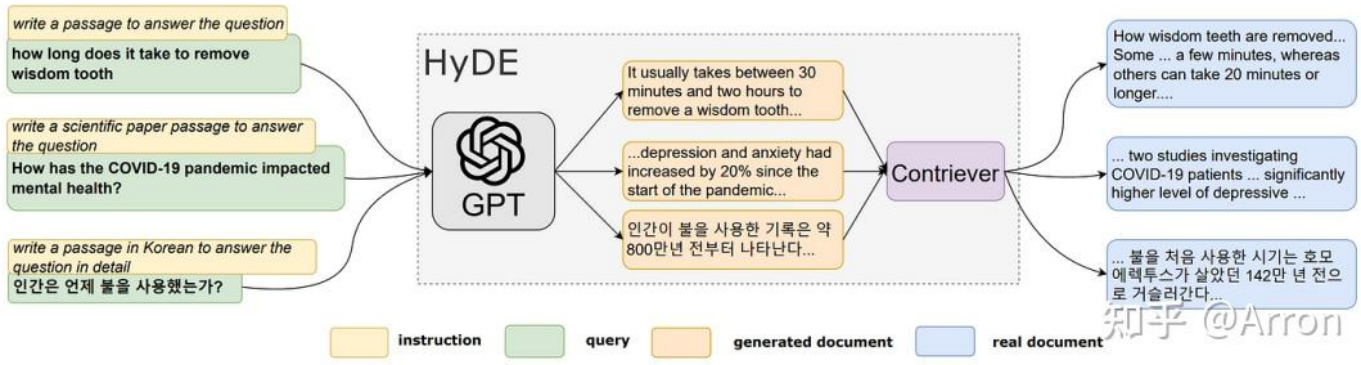
示例代码：

```
import os

from langchain_community.utilities import DuckDuckGoSearchAPIWrapper
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnablePassthrough
from langchain_openai import ChatOpenAI
os.environ["DASHSCOPE_API_KEY"]="<your-api-key>"
os.environ["ALIYUN_BASE_URL"]="https://dashscope.aliyuncs.com/compatible-mode/v1"



model = ChatOpenAI(
    api_key=os.getenv("DASHSCOPE_API_KEY"),
    base_url="https://dashscope.aliyuncs.com/compatible-mode/v1",
    model="qwen-plus",  # 此处以qwen-plus为例，您可按需更换模型名称。模型列表：
https://help.aliyun.com/zh/model-studio/getting-started/models
    # other params...
)

search = DuckDuckGoSearchAPIWrapper()

base_template = """Answer the users question based only on the following context:
<context>
{context}
</context>
Question: {question}
"""

base_prompt = ChatPromptTemplate.from_template(base_template)

def june_print(msg, res):
    print('-' * 100)
    print(msg)
    print(res)
def retriever(query):
    return search.run(query)
```

```
def withoutRewrite(query):
    chain = (
        {"context": retriever, "question": RunnablePassthrough()}
        | base_prompt
        | model
        | StrOutputParser()
    )

    june_print(
        'The result of query:',
        chain.invoke(query)
    )
    june_print(
        'The result of the searched contexts:',
        retriever(query)
    )


def withRewrite(query):

    rewrite_template = """Provide a better search query for \
    web search engine to answer the given question, end \
    the queries with '**'. Question: \
    {x} Answer:"""
    rewrite_prompt = ChatPromptTemplate.from_template(rewrite_template)
    def _parse(text):
        return text.strip("**")
    rewriter = rewrite_prompt |model | StrOutputParser() | _parse
    june_print(
        'Rewritten query:',
        rewriter.invoke({"x": query})
    )

    rewrite_retrieve_read_chain = (
        {
            "context": {"x": RunnablePassthrough()} | rewriter | retriever,
            "question": RunnablePassthrough(),
        }
        | base_prompt
        | model
        | StrOutputParser()
    )
    june_print(
        'The result of the rewrite_retrieve_read_chain:',
        rewrite_retrieve_read_chain.invoke(query)
    )

query = "The NBA champion of 2020 is the Los Angeles Lakers! Tell me what is langchain
framework?"
withRewrite(query)
```

2.1.1 假设文档嵌入（HyDE）

论文《Precise Zero-Shot Dense Retrieval without Relevance Labels》提出了一种基于假设文档嵌入（HyDE）的方法，主要过程如图2所示：

