RAG 优化策略与知识库模块功能解析

LangChain RAG 开发多阶段优化策略

- 学习掌握在 RAG 不同开发阶段优化的相关思路，涵盖了查询转换、路由、问题构造、索引、检索和生成 6 个阶段。

- 学习这 6 个不同阶段优化策略的案例实现与 Prompt 编写（近 20 种优化策略），掌握对复杂 RAG 应用优化策略的快速拆解与代码编写。

- 学习掌握 LangChain 中将一个复杂应用（组件≥ 30 ）从流程图到 LCEL 链应用的实现思路。

LLMOps 知识库模块功能解析

- 学习了解 LLMOps 知识库模块 UI 功能拆解，完成 API 文档撰写，数据库表 ER 图设计。

- 学习掌握多知识库、多用户、多应用关联的 AI 应用产品设计思路，掌握各个模块的功能关联。

- 学习掌握向量数据库元数据结构的设计与 weaviate 数据库的复杂条件筛选，并实现对数据的隔离与同步。

RAG 开发 6 个阶段优化策略分析

RAG 开发 6 个阶段优化策略分析

01. LLM 应用中的 RAG 开发阶段

在 RAG 应用开发中，无论架构多复杂，接入了多少组件，使用了多少优化策略与特性，所有优化的最终目标都是`提升LLM生成内容的准确性`，而对于`Transformer架构类型`的大模型来说，要实现这个目标，一般只需要 3 个步骤：

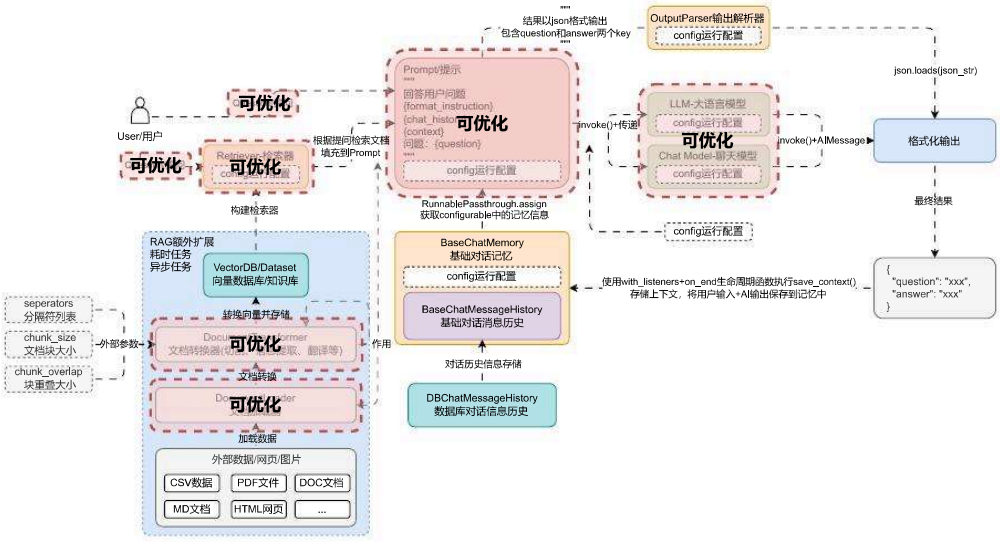
1. `传递更准确的内容`：传递和提问准确性更高的内容，会让 LLM 能识别到关联的内容， 生成的内容准确性更高。
2. `让重要的内容更靠前`：GPT 模型的注意力机制会让传递`Prompt`中更靠前的内种权重更高，越靠后权重越低。
3. `尽可能不传递不相关内容`：缩短每个块的大小，尽可能让每个块只包含关联的内容，缩小不相关内容的比例。

看起来很简单，但是目前针对这 3 个步骤 N 多研究员提出了不少方案，比较遗憾的是，目前也没有一种统一的方案，不同的场合仍然需要考虑不同的方案结合才能实现相对好一点的效果，并不是所有场合都适合配置很复杂的优化策略。

> **Important**
>
> 在 RAG 应用开发中，使用的优化策略越多，单次响应成本越高，性能越差，需要合理使用。

映射到`RAG`中，其实就是`切割合适的文档块`、`更准确的搜索语句`、`正确地排序文档`、`剔除重复无关的检索内容`，所以在`RAG应用开发`中，想进行优化，可以针对`query(提问查询)`、`TextSplitter(文本分割器)`、`VectorStore(向量数据库)`、`Retriever(检索器)`、`Prompt(基础prompt编写)`这几个组件。

在前面学习的 LLM 应用开发中，我们构建了一个应用开发流程图，涵盖了`向量数据库`、`检索器`、`Prompt`、`记忆`、`输出解析器`、`大语言模型`、`运行时配置`、`大模型生成`等阶段，可以优化 RAG 的组件使用红圈覆盖如下：



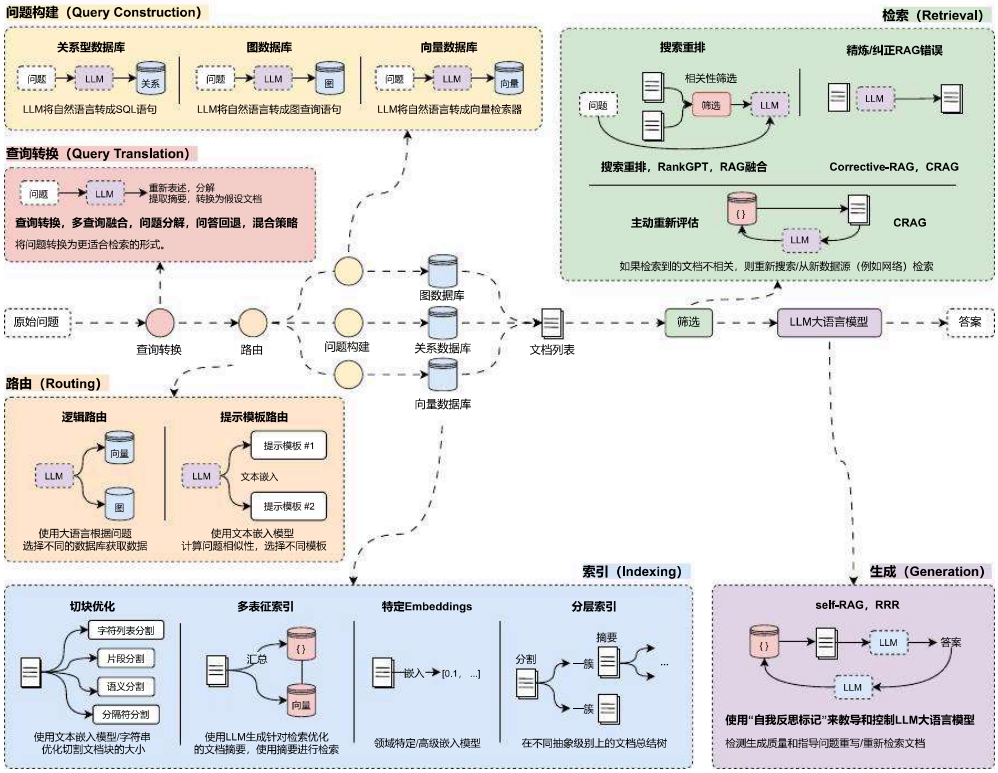
在完整的 LLM 应用流程中拆解 RAG 开发阶段并进行优化看起来相对繁琐，可以考虑单独将 RAG 开发阶段的流程拎出来，并针对性对每个阶段进行优化与调整，按照不同的功能模块，共可以划分成 6 个阶段：`查询转换`、`路由`、`查询构建`、`索引`、`检索`和`生成`。

02. RAG 开发 6 个阶段优化策略

在 RAG 开发的 6 个阶段中，不同的阶段拥有不同的优化策略，需要针对不同的应用进行特定性的优化，目前市面上常见的优化方案有：`问题转换`、`多路召回`、`混合检索`、`搜索重排`、`动态路由`、`图查询`、`问题重建`、`自检索` 等数十种优化策略，每种策略所在的阶段并不一致，效果也有差异，并且相互影响。

