使用检索器逻辑路由缩减检索范围

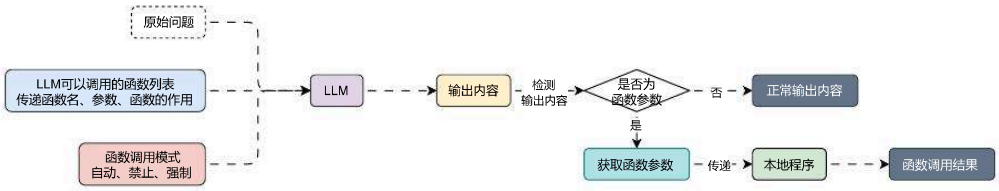
使用检索器逻辑路由缩减检索范围

01. 大模型的函数回调与规范化输出

让 LLM 执行 `函数回调` 听起来是一个很高级的技术，但是理解起来其实很简单，简单来说，就是传递给 LLM 一大堆工具/函数（传递函数名字、参数描述、函数作用等），让 LLM 自行识别，在当前用户的提问下，识别出最适合调用的函数（选择一个、多个、或者不调用、亦或者强制调用其中某个），然后把要调用的函数的参数作为 LLM 的输出内容。

执行完上面这一步，虽然说是 LLM 函数回调，但是 LLM 并不会真正调用本地的函数，本地的函数仍然是需要本地的程序根据 LLM 的输出内容来调用，拥有 `function call` 就意味着 LLM 可以智能选择不同的工具，并且规范化输出。

所以对于一个完整的 `函数回调` 运行过程来说，**除了要有 LLM 的参与，还要有本地程序的参与**，完整运行流程如下：



既然 LLM 可以强制调用某个函数，并且函数某个函数的参数输出对应的数据，所以其实**可以考虑利用 `函数回调` 这个功能来执行相应的规范化输出，我们构建一个假函数，并告知 LLM 强制调用这个函数，让 LLM 返回其函数的参数信息**，只需要将需要规范化的数据写成函数参数，并配上对应的解释，其实就可以实现规范化输出。

通过这种方式约束大语言模型生成的内容，比 Prompt 可靠性更高，而且性能更佳。

OpenAI 大语言模型函数回调文档：https://platform.openai.com/docs/api-reference/chat/create

在 LangChain 中使用 OpenAI 的 `函数回调` 来执行规范化输出，其实非常简单，调用大语言模型的 `.with_structured_output()` 方法并传递一个 `BaseModel` 的子类即可，在底层，这个函数会自动将对应的 `BaseModel` 转换成函数回调，并强制让 LLM 调用。

例如：

```
from langchain_core.pydantic_v1 import BaseModel, Field

class RouteQuery(BaseModel):
    """将用户查询映射到最相关的数据源"""
    datasource: Literal["python_docs", "js_docs", "golang_docs"] = Field(
        description="根据给定用户问题，选择哪个数据源最相关以回答他们的问题"
    )

llm = ChatOpenAI(model="gpt-3.5-turbo-16k", temperature=0)
structured_llm = llm.with_structured_output(RouteQuery)
```

使用起来传递对应的 `prompt` 给大语言模型，大语言模型会自动按照传递的模型规范，生成对应的类实例，从而实现规范输出。

如果单纯靠 `prompt` 来约束大语言模型的输出，例如有这么一段 `prompt`：

```
根据给定用户问题，选择哪个数据源最相关以回答他们的问题。

目前有3个数据源：python_docs、js_docs、golang_docs。

用户的问题是：

---
为什么下面的代码不工作了，请帮我检查下：

from langchain_core.prompts import ChatPromptTemplate

prompt = ChatPromptTemplate.from_messages(["human", "speak in {language}"])
prompt.invoke("中文")
---
```

大语言模型会非常热情地帮助我们解答问题，得到的回复是：

```
根据用户的问题，涉及到 `langchain_core` 中的 `ChatPromptTemplate` 类的使用问题。这个类可能与语言处理或者自然语言生成有关。考虑到问题中的代码以及 `"中文"` 这一指示，最相关的数据源应该是 `python_docs`，因为这里涉及到 Python 代码的调用和可能的语法或库的使用问题。

因此，建议查看 `python_docs` 数据源以获取与 `ChatPromptTemplate` 类和相关 Python 代码的信息，以便更好地帮助用户解决问题。
```

虽然我们人类可以从这么一大长串内容中看出，是需要寻找 `python_docs` 的数据源，但是对于程序来说，要的其实只是 `python_docs` 这个字符串，并不是要这么一大长串带有语义场景的文本，大语言模型越热情，返回的数据越难处理。

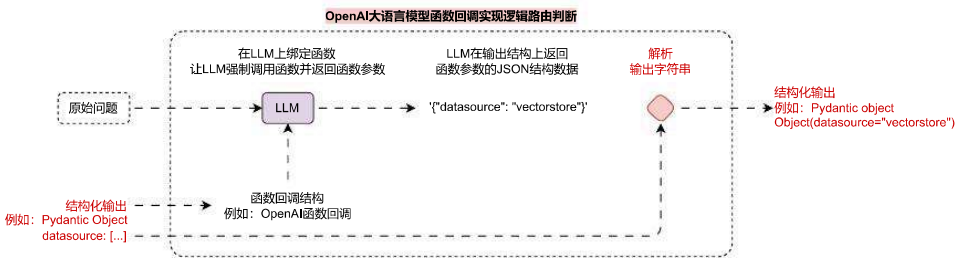
所以 `函数回调` 在进行规范化输出时，作用特别大！而且不仅仅是规范化输出，函数回调的作用还远远不仅如此，在下一章的课时中，我们会来重点学习函数回调/Agent/LangGraph，去构建更加智能的应用！

02. 检索器的逻辑路由实现

在 RAG 应用开发中，想根据不同的问题检索不同的 `检索器/向量数据库`，其实只需要设定要对应的 Prompt，然后让 LLM 根据传递的问题返回需要选择的 `检索器/向量数据库` 的名称，然后根据得到的名称选择不同的 `检索器` 即可。

但是对于 LLM 来说，如果使用普通的 prompt 来约束输出内容的格式与规范，因为 LLM 的特性，很难保证输出格式符合特定的需求，所以可以考虑使用 `函数回调` 来实现，即设定一个 `虚假的函数`，告诉 LLM，这个函数有对应的参数，让 LLM 强制调用这个函数，这个时候 LLM 就会输出函数的调用参数，从而保证输出的统一性。

使用 `函数回调` 实现的检索器逻辑路由运行流程图如下：



假设目前有 3 个向量数据库/集合，分别代表 `python_docs`、`js_docs`、`golang_docs`，需要根据用户传递的问题判断与哪个向量数据库最接近，使用最接近的向量数据库进行检索，代码示例：

```
from typing import Literal

import dotenv
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.pydantic_v1 import BaseModel, Field
from langchain_core.runnables import RunnablePassthrough
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()


class RouteQuery(BaseModel):
    """将用户查询映射到最相关的数据源"""
    datasource: Literal["python_docs", "js_docs", "golang_docs"] = Field(
        description="根据给定用户问题，选择哪个数据源最相关以回答他们的问题"
    )


def choose_route(result: RouteQuery):
    if "python_docs" in result.datasource.lower():
        return "chain for python_docs"
    elif "js_docs" in result.datasource.lower():
        return "chain for js_docs"
    else:
        return "golang_docs"


# 1.构建大语言模型并进行结构化输出
llm = ChatOpenAI(model="gpt-3.5-turbo-16k", temperature=0)
structured_llm = llm.with_structured_output(RouteQuery)

# 2.创建路由逻辑链
prompt = ChatPromptTemplate.from_messages([
    ("system", "你是一个擅长将用户问题路由到适当的数据源的专家。\n请根据问题涉及的编程语言，将其路由到相关数据源"),
    ("human", "{question}")
])
router = {"question": RunnablePassthrough()} | prompt | structured_llm |
choose_route

# 3.执行相应的提问，检查映射的路由
question = """为什么下面的代码不工作了，请帮我检查下：

from langchain_core.prompts import ChatPromptTemplate

prompt = ChatPromptTemplate.from_messages(["human", "speak in {language}"])
prompt.invoke("中文")"""

# 4.选择不同的数据库
print(router.invoke(question))
```

输出内容：

```
datasource='python_docs'
chain for python_docs
```

使用语义路由选择不同的 Prompt 模板

使用语义路由选择不同的 Prompt 模板

在 RAG 应用开发中，针对不同场景的问题使用 `特定化的prompt模板` 效果一般都会比通用模板会好一些，例如在 `教培场景`，制作一个可以教学 `物理+数学` 的授课机器人，如果使用通用的 `prompt模板`，会导致 prompt 编写变得非常复杂；反过来如果 prompt 写的简单，有可能没法起到很好的回复效果。

如果能针对用户的提问，例如用户提问的内容是数学相关的则使用数学的模板，提问的内容是物理相关的则使用物理的模板，针对性选择不同的模板，LLM 生成的内容会比使用通用模板会更好，例如下方有两个 `prompt模板`：