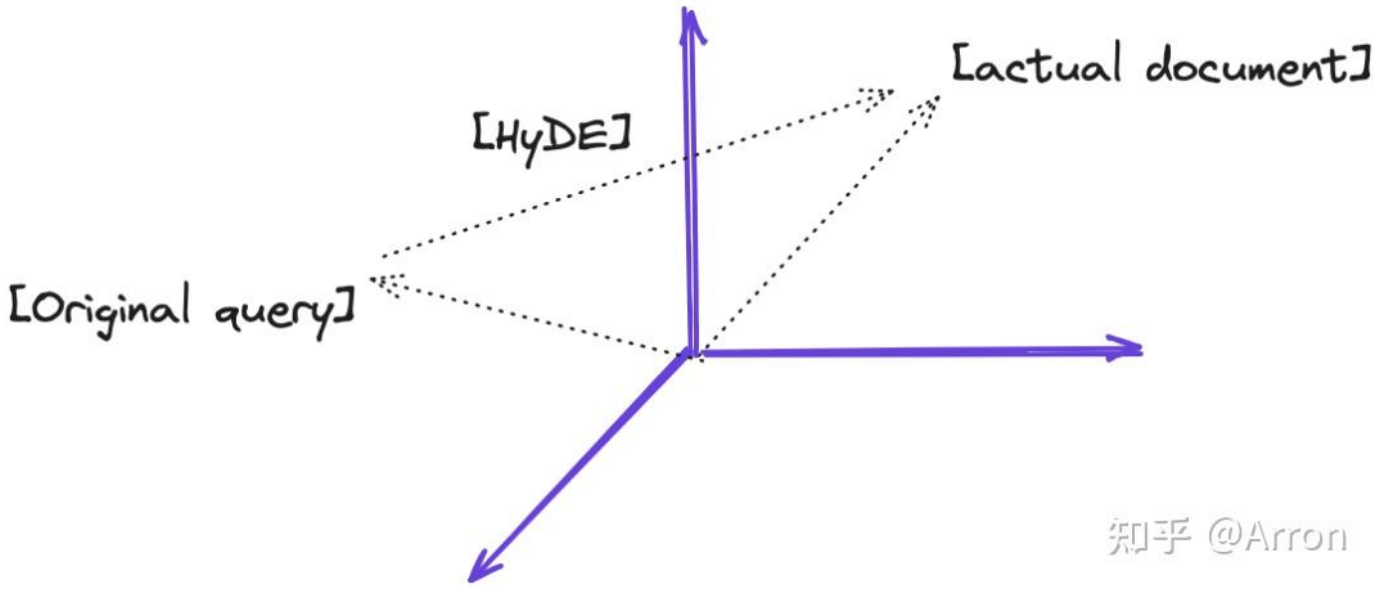


该过程主要分为四个步骤：

1. 使用LLM基于查询生成k个假设文档。这些生成的文件可能不是事实，也可能包含错误，但它们应该于相关文件相似。此步骤的目的是通过LLM解释用户的查询。
2. 将生成的假设文档输入编码器，将其映射到密集向量f(dk)，编码器具有过滤功能，过滤掉假设文档中的噪声。这里，dk表示第k个生成的文档，f表示编码器操作。
3. 使用给定的公式计算以下k个矢量的平均值:

$$v = \frac{1}{N}\sum_{k=1}^{N} f(d_k) \tag{2}$$

4. 使用向量v从文档库中检索答案。如步骤3中所建立的，该向量保存来自用户的查询和所需答案模式的信息，这可以提高回忆。



示例代码：

```
import os
from langchain_community.vectorstores import FAISS
from langchain_core.prompts import PromptTemplate
```

```
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnablePassthrough, RunnableLambda
from openai import OpenAI as OpenAI_Client
from typing import List
from langchain.embeddings.base import Embeddings
from langchain_openai import ChatOpenAI
os.environ["DASHSCOPE_API_KEY"]="<your-api-key>"
os.environ["ALIYUN_BASE_URL"]="https://dashscope.aliyuncs.com/compatible-mode/v1"
client = OpenAI_Client(
    # 若没有配置环境变量，请用阿里云百炼API Key将下行替换为：api_key=
    api_key=os.getenv("DASHSCOPE_API_KEY"),
    base_url=os.getenv("ALIYUN_BASE_URL"),
)


class AliyunEmbeddings(Embeddings):
    def __init__(self, client, model="text-embedding-v3"):
        self.client = client
        self.model = model

    def embed_documents(self, texts: List[str]) -> List[List[float]]:
        """对多个文档进行嵌入"""
        return [self.embed_query(text) for text in texts]

    def embed_query(self, text: str) -> List[float]:
        """对单个查询进行嵌入"""
        print(text)
        response = self.client.embeddings.create(
            input=text,
            model=self.model
        )
        return response.data[0].embedding


# 示例文档 - 可替换为实际业务数据
docs = [
    "阿里云创立于2009年，是全球领先的云计算和人工智能科技公司",
    "通义千问（Qwen）是阿里云推出的大语言模型系列，包括多个不同规模版本",
    "Qwen-plus是基于混合专家架构的先进大语言模型，支持中文、英文等多语言任务",
    "Text-Embedding-V3是阿里云推出的文本嵌入模型，支持中文向量化任务",
    "阿里云在杭州、上海、北京和深圳设有研发中心",
    "通义千问支持128K上下文长度，适用于长文本理解任务",
    "DashScope是阿里云推出的模型服务平台，提供Qwen等多种模型API",
    "2023年阿里云发布了Qwen-VL多模态大模型，支持图像理解任务",
    "PaiRAG是阿里云提供的检索增强生成解决方案，支持定制化知识库",
    "阿里云机器学习平台PAI支持模型训练、部署和推理全流程"
]

# 使用阿里云Text-Embedding-V3创建向量数据库
embeddings = AliyunEmbeddings(client=client, model="text-embedding-v3")

vectorstore = FAISS.from_texts(docs, embeddings)
```

```
retriever = vectorstore.as_retriever(search_kwargs={"k": 3})

# 1. 生成假设文档的提示模板
hyde_prompt = PromptTemplate.from_template(
    """根据用户的问题，生成一个假设性的答案文档。即使你不确定，也请生成包含关键信息的全面回答。

问题：{query}
假设性回答：
"""
)

# 2. 最终答案生成的提示模板
qa_prompt = PromptTemplate.from_template(
    """基于以下上下文信息，回答问题：
{context}

原始问题：{query}
如果上下文无法回答问题，请说明该信息不在知识库中。

正式回答：
"""
)

# 初始化Qwen-plus模型
llm=ChatOpenAI(
    api_key=os.getenv("DASHSCOPE_API_KEY"),
    base_url=os.environ["ALIYUN_BASE_URL"],
    model="qwen-plus",  # 此处以qwen-plus为例，您可按需更换模型名称。模型列表：
https://help.aliyun.com/zh/model-studio/getting-started/models
    # other params...
)

# 创建生成假设文档的链
hyde_chain = hyde_prompt | llm | StrOutputParser()


# 定义格式化文档的函数
def format_docs(docs):
    return "\n\n".join(f"• {doc.page_content}" for doc in docs)


# 构建完整的HyDE RAG流程
hyde_rag_chain = (
        RunnablePassthrough.assign(query=lambda x: x)
        .assign(hypothetical_document=hyde_chain)
        .assign(context=lambda x:
format_docs(retriever.invoke(x["hypothetical_document"])))
        .assign(answer=qa_prompt | llm | StrOutputParser())
        | {
            "original_query": lambda x: x["query"],
            "generated_hypothetical": lambda x: x["hypothetical_document"],
            "retrieved_context": lambda x: x["context"],
```

```
            "final_answer": lambda x: x["answer"]
        }
)


# 运行链并打印结果
def run_query(query):
    print(f"\n{'=' * 50}\n查询: {query}\n{'=' * 50}")
    result = hyde_rag_chain.invoke({"query": query})

    print(f"\n生成的假设文档:\n{'-' * 40}")
    print(result["generated_hypothetical"])

    print(f"\n检索到的相关上下文:\n{'-' * 40}")
    print(result["retrieved_context"])

    print(f"\n最终回答:\n{'-' * 40}")
    print(result["final_answer"])

    return result


# 示例查询
if __name__ == "__main__":
    queries = [
        "阿里云的主要研发中心在哪里？",
        # "Qwen-plus有哪些技术特点？",
        # "Text-Embedding-V3模型的主要用途是什么？",
        # "通义千问支持多模态任务吗？"
    ]

    for query in queries:
        run_query(query)
        print("\n\n")
```