并且 RAG 优化和 LangChain 并没有关系，无论使用任何框架、任何编程语言，进行 RAG 开发时，掌握优化的思路才是最重要的！

将对应的优化策略整理到 RAG 运行流程中，优化策略与开发阶段对应图如下：



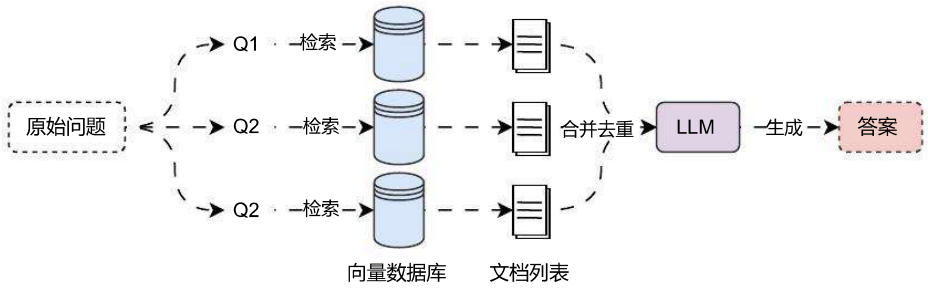
多查询重写策略提升检索准确性

多查询重写策略提升检索准确性

01. Multi-Query 多查询策略

`多查询策略` 也被称为 `子查询`，是一种用于生成子问题的技术，其核心思想是在问答过程中，为了更好地理解和回答主问题，系统会自动生成并提出与主问题相关的子问题，这些子问题通常具有更具体的细节，可以帮助大语言模型更深入地理解主问题，从而进行更加准确的检索并提供正确的答案。

`多查询策略` 会从多个角度重写用户问题，为每个重写的文件执行检索，并将检索到的文档列表进行合并后去重，返回唯一文档，该策略的运行流程非常简单，如下：



在 LangChain 中，针对 `多查询策略` 封装了一个检索器 `MultiQueryRetriever`，该检索器可以通过构造函数亦或者 `from_llm` 类方法进行实例化，参数如下：

1. `retriever`：基础检索器，必填参数。
2. `llm`：大语言模型，用于将原始问题转换成多个问题，必填参数。
3. `prompt`：转换原始问题为多个问题的提示模板，非必填，已有默认值。
4. `parser_key`：解析键，该参数在未来将被抛弃，非必填，已弃用，新版本中保留参数，但没有任何使用的地方。
5. `inclued_original`：是否保留原始问题，默认为 `False`，如果设置为 `True`，则除了检索新问题，还会检索原始问题。

例如以 `weaviate` 向量数据库作为检索器，使用 `多查询策略` 优化普通的 `RAG检索缓解`，对应的代码具象化如下：

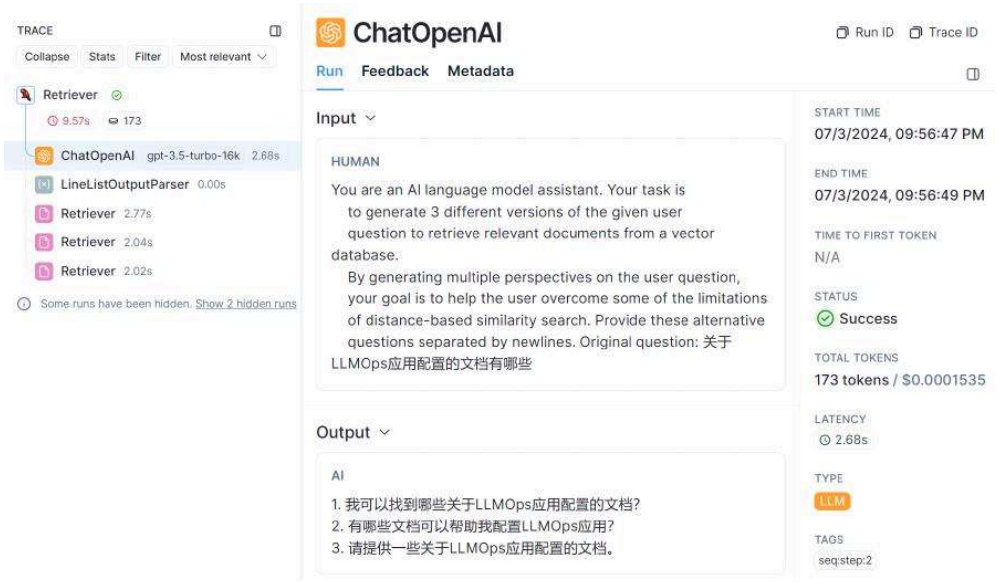
```
import dotenv
import weaviate
from langchain.retrievers import MultiQueryRetriever
from langchain_openai import OpenAIEmbeddings, ChatOpenAI
from langchain_weaviate import WeaviateVectorStore
from weaviate.auth import AuthApiKey

dotenv.load_dotenv()

# 1.构建向量数据库与检索器
db = WeaviateVectorStore(
    client=weaviate.connect_to_wcs(
        cluster_url="https://eftofnujtxqcsaOsn272jw.cO.us-
west3.gcp.weaviate.cloud",
        auth_credentials=AuthApiKey("21pzYyOorl2dxH9xCoZG1O2bOeuDeKJNEbBO"),
    ),
    index_name="DatasetDemo",
```

```
    text_key="text",
    embedding=OpenAIEmbeddings(model="text-embedding-3-small"),
)
retriever = db.as_retriever(search_type="mmr")

# 2.创建多查询检索器
multi_query_retriever = MultiQueryRetriever.from_llm(
    retriever=retriever,
    llm=ChatOpenAI(model="gpt-3.5-turbo-16k", temperature=0),
)

# 3.执行检索
docs = multi_query_retriever.invoke("关于LLMOps应用配置的文档有哪些")
print(docs)
print(len(docs))
```