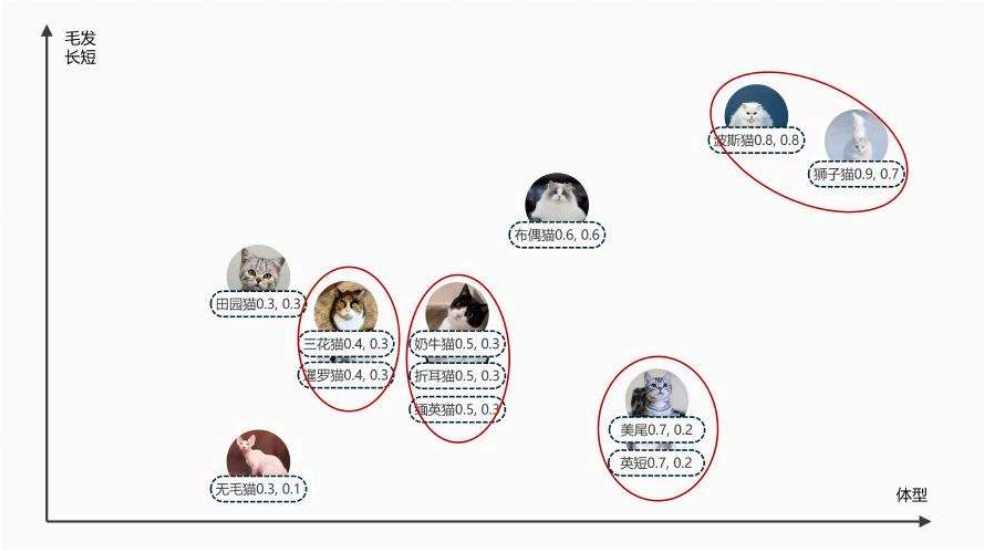
```
physics_template = """你是一位非常聪明的物理教授。
你擅长以简洁易懂的方式回答物理问题。
当你不知道问题的答案时，你会坦率承认自己不知道。

这是一个问题：
{query}"""
math_template = """你是一位非常优秀的数学家。你擅长回答数学问题。
你之所以如此优秀，是因为你能将复杂的问题分解成多个小步骤。
并且回答这些小步骤，然后将它们整合在一起回来更广泛的问题。

这是一个问题：
{query}"""
```

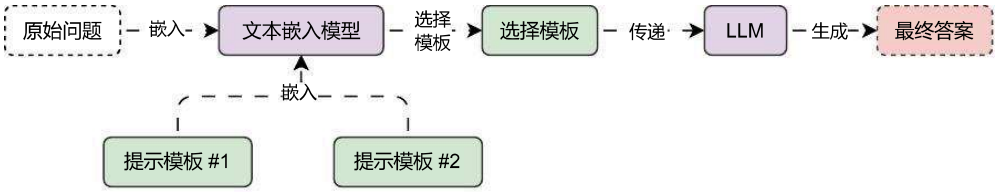
基于这个思想和我们前面学习的 `向量` 与 `文本嵌入模型`，我们其实可以猜测，提问如果是 `数学问题(能介绍下余弦计算公式么？)`，在向量空间上，正常情况下和 `数学模板` 的向量靠的更近；映射到 `物理问题(黑洞是什么?)` 上也是一样的。

用前面的 `猫猫向量` 表示就是更接近，相似度更高，如下：



所以利用向量执行相似性搜索，不仅可以作用于 `向量数据库`，我们还可以利用 `原始问题` 与 `prompt模板` 的相似性，来找到类型、语义上更接近的模板，从而实现对 `prompt模板` 的动态路由。

该语义路由的运行流程其实也非常简单，如下：



在 LangChain 中，实现的代码示例如下：

```
import dotenv
from langchain.utils.math import cosine_similarity
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnablePassthrough, RunnableLambda
from langchain_openai import OpenAIEmbeddings, ChatOpenAI

dotenv.load_dotenv()

# 1.定义两份不同的prompt模板(物理模板、数学模板)
physics_template = """你是一位非常聪明的物理教程。
你擅长以简洁易懂的方式回答物理问题。
当你不知道问题的答案时，你会坦率承认自己不知道。

这是一个问题：
{query}"""
math_template = """你是一位非常优秀的数学家。你擅长回答数学问题。
你之所以如此优秀，是因为你能将复杂的问题分解成多个小步骤。
并且回答这些小步骤，然后将它们整合在一起回来更广泛的问题。

这是一个问题：
{query}"""

# 2.创建文本嵌入模型，并执行嵌入
embeddings = OpenAIEmbeddings(model="text-embedding-3-small")
prompt_templates = [physics_template, math_template]
prompt_embeddings = embeddings.embed_documents(prompt_templates)


def prompt_router(input) -> ChatPromptTemplate:
    """根据传递的query计算返回不同的提示模板"""
    # 1.计算传入query的嵌入向量
    query_embedding = embeddings.embed_query(input["query"])

    # 2.计算相似性
    similarity = cosine_similarity([query_embedding], prompt_embeddings)[0]
    most_similar = prompt_templates[similarity.argmax()]
    print("使用数学模板" if most_similar == math_template else "使用物理模板")

    # 3.构建提示模板
    return ChatPromptTemplate.from_template(most_similar)


chain = (
        {"query": RunnablePassthrough()}
        | RunnableLambda(prompt_router)
        | ChatOpenAI(model="gpt-3.5-turbo-16k")
```

```
        | StrOutputParser()
)

print(chain.invoke("黑洞是什么?"))
print("======================")
print(chain.invoke("能介绍下余弦计算公式么？"))
```

输出内容:

```
使用物理模板
黑洞是一种极为密集的天体，其引力非常强大，甚至连光都无法逃离它的吸引力。黑洞形成于恒星坍缩或者质量非常大的天体的坍塌过程中。在黑洞的中心，存在一个称为“奇点”的点，其中物质密度无限大，空间弯曲度也达到了极限。黑洞周围的区域被称为“事件视界”，在这个视界内，一切进入的物质都无法逃脱黑洞的吸引力。黑洞是宇宙中极为神秘而又引人入胜的天体。
======================
使用数学模板
余弦（cosine）计算公式是三角函数中的一种，用于计算一个三角形的两个边长和夹角之间的关系。

在一个直角三角形中，余弦的定义是：余弦值等于直角边上的斜边与该直角边的比值。

余弦的计算公式如下：
cos(θ) = adjacent / hypotenuse

其中，θ表示夹角的大小（以弧度为单位），adjacent表示夹角边的邻边长度，hypotenuse表示斜边的长度。

需要注意的是，余弦计算公式只适用于直角三角形，当夹角不是直角时，需要使用其他三角函数（如正弦、正切等）来计算。

通过将复杂的问题分解成小步骤，我们可以先计算出夹角的余弦值，然后再将其应用到更广泛的问题中，如求解三角形的面积、边长等。
```

自查询检索器实现动态元数据过滤

自查询检索器实现动态元数据过滤

01. 查询构建与自查询检索器

在 RAG 应用开发中，检索外部数据时，前面的优化案例中，无论是生成的 `子查询`、`问题分解`、`生成假设性文档`，最后在执行检索的时候使用的都是固定的筛选条件（没有附加过滤的相似性搜索）。

但是在某些情况下，用户发起的原始提问其实隐式携带了 `筛选条件`，例如提问：

> 请帮我整理下关于2023年全年关于AI的新闻汇总。

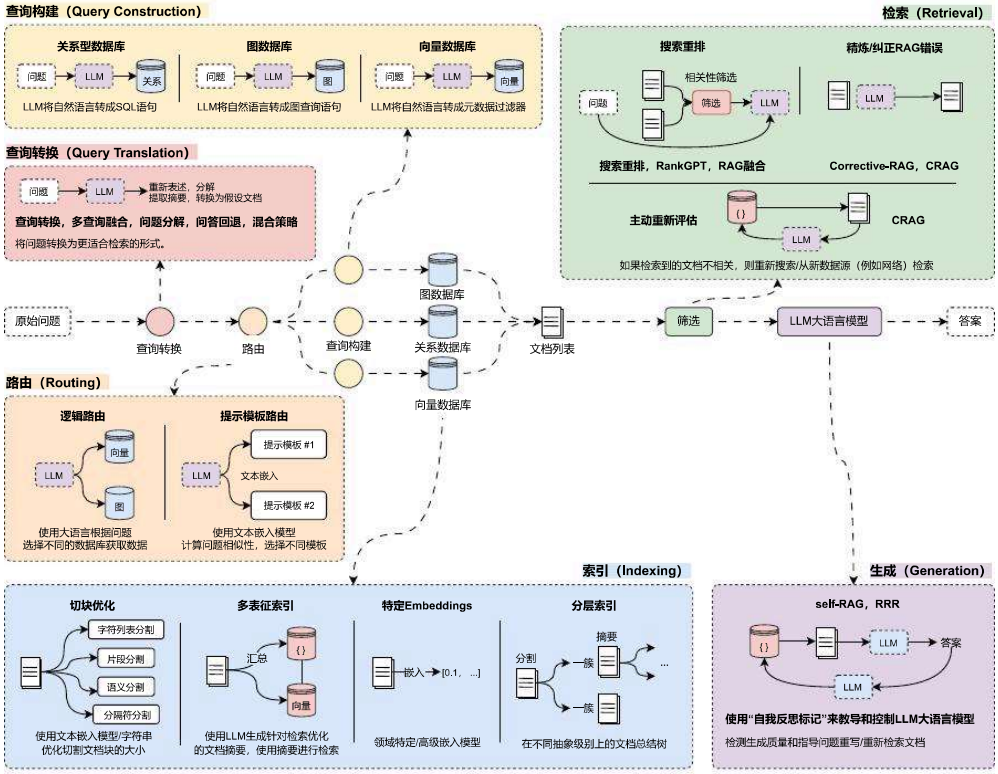
在这段 `原始提问` 中，如果执行相应的向量数据库相似性搜索，其实是附加了 `筛选条件` 的，即 `year=2023`，但是在普通的相似性搜索中，是不会考虑 2023 年这个条件的（因为没有添加元数据过滤器，2022年和2023年数据在高维空间其实很接近），存在很大概率会将其他年份的数据也检索出来。

那么有没有一种策略，能根据用户传递的 `原始问题` 构建相应的 `元数据过滤器` 呢？这样在搜索的时候带上对应的元数据过滤器，不仅可以压缩检索范围，还能提升搜索的准确性。这个思想其实就是 `查询构建` 或者称为 `自查询`。