2.1.2 Step-Back提示

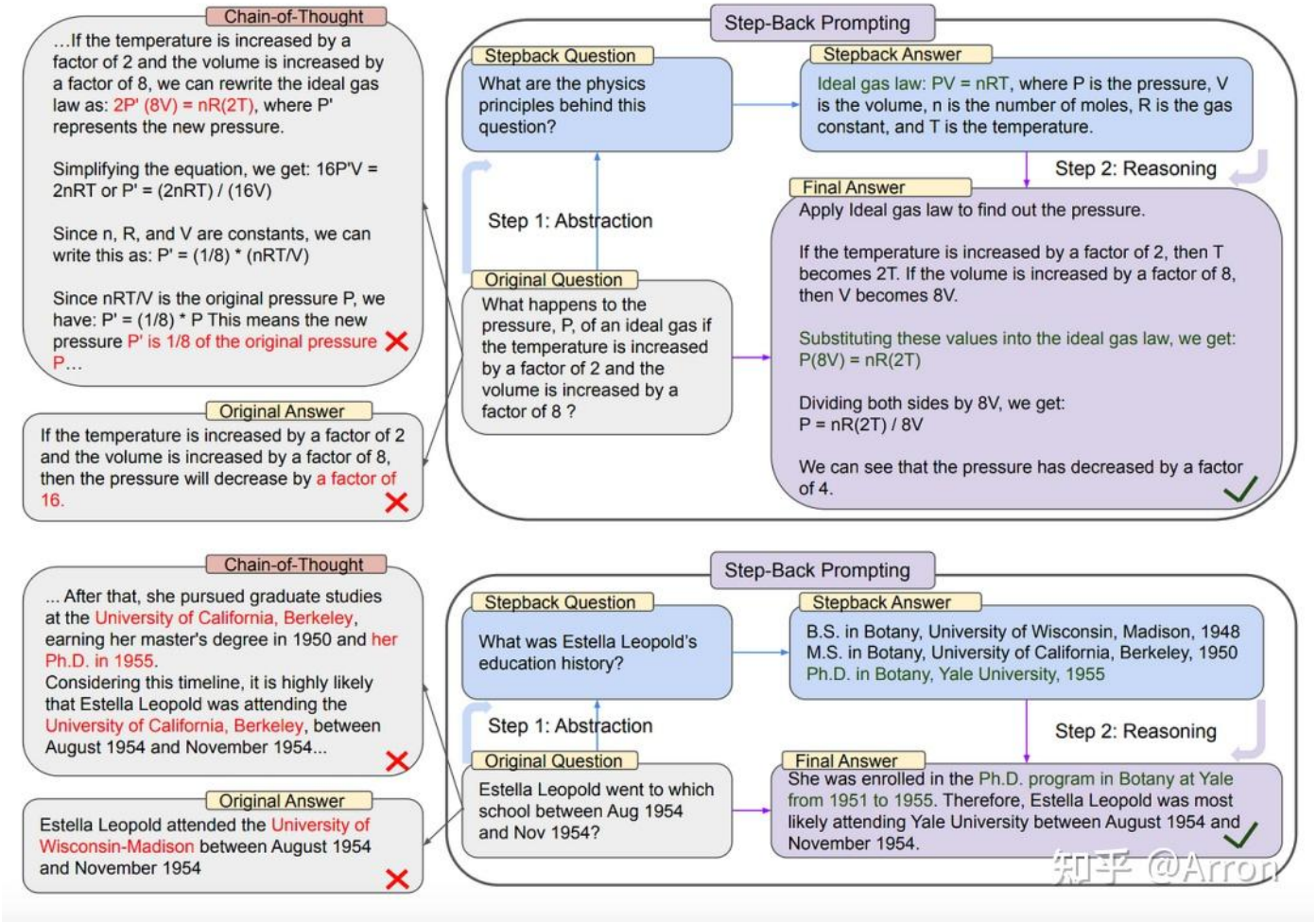
STEP-BACK PROMPING是一种简单的提示技术，使LLM能够从包含特定细节的实例中抽象、提取高级概念和基本原理。其思想是将“step-back问题”定义为从原始问题派生出的更抽象的问题。

包括两个基本步骤：

- **抽象**：最初，我们提示LLM提出一个关于高级概念或原理的广泛问题，而不是直接响应查询。然后，我们检索关于所述概念或原理的相关事实。
- **推理**：LLM可以根据这些关于高级概念或原理的事实推导出原始问题的答案。我们称之为抽象推理。

例如，如果查询包含大量细节，LLM很难检索相关事实来解决任务。如图5中的第一个例子所示，对于物理问题“如果温度增加2倍，体积增加8倍，理想气体的压力P会发生什么？”在直接推理该问题时，LLM可能会偏离理想气体定律的第一原理。