输出内容:

```
[Document(metadata={'source': './项目API文档.md', 'start_index': 0.0},
page_content='LLMOps 项目 API 文档\n\n应用 API 接口统一以 JSON 格式返回，并且包含 3 个字段：code、data 和 message，分别代表业务状态码、业务数据和接口附加信息。\n\n业务状态码共有 6 种，其中只有 success(成功) 代表业务操作成功，其他 5 种状态均代表失败，并且失败时会附加相关的信息：fail(通用失败)、not_found(未找到)、unauthorized(未授权)、forbidden(无权限)和 validate_error(数据验证失败)。\n\n接口示例：\n\njson { "code": "success", "data": { "redirect_url": "https://github.com/login/oauth/authorize?client_id=f69102c6b97d90d69768&redirect_uri=http%3A%2F%2Flocalhost%3A5001%2Foauth%2Fauthorize%2Fgithub&scope=user%3Aemail" }, "message": "" }'),
Document(metadata={'source': './项目API文档.md', 'start_index': 3042.0},
page_content='1.2 [todo]更新应用草稿配置信息\n\n接口说明：更新应用的草稿配置信息，涵盖：模型配置、长记忆模式等，该接口会查找该应用原始的草稿配置并进行更新，如果没有原始草稿配置，则创建一个新配置作为草稿配置。\n\n接口信息：授权+POST:/apps/:app_id/config\n\n接口参数：\n\n请求参数：\n\napp_id -> str：需要修改配置的应用 id。\n\nmodel_config -> json：模型配置信息。\n\ndialog_round -> int：携带上下文轮数，类型为非负整型。\n\nmemory_mode -> string：记忆类型，涵盖长记忆 long_term_memory 和 none 代表无。\n\n请求示例：\n\njson { "model_config": { "dialog_round": 10 }, "memory_mode": "long_term_memory" }\n\n响应示例：\n\njson { "code": "success", "data": {}, "message": "更新AI应用配置成功" }\n\n1.3 [todo]获取应用调试长记忆'), Document(metadata={'source': './项目API文档.md', 'start_index': 5818.0}, page_content='json { "code": "success", "data": { "list": [ { "id": "1550b71a-1444-47ed-a59d-c2f080fbae94", "conversation_id": "2d7d3e3f-95c9-4d9d-ba9c-9daaf09cc8a8", "query": "能详细讲解下LLM是什么吗？", "answer": "LLM 即 Large Language Model，大语言模型，是一种基于深度学习的自然语言处理模型，具有很高的语言理解和生成能力，能够处理各式各样的自然语言任务，例如文本生成、问答、翻译、摘要等。它通过在大量的文本数据上进行训练，学习到语言的模式、结构和语义知识'), Document(metadata={'source': './项目API文档.md', 'start_index': 675.0}, page_content='json { "code": "success", "data": { "list": [ { "app_count": 0, "created_at": 1713105994, "description": "这是专门用来存储慕课LLMOps课程信息的知识库", "document_count": 13, "icon": "https://imooc-llmops-1257184990.cos.ap-guangzhou.myqcloud.com/2024/04/07/96b5e270-c54a-4424-aece-ff8a2b7e4331.png", "id": "c0759ca8-2d35-4480-83a8-1f41f29d1401", "name": "慕课LLMOps课程知识库", "updated_at": 1713106758, "word_count": 8850 } ], "paginator": { "current_page": 1, "page_size": 20, "total_page": 1, "total_record": 2 } }'),
Document(metadata={'source': './项目API文档.md', 'start_index': 2324.0},
page_content='json { "code": "success", "data": { "id": "5e7834dc-bbca-4ee5-9591-8f297f5acded", "name": "慕课LLMOps聊天机器人", "icon": "https://imooc-llmops-1257184990.cos.ap-guangzhou.myqcloud.com/2024/04/23/e4422149-4cf7-41b3-ad55-ca8d2caa8f13.png", "description": "这是一个慕课LLMOps的Agent应用", "published_app_config_id": null, "drafted_app_config_id": null, "debug_conversation_id": "1550b71a-1444-47ed-a59d-c2f080fbae94", "published_app_config": null, "drafted_app_config": { "id": "755dc464-67cd-42ef-9c56-b7528b44e7c8"'), Document(metadata={'source': './项目API文档.md', 'start_index': 2042.0}, page_content='dialog_round -> int：携带上下文轮数，类型为非负整型。\n\nmemory_mode -> string：记忆类型，涵盖长记忆 long_term_memory 和 none 代表无。\n\nstatus -> string：应用配置的状态，drafted 代表草稿、published 代表已发布配置。\n\nupdated_at -> int：应用配置的更新时间。\n\ncreated_at -> int：应用配置的创建时间。\n\nupdated_at -> int：应用的更新时间。\n\ncreated_at -> int：应用的创建时间。\n\n响应示例：')]
6
```


亦或者在 LangSmith 平台上也可以观测到整个执行的流程，如下：



02. 核心及注意事项

从 LangSmith 平台记录的运行流程，可以很清晰看到这个检索器会先调用大语言模型生成 3 条与原始问题相关的 `子问题`，然后再逐个使用传递的检索器检索 3 个子问题，得到对应的文档列表，最后再将所有文档列表进行合并去重，得到最终的文档。

在 `MultiQueryRetriever` 这个检索器中，预设了一段 `prompt`，用于将原始问题生成 3 个关联子问题，并使用 `\n` 分割得到具体问题。

这段 `prompt` 如下：

```
# langchain/retrievers/multi_query.py
DEFAULT_QUERY_PROMPT = PromptTemplate(
    input_variables=["question"],
    template="""You are an AI language model assistant. Your task is to generate
3 different versions of the given user question to retrieve relevant documents
from a vector  database. By generating multiple perspectives on the user
question, your goal is to help the user overcome some of the limitations of
distance-based similarity search. Provide these alternative questions separated
by newlines. Original question: {question}""",
)
```

在 LangChain 中，所有预设的 prompt 绝大部分场景都是使用 OpenAI 的大语言模型进行调试的，所以效果会比较好，对于其他的模型，例如国内的模型，一般来说还需要将对应的提示换成 `中文语言`，所以可以考虑使用 ChatGPT 翻译原有的 `prompt`，更新后：

```
multi_query_retriever = MultiQueryRetriever.from_llm(
    retriever=retriever,
    llm=ChatOpenAI(model="gpt-3.5-turbo-16k", temperature=0),
    prompt=ChatPromptTemplate.from_template(
        "你是一个AI语言模型助手。你的任务是生成给定用户问题的3个不同版本，以从向量数据库中检索相关文档。"
        "通过提供用户问题的多个视角，你的目标是帮助用户克服基于距离的相似性搜索的一些限制。"
        "请用换行符分隔这些替代问题。"
        "原始问题：{question}"
    )
)
```

基于中文 `prompt` 生成的问题列表如下：

```
1. LLMOps应用配置的文档有哪些资源可供参考？
2. 我可以在哪里找到关于LLMOps应用配置的文档？
3. 有哪些文档可以帮助我了解LLMOps应用配置的相关信息？
```

对于该检索器，不同的模型生成的 `query` 格式可能并不一样，某些模型生成的多条 `query` 可能并不是按照 `\n` 进行分割，这个时候查询的效果可能不如原始问题，所以在使用该检索器时，一定要多次测试 `prompt` 的效果，或者设置 `inclued_original` 为 `True`，确保生成内容不符合规范时，仍然可以使用原始问题进行检索。

另外，在 `MultiQueryRetriever` 的底层进行合并去重是，并没有任何特别的，仅仅只做了循环遍历并记录唯一的文档而已，核心代码：

```
# langchain/retrievers/multi_query.py
def _unique_documents(documents: Sequence[Document]) -> List[Document]:
    return [doc for i, doc in enumerate(documents) if doc not in documents[:i]]
```

多查询策略是最基础+最简单的 RAG 优化，不涉及到复杂的逻辑与算法，会稍微影响单次对话的耗时。

并且由于需要转换 `query` 一般较小，以及生成 `sub-queries` 时对 LLM 的能力要求并不高，在实际的 LLM 应用开发中，通常使用参数较小的本地模型+针对性优化的 prompt 即可完成任务。

而且为了减少模型的幻觉以及胡说八道，一般都将 `temperature` 设置为 `0`，确保生成的文本更加有确定性。

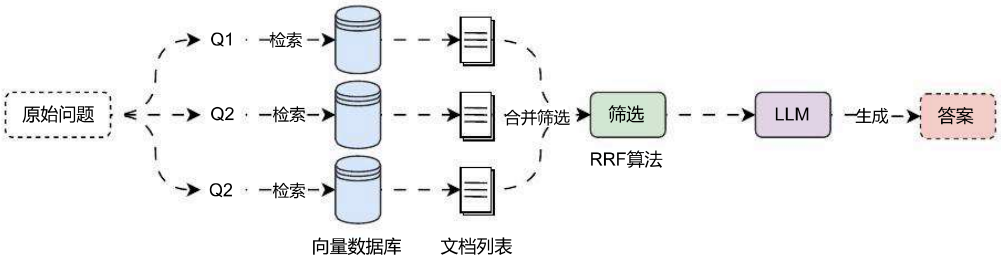
RAG 多查询结果融合策略

RAG 多查询结果融合策略

01. 多查询结果融合策略及 RRF

在 `多查询重写策略` 中，虽然可以生成多条查询并执行多次检索器检索，但是在合并数据的时候，并没有考虑最终结果的文档数，极端情况下，原始的 `k` 设置为 4，可能会返回 16 个文档（3 条子查询的文档，1 条原始问题查询的文档），除此之外，`多查询重写策略` 并不会考虑对应文档的权重，只按默认顺序进行合并。

于是就诞生了 `RAG融合` 的概念，它的主要思想是在 Multi-Query 的基础上，对其检索结果进行重新排序（即 reranking）后输出 Top K 个结果，最后再将这 Top K 个结果喂给 LLM 并生成最终答案，运行流程如下：