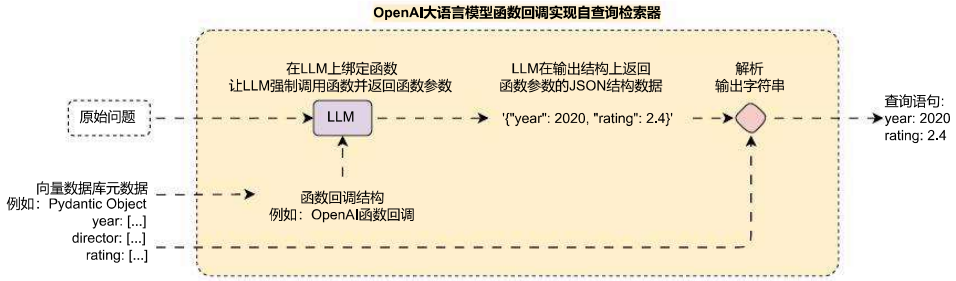
并且除了 `向量数据库`，类比映射到 `关系型数据库`、`图数据库` 也是同样的操作技巧，即：

1. `关系型数据库自查询`：使用 LLM 将自然语言转换成 SQL 过滤语句。
2. `图数据库自查询`：使用 LLM 将自然语言转换成图查询语句。
3. `向量数据库`：使用 LLM 将自然语言转换成元数据过滤器/向量检索器。

这就是 `查询构建` 概念的由来，但是并不是所有的数据都支持 `查询构建` 的，需要看存储的 `Document` 是否存在元数据，对应的数据库类型是否支持筛选，在 LangChain 中是否针对性做了封装（如果没封装，自行实现难度比较大）。



将 `查询构建` 这个步骤单独拎出来，它的运行流程其实很简单，但是底层的操作非常麻烦，如下：



在 LangChain 中，针对一些高频使用的向量数据库封装了 `自查询检索器` 的相关支持——`SelfQueryRetriever`，无需自行构建转换语句与解析，使用该类进行二次包装即可。

所有支持 `自查询检索器` 的向量数据库都在这个链接内部可以看到使用示例：https://imooc-langchain.shortvar.com/docs/integrations/retrievers/self_query/（但是因为向量数据库的更新频率过快，LangChain 封装的部分向量数据库已经更新，但是 `SelfQueryRetriever` 内部的逻辑还未更新）。

`SelfQueryRetriever` 使用起来也非常简单，以 `Pinecone` 向量数据库为例，首先安装对应的依赖：

```
pip install --upgrade --quiet lark
```

定义好 `带元数据的文档`、`支持过滤的元数据`、`包装的向量数据库`、`文档内容的描述` 等信息，即可进行快速包装，示例代码如下：

```
import dotenv
from langchain.chains.query_constructor.base import AttributeInfo
from langchain.retrievers import SelfQueryRetriever
from langchain_core.documents import Document
```

```python
from langchain_openai import OpenAIEmbeddings, ChatOpenAI
from langchain_pinecone import PineconeVectorStore

dotenv.load_dotenv()

# 1.构建文档列表并上传到数据库
documents = [
    Document(
        page_content="肖申克的救赎",
        metadata={"year": 1994, "rating": 9.7, "director": "弗兰克·德拉邦特"},
    ),
    Document(
        page_content="霸王别姬",
        metadata={"year": 1993, "rating": 9.6, "director": "陈凯歌"},
    ),
    Document(
        page_content="阿甘正传",
        metadata={"year": 1994, "rating": 9.5, "director": "罗伯特·泽米吉斯"},
    ),
    Document(
        page_content="泰坦尼克号",
        metadat={"year": 1997, "rating": 9.5, "director": "詹姆斯·卡梅隆"},
    ),
    Document(
        page_content="千与千寻",
        metadat={"year": 2001, "rating": 9.4, "director": "宫崎骏"},
    ),
    Document(
        page_content="星际穿越",
        metadat={"year": 2014, "rating": 9.4, "director": "克里斯托弗·诺兰"},
    ),
    Document(
        page_content="忠犬八公的故事",
        metadat={"year": 2009, "rating": 9.4, "director": "莱塞·霍尔斯道姆"},
    ),
    Document(
        page_content="三傻大闹宝莱坞",
        metadat={"year": 2009, "rating": 9.2, "director": "拉库马·希拉尼"},
    ),
    Document(
        page_content="疯狂动物城",
        metadat={"year": 2016, "rating": 9.2, "director": "拜伦·霍华德"},
    ),
    Document(
        page_content="无间道",
        metadat={"year": 2002, "rating": 9.3, "director": "刘伟强"},
    ),
]
db = PineconeVectorStore(
    index_name="llmops",
    embedding=OpenAIEmbeddings(model="text-embedding-3-small"),
    namespace="dataset",
    text_key="text"
)
db.add_documents(documents)
```

```
# 2.创建自查询元数据
metadata_filed_info = [
    AttributeInfo(
        name="year",
        description="电影的发布年份",
        type="integer",
    ),
    AttributeInfo(
        name="rating",
        description="电影的评分",
        type="float",
    ),
    AttributeInfo(
        name="director",
        description="电影的导演",
        type="string",
    ),
]

# 3.创建子查询检索
self_query_retriever = SelfQueryRetriever.from_llm(
    llm=ChatOpenAI(model="gpt-3.5-turbo-16k", temperature=0),
    vectorstore=db,
    document_contents="电影的名字",
    metadata_field_info=metadata_filed_info,
    enable_limit=True,
)

# 4.检索示例
search_docs = self_query_retriever.invoke("查找下评分高于9.5分的电影")

print(search_docs)
print(len(search_docs))
```

输出内容：

```
[Document(metadata={'director': '陈凯歌', 'rating': 9.6, 'year': 1993.0},
page_content='霸王别姬'), Document(metadata={'director': '弗兰克·德拉邦特', 'rating':
9.7, 'year': 1994.0}, page_content='肖申克的救赎')]
2
```

02. 自查询检索器的运行逻辑与衍生

在 LangChain 中，涉及调用第三方服务或者调用本地自定义工具的，例如课程中学习的 `自查询检索器`、`检索器逻辑路由` 等，在底层都是通过一个预设好的 Prompt 生成符合相应规则的内容（字符串、JSON），然后通过 `解析器` 解析生成的内容，并将解析出来的结构化内容调用特定的接口、服务亦或者本地函数实现。

例如在 `子查询检索器` 底层，首先使用 `FewShotPromptTemplate` + `函数回调/结构化输出` 生成特定规则的 `查询语句`，这段提示代码如下：

```
DEFAULT_SCHEMA = """\
<< Structured Request Schema >>
```

```
When responding use a markdown code snippet with a JSON object formatted in the
following schema:

```json
{{{{
    "query": string \\ text string to compare to document contents
    "filter": string \\ logical condition statement for filtering documents
}}}}
```

The query string should contain only text that is expected to match the contents
of documents. Any conditions in the filter should not be mentioned in the query
as well.

A logical condition statement is composed of one or more comparison and logical
operation statements.

A comparison statement takes the form: `comp(attr, val)`:
- `comp` ({allowed_comparators}): comparator
- `attr` (string): name of attribute to apply the comparison to
- `val` (string): is the comparison value

A logical operation statement takes the form `op(statement1, statement2, ...)`:
- `op` ({allowed_operators}): logical operator
- `statement1`, `statement2`, ... (comparison statements or logical operation
statements): one or more statements to apply the operation to

Make sure that you only use the comparators and logical operators listed above
and no others.
Make sure that filters only refer to attributes that exist in the data source.
Make sure that filters only use the attributed names with its function names if
there are functions applied on them.
Make sure that filters only use format `YYYY-MM-DD` when handling date data typed
values.
Make sure that filters take into account the descriptions of attributes and only
make comparisons that are feasible given the type of data being stored.
Make sure that filters are only used as needed. If there are no filters that
should be applied return "NO_FILTER" for the filter value.\
"""
```

原始问题如下：

```
查找下评分高于9.5分的电影
```

生成的查询语句原文如下：

```
{
    "query": "",
    "filter": "gt(\"rating\", 9.5)"
}
```

接下来使用特定的转换器，将生成的查询语句转换成适配向量数据库的`过滤器`，并在检索时传递该参数，从而完成自查询构建的全过程，不同的向量数据库对应的转换器差异也非常大。

这个思想其实已经涉及到将 LLM 与企业自有应用/API进行快速对接智能化，即如何将 LLM 生成的 `文字信息` 对接到当前业务系统中。

例如企业自有一套 PPT 生成 API 接口，通过传递设定的参数即可生成对应的 PPT，假设有这样一段参数规则：

```
[
    {
        "page": 1, # PPT的页数
        "background": {
            "size": [400, 600],  # 背景图片大小
            "position": [0, 0],  # 背景图片位置
            "image_url": "xxx",  # 图片URL
            ...
        },
        "objects": [
            {
                "type": "title",  # 对象类型
                "attribute": {
                    "content": "求知若渴，虚心若愚",  # 标题内容
                    "size": 20,  # 标题字体大小
                    "color": "#000000",  # 标题颜色
                    "position": [240, 128],  # 标题位置
                    "font": "微软雅黑",  # 标题字体
                    ...
                }
            },
            ...
        ],
        ...
    }
]
```

如果想通过 LLM 构建一个 PPT 自动生成工具，只需要设定好 `prompt`，让 LLM 按照特定的规则生成一段用于描述 `PPT信息的参数`，接下来解析这段参数，并将相应的参数传递给现成没有智能的 PPT 生成工具，即可快速实现 `自然语言->PPT` 的过程。

当然在这类应用的开发过程中，需要考虑的其他因素其实还非常多，例如：

1. `大语言模型上下文长度`：涉及到的生成内容过多，没法一次性生成，需要确保多次生成参数之间的连贯性。
2. `prompt描述`：对于一些复杂的应用，原有参数可能非常复杂，需要通过合适的方式将参数描述添加到 prompt 中。
3. `生成内容To调用参数`：对于生成的内容，有些是字符串的形式（函数回调生成的参数数量是有限制的），如何通过合适方式将字符串无损转换成调用参数。
4. `参数校验`：因为大语言模型输出的随机性，需要对生成的参数进行校验，校验的复杂性。

至此，学习到这里，其实相信大家对如何将 LLM 接入到原有应用已经有了一个初步的概念了，在下一章的课时中，我们将会学习 `Agent`、`AI 工作流` 等内容，让 LLM 成为一个真正的大脑，让其能更智能地和现有应用对接。