同样，由于特定的时间范围限制，“Estella Leopold在1954年8月至1954年11月期间上过哪所学校？”这个问题很难直接解决。



示例代码：

```
import os
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate, FewShotChatMessagePromptTemplate
from langchain_core.runnables import RunnableLambda
from langchain_openai import ChatOpenAI
from langchain_community.utilities import DuckDuckGoSearchAPIWrapper

os.environ["DASHSCOPE_API_KEY"]="<your-api-key>"
os.environ["ALIYUN_BASE_URL"]="https://dashscope.aliyuncs.com/compatible-mode/v1"


model = ChatOpenAI(
    api_key=os.getenv("DASHSCOPE_API_KEY"),
    base_url="https://dashscope.aliyuncs.com/compatible-mode/v1",
    model="qwen-plus",  # 此处以qwen-plus为例，您可按需更换模型名称。模型列表：
https://help.aliyun.com/zh/model-studio/getting-started/models
    # other params...
)

def june_print(msg, res):
```

```
    print('-' * 100)
    print(msg)
    print(res)

question = "如果温度增加2倍，体积增加8倍，理想气体的压力P会发生什么？"

base_prompt_template = """You are an expert of world knowledge. I am going to ask you a
question. Your response should be comprehensive and not contradicted with the following
context if they are relevant. Otherwise, ignore them if they are not relevant.

{normal_context}
Original Question: {question}
Answer:"""

base_prompt = ChatPromptTemplate.from_template(base_prompt_template)

search = DuckDuckGoSearchAPIWrapper(max_results=4)
def retriever(query):
    return search.run(query)
def base():
    base_chain = (
        {
            # Retrieve context using the normal question (only the first 3 results)
            "normal_context": RunnableLambda(lambda x: x["question"]) | retriever,
            # Pass on the question
            "question": lambda x: x["question"],
        }
        | base_prompt
        | model
        | StrOutputParser()
    )

    june_print('The searched contexts of the original question:', retriever(question))
    june_print('The result of base_chain:', base_chain.invoke({"question": question}) )

def step_back():
    # Few Shot Examples
    examples = [
        {
            "input": "Could the members of The Police perform lawful arrests?",
            "output": "what can the members of The Police do?",
        },
        {
            "input": "Jan Sindel’s was born in what country?",
            "output": "what is Jan Sindel’s personal history?",
        },
    ]
    # We now transform these to example messages
    example_prompt = ChatPromptTemplate.from_messages(
        [
            ("human", "{input}"),
            ("ai", "{output}"),
```

```
        ]
    )
    few_shot_prompt = FewShotChatMessagePromptTemplate(
        example_prompt=example_prompt,
        examples=examples,
    )
    step_back_prompt = ChatPromptTemplate.from_messages(
        [
            (
                "system",
                """You are an expert at world knowledge. Your task is to step back and
paraphrase a question to a more generic step-back question, which is easier to answer.
Here are a few examples:""",
            ),
            # Few shot examples
            few_shot_prompt,
            # New question
            ("user", "{question}"),
        ]
    )
    step_back_question_chain = step_back_prompt | model | StrOutputParser()


    june_print('The step-back question:', step_back_question_chain.invoke({"question":
question}))
    june_print('The searched contexts of the step-back question:',
               retriever(step_back_question_chain.invoke({"question": question})))
    response_prompt_template = """You are an expert of world knowledge. I am going to ask
you a question. Your response should be comprehensive and not contradicted with the
following context if they are relevant. Otherwise, ignore them if they are not relevant.
    {normal_context}
    {step_back_context}
    Original Question: {question}
    Answer:"""
    response_prompt = ChatPromptTemplate.from_template(response_prompt_template)


    step_back_chain = (
            {
                # Retrieve context using the normal question
                "normal_context": RunnableLambda(lambda x: x["question"]) | retriever,
                # Retrieve context using the step-back question
                "step_back_context": step_back_question_chain | retriever,
                # Pass on the question
                "question": lambda x: x["question"],
            }
            | response_prompt
            | model
            | StrOutputParser()
    )
    june_print('The result of step_back_chain:', step_back_chain.invoke({"question":
question}))
```

```
step_back()
```

2.2 混合检索

- 使用 Elasticsearch 作为传统搜索机制，并使用 faiss 作为向量数据库进行语义搜索。代码
- 使用bm25 和向量检索实现。

```
    def get_ensemble_retriever(self, k=3):
        bm25 = self.get_bm25_retriever(k)
        vector = self.get_vector_retriever(k)
        return EnsembleRetriever(
            retrievers=[bm25, vector],
            weights=[0.5, 0.5]
        )
```

3. 重新排序和过滤

在 RAG 工作流中，从检索器（Retriever）返回的候选文档通常包含冗余或无关的内容。通过重新排序（Reranking）和过滤，可以提升模型生成的最终答案的准确性和相关性。重新排序步骤主要依赖于预训练的交叉编码器模型（Cross Encoder），该模型能根据输入和上下文之间的语义相关性重新调整候选文档的优先级。