在 `RAG融合` 中，对文档列表进行排序&去重合并的算法为 `RRF(Reciprocal Rank Fusion)`，即倒排序排名算法，该算法是滑铁卢大学（CAN）和 Google 合作开发的，而且该算法的原理其实非常简单，公式如下：

$$RRF_{score}(d \in D) = \sum_{r \in R} \frac{1}{k + r(d)}$$

论文原文链接：https://plg.uwaterloo.ca/~gvcormac/cormacksigir09-rrf.pdf

在 RRF 算法中，D 表示相关文档的全集，k 是固定常数 60，r(d) 表示当前文档 d 在其子集中的位置，该算法会对全集 D 进行二重遍历，外层遍历文档全集 D，内层遍历文档子集，在做内层变量的时候，我们会累计当前文档在其所在子集中的位置并取倒数作为其权重。

常数 k 被设定为 60，这个值是在进行初步调查时确定的，在论文中，通过四个试点实验，每个实验结合了 30 种搜索配置应用于不同的 TREC 集合的结果，发现 k=60 接近最优值，k 值是多少并不是关键，主要是通过 k 值，可以很容易发现一个事实：

> **Important**
>
> 虽然高排名的文档更加重要，但低排名文档的重要性并不会像使用指数函数那样消失。

RRF 算法的 Python 实现具象化如下：

```python
def rrf(results: list[list], k: int = 60) -> list[tuple]:
    """倒数排名融合RRF算法，用于将多个结果生成单一、统一的排名"""

    # 1.初始化一个字典，用于存储每一个唯一文档的得分
    fused_scores = {}

    # 2.遍历每个查询对应的文档列表
    for docs in results:
        # 3.内层遍历文档列表得到每一个文档
```

```
        for rank, doc in enumerate(docs):
            # 4.将文档使用langchain提供的dump工具转换成字符串
            doc_str = dumps(doc)
            # 5.检测该字符串是否存在得分，如果不存在则赋值为0
            if doc_str not in fused_scores:
                fused_scores[doc_str] = 0
            # 6.计算多结果得分，排名越小越靠前，k为控制权重的参数
            fused_scores[doc_str] += 1 / (rank + k)

    # 7.提取得分并进行排序
    reranked_results = [
        (loads(doc), score)
        for doc, score in sorted(fused_scores.items(), key=lambda x: x[1],
reverse=True)
    ]

    return reranked_results
```

02. 多查询结果融合策略实现

在 LangChain 中并没有直接实现 `RAG多查询结果融合策略` 的检索器，所以可以考虑自定义实现，或者是继承 `MultiQueryRetriever` 并重写 `retrieve_docments()` 与 `unique_union()` 方法来实现对文档的 RRF 排名计算与合并。

重写方法的思路其实也非常简单，在方法内部将每次检索到的内容填充到一个两层列表中，然后传递给 RRF 函数即可。

完整代码实现如下：

```
from typing import List

import dotenv
import weaviate
from langchain.load import dumps, loads
from langchain.retrievers import MultiQueryRetriever
from langchain_core.callbacks import CallbackManagerForRetrieverRun
from langchain_core.documents import Document
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
from langchain_weaviate import WeaviateVectorStore
from weaviate.auth import AuthApiKey

dotenv.load_dotenv()


class RAGFusionRetriever(MultiQueryRetriever):
    """RAG多查询结果融合检索器"""
    k: int = 4

    def __init__(self, k: int = 4, **kwargs):
        super().__init__(**kwargs)
        self.k = k

    def retrieve_documents(
            self, queries: List[str], run_manager: CallbackManagerForRetrieverRun
    ) -> List[List]:
```

```
        """重写检索文档，返回二层嵌套的列表"""
        documents = []
        for query in queries:
            docs = self.retriever.invoke(
                query, config={"callbacks": run_manager.get_child()}
            )
            documents.append(docs)
        return documents

    def unique_union(self, documents: List[List]) -> List[Document]:
        """使用RRF算法对文档列表进行排序&合并"""
        # 1.初始化一个字典，用于存储每一个唯一文档的得分
        fused_scores = {}

        # 2.遍历每个查询对应的文档列表
        for docs in documents:
            # 3.内层遍历文档列表得到每一个文档
            for rank, doc in enumerate(docs):
                # 4.将文档使用langchain提供的dump工具转换成字符串
                doc_str = dumps(doc)
                # 5.检测该字符串是否存在得分，如果不存在则赋值为0
                if doc_str not in fused_scores:
                    fused_scores[doc_str] = 0
                # 6.计算多结果得分，排名越小越靠前，k为控制权重的参数
                fused_scores[doc_str] += 1 / (rank + 60)

        # 7.提取得分并进行排序
        reranked_results = [
            (loads(doc), score)
            for doc, score in sorted(fused_scores.items(), key=lambda x: x[1],
reverse=True)
        ]

        return [item[0] for item in reranked_results[:self.k]]


# 1.构建向量数据库与检索器
db = WeaviateVectorStore(
    client=weaviate.connect_to_wcs(
        cluster_url="https://eftofnujtxqcsaOsn272jw.cO.us-
west3.gcp.weaviate.cloud",
        auth_credentials=AuthApiKey("21pzYyOorl2dxH9xCoZG1O2bOeuDeKJNEbBO"),
    ),
    index_name="DatasetDemo",
    text_key="text",
    embedding=OpenAIEmbeddings(model="text-embedding-3-small"),
)
retriever = db.as_retriever(search_type="mmr")

rag_fusion_retriever = RAGFusionRetriever.from_llm(
    retriever=retriever,
    llm=ChatOpenAI(model="gpt-3.5-turbo-16k", temperature=0),
)

# 3.执行检索
docs = rag_fusion_retriever.invoke("关于LLMOps应用配置的文档有哪些")
```

```
print(docs)
print(len(docs))
```

输出内容:

```
[Document(metadata={'source': './项目API文档.md', 'start_index': 0.0},
page_content='LLMOps 项目 API 文档\n\n应用 API 接口统一以 JSON 格式返回，并且包含 3 个字段：code、data 和 message，分别代表业务状态码、业务数据和接口附加信息。\n\n业务状态码共有 6 种，其中只有 success(成功) 代表业务操作成功，其他 5 种状态均代表失败，并且失败时会附加相关的信息：fail(通用失败)、not_found(未找到)、unauthorized(未授权)、forbidden(无权限)和 validate_error(数据验证失败)。\n\n接口示例：\n\njson { "code": "success", "data": { "redirect_url": "https://github.com/login/oauth/authorize?client_id=f69102c6b97d90d69768&redirect_uri=http%3A%2F%2Flocalhost%3A5001%2Foauth%2Fauthorize%2Fgithub&scope=user%3Aemail" }, "message": "" }'),
Document(metadata={'source': './项目API文档.md', 'start_index': 5818.0},
page_content='json { "code": "success", "data": { "list": [ { "id": "1550b71a-1444-47ed-a59d-c2f080fbae94", "conversation_id": "2d7d3e3f-95c9-4d9d-ba9c-9daaf09cc8a8", "query": "能详细讲解下LLM是什么吗？", "answer": "LLM 即 Large Language Model，大语言模型，是一种基于深度学习的自然语言处理模型，具有很高的语言理解和生成能力，能够处理各式各样的自然语言任务，例如文本生成、问答、翻译、摘要等。它通过在大量的文本数据上进行训练，学习到语言的模式、结构和语义知识'), Document(metadata={'source': './项目API文档.md', 'start_index': 3042.0}, page_content='1.2 [todo]更新应用草稿配置信息\n\n接口说明：更新应用的草稿配置信息，涵盖：模型配置、长记忆模式等，该接口会查找该应用原始的草稿配置并进行更新，如果没有原始草稿配置，则创建一个新配置作为草稿配置。\n\n接口信息：授权 +POST:/apps/:app_id/config\n\n接口参数：\n\n请求参数：\n\napp_id -> str：需要修改配置的应用 id。\n\nmodel_config -> json：模型配置信息。\n\ndialog_round -> int：携带上下文轮数，类型为非负整型。\n\nmemory_mode -> string：记忆类型，涵盖长记忆 long_term_memory 和 none 代表无。\n\n请求示例：\n\njson { "model_config": { "dialog_round": 10 }, "memory_mode": "long_term_memory" }\n\n响应示例：\n\njson { "code": "success", "data": {}, "message": "更新AI应用配置成功" }\n\n1.3 [todo]获取应用调试长记忆'),
Document(metadata={'source': './项目API文档.md', 'start_index': 675.0},
page_content='json { "code": "success", "data": { "list": [ { "app_count": 0, "created_at": 1713105994, "description": "这是专门用来存储慕课LLMOps课程信息的知识库", "document_count": 13, "icon": "https://imooc-llmops-1257184990.cos.ap-guangzhou.myqcloud.com/2024/04/07/96b5e270-c54a-4424-aece-ff8a2b7e4331.png", "id": "c0759ca8-2d35-4480-83a8-1f41f29d1401", "name": "慕课LLMOps课程知识库", "updated_at": 1713106758, "word_count": 8850 } ], "paginator": { "current_page": 1, "page_size": 20, "total_page": 1, "total_record": 2 } }')]
4
```