MultiVector 实现多向量检索文档

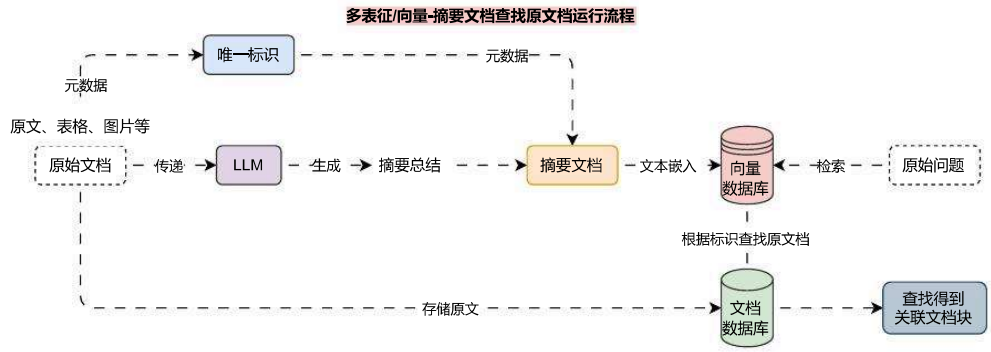
MultiVector实现多向量检索检索文档

01. 多表征/向量索引

在前面的课时中，我们为每一个文档块生成一条向量用于记录该文本的特征信息，如果能从多个维度记录该文档块的信息，会大大增加该文档块被检索到的概率，`多个维度记录信息`等同于为文档块生成`多个向量`，支持的方法如下：

1. `把文档切割成更小的块`：通过检索更小的块，但是查找其父类文档（ParentDocumentRetriever）。
2. `摘要`：使用 LLM 为每个文档块生成一段摘要，将其和原文档一起嵌入或者代替，返回时返回原文档。
3. `假设性问题`：使用 LLM 为每个文档块生成适合回答的假设性问题，将其和原文档一起嵌入或者代替，返回时返回原文档。

通过这种方式可以为一个文档块生成多条特征/向量，**在检索时能提升关联文档被检索到的概率**，多向量检索的运行流程其实也非常简单，以`摘要文档`检索`原文档`为例，运行流程图如下：



通过上面的运行流程，可以很容易知道在`原始文档`和`摘要文档`中都在元数据中设置了`唯一标识`，从向量数据库中找到符合规则的数据后，通过查找其`元数据`的唯一标识，即可在`文档数据库`中匹配出原文档，完成整个多表征/向量的检索。

02. 多向量索引示例

在 LangChain 中，为多向量索引的集成封装了`MultiVectorRetriever`类，实例化该类只需要传递`向量数据库`、`字节存储数据库(文档数据库)`、`id标识(关联标识)`即可快速完成整个运行流程的集成。

以`FAISS向量数据库`和`本地文件存储库`为例，构建一个`存储摘要->检索原文`的优化策略，代码示例如下：

```
import uuid

import dotenv
from langchain.retrievers import MultiVectorRetriever
from langchain.storage import LocalFileStore
from langchain_community.document_loaders import UnstructuredFileLoader
from langchain_community.vectorstores import FAISS
from langchain_core.documents import Document
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
```

```python
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
from langchain_text_splitters import RecursiveCharacterTextSplitter

dotenv.load_dotenv()

# 1.创建加载器、文本分割器并处理文档
loader = UnstructuredFileLoader("./电商产品数据.txt")
text_splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=50)
docs = loader.load_and_split(text_splitter)

# 2.定义摘要生成链
summary_chain = (
        {"doc": lambda x: x.page_content}
        | ChatPromptTemplate.from_template("总结以下文档的内容: \n\n{doc}")
        | ChatOpenAI(model="gpt-3.5-turbo-16k", temperature=0)
        | StrOutputParser()
)

# 3.批量生成摘要与唯一标识
summaries = summary_chain.batch(docs, {"max_concurrency": 5})
doc_ids = [str(uuid.uuid4()) for _ in enumerate(docs)]

# 4.构建摘要文档
summary_docs = [
    Document(page_content=summary, metadata={"doc_id": doc_ids[idx]})
    for idx, summary in enumerate(summaries)
]

# 5.构建文档数据库与向量数据库
byte_store = LocalFileStore("./multy-vector")
db = FAISS.from_documents(
    summary_docs,
    embedding=OpenAIEmbeddings(model="text-embedding-3-small"),
)

# 6.构建多向量检索器
retriever = MultiVectorRetriever(
    vectorstore=db,
    byte_store=byte_store,
    id_key="doc_id",
)

# 7.将摘要文档和原文档存储到数据库中
retriever.docstore.mset(list(zip(doc_ids, docs)))

# 8.执行检索
search_docs = retriever.invoke("推荐一些潮州特产?")
print(search_docs)
print(len(search_docs))
```

输出内容:

```
[Document(metadata={'source': './电商产品数据.txt'}, page_content='产品名称：潮汕鱼丸\n\n电商网址：shop.example.com/fishballs\n\n产品描述：潮汕鱼丸采用新鲜鱼肉，加入少量淀粉和调味料，手工捶打成丸，Q弹爽滑，鱼香浓郁。\n\n产品特点:\n\n原材料：新鲜鱼肉、淀粉、盐、胡椒粉\n\n制作工艺：传统手工捶打\n\n口感：Q弹爽滑，鲜美可口\n\n净重：500克/袋、1000克/袋\n\n保质期：6个月（冷冻保存）\n\n发货方式：顺丰冷链配送，确保新鲜\n\n物流信息：24小时内发货，预计2\n\n3天到货\n\n推荐菜系:\n\n鱼丸火锅：搭配各类蔬菜、菌类，煮至鱼丸浮起即可。\n\n鱼丸煮汤：与蔬菜同煮，味道鲜美。\n\n价格:\n\n500克：55元/袋\n\n1000克：100元/袋\n\n6. 潮汕豆腐花\n\n产品名称：潮汕豆腐花\n\n电商网址：shop.example.com/tofupudding\n\n产品描述：潮汕豆腐花使用优质黄豆，传统工艺制作，质地细腻，入口即化，豆香浓郁。\n\n产品特点:\n\n原材料：黄豆、水、石膏\n\n制作工艺：传统手工点浆\n\n口感：细腻嫩滑，豆香浓郁\n\n净重：450克/盒\n\n保质期：5天（冷藏保存）'), Document(metadata={'source': './电商产品数据.txt'}, page_content='产品特点:\n\n原材料：猪后腿肉、香料、盐、糖\n\n制作工艺：精细切割，手工卷制\n\n口感：鲜嫩多汁，咸香可口\n\n净重：400克/袋、800克/袋\n\n保质期：3个月（冷冻保存）\n\n发货方式：顺丰冷链配送，确保新鲜\n\n物流信息：24小时内发货，预计2\n\n3天到货\n\n推荐菜系:\n\n猪肉卷煎烤：切片后煎至金黄，外脆里嫩。\n\n猪肉卷炖煮：切块后与蔬菜同炖，风味更佳。\n\n价格:\n\n400克：58元/袋\n\n800克：108元/袋\n\n3. 潮汕三宝（酱油、甜醋、虾酱）\n\n产品名称：潮汕三宝\n\n电商网址：shop.example.com/chaoshanthree\n\n产品描述：潮汕三宝包含酱油、甜醋和虾酱。酱油由大豆、麦子自然发酵而成，甜醋以糯米酿制，虾酱选用新鲜海虾发酵，是潮汕菜肴必备调味品。\n\n产品特点:\n\n酱油：大豆、麦子自然发酵，500ml/瓶\n\n甜醋：糯米酿制，500ml/瓶\n\n虾酱：新鲜海虾发酵，200克/瓶\n\n保质期：酱油和甜醋12个月，虾酱6个月\n\n发货方式：顺丰配送，确保完好\n\n物流信息：24小时内发货，预计2\n\n3天到货\n\n推荐菜系:'), Document(metadata={'source': './电商产品数据.txt'}, page_content='口感：鲜嫩多汁，味道浓郁\n\n净重：500克/袋、1000克/袋\n\n保质期：3个月（冷冻保存）\n\n发货方式：顺丰冷链配送，确保新鲜\n\n物流信息：24小时内发货，预计2\n\n3天到货\n\n推荐菜系:\n\n红烧狮子头：加热后直接食用，适合作为主菜。\n\n狮子头炖菜：与蔬菜同炖，味道更佳。\n\n价格:\n\n500克：60元/袋\n\n1000克：110元/袋\n\n10. 潮汕香菇肉酱\n\n产品名称：潮汕香菇肉酱\n\n电商网址：shop.example.com/mushroomsauce\n\n产品描述：潮汕香菇肉酱采用香菇和猪肉为主要原料，加入特制酱料炒制而成，香气扑鼻，味道鲜美。\n\n产品特点:\n\n原材料：香菇、猪肉、酱料\n\n制作工艺：精细切割，炒制均匀\n\n口感：鲜香可口，酱香浓郁\n\n净重：200克/瓶、400克/瓶\n\n保质期：6个月（常温保存）\n\n发货方式：顺丰配送，确保完好\n\n物流信息：24小时内发货，预计2\n\n3天到货\n\n推荐菜系:\n\n拌饭：加入米饭中，提升口感。\n\n拌面：加入面条中，风味独特。\n\n价格:\n\n200克：35元/瓶\n\n400克：65元/瓶'), Document(metadata={'source': './电商产品数据.txt'}, page_content='口感：细腻嫩滑，豆香浓郁\n\n净重：450克/盒\n\n保质期：5天（冷藏保存）\n\n发货方式：顺丰冷链配送，确保新鲜\n\n物流信息：24小时内发货，预计2\n\n3天到货\n\n推荐菜系:\n\n甜食：加糖水、红豆、芝麻食用。\n\n咸食：加入虾米、葱花、酱油食用。\n\n价格：25元/盒\n\n7. 潮汕鱼露\n\n产品名称：潮汕鱼露\n\n电商网址：shop.example.com/fishsauce\n\n产品描述：潮汕鱼露以新鲜小鱼为原料，经过发酵、过滤而成，味道鲜美，是潮汕菜肴必备调味品。\n\n产品特点:\n\n原材料：小鱼、盐\n\n制作工艺：自然发酵，传统工艺\n\n口感：鲜美咸香\n\n净重：500ml/瓶\n\n保质期：12个月\n\n发货方式：顺丰配送，确保完好\n\n物流信息：24小时内发货，预计2\n\n3天到货\n\n推荐菜系:\n\n凉拌菜：作为调味料使用，提升菜肴鲜味。\n\n炒菜：适合炒菜提鲜。\n\n价格：38元/瓶\n\n8. 潮汕糯米肠\n\n产品名称：潮汕糯米肠\n\n电商网址：shop.example.com/glutinousrice')]
4
```