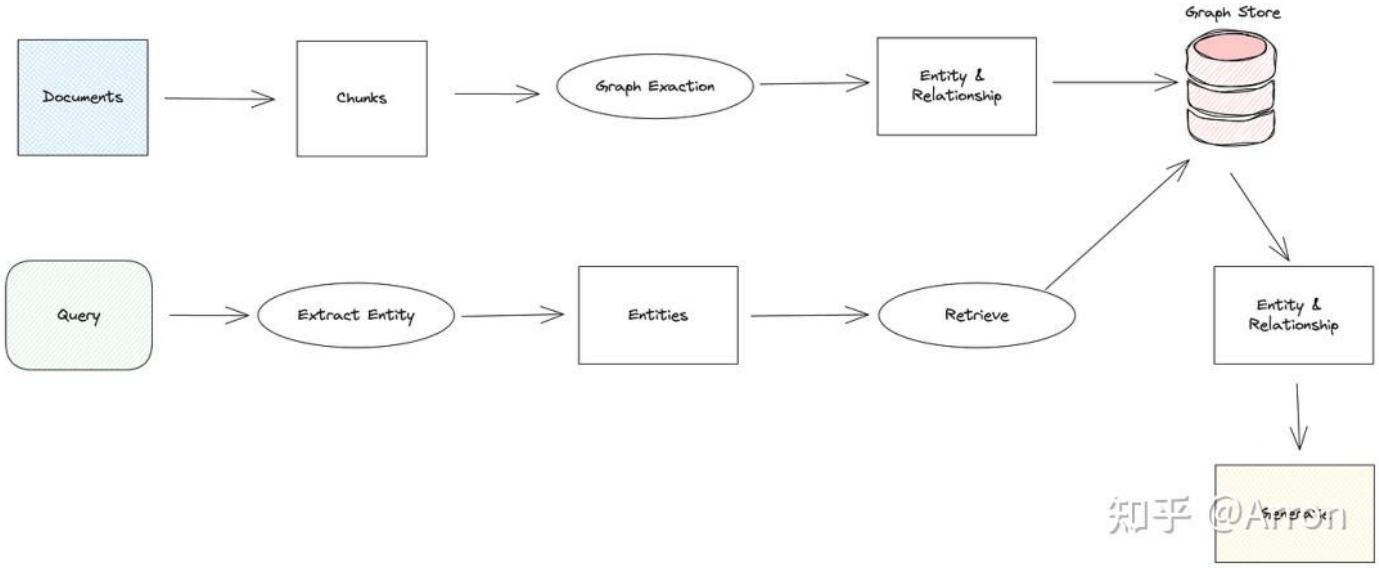
示例代码

以下代码展示了如何实现基于 `HuggingFaceCrossEncoder` 的文档重新排序器：

```
    def local_reranker(retriever, top_n=3):
        model = HuggingFaceCrossEncoder(model_name="BAAI/bge-reranker-base")
        compressor = CrossEncoderReranker(model=model, top_n=top_n)
        return ContextualCompressionRetriever(
            base_compressor=compressor,
            base_retriever=retriever
        )
```

4. 使用知识图谱改进 RAG 检索

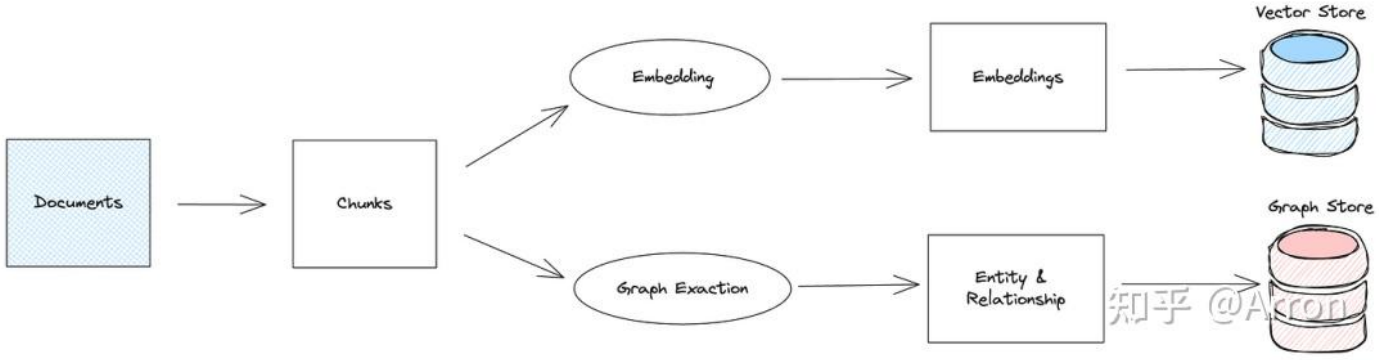
知识图谱是使用图结构来表示实体及其在现实世界中的关系并对其进行建模的一种技术方法。它将信息组织为节点（实体）和边（关系），形成一个有机网络，可以有效地存储、查询和分析复杂的知识。知识图谱的核心在于它使用三元组 （entity-relationship-entity） 来描述实体之间的关联。



通过将文档提取到实体和关系中，知识图谱可以显著压缩文档块，从而可以将所有相关文档提交到LLM。

知识图谱RAG与Base RAG区别

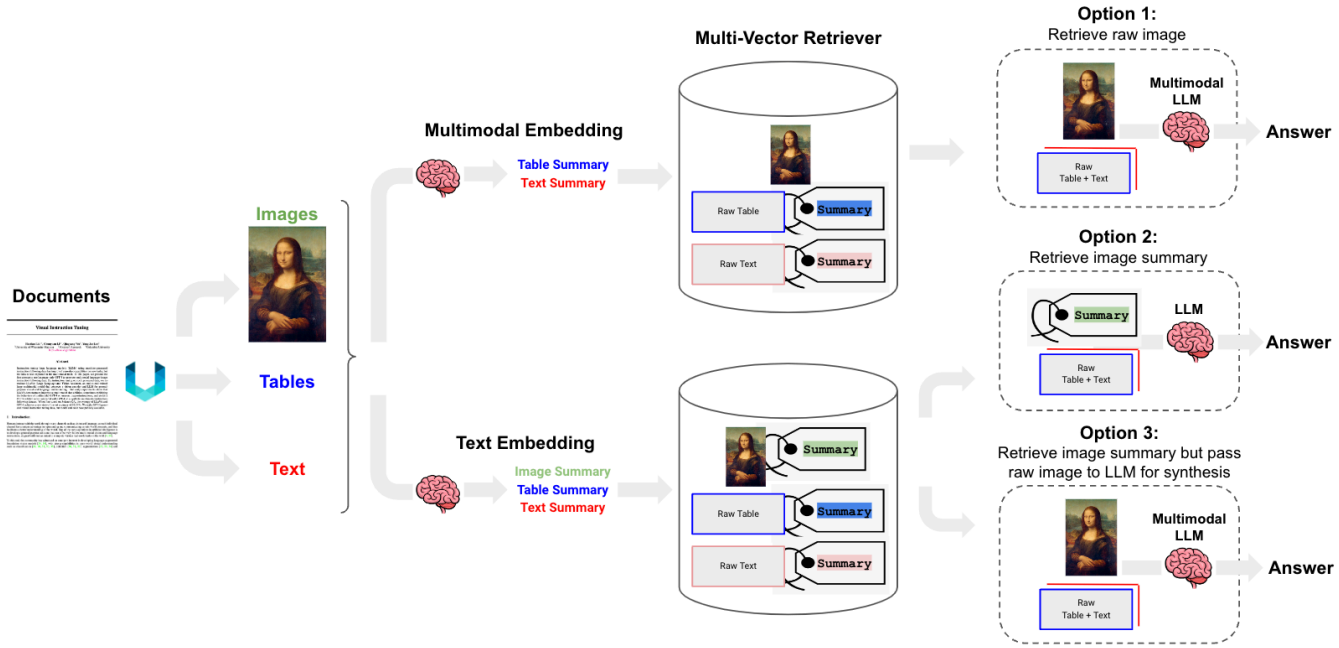
- 知识图谱 RAG 使用图形结构来表示和存储信息，从而捕获实体之间的复杂关系，而Base RAG 通常使用矢量化文本数据。
- 知识图谱 RAG 通过图遍历和子图搜索来检索信息，而Base RAG 依赖于向量相似性搜索。
- 知识图谱 RAG 可以更好地理解实体之间的关系和层次结构，从而提供更丰富的上下文，而Base RAG 在处理复杂关系方面受到限制




```
from llama_index.graph_stores.neo4j import Neo4jGraphStore
from llama_index.core import Settings
from llama_index.llms.dashscope import DashScope
from llama_index.embeddings.dashscope import DashScopeEmbedding # <--- Changed this line
import os

os.environ["DASHSCOPE_API_KEY"]="<your-api-key>"
os.environ["ALIYUN_BASE_URL"]="https://dashscope.aliyuncs.com/compatible-mode/v1"
username = "neo4j"
password = "12345678"
url = "bolt://localhost:7687"
database = "neo4j"
graph_store = Neo4jGraphStore(
    username=username,
    password=password,
```

```
    url=url,
    database=database,
)

#设置 llm
Settings.llm = DashScope(
    api_key=os.getenv("DASHSCOPE_API_KEY"),
    model="qwen-plus"
)
# 配置嵌入模型 ⭐ 新增设置
Settings.embed_model = DashScopeEmbedding(
    model_name="text-embedding-v3",  # 百炼嵌入模型
    api_key=os.getenv("DASHSCOPE_API_KEY"),
)

from llama_index.core import StorageContext, SimpleDirectoryReader, KnowledgeGraphIndex

# 使用 SimpleDirectoryReader 加载文档数据。
documents = SimpleDirectoryReader("./data").load_data()
# 使用 StorageContext 创建存储上下文对象，并传入图形存储对象/
storage_context = StorageContext.from_defaults(graph_store=graph_store)
# 使用 KnowledgeGraphIndex 从文档创建知识图谱索引对象。
index = KnowledgeGraphIndex.from_documents(
    documents,
    storage_context=storage_context,
    max_triplets_per_chunk=2,#指定每个文档块将被提取为最多两个三元组。
    include_embeddings=True,#表示提取的三元组将被转换为嵌入向量并保存。
)

query_engine = index.as_query_engine(
    include_text=True,
    response_mode="tree_summarize",
    embedding_mode="hybrid",
    similarity_top_k=5,
    verbose=True,
)
response = query_engine.query("白龍馬身世?")
print(f"Response: {response}")
```