问题分解策略提升复杂问题检索正确率

问题分解策略提升复杂问题检索正确率

01. 复杂问题检索的难点与分解

在 RAG 应用开发中，对于一些提问相对复杂的原始问题来说，无论是使用原始问题进行检索，亦或者生成多个相关联的问题进行检索，往往都很难在向量数据库中找到关联性高的文档，导致 RAG 效果偏差。

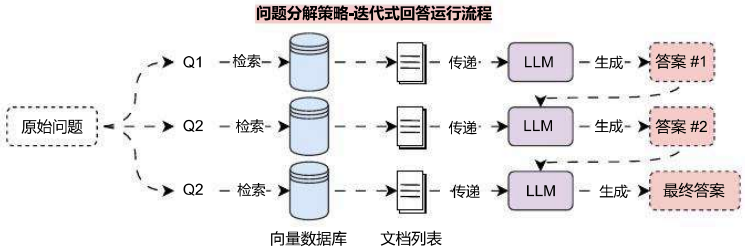
例如向量数据库中存储了一份 `机器的说明文档`，对于这类数据，如果提问 `如何完成某个部件的维修` 这类问题，一般都会涉及到多个步骤与顺序，执行相似性搜索会有很大概率没法找到有关联的文档。

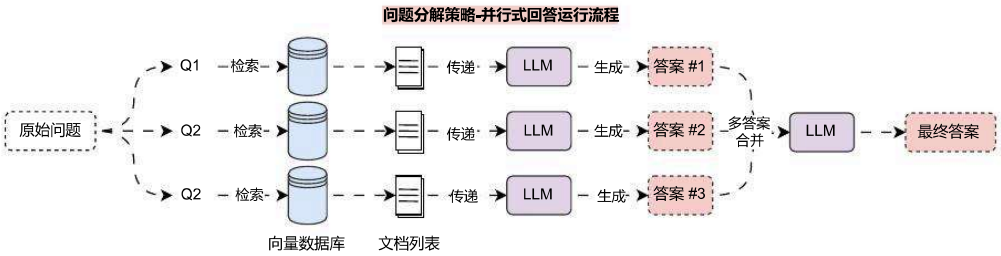
造成这个问题的原因有几种：

1. 复杂问题由多个问题按顺序步骤组成，执行相似性搜索时，向量数据库存储的都是基础文档数据，往往相似度低，但是这些数据在现实世界又可能存在很大的关联（文本嵌入模型的限制，一条向量不可能无损记录段落信息）。
2. 问题复杂度高或者涉及到数学问题，导致 LLM 没法一次性完成答案的生成，一次性传递大量的相关性文档，极大压缩了大语言模型生成内容上下文长度的限制。

对于这类 RAG 应用场景，可以使用 `Decomposition问题分解策略`，将一个复杂问题分解成多个子问题，和 `多查询重写策略` 不一样的是，这个策略生成的子问题使用的是 `深度优先`，即解决完第一个问题后，对应的资料传递给第二个问题，以此类推；亦或者是并行将每个问题的答案合并成最终问题。

所以 `Decomposition问题分解策略` 可以划分成两种方案：`迭代式回答` 与 `并行式回答`，两种方案的运行流程如下：





其中迭代式回答，会将上一次的 `提问+答案`，还有这一次的 `检索上下文` 一起传递给 LLM，让其生成答案，迭代到最后一次，就是最终答案。而 `并行式回答` 则会同时检索，并同时调用 LLM 生成答案，最后在将答案进行汇总，让 LLM 整理生成最终答案。

02. 问题分解策略-迭代式回答实现

在 LangChain 中，并没有针对 `问题分解策略` 实现对应的 `检索器` 或者 `预设链`，所以只能自行实现这个优化策略，由于问题分解策略同样也是先生成对应的子问题（深入优先），所以需要单独构建一条链先进行问题的分解，然后迭代执行响应的检索，得到上下文，并使用 LLM 回复该问题，将得到的 `迭代答案+问题`，传递给下一个子问题。

代码实现：

```
from operator import itemgetter

import dotenv
import weaviate
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnablePassthrough
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
from langchain_weaviate import WeaviateVectorStore
from weaviate.auth import AuthApiKey

dotenv.load_dotenv()


def format_qa_pair(question: str, answer: str) -> str:
    """格式化传入的问题+答案"""
    return f"Question: f{question}\nAnswer: {answer}\n\n".strip()


# 1.定义分解子问题的prompt
decomposition_prompt = ChatPromptTemplate.from_template(
    "你是一个乐于助人的AI助理，可以针对一个输入问题生成多个相关的子问题。\n"
    "目标是将输入问题分解成一组可以独立回答的子问题或子任务。\n"
    "生成与以下问题相关的多个搜索查询：{question}\n"
    "并使用换行符进行分割，输出（3个子问题/子查询）："
)

# 2.构建分解问题链
decomposition_chain = (
        {"question": RunnablePassthrough()}
        | decomposition_prompt
        | ChatOpenAI(model="gpt-3.5-turbo-16k", temperature=0)
        | StrOutputParser()
        | (lambda x: x.strip().split("\n"))
)

# 3.构建向量数据库与检索器
db = WeaviateVectorStore(
    client=weaviate.connect_to_wcs(
        cluster_url="https://mbakeruerziae6psyex7ng.c0.us-
west3.gcp.weaviate.cloud",
        auth_credentials=AuthApiKey("ZltPVa9ZSOxUcfafelsggGyyH6tnTYQYJvBx"),
    ),
    index_name="DatasetDemo",
    text_key="text",
    embedding=OpenAIEmbeddings(model="text-embedding-3-small"),
)
retriever = db.as_retriever(search_type="mmr")

# 4.执行提问获取子问题
question = "关于LLMOps应用配置的文档有哪些"
sub_questions = decomposition_chain.invoke(question)
```

```
# 5.构建迭代问答链
prompt = ChatPromptTemplate.from_template("""这是你需要回答的问题：
---
{question}
---

这是所有可用的背景问题和答案对：
---
{qa_pairs}
---

这是与问题相关的额外背景信息：
---
{context}
---

使用上述背景信息和所有可用的背景问题和答案对来回答这个问题：

{question}""")
chain = (
        {
            "context": itemgetter("question") | retriever,
            "question": itemgetter("question"),
            "qa_pairs": itemgetter("qa_pairs"),
        }
        | prompt
        | ChatOpenAI(model="gpt-3.5-turbo-16k", temperature=0)
        | StrOutputParser()
)

# 5.循环遍历所有子问题进行检索并获取答案
qa_pairs = ""
for sub_question in sub_questions:
    answer = chain.invoke({"question": sub_question, "qa_pairs": ""})
    qa_pairs += "\n---\n" + format_qa_pair(sub_question, answer)
    print(f"问题: {sub_question}")
    print(f"答案: {answer}")
    print("====================")
```