除了使用 `摘要` 来检索全文，多向量检索一般还适用于 `子文档检索父文档` 和 `假设性查询检索`，其中 `假设性查询检索` 是利用 LLM 对切块后的文档生成多个 `假设性标题`，在向量数据库中存储 `假设性标题` 文档块，使用检索到的数据查找 `原始文档`。

核心代码修正如下：

```
from typing import List

import dotenv
from langchain_core.documents import Document
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.pydantic_v1 import BaseModel, Field
```

```
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()


class HypotheticalQuestions(BaseModel):
    """生成假设性问题"""
    questions: List[str] = Field(
        description="假设性问题列表，类型为字符串列表",
    )


# 1.构建一个生成假设性问题的prompt
prompt = ChatPromptTemplate.from_template("生成一个包含3个假设性问题的列表，这些问题可以用于回答下面的文档:\n\n{doc}")

# 2.创建大语言模型，并绑定对应的规范化输出结构
llm = ChatOpenAI(model="gpt-3.5-turbo-16k", temperature=0)
structured_llm = llm.with_structured_output(HypotheticalQuestions)

# 3.创建链应用
chain = (
        {"doc": lambda x: x.page_content}
        | prompt
        | structured_llm
)

hypothetical_questions: HypotheticalQuestions = chain.invoke(
    Document(page_content="我叫慕小课，我喜欢打篮球，游泳")
)
print(hypothetical_questions)
```

输出内容：

```
questions=['如果你不能打篮球，你会选择什么运动？', '如果你不能游泳，你会选择什么运动？', '如果你不能进行任何体育运动，你会选择什么爱好？']
```

接下来针对每个文档生成的 `假设性查询` 创建 `Document列表`，并添加 `doc_id`，添加到向量数据库中，并将 `doc_id` 与原始文档进行绑定，存储到 `文档数据库/字节数据库` 即可。

父文档检索器实现拆分和存储平衡

父文档检索器实现拆分和存储平衡

01. 拆分文档与检索的冲突

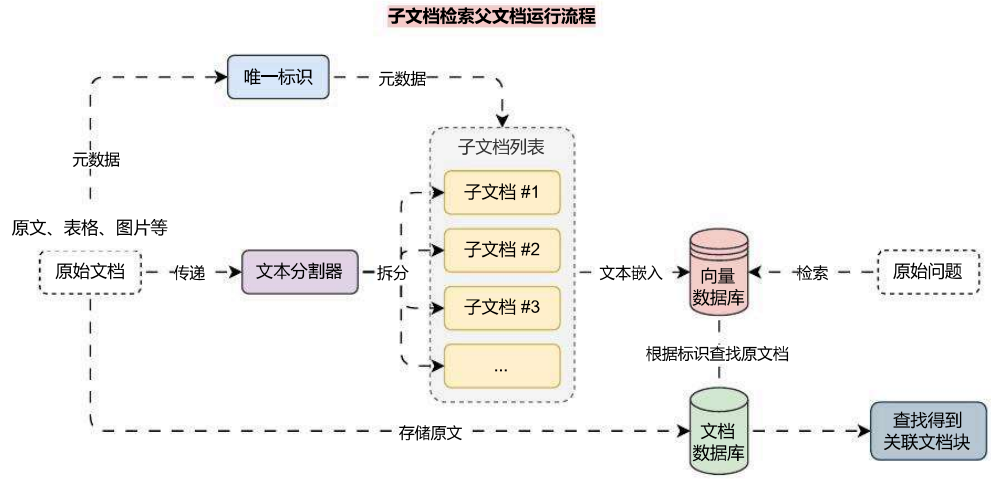
在 RAG 应用开发中，`文档拆分` 和 `文档检索` 通常存在相互冲突的愿望，例如：

1. 我们可能希望拥有小型文档，以便它们的嵌入可以最准确地反映它们的含义，如果太长，嵌入/向量没法记录太多文本特征。
2. 但是又希望文档足够长，这样能保留每个块的上下文。

这个时候就可以考虑通过 `拆分子文档块`，检索 `父文档块` 的策略来实现这种平衡，即在检索中，首先获取小块，然后再根据小块元数据中存储的 `id`，使用 `id` 来查找这些块的父文档，并返回那些更大的文档，该策略适合一些不是特别能拆分的文档，或者是文档上下文关联性很强的场景。

> **Important**
>
> 请注意，这里的“父文档”指的是小块来源的文档，可以是整个原始文档，也可以是切割后比较大的文档块。

`子文档->父文档` 的运行流程也非常简单，其实和 `多向量检索器` 一模一样，如下：



除了使用 `MultiVectorRetriever` 来实现该运行流程，在 LangChain 中，还封装了 `ParentDocumentRetriever`，可以更加便捷地完成该功能，使用技巧也非常简单，传递 `向量数据库`、`文档数据库` 和 `子文档分割器` 即可。

代码示例：

```
import dotenv
import weaviate
from langchain.retrievers import ParentDocumentRetriever
from langchain.storage import LocalFileStore
from langchain_community.document_loaders import UnstructuredFileLoader
from langchain_openai import OpenAIEmbeddings
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_weaviate import WeaviateVectorStore
from weaviate.auth import AuthApiKey

dotenv.load_dotenv()
```

```
# 1.创建加载器与文档列表，并加载文档
loaders = [
    UnstructuredFileLoader("./电商产品数据.txt"),
    UnstructuredFileLoader("./项目API文档.md"),
]
docs = []
for loader in loaders:
    docs.extend(loader.load())

# 2.创建文本分割器
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=500,
    chunk_overlap=50,
)

# 3.创建向量数据库与文档数据库
vector_store = WeaviateVectorStore(
    client=weaviate.connect_to_wcs(
        cluster_url="https://mbakeruerziae6psyex7ng.c0.us-
west3.gcp.weaviate.cloud",
        auth_credentials=AuthApiKey("ZltPVa9ZSOxUcfafelsggGyyH6tnTYQYJvBx"),
    ),
    index_name="ParentDocument",
    text_key="text",
    embedding=OpenAIEmbeddings(model="text-embedding-3-small"),
)
store = LocalFileStore("./parent-document")

# 4.创建父文档检索器
retriever = ParentDocumentRetriever(
    vectorstore=vector_store,
    byte_store=store,
    child_splitter=text_splitter,
)

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

# 6.检索并返回内容
search_docs = retriever.invoke("分享关于LLMOps的一些应用配置")
print(search_docs)
print(len(search_docs))
```

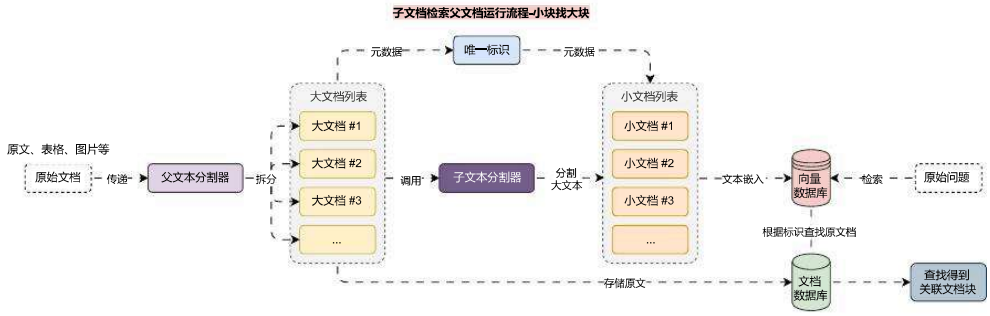
输出内容会返回完整的文档片段，而不是拆分后的片段（但是在向量数据库中存储的是分割后的片段）：

```
[Document(metadata={'source': './项目API文档.md'}, 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\n{\n    "code": "success",\n    "data": {\n        "redirect_url": "https://github.com/login/oauth/authorize?client_id=f69102c6b97d90d69768&redirect_uri=http%3A%2F%2Flocalhost%3A5001%2Foauth%2Fauthorize%2Fgithub&scope=user%3Aemail"\n    },\n    "message":...')]
1
```

02. 父文档检索器检索较大块

在上面的示例中，我们使用拆分的文档块检索数据原文档，但是有时候完整文档可能太大，我们不希望按原样检索它们。在这种情况下，我们真正想要做的是先将原始文档拆分成较大的块（例如 1000-2000 个 Token），然后将其拆分为较小块，接下来索引较小块，但是检索时返回较大块（非原文档）。

运行流程变更如下：



在 `ParentDocumentRetriever` 中，只需要传递多一个 `父文档分割器` 即可，其他流程无需任何变化，更新后的部分代码如下：

```python
parent_splitter = RecursiveCharacterTextSplitter(chunk_size=2000)
child_splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=50)

retriever = ParentDocumentRetriever(
    vectorstore=vector_store,
    byte_store=store,
    parent_splitter=parent_splitter,
    child_splitter=child_splitter,
)
```