5. 多模态RAG



方案 1:

使用多模态嵌入（如 CLIP）对图像和文本进行嵌入通过相似度搜索同时检索图像和文本将原始图像和文本块传递给多模态大语言模型进行答案合成

方案 2:

使用多模态大语言模型（如 GPT-4V、LLaVA 或 FUYU-8b）从**图像生成文本摘要**对文本摘要进行嵌入和检索将文本块传递给**大语言模**型进行答案合成

方案 3:

使用多模态大语言模型（如 GPT-4V、LLaVA 或 FUYU-8b）从图像生成文本摘要嵌入并检索图像摘要（附带原始图像引用）将**原始图像和文本块传递给多模态大语言模型**进行答案合成

6. 综合实战

6.1 数据集

数据集：使用数据为chinese-simplified-xlsum-v2新闻数据集，摘取`chinese_simplified_train.jsonl`前八条

6.2 数据加载与分块

> 数据清洗分块:由于数据集为jsonl格式，直接按照每个json单元分块

数据加载与分块 - 针对特定JSONL格式优化

```
class NewsJsonlDataLoader:
    def __init__(self, file_path):
        self.file_path = file_path

    def load_and_chunk(self):
        """加载JSONL文件并分块处理"""
        # 直接处理JSONL文件
```

```
        loader = JSONLoader(
            file_path=self.file_path,
            jq_schema=".",  # 整个对象作为文档
            content_key="text",  # 使用text字段作为内容
            metadata_func=self.extract_metadata,#指定如何提取每条记录的元数据
            json_lines=True  # 处理JSONL格式
        )

        # 每个新闻作为一个分块
        text_splitter = RecursiveCharacterTextSplitter(
            chunk_size=1000,
            chunk_overlap=100,
            separators=["\n\n", "\n", "。", "！", "？", ". "]
        )

        return loader.load_and_split(text_splitter=text_splitter)

    def extract_metadata(self, record: dict, metadata: dict) -> dict:
        """提取元数据"""
        return {
            "id": record.get("id", ""),
            "url": record.get("url", ""),
            "title": record.get("title", ""),
            "summary": record.get("summary", ""),
        }
```

RecursiveCharacterTextSplitter 原理：该工具以递归为核心机制。先按指定字符集顺序尝试分割文本，评估分割结果是否小于指定块大小，不满足则换用下一个字符，直到符合条件。