输出内容：

```
问题: 1. 如何配置LLMOps应用？
答案: 根据提供的背景信息，LLMOps应用的配置可以通过以下步骤完成：

1. 获取应用的长记忆内容：使用授权+GET:/apps/:app_id/long-term-memory接口，其中app_id参数为需要获取长记忆的应用id。该接口将返回该应用最新调试会话的长记忆内容。

2. 获取应用的详细信息：使用授权+GET:/apps/:app_id接口，其中app_id参数为需要获取详细信息的应用id。该接口将返回该应用的id、名称、图标、描述等信息。

3. 根据获取到的应用信息和长记忆内容进行配置：根据获取到的应用信息和长记忆内容，可以进行相应的配置操作，例如设置应用的名称、图标、描述等。

需要注意的是，具体的配置操作可能会根据LLMOps应用的具体需求而有所不同，以上步骤仅提供了一般的配置流程。
```

```
====================
问题: 2. LLMOps应用配置的最佳实践是什么?
答案: LLMOps应用配置的最佳实践是根据具体需求和业务场景进行配置，并遵循以下几个原则:

1. 状态管理: LLMOps应用配置有两种状态，即草稿（drafted）和已发布（published）。在进行配置时，应确保配置的状态正确，并根据需要进行相应的状态转换。

2. 更新和创建时间: 配置应包含更新时间（updated_at）和创建时间（created_at），以便跟踪配置的变更历史和创建时间。

3. 记忆类型: 配置中的记忆类型（memory_mode）可以选择长期记忆（long_term_memory）或无记忆（none）。根据应用的需求，选择适当的记忆类型。

4. 应用更新时间: 配置中还应包含应用的更新时间（updated_at）和创建时间（created_at），以便跟踪应用的变更历史和创建时间。

综上所述，LLMOps应用配置的最佳实践是根据具体需求和业务场景进行配置，并确保状态管理、时间跟踪和记忆类型的正确设置。
====================
问题: 3. 如何在LLMOps应用中查找相关的配置文档?
答案: 在LLMOps应用中查找相关的配置文档，可以参考项目的API文档。根据提供的背景信息，可以看到API文档中包含了关于应用配置的接口说明和示例。

具体来说，可以查找到更新应用草稿配置信息的接口说明，该接口的路径为`POST:/apps/:app_id/config`，需要传递`app_id`参数和`model_config`、`dialog_round`、`memory_mode`等配置信息作为请求参数。接口的响应示例中包含了`code`、`data`和`message`字段，其中`data`字段为空，`message`字段为"更新AI应用配置成功"。

通过阅读API文档中的其他接口说明和示例，可以进一步了解LLMOps应用的配置相关信息。
====================
```

> **Note**
>
> 课后练习：请修改 `迭代式回答示例`，将其修改成 `并行式回答` 的优化策略。
>
> 知识点：使用 `Runnable.map()` 的形式将 `并行式回答` 封装成一条链，而非使用循环方式。
>
> 并尝试下能否将 `迭代式回答` 的优化策略也单独封装成链或者组件应用。

Step-Back 回答回退策略实现前置检索

Step-Back 回答回退策略扩大检索范围

01. LangChain 少量示例提示模板

在与 LLM 的对话中，提供少量这样的示例被称为 `少量示例`，这是一种简单但强大的指导生成的方式，在某些情况下可以显著提高模型性能（与之对应的是零样本），少量示例可以降低 Prompt 的复杂度，快速告知 LLM 生成内容的规范。

在 LangChain 中，针对少量示例也封装对应的提示模板——`FewShotPromptTemplate`，这个提示模板只需要传递 `示例列表` 与 `示例模板` 即可快速构建 `少量示例提示模板`，使用示例如下：

```
import dotenv
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate,
FewShotChatMessagePromptTemplate
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()

# 1.构建示例模板与示例
example_prompt = ChatPromptTemplate.from_messages([
    ("human", "{question}"),
    ("ai", "{answer}"),
])
examples = [
    {"question": "帮我计算下2+2等于多少？", "answer": "4"},
    {"question": "帮我计算下2+3等于多少？", "answer": "5"},
    {"question": "帮我计算下20*15等于多少？", "answer": "300"},
]

# 2.构建少量示例提示模板
few_shot_prompt = FewShotChatMessagePromptTemplate(
    example_prompt=example_prompt,
    examples=examples,
)
print("少量示例模板:", few_shot_prompt.format())

# 3.构建最终提示模板
prompt = ChatPromptTemplate.from_messages([
    ("system", "你是一个可以计算复杂数学问题的聊天机器人"),
    few_shot_prompt,
    ("human", "{question}"),
])

# 4.创建大语言模型与链
llm = ChatOpenAI(model="gpt-3.5-turbo-16k", temperature=0)
chain = prompt | llm | StrOutputParser()

# 5.调用链获取结果
print(chain.invoke("帮我计算下14*15等于多少"))
```

输出内容：

```
少量示例模板: Human: 帮我计算下2+2等于多少?
AI: 4
Human: 帮我计算下2+3等于多少?
AI: 5
Human: 帮我计算下20*15等于多少?
AI: 300
210
```

`少量示例提示模板` 在底层会根据传递的 `示例模板` 与 `示例` 格式化对应的 `消息列表` 或者 `字符串`，从而将对应的示例参考字符串信息添加到完整的提示模板中，简化了 Prompt 编写的繁琐程度。

对于聊天模型可以使用 `FewShotChatMessagePromptTemplate`，而文本补全基座模型可以使用 `FewShotPromptTemplate`。

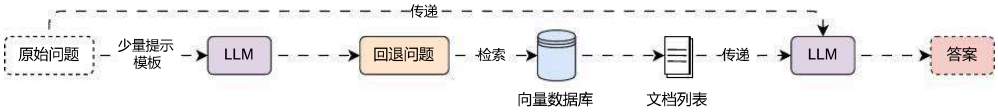
02. Step-Back 回答回退策略的优点

对于一些复杂的问题，除了使用 `问题分解` 来得到子问题亦或者依赖问题，还可以为复杂问题生成一个前置问题，通过前置问题来执行相应的检索，这就是 `Setp-Back 回答回退策略（后退提示）`，这是是一种用于增强语言模型的推理和问题解决能力的技巧，它鼓励 LLM 从一个给定的问题或问题后退一步，提出一个更抽象、更高级的问题，涵盖原始查询的本质。

> 概念来源：https://arxiv.org/pdf/2310.06117.pdf

后退提示背后的概念是，许多复杂的问题或任务包含很多复杂的细节和约束，这使 LLM 难以直接检索和应用相关信息。通过引入一个后退问题，这个问题通常更容易回答，并且围绕一个更广泛的概念或原则，让 LLM 可以更有效地构建它们的推理。

Step-Back 回答回退策略的运行流程也非常简单，构建一个 `少量示例提示模板`，让 LLM 根据传递的问题生成一个后退问题，使用 `后退问题` 执行相应的检索，利用检索到的文档+原始问题执行 RAG 索引增强生成，运行流程如下：