递归文档树检索实施高级 RAG 优化理解

递归文档树检索实施高级RAG优化理解

01. RAPTOR 递归文档数策略

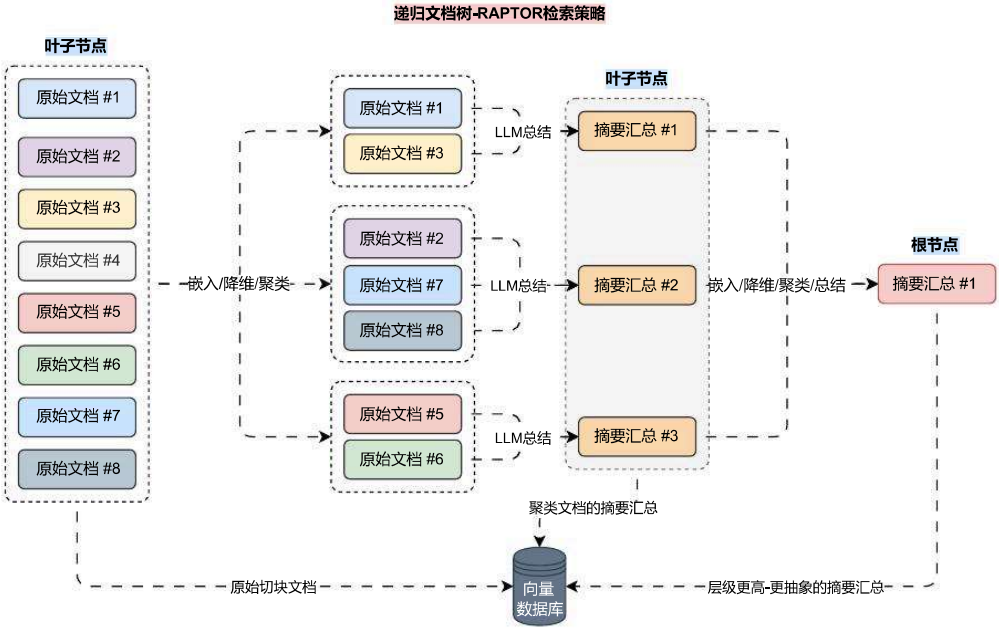
在传统的 RAG 中，我们通常依靠检索短的连续文本块来进行检索。但是，当我们处理的是长上下文时，我们就不能仅仅将文档分块嵌入到其中，或者仅仅使用上下文填充所有文档。相反，我们希望为 LLM 的长下文找到一种好的最小化分块方法，这就是 RAPTOR 的用武之地，在 RAPTOR 中，均衡了`多文档`、`超长上下文`、`高准确性`、`超低成本`等特性。

RAPTOR 其实是一种用树状组织检索的递归抽象处理技术，它采用了一种自下而上的方法，通过对文本片段（块）进行聚类和归纳来形成一种分层结构，该技术在 https://arxiv.org/pdf/2401.18059 论文中被首次提出。

RAPTOR 的运行流程其实很简单，主要步骤如下：

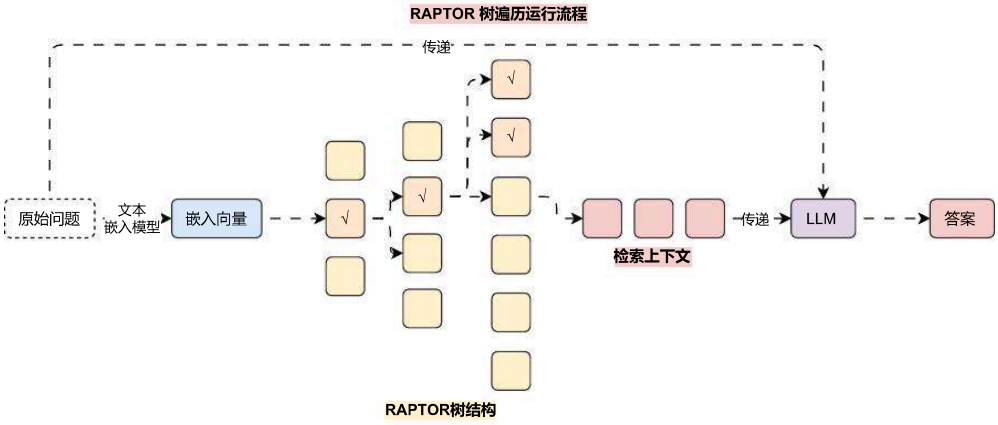
1. 对原始文本进行分块，拆分成合适的大小；
2. 对拆分的文档块进行嵌入/向量化，向量目前处于高维，并将数据存储到向量数据库；
3. 将高维向量进行降维，降低运算成本，例如降低成 2 维或者 3 维；
4. 对降维向量进行聚类，找出同一类的文档组；
5. 合并文档组的文本，使用 LLM 对合并文档进行摘要汇总得到新的文本并文档，重复 ②-⑤ 的步骤；
6. 直到最后只剩下一个`文档`并且该文档的长度符合大小时，结束整个流程；

可视化其运行流程后如下：

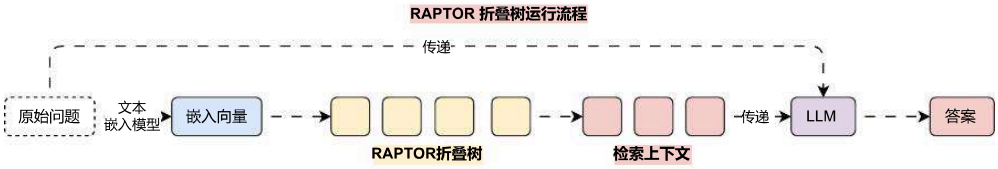


在执行检索的时候，RAPTOR 模型采用了两种主要的检索策略——`树遍历`和`折叠树`，对比两种策略，论文中更推荐`折叠树`检索策略。

`树遍历`方法从树的根节点开始，逐层向下进行检索，在每一层它根据查询向量与节点嵌入的余弦相似性选择最相关的前 k 个节点，然后将这些节点的子节点作为下一层的候选节点集，并重复选择过程，直到达到叶节点，最终将所有选中的节点文本进行拼接，形成检索到的上下文。



`折叠树` 方法会先将整个 RAPTOR 树展平为单个层，即所有节点在同一层级上进行比较，计算查询向量与所有节点嵌入的余弦相似性，并选择最相关的前 k 个节点。



并且因为 `折叠树` 和 `向量数据库基础搜索` 一样都是遍历所有向量（同层），所以只需要将数据存储到向量数据库中，并执行 `相似性搜索` 其实就实现了 `折叠树` 检索策略。

对比其他的 RAG 优化策略，`RAPTOR递归文档树` 不仅保留了原始文档，还将同类文档进行层层抽象（层层总结），而且在 `文本嵌入模型` 参数较小的情况下，一般也能实现不错的效果（低成本、高性能），不过因为该数据存储的是 `树状结构`，每次在更新/新增数据时，操作对比其他优化策略麻烦很多。

该论文并没有提供更新/新增数据的策略，但是对于树状结构，一般的更新策略有：

1. `重新构建树结构`：新增了大量文档时，可以重新构建整个 RAPTOR 树，确保数据不会遗漏。
2. `增量更新`：如果新增的文档不是很多，可以考虑只更新部分树结构，将新文档的文本块作为新的叶子节点添加到树中，然后根据这些新节点的内容，调整或更新其上层节点的摘要信息。
3. `利用现有节点`：如果新文档与现有树中的某些节点内容相似，可以考虑将新文档的文本块与现有节点合并，然后重新生成这些节点的摘要，以此来更新树结构。
4. `层次化更新`：根据新文档的内容和重要性，选择在树的哪个层次上进行更新。如果新文档提供了对整个文档集的重要概述，可能需要在较高的层次上更新；如果新文档提供了细节信息，可能只需要在较低层次上更新。

02. RAPTOR 递归文档树的视线

在 LangChain 中并没有针对 RAPTOR 的实现，并且在该策略中涵盖了 `高维数据降维`、`低维数据聚类`、`递归检索` 等内容，对于该检索策略，目前使用得较少（更新与新增缺陷导致外挂文档增加时复杂度成倍递增），所以仅需要了解 `RAPTOR` 的运行流程即可。