6.3 向量存储

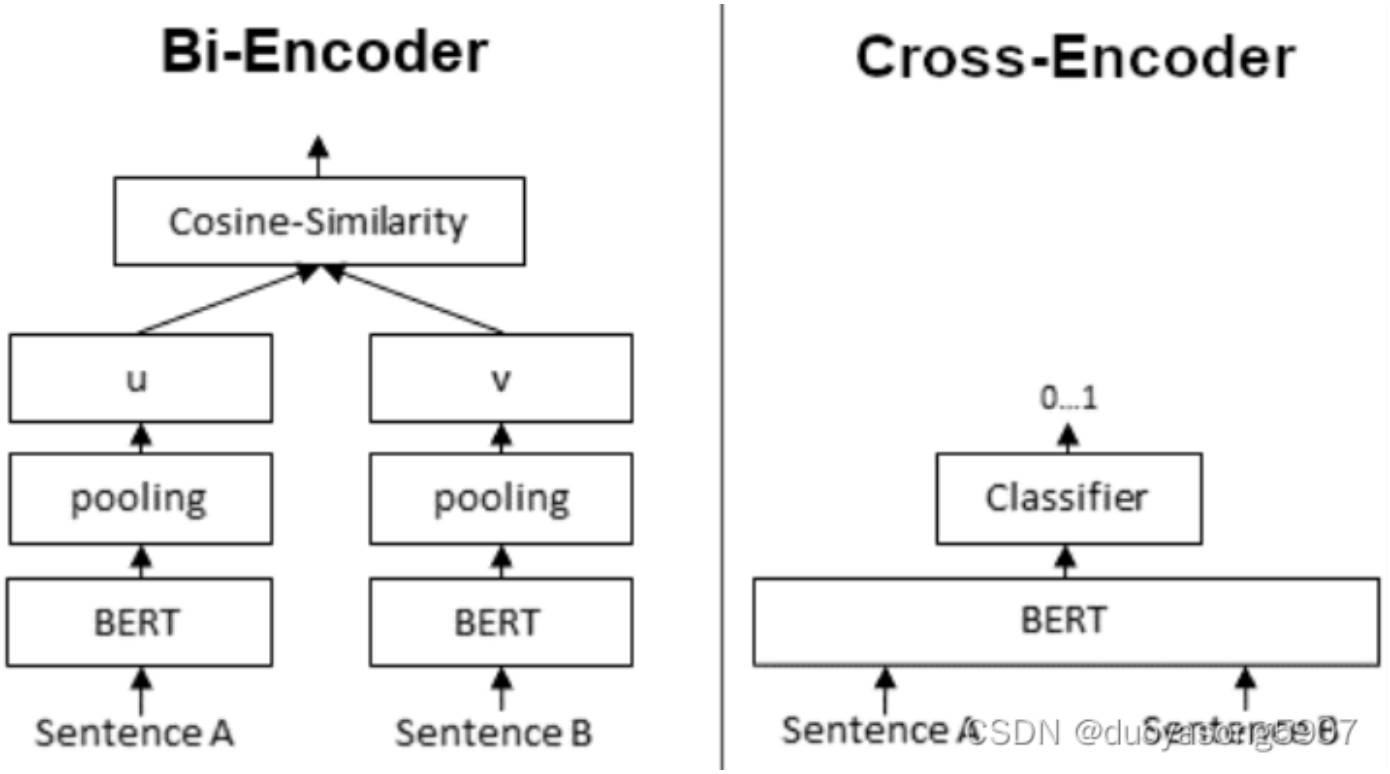
```
    # # 构建向量数据库（首次运行）
    # vector_store = Chroma.from_documents(
    #     documents=documents,
    #     embedding=embeddings,
    #     persist_directory="./chroma_db"
    # )

    # # 显式保存到磁盘（Chroma 会自动保存，但建议显式调用）
    # vector_store.persist()
    # print(f"向量索引已保存至目录: ./chroma_db")

    #  二次使用
    # 从持久化目录加载向量存储
    vector_store = Chroma(
        persist_directory="./chroma_db",  # 与保存时相同的目录
        embedding_function=embeddings
    )
```

6.4 检索器系统

```
class RetrievalSystem:
    def __init__(self, documents, vector_store):
        self.documents = documents
        self.vector_store = vector_store
#关键词检索
    def get_bm25_retriever(self, k=3):
        return BM25Retriever.from_documents(
            documents=self.documents,
            k=k
        )
#向量检索
    def get_vector_retriever(self, k=3):
        return self.vector_store.as_retriever(
            search_type="mmr",
            search_kwargs={"k": k}
        )
#混合检索
    def get_ensemble_retriever(self, k=3):
        bm25 = self.get_bm25_retriever(k)
        vector = self.get_vector_retriever(k)
        return EnsembleRetriever(
            retrievers=[bm25, vector],
            weights=[0.5, 0.5]
        )
```

6.5 重排

```
class Reranker:
#基于本地模型的重排
    @staticmethod
    def local_reranker(retriever, top_n=3):
        model = HuggingFaceCrossEncoder(model_name="BAAI/bge-reranker-base")
        compressor = CrossEncoderReranker(model=model, top_n=top_n)
        return ContextualCompressionRetriever(
            base_compressor=compressor,
            base_retriever=retriever
        )
#基于阿里云api的重排
    @staticmethod
    def cloud_reranker(retriever, top_n=3):
        # 实际使用需要替换为DashScope实现
        compressor = DashScopeRerank()
        compression_retriever = ContextualCompressionRetriever(
            base_compressor=compressor, base_retriever=retriever
        )
        return compression_retriever  # 降级处理，直接返回原始检索器
```



Bi-Encoder会用BERT对输入文本编码，再根据cosine相似度分数筛选文本。Cross-Encoder会直接计算两个句子的相关性分数。

- 有一组预先定义好的句子对，并想对其进行打分时，就可以使用cross-Encoder
- 需要在向量空间中获得句子嵌入以进行高效比较的情况，使用BiEncoder
- ContextualCompressionRetriever：将交叉编码器包装为 `Compressor` 对象，负责对检索结果重排并截断。
  - 接收基础检索器返回的文档列表。
  - 用交叉编码器计算每个文档相对于查询的相关性得分。
  - 按得分降序排序，保留前 `top_n` 个文档。
- ContextualCompressionRetriever：创建压缩检索器

6.6 质量评估

```
class AnswerGrader:
    def __init__(self, llm):
        self.llm = llm

    def hallucination_grader(self):
        class GradeHallucinations(BaseModel):
            binary_score: str = Field(description="答案是否虚构。('yes' or 'no')")

        instruction = """
        你是一个评分人员，负责确认LLM的回复是否为虚构的。
        以下会给你一个文件与相对应的LLM回复，请输出 'yes' or 'no'作为判断结果。
```

```
        'Yes' 代表LLM的回答是虚构的，未基于文件内容 'No' 则代表LLM的回答并未虚构，而是基于文件内容得出。
        """

        prompt = ChatPromptTemplate.from_messages([
            ("system", instruction),
            ("human", "文件: \n\n {documents} \n\n LLM 回复: {generation}")
        ])

        return prompt | self.llm.with_structured_output(GradeHallucinations)

    def answer_grader(self):
        class GradeAnswer(BaseModel):
            binary_score: str = Field(description="答案是否回应问题。('yes' or 'no')")

        instruction = """
        你是一个评分人员，负责确认答案是否回应了问题。
        输出 'yes' or 'no'。 'Yes' 代表答案确实回应了问题， 'No' 则代表答案并未回应问题。
        """

        prompt = ChatPromptTemplate.from_messages([
            ("system", instruction),
            ("human", "用户问题: \n\n {question} \n\n 答案: {generation}")
        ])

        return prompt | self.llm.with_structured_output(GradeAnswer)
```