在 LangChain 中并没有封装好的 `回答回退策略检索器`，所以可以执行相应的封装，实现一个自定义检索器，实现代码如下：

```
from typing import List

import dotenv
import weaviate
from langchain_core.callbacks import CallbackManagerForRetrieverRun
from langchain_core.documents import Document
from langchain_core.language_models import BaseLanguageModel
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate,
FewShotChatMessagePromptTemplate
from langchain_core.retrievers import BaseRetriever
from langchain_core.runnables import RunnablePassthrough
from langchain_openai import OpenAIEmbeddings, ChatOpenAI
from langchain_weaviate import WeaviateVectorStore
from weaviate.auth import AuthApiKey
```

```
dotenv.load_dotenv()


class StepBackRetriever(BaseRetriever):
    """回答回退检索器"""
    retriever: BaseRetriever
    llm: BaseLanguageModel

    def _get_relevant_documents(
            self, query: str, *, run_manager: CallbackManagerForRetrieverRun
    ) -> List[Document]:
        """根据传递的query执行问题回退并检索"""
        # 1.构建少量提示模板
        examples = [
            {"input": "慕课网上有关于AI应用开发的课程吗？", "output": "慕课网上有哪些课
程？"},
            {"input": "慕小课出生在哪个国家？", "output": "慕小课的个人经历是怎样的？"},
            {"input": "司机可以开快车吗？", "output": "司机可以做什么？"},
        ]
        example_prompt = ChatPromptTemplate.from_messages([
            ("human", "{input}"),
            ("ai", "{output}")
        ])
        few_show_prompt = FewShotChatMessagePromptTemplate(
            examples=examples,
            example_prompt=example_prompt,
        )

        # 2.构建生成回退问题提示
        system_prompt = "你是一个世界知识的专家。你的任务是回退问题，将问题改述为更一般或者前
置问题，这样更容易回答，请参考示例来实现。"
        prompt = ChatPromptTemplate.from_messages([
            ("system", system_prompt),
            few_show_prompt,
            ("human", "{question}")
        ])

        # 3.构建生成回退问题的链
        chain = (
                {"question": RunnablePassthrough()}
                | prompt
                | self.llm
                | StrOutputParser()
                | self.retriever
        )

        return chain.invoke(query)


# 1.构建向量数据库与检索器
db = WeaviateVectorStore(
    client=weaviate.connect_to_wcs(
        cluster_url="https://mbakeruerziae6psyex7ng.c0.us-
west3.gcp.weaviate.cloud",
        auth_credentials=AuthApiKey("ZltPVa9ZSOxUcfafelsggGyyH6tnTYQYJvBx"),
    ),
```

```
    index_name="DatasetDemo",
    text_key="text",
    embedding=OpenAIEmbeddings(model="text-embedding-3-small"),
)
retriever = db.as_retriever(search_type="mmr")

# 2.创建回答回退检索器
step_back_retriever = StepBackRetriever(
    retriever=retriever,
    llm=ChatOpenAI(model="gpt-3.5-turbo-16k", temperature=0),
)

# 3.检索文档
documents = step_back_retriever.invoke("人工智能会让世界发生翻天覆地的变化吗?")
print(documents)
print(len(documents))
```

输出内容:

```
[Document(metadata={'source': './项目API文档.md', 'start_index': 5818.0},
page_content='json { "code": "success", "data": { "list": [ { "id": "1550b71a-
1444-47ed-a59d-c2f080fbae94", "conversation_id": "2d7d3e3f-95c9-4d9d-ba9c-
9daaf09cc8a8", "query": "能详细讲解下LLM是什么吗？", "answer": "LLM 即 Large Language
Model，大语言模型，是一种基于深度学习的自然语言处理模型，具有很高的语言理解和生成能力，能够处理各
式各样的自然语言任务，例如文本生成、问答、翻译、摘要等。它通过在大量的文本数据上进行训练，学习到语
言的模式、结构和语义知识'), Document(metadata={'source': './项目API文档.md',
'start_index': 6359.0}, page_content='1.7 [todo]删除特定的调试消息\n\n接口说明：用于删
除 AI 应用调试对话过程中指定的消息，该删除会在后端执行软删除操作，并且只有当会话 id 和消息 id
都匹配上时，才会删除对应的调试消息。\n\n接口信息：授权
+POST:/apps/:app_id/messages/:message_id/delete\n\n接口参数：\n\n请求参数：
\n\napp_id -> uuid：路由参数，需要删除消息归属的应用 id，格式为 uuid。\n\nmessage_id ->
uuid：路由参数，需要删除的消息 id，格式为 uuid。\n\n请求示例：\n\njson { "app_id":
"1550b71a-1444-47ed-a59d-c2f080fbae94", "message_id": "2d7d3e3f-95c9-4d9d-ba9c-
9daaf09cc8a8" }\n\n响应示例：\n\njson { "code": "success", "data": {}, "message":
"删除调试信息成功" }'), Document(metadata={'source': './项目API文档.md',
'start_index': 490.0}, page_content='带有分页数据的接口会在 data 内固定传递 list 和
paginator 字段，其中 list 代表分页后的列表数据，paginator 代表分页的数据。\n\npaginator 内
存在 4 个字段：current_page(当前页数) 、page_size(每页数据条数)、total_page(总页数)、
total_record(总记录条数)，示例数据如下：'), Document(metadata={'source': './项目API文
档.md', 'start_index': 2042.0}, page_content='dialog_round -> int：携带上下文轮数，类
型为非负整型。\n\nmemory_mode -> string：记忆类型，涵盖长记忆 long_term_memory 和 none
代表无。\n\nstatus -> string：应用配置的状态，drafted 代表草稿、published 代表已发布配置。
\n\nupdated_at -> int：应用配置的更新时间。\n\ncreated_at -> int：应用配置的创建时间。
\n\nupdated_at -> int：应用的更新时间。\n\ncreated_at -> int：应用的创建时间。\n\n响应示
例：')]
4
```

对比 `问题分解策略`，`回答回退策略` 仅仅多调用一次 LLM，所以相应速度更快，性能更高，并且复杂度更低，对于一些参数量较小的模型，也可以实现不错的效果，对于 `问题分解策略-迭代式回答`，再一些极端的情况下，模型输出了有偏差的内容，每次都在有偏差的 `问题+答案` 生成新内容，很有可能会导致最后的输出完全偏离开始的预设。

就像早些年很火的 `谷歌翻译将同一句话翻译20次`，输出的内容就完全偏离了原来的预设:

> 原文：吕布拜义父。
> 使用谷歌翻译20次后：在这个世界上，还有很多儿子等着我去当。

本质上就是因为无论是`谷歌翻译`还是`LLM大语言模型`，执行转换操作时，信息产生了丢失，并且随着迭代的增加，信息丢失得越来越多。

集成多种检索器算法实现混合检索

多检索器集成多种算法实现混合检索

01. 集成检索器的优势与使用

在 LangChain 中，封装了一个集成检索器 `EnsembleRetriever`，这个检索器接受一个检索器列表作为输入，并根据 RRF 算法对每个检索器的 `get_relevant_documents()` 方法产生的文档列表进行集成和重新排序。

集成检索器可以利用不同算法的优势，从而获得比任何单一算法更好的性能。集成检索器一个常见的案例是将 `稀疏检索器(如BM25)` 和 `密集检索器(如嵌入相似度)` 结合起来，因为它们的优势是互补的，这种检索方式也被称为混合检索（稀疏检索器擅长基于关键词检索，密集检索器擅长基于语义相似性检索）。

`混合检索器` 被广泛应用于各类 AI 应用开发平台，例如：`Dify`、`Coze`、`智谱` 等平台，也是课程 LLMOps 项目会使用的检索策略。

在 LangChain 中使用 `EnsembleRetriever` 非常简单，传递 `retrievers(检索器列表)` 和 `weights(检索器权重)` 即可。

例如使用 `BM25关键词搜索` 和 `FASISS相似性搜索` 进行结合，实现混合搜索，首先安装 `rank_bm25` 包，命令如下

```
pip install -U rank_bm25
```