首先安装特定的第三方 Python 包：

```
```

实现代码如下：

```
from typing import Optional

import dotenv
import numpy as np
import pandas
import pandas as pd
import umap
import weaviate
from langchain_community.document_loaders import UnstructuredFileLoader
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_huggingface import HuggingFaceEmbeddings
from langchain_openai import ChatOpenAI
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_weaviate import WeaviateVectorStore
from sklearn.mixture import GaussianMixture
from weaviate.auth import AuthApiKey

dotenv.load_dotenv()

# 1.定义随机数种子、文本嵌入模型、大语言模型、向量数据库
RANDOM_SEED = 224
embd = HuggingFaceEmbeddings(
    model_name="thenlper/gte-small",
    cache_folder="./embeddings/",
    encode_kwargs={"normalize_embeddings": True},
)
model = ChatOpenAI(model="gpt-3.5-turbo-16k", temperature=0)
db = WeaviateVectorStore(
    client=weaviate.connect_to_wcs(
        cluster_url="https://mbakeruerziae6psyex7ng.c0.us-
west3.gcp.weaviate.cloud",
        auth_credentials=AuthApiKey("ZltPVa9ZSOxUcfafelsggGyyH6tnTYQYJvBx"),
    ),
    index_name="RaptorRAG",
    text_key="text",
    embedding=embd,
)


def global_cluster_embeddings(
        embeddings: np.ndarray, dim: int, n_neighbors: Optional[int] = None,
metric: str = "cosine",
) -> np.ndarray:
    """
    使用UMAP对传递嵌入向量进行全局降维

    :param embeddings: 需要降维的嵌入向量
    :param dim: 降低后的维度
    :param n_neighbors: 每个向量需要考虑的邻居数量，如果没有提供默认为嵌入数量的开方
    :param metric: 用于UMAP的距离度量，默认为余弦相似性
    :return: 一个降维到指定维度的numpy嵌入数组
    """
```

```
    if n_neighbors is None:
        n_neighbors = int((len(embeddings) - 1) ** 0.5)
    return umap.UMAP(n_neighbors=n_neighbors, n_components=dim,
metric=metric).fit_transform(embeddings)


def local_cluster_embeddings(
        embeddings: np.ndarray, dim: int, n_neighbors: int = 10, metric: str =
"cosine",
) -> np.ndarray:
    """
    使用UMAP对嵌入进行局部降维处理，通常在全局聚类之后进行。

    :param embeddings: 需要降维的嵌入向量
    :param dim: 降低后的维度
    :param n_neighbors: 每个向量需要考虑的邻居数量
    :param metric: 用于UMAP的距离度量，默认为余弦相似性
    :return: 一个降维到指定维度的numpy嵌入数组
    """
    return umap.UMAP(
        n_neighbors=n_neighbors, n_components=dim, metric=metric,
    ).fit_transform(embeddings)


def get_optimal_clusters(
        embeddings: np.ndarray, max_clusters: int = 50, random_state: int =
RANDOM_SEED,
) -> int:
    """
    使用高斯混合模型结合贝叶斯信息准则（BIC）确定最佳的聚类数目。

    :param embeddings: 需要聚类的嵌入向量
    :param max_clusters: 最大聚类数
    :param random_state: 随机数
    :return: 返回最优聚类数
    """
    # 1.获取最大聚类树，最大聚类数不能超过嵌入向量的数量
    max_clusters = min(max_clusters, len(embeddings))
    n_clusters = np.arange(1, max_clusters)

    # 2.逐个设置聚类树并找出最优聚类数
    bics = []
    for n in n_clusters:
        # 3.创建高斯混合模型，并计算聚类结果
        gm = GaussianMixture(n_components=n, random_state=random_state)
        gm.fit(embeddings)
        bics.append(gm.bic(embeddings))

    return n_clusters[np.argmin(bics)]


def gmm_cluster(embeddings: np.ndarray, threshold: float, random_state: int = 0)
-> tuple[list, int]:
    """
    使用基于概率阈值的高斯混合模型（GMM）对嵌入进行聚类。
```

```
    :param embeddings: 需要聚类的嵌入向量（降维）
    :param threshold: 概率阈值
    :param random_state: 用于可重现的随机性种子
    :return: 包含聚类标签和确定聚类数目的元组
    """
    # 1.获取最优聚类数
    n_clusters = get_optimal_clusters(embeddings)

    # 2.创建高斯混合模型对象并嵌入数据
    gm = GaussianMixture(n_components=n_clusters, random_state=random_state)
    gm.fit(embeddings)

    # 3.预测每个样本属于各个聚类的概率
    probs = gm.predict_proba(embeddings)

    # 4.根据概率阈值确定每个嵌入的聚类标签
    labels = [np.where(prob > threshold)[0] for prob in probs]

    # 5.返回聚类标签和聚类数目
    return labels, n_clusters


def perform_clustering(embeddings: np.ndarray, dim: int, threshold: float) ->
list[np.ndarray]:
    """
    对嵌入进行聚类，首先全局降维，然后使用高斯混合模型进行聚类，最后在每个全局聚类中进行局部聚
类。

    :param embeddings: 需要执行操作的嵌入向量列表
    :param dim: 指定的降维维度
    :param threshold: 概率阈值
    :return: 包含每个嵌入的聚类ID的列表，每个数组代表一个嵌入的聚类标签。
    """
    # 1.检测传入的嵌入向量，当数据量不足时不进行聚类
    if len(embeddings) <= dim + 1:
        return [np.array([0]) for _ in range(len(embeddings))]

    # 2.调用函数进行全局降维
    reduced_embeddings_global = global_cluster_embeddings(embeddings, dim)

    # 3.对降维后的数据进行全局聚类
    global_clusters, n_global_clusters = gmm_cluster(reduced_embeddings_global,
threshold)

    # 4.初始化一个空列表，用于存储所有嵌入的局部聚类标签
    all_local_clusters = [np.array([]) for _ in range(len(embeddings))]
    total_clusters = 0

    # 5.遍历每个全局聚类以执行局部聚类
    for i in range(n_global_clusters):
        # 6.提取属于当前全局聚类的嵌入向量
        global_cluster_embeddings_ = embeddings[
            np.array([i in gc for gc in global_clusters])
        ]

        # 7.如果当前全局聚类中没有嵌入向量则跳过循环
```

```
        if len(global_cluster_embeddings_) == 0:
            continue

        # 8.如果当前全局聚类中的嵌入量很少，直接将它们分配到一个聚类中
        if len(global_cluster_embeddings_) <= dim + 1:
            local_clusters = [np.array([0]) for _ in global_cluster_embeddings_]
            n_local_clusters = 1
        else:
            # 9.执行局部降维和聚类
            reduced_embeddings_local =
local_cluster_embeddings(global_cluster_embeddings_, dim)
            local_clusters, n_local_clusters =
gmm_cluster(reduced_embeddings_local, threshold)

        # 10.分配局部聚类ID，调整已处理的总聚类数目
        for j in range(n_local_clusters):
            local_cluster_embeddings_ = global_cluster_embeddings_[
                np.array([j in lc for lc in local_clusters])
            ]
            indices = np.where(
                (embeddings == local_cluster_embeddings_[:, None]).all(-1)
            )[1]
            for idx in indices:
                all_local_clusters[idx] = np.append(all_local_clusters[idx], j +
total_clusters)

        total_clusters += n_local_clusters

    return all_local_clusters


def embed(texts: list[str]) -> np.ndarray:
    """
    将传递的的文本列表转换成嵌入向量列表

    :param texts: 需要转换的文本列表
    :return: 生成的嵌入向量列表并转换成numpy数组
    """
    text_embeddings = embd.embed_documents(texts)
    return np.array(text_embeddings)


def embed_cluster_texts(texts: list[str]) -> pandas.DataFrame:
    """
    对文本列表进行嵌入和聚类,并返回一个包含文本、嵌入和聚类标签的数据框。
    该函数将嵌入生成和聚类结合成一个步骤。

    :param texts: 需要处理的文本列表
    :return: 返回包含文本、嵌入和聚类标签的数据框
    """
    text_embeddings_np = embed(texts)
    cluster_labels = perform_clustering(text_embeddings_np, 10, 0.1)
    df = pd.DataFrame()
    df["text"] = texts
    df["embd"] = list(text_embeddings_np)
    df["cluster"] = cluster_labels
```

```
    return df


def fmt_txt(df: pd.DataFrame) -> str:
    """
    将数据框中的文本格式化成单个字符串

    :param df: 需要处理的数据框，内部涵盖text、embd、cluster三个字段
    :return: 返回合并格式化后的字符串
    """
    unique_txt = df["text"].tolist()
    return "--- --- \n --- ---".join(unique_txt)


def embed_cluster_summarize_texts(texts: list[str], level: int) ->
tuple[pd.DataFrame, pd.DataFrame]:
    """
    对传入的文本列表进行嵌入、聚类和总结。
    该函数首先问文本生成嵌入，基于相似性对他们进行聚类，扩展聚类分配以便处理，然后总结每个聚类中
的内容。

    :param texts: 需要处理的文本列表
    :param level: 一个整数，可以定义处理的深度
    :return: 包含两个数据框的元组
    - 第一个 DataFrame (df_clusters) 包括原始文本、它们的嵌入以及聚类分配。
    - 第二个 DataFrame (df_summary) 包含每个聚类的摘要信息、指定的处理级别以及聚类标识符。
    """
    # 1.嵌入和聚类文本，生成包含text、embd、cluster的数据框
    df_clusters = embed_cluster_texts(texts)

    # 2.定义变量，用于扩展数据框，以便更方便地操作聚类
    expanded_list = []

    # 3.扩展数据框条目，将文档和聚类配对，便于处理
    for index, row in df_clusters.iterrows():
        for cluster in row["cluster"]:
            expanded_list.append(
                {"text": row["text"], "embd": row["embd"], "cluster": cluster}
            )

    # 4.从扩展列表创建一个新的数据框
    expanded_df = pd.DataFrame(expanded_list)

    # 5.获取唯一的聚类标识符以进行处理
    all_clusters = expanded_df["cluster"].unique()

    # 6.创建汇总Prompt、汇总链
    template = """Here is a sub-set of LangChain Expression Language doc.

    LangChain Expression Language provides a way to compose chain in LangChain.

    Give a detailed summary of the documentation provided.

    Documentation:
    {context}
    """
```

```
    prompt = ChatPromptTemplate.from_template(template)
    chain = prompt | model | StrOutputParser()

    # 7.格式化每个聚类中的文本以进行总结
    summaries = []
    for i in all_clusters:
        df_cluster = expanded_df[expanded_df["cluster"] == i]
        formatted_txt = fmt_txt(df_cluster)
        summaries.append(chain.invoke({"context": formatted_txt}))

    # 8.创建一个DataFrame来存储总结及其对应的聚类和级别
    df_summary = pd.DataFrame(
        {
            "summaries": summaries,
            "level": [level] * len(summaries),
            "cluster": list(all_clusters),
        }
    )

    return df_clusters, df_summary


def recursive_embed_cluster_summarize(
        texts: list[str], level: int = 1, n_levels: int = 3,
) -> dict[int, tuple[pd.DataFrame, pd.DataFrame]]:
    """
    递归地嵌入、聚类和总结文本，直到达到指定的级别或唯一聚类数变为1，将结果存储在每个级别处。

    :param texts: 要处理的文本列表
    :param level: 当前递归级别（从1开始）
    :param n_levels: 递归地最大深度（默认为3）
    :return: 一个字典，其中键是递归级别，值是包含该级别处聚类DataFrame和总结DataFrame的元
组。
    """
    # 1.定义字典用于存储每个级别处的结果
    results = {}

    # 2.对当前级别执行嵌入、聚类和总结
    df_clusters, df_summary = embed_cluster_summarize_texts(texts, level)

    # 3.存储当前级别的结果
    results[level] = (df_clusters, df_summary)

    # 4.确定是否可以继续递归并且有意义
    unique_clusters = df_summary["cluster"].nunique()
    if level < n_levels and unique_clusters > 1:
        # 5.使用总结作为下一级递归的输入文本
        new_texts = df_summary["summaries"].tolist()
        next_level_results = recursive_embed_cluster_summarize(
            new_texts, level + 1, n_levels
        )

        # 6.将下一级的结果合并到当前结果字典中
        results.update(next_level_results)

    return results
```

```
# 2.定义文档加载器、文本分割器(中英文场景)
loaders = [
    UnstructuredFileLoader("./流浪地球.txt"),
    UnstructuredFileLoader("./电商产品数据.txt"),
    UnstructuredFileLoader("./项目API文档.md"),
]
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=500,
    chunk_overlap=0,
    separators=["\n\n", "\n", "。|！|？", "\.\s|\!\s|\?\s", "；|;\s", "，|,\s", "
", ""],
    is_separator_regex=True,
)

# 3.循环分割并加载文本
docs = []
for loader in loaders:
    docs.extend(loader.load_and_split(text_splitter))

# 4.构建文档树，最多3层
leaf_texts = [doc.page_content for doc in docs]
results = recursive_embed_cluster_summarize(leaf_texts, level=1, n_levels=3)

# 5.遍历文档树结果，从每个级别提取总结并将它们添加到all_texts中
all_texts = leaf_texts.copy()
for level in sorted(results.keys()):
    summaries = results[level][1]["summaries"].tolist()
    all_texts.extend(summaries)

# 6.将all_texts添加到向量数据库
db.add_texts(all_texts)

# 7.执行相似性检索（折叠树）
retriever = db.as_retriever(search_type="mmr")
search_docs = retriever.invoke("流浪地球中的人类花了多长时间才流浪到新的恒星系？")

print(search_docs)
print(len(search_docs))
```