- 幻觉检测
  - 生成文本+RAG 文档给llm 判断
  - `with_structured_output`:LangChain 的方法，用于指定 LLM 的输出格式。
  - `|`:在 LangChain 中，`|` 是用于将两个组件串联的语法，类似于管道操作符。它将 `hallucination_prompt` 的输出传递给 `structured_llm_grader`。
- answer_grader：确认答案是否回应问题

6.7 RAG问答系统

```
class RAGSystem:
    def __init__(self, retriever, client):
        self.retriever = retriever
        self.client = client

    def format_docs(self, docs: List[Document]) -> str:
        """格式化检索到的文档用于提示词"""
        return "\n\n".join([
            f"标题: {doc.metadata['title']}\n"
            f"日期: {doc.metadata.get('date', '未知')}\n"
            f"摘要: {doc.metadata.get('summary', '')}\n"
            f"内容: {doc.page_content[:500]}...\n"
```

```
            f"来源：{doc.metadata['url']}"
            for doc in docs
        ])

    def generate_answer(self, question: str) -> str:
        print(f"\n检索相关文档：{question}...")
        context_docs = self.retriever.get_relevant_documents(question)

        # 准备参考文档摘要
        reference_summary = "参考文档摘要：\n"
        for i, doc in enumerate(context_docs):
            title = doc.metadata.get('title', '无标题')
            url = doc.metadata.get('url', '未知链接')
            summary = doc.metadata.get('summary', '')
            reference_summary += f"{i + 1}. 来源：{url} | 标题：{title} | 摘要：{summary}\n"

        print(f"格式化 {len(context_docs)} 个相关文档...")
        context = self.format_docs(context_docs)

        template = """
        你是一个新闻分析助手，需要根据以下相关新闻片段回答问题。
        如果问题涉及多个新闻，请综合多篇新闻内容进行回答。
        回答时需要注明新闻来源，如可能请包含发布日期。

        相关新闻片段：
        {context}
        """
        prompt = PromptTemplate.from_template(template)
        system_prompt = prompt.format(context=context)

        print("生成回答...")
        completion = self.client.chat.completions.create(
            model="qwen-plus",
            messages=[
                {'role': 'system', 'content': system_prompt},  # 中文系统角色
                {'role': 'user', 'content': question},  # 使用生成的提示
            ]
        )
        answer = completion.choices[0].message.content
        combined_response = f"{answer}\n\n{reference_summary}"

        return combined_response
```

6.7 主程序

```
if __name__ == "__main__":
    # 配置
    JSONL_FILE = "../data/chinese-simplified-xlsum-
v2/chinese_simplified_XLSum_v2.0/test.jsonl"  # 替换为实际文件路径

    # 步骤1：加载数据并分块
```

```
print("正在加载新闻数据...")
loader = NewsJsonlDataLoader(JSONL_FILE)
documents = loader.load_and_chunk()
print(f"成功加载 {len(documents)} 条新闻")
# 步骤2: 初始化LLM
print("初始化语言模型...")

client = OpenAI(
    # 若没有配置环境变量，请用阿里云百炼API Key将下行替换为：api_key=
    api_key=os.getenv("DASHSCOPE_API_KEY"),
    base_url=os.getenv("ALIYUN_BASE_URL"),
)
llm = ChatOpenAI(
    openai_api_key=os.getenv("DASHSCOPE_API_KEY"),
    base_url=os.getenv("ALIYUN_BASE_URL"),
    model='qwen-plus',
)
# 步骤3: 创建向量存储
print("正在构建向量索引...")
# 创建 Chroma 向量存储并指定持久化目录


embeddings = AliyunEmbeddings(client=client, model="text-embedding-v3")

# # 构建向量数据库（首次运行）
# vector_store = Chroma.from_documents(
#     documents=documents,
#     embedding=embeddings,
#     persist_directory="./chroma_db"
# )

# # 显式保存到磁盘（Chroma 会自动保存，但建议显式调用）
# vector_store.persist()
# print(f"向量索引已保存至目录: ./chroma_db")

#  二次使用
# 从持久化目录加载向量存储
vector_store = Chroma(
    persist_directory="./chroma_db",  # 与保存时相同的目录
    embedding_function=embeddings
)

print("已成功加载本地向量存储")


# 步骤4: 创建检索系统
print("构建检索系统...")
retrieval_sys = RetrievalSystem(documents, vector_store)

# 选择检索方式: ensemble_retriever | self_query_retriever
retriever = retrieval_sys.get_vector_retriever(k=3)
print("自查询检索器准备就绪")
```

```python
# 步骤5：添加重排
print("添加结果重排...")
reranked_retriever = Reranker.local_reranker(retriever, top_n=2)

# 步骤6：创建RAG系统
print("初始化问答系统...")
rag_system = RAGSystem(reranked_retriever, client)

# 示例问题
questions = [
    "心理健康新闻",
    "足球新闻"
]

for question in questions:
    print("\n" + "=" * 50)
    print(f"问题：{question}")

    # 生成答案
    answer = rag_system.generate_answer(question)
    print("\n回答:")
    print(answer)

    # 步骤7：回答质量评估
    print("\n" + "-" * 50)
    print("\n质量评估:")
    grader = AnswerGrader(llm)

    # # 幻觉评估
    print("\n" + "-" * 50)
    hallucination_result = grader.hallucination_grader().invoke({
        "documents": "相关新闻摘要...",
        "generation": answer
    })
    print(f"幻觉评估：{hallucination_result.binary_score}")

    # 回答相关性评估
    print("\n" + "-" * 50)
    relevance_result = grader.answer_grader().invoke({
        "question": question,
        "generation": answer
    })
    print(f"答案相关性：{relevance_result.binary_score}")

print("\n问答系统完成")
```