混合搜索实例代码如下：

```
import dotenv
from langchain.retrievers import EnsembleRetriever
from langchain_community.retrievers import BM25Retriever
from langchain_community.vectorstores import FAISS
from langchain_core.documents import Document
from langchain_openai import OpenAIEmbeddings

dotenv.load_dotenv()

# 1.创建文档列表
documents = [
    Document(page_content="笨笨是一只很喜欢睡觉的猫咪", metadata={"page": 1}),
    Document(page_content="我喜欢在夜晚听音乐，这让我感到放松。", metadata={"page":
2}),
    Document(page_content="猫咪在窗台上打盹，看起来非常可爱。", metadata={"page": 3}),
    Document(page_content="学习新技能是每个人都应该追求的目标。", metadata={"page":
4}),
    Document(page_content="我最喜欢的食物是意大利面，尤其是番茄酱的那种。", metadata=
{"page": 5}),
    Document(page_content="昨晚我做了一个奇怪的梦，梦见自己在太空飞行。", metadata=
{"page": 6}),
    Document(page_content="我的手机突然关机了，让我有些焦虑。", metadata={"page": 7}),
    Document(page_content="阅读是我每天都会做的事情，我觉得很充实。", metadata={"page":
8}),
    Document(page_content="他们一起计划了一次周末的野餐，希望天气能好。", metadata=
{"page": 9}),
    Document(page_content="我的狗喜欢追逐球，看起来非常开心。", metadata={"page": 10}),
]
```

```
# 2.构建BM25关键词检索器
bm25_retriever = BM25Retriever.from_documents(documents)
bm25_retriever.k = 4

# 3.创建FAISS向量数据库检索
faiss_db = FAISS.from_documents(documents,
embedding=OpenAIEmbeddings(model="text-embedding-3-small"))
faiss_retriever = faiss_db.as_retriever(search_kwargs={"k": 4})

# 4.初始化集成检索器
ensemble_retriever = EnsembleRetriever(
    retrievers=[bm25_retriever, faiss_retriever],
    weights=[0.5, 0.5],
)

# 5.执行检索
docs = ensemble_retriever.invoke("除了猫，你养了什么宠物呢？")
print(docs)
print(len(docs))
```

输出内容：

```
[Document(metadata={'page': 10}, page_content='我的狗喜欢追逐球，看起来非常开心。'),
Document(metadata={'page': 3}, page_content='猫咪在窗台上打盹，看起来非常可爱。'),
Document(metadata={'page': 9}, page_content='他们一起计划了一次周末的野餐，希望天气能
好。'), Document(metadata={'page': 1}, page_content='笨笨是一只很喜欢睡觉的猫咪'),
Document(metadata={'page': 8}, page_content='阅读是我每天都会做的事情，我觉得很充实。'),
Document(metadata={'page': 7}, page_content='我的手机突然关机了，让我有些焦虑。'),
Document(metadata={'page': 5}, page_content='我最喜欢的食物是意大利面，尤其是番茄酱的那
种。')]
7
```

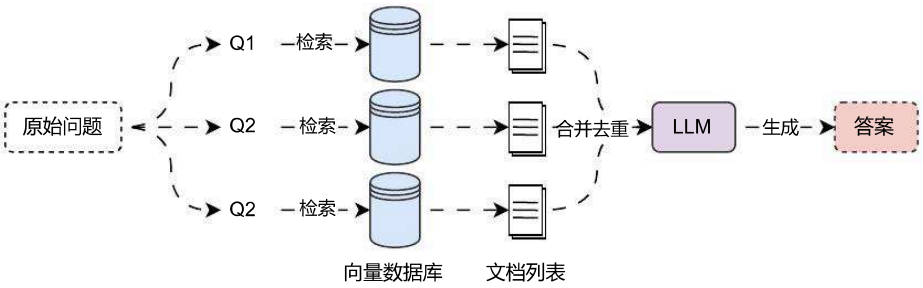
在实际的开发中，除了硬编码不同检索器与对应的权重，还可以在运行时配置检索器，在检索时动态控制某个检索器输出内容的数量、权重等，例如：

```
faiss_retriever = faiss_db.as_retriever(search_kwargs={"k":
4}).configurable_fields(
    search_kwargs=ConfigurableField(
        id="search_kwargs_faiss",
        name="搜索参数",
        description="要使用的搜索参数",
    )
)

config = {"configurable": {"search_kwargs_faiss": {"k": 1}}}
docs = ensemble_retriever.invoke("苹果", config=config)
```

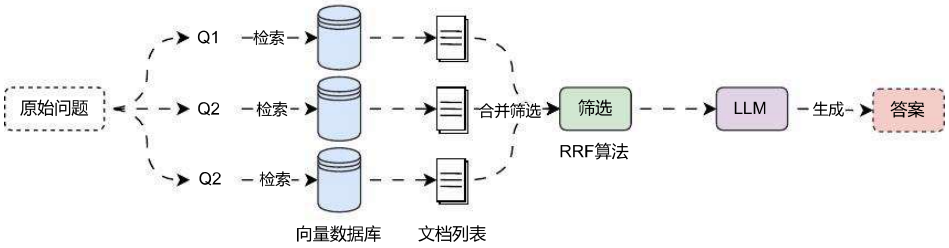
02. 查询转换阶段优化策略总结

至此，在 RAG 的 `查询转换` 阶段，目前市面上主流的优化策略其实我们都已经讲解完了，涵盖了：**多查询重写、RAG 多查询结果融合、问题分解策略、回答回退策略、HyDE 混合策略、集成检索器策略**等，不同的优化策略有不同的优缺点：

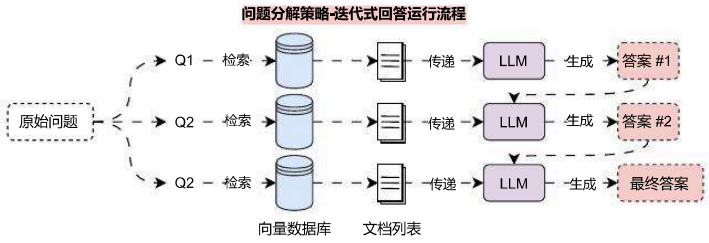
1. `多查询重写`：实现简单，使用参数较小的模型也可以完成对查询的转换（不涉及回答），因为多查询可以并行检索，所以性能较高，但是在合并的时候，没有考虑到不同文档的权重，仅仅按照顺序进行合并，会让某些高权重的文档在使用时可能被剔除。

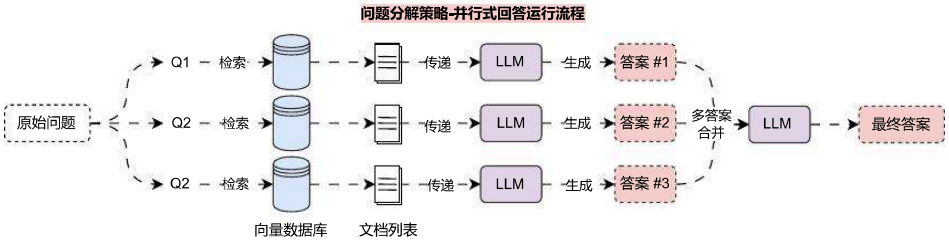


2. `多查询融合`：在 `多查询重写` 的基础上引入了 RRF 算法，对不同查询检索到的文档列表的每一篇文档计算对应的权重，将排名靠前亦或者是出现次数更多的文档放置到合并文档的靠前部分。

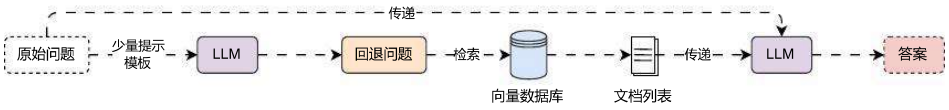


3. `问题分解策略`：通过将复杂问题/数学问题分解成多个子问题，从而实现对每个子问题的 `检索-生成` 流程，最后再将所有子问题的 `答案` 进行合并，在转换缓解涉及到对子问题的回答，所以对于中间 LLM 的要求比较高，性能相对也比较差，在上下文长度不足的情况下，拆分问题并迭代回答，可能会让最终答案偏离原始的提问。





4. `回答回退策略`：通过提出一个前置问题/通用问题用于优化原始的复杂问题，从而获得更大的搜索范围，提升检索到相关性高的文档的概率，因为中间 LLM 没涉及到回答，所以可以使用参数量较小的模型来实现，性能相对较高。



5. `HyDE混合策略`：通过将 `query` 转换成 `doc` 的思想，并执行 `doc-doc` 实现对称性检索，从而提升找到相似性文档的概率，但是对于上下文不足，或者是开放性问题，`HyDE混合策略` 的效果相对较差。

6. `集成检索器策略`：目前使用频率最高的检索器策略，集成多种不同的检索方式，充分利用不同算法的优点，最后再使用 RRF 算法对不同检索器检索到的数据进行合并，从而得到最终文档列表。