输出内容:

```
[Document(page_content='我们的船继续航行，到了地球黑夜的部分，在这里，阳光和地球发动机的光柱都照不到，在大西洋清凉的海风中，我们这些孩子第一次看到了星空。天啊，那是怎样的景象啊，美得让我们心醉。小星老师一手搂着我们，一手指着星空，看，孩子们，那就是半人马座，那就是比邻星，那就是我们的新家！说完她哭了起来，我们也都跟着哭了，周围的水手和船长，这些铁打的汉子也流下了眼泪。所有的人都用泪眼探望着老师指的方向，星空在泪水中扭曲抖动，唯有那个星星是不动的，那是黑夜大海狂浪中远方陆地的灯塔，那是冰雪荒原中快要冻死的孤独旅人前方隐现的火光，那是我们心中的太阳，是人类在未来一百代人的苦海中惟一的希望和支撑……\n\n在回家的航程中，我们看到了启航的第一个信号：夜空中出现了一个巨大的彗星，那是月球。人类带不走月球，就在月球上也安装了行星发动机，把它推离地球轨道，以免在地球加速时相撞。月球上行星发动机产生的巨大彗尾使大海笼罩在一片蓝光之中，群星看不见了。月球移动产生的引力潮汐使大海巨浪冲天，我们改乘飞机向南半球的家飞去。\n\n启航的日子终于到了！'), Document(page_content='离开木星时，地球已达到了逃逸速度，它不再需要返回潜藏着死亡的太阳，向广漠的外太空飞去，漫长的流浪时代开始了。\n\n就在木星暗红色的阴影下，我的儿子在地层深处出生了。\n\n第3章 叛乱\n\n离开木星后，亚洲大陆上一万多台地球发动机再次全功率开动，这一次它们要不停地运行500年，不停地加速地球。这500年中，发动机将把亚洲大陆上一半的山脉用做燃料消耗掉。\n\n从四个多世纪死亡的恐惧中解脱出来，人们长出了一口气。但预料中的狂欢并没有出现，接下来发生的事情出乎所有人的想像。\n\n在地下城的庆祝集会后，我一个人穿上密封服来到地面。童年时熟悉的群山已被超级挖掘机夷为平地，大地上只有裸露的岩石和坚硬的冻土，冻土上到处有白色的斑块，那是大海潮留下的盐渍。面前那座爷爷和爸爸度过了一生的曾有千万人口的大城市现在已是一片废墟，高楼钢筋外露的残骸在地球发动机光柱的蓝光中拖着长长的影子，好像是史前巨兽的化石……一次次的洪水和小行星的撞击已摧毁了地面上的一切，各大陆上的城市和植被都荡然无存，地球表面已变成火星一样的荒漠。'), Document(page_content='“可老师，我们来不及的，地球来不及的，它还来不及加速到足够快，航行足够远，太阳就爆炸了！”\n\n“时间是够的，要相信联合政府！这我说了多少遍，如果你们还不相信，我们就退一万步说：人类将自豪地去死，因为我们尽了最大的努力！”\n\n人类的逃亡分为五步：第一步，用地球发动机使地球停止转动，使发动机喷口固定在地球运行的反方向；第二步，全功率开动地球发动机，使地球加速到逃逸速度，飞出太阳系；第三步，在外太空继续加速，飞向比邻星；第四步，在中途使地球重新自转，掉转发动机方向，开始减速；第五步，地球泊入比邻星轨道，成为这颗恒星的卫星。人们把这五步分别称为刹车时代、逃逸时代、流浪时代I（加速）、流浪时代II（减速）、新太阳时代。\n\n整个移民过程将延续两千五百年时间，一百代人。'), Document(page_content='我们很快到达了海边，看到城市摩天大楼的尖顶伸出海面，退潮时白花花的海水从大楼无数的窗子中流出，形成一道道瀑布……刹车时代刚刚结束，其对地球的影响已触目惊心：地球发动机加速造成的潮汐吞没了北半球三分之二的大城市，发动机带来的全球高温融化了极地冰川，更给这大洪水推波助澜，波及到南半球。爷爷在三十年前亲眼目睹了百米高的巨浪吞没上海的情景，他现在讲这事的时候眼还直勾勾的。事实上，我们的星球还没启程就已面目全非了，谁知道在以后漫长的外太空流浪中，还有多少苦难在等着我们呢？')]
4
```

感兴趣的同学也可以阅读代码，在代码中，每一行我都添加上了对应的注释。

除了 RAPTOR 的优化思想，LangChain 团队在路演中，还演示了一种基于 `Token2Text` 的优化策略——ColBERT，该策略会为段落中的每个 Token 生成一个受上下文影响的向量，在计算文档得分时并不是使用文档的向量相似性计算，而是通过 `查询嵌入` 与 `每个Token嵌入向量` 的最大相似性之和，**不过由于性能偏低，并且需要高昂的计算成本，带来的提升不大，导致目前应用并不广，接受度较低。**

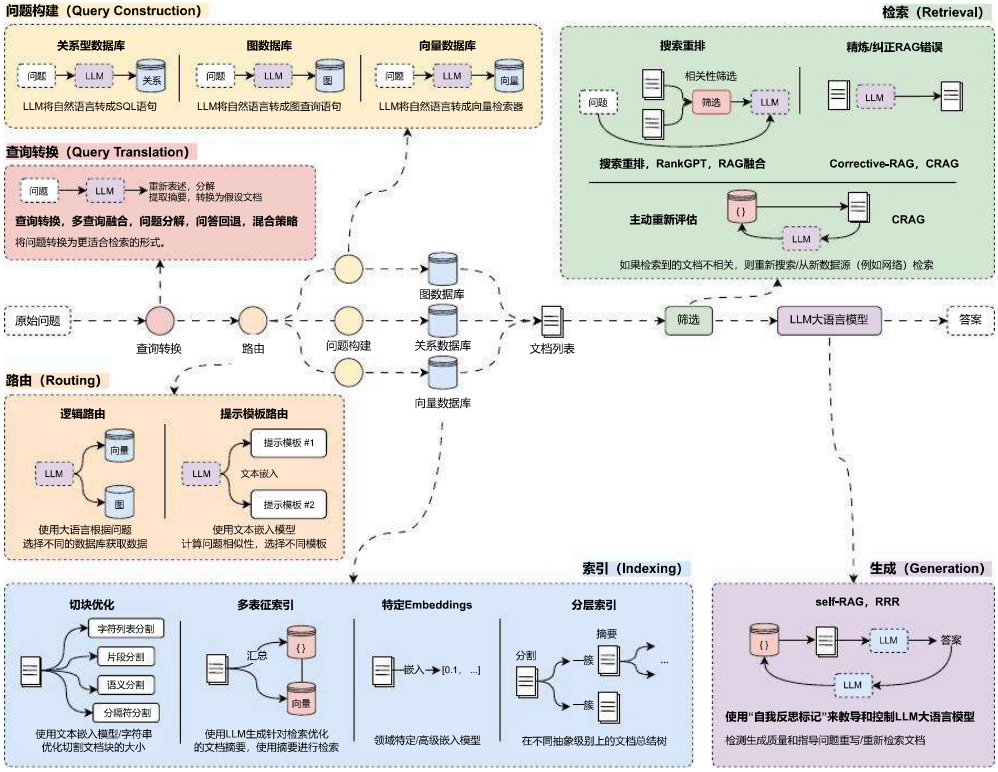
感兴趣的同学可以通过以下资料进行阅读学习：

1. LangChain 文档：https://python.langchain.com/v0.2/docs/integrations/retrievers/ragatouille/
2. ColBERT 英文博文：https://hackernoon.com/how-colbert-helps-developers-overcome-the-limits-of-rag

03. 索引阶段优化策略总结

至此，在 RAG 的 `索引阶段`，这些具有代表性的优化策略其实我们已经讲解完了，涵盖了 `分割策略(语义分割、递归字符分割)`、`多向量/表征检索`、`RAPTOR递归文档树检索`，其中 `特定Embeddings` 即考虑使用维度更高的文本嵌入模型，并没有复杂的使用技巧。

目前在`索引阶段`，使用最多的优化策略还是在`文档加载/分割`与`特定Embeddings`上，因为无论是`多向量/表征索引`还是`RAPTOR递归文档树`，都会使用额外的 LLM 对文档进行相应的总结、信息提取、分类汇总等，**数据越多，信息失真的概率会越大**（所以谨慎使用）。



ReRank 重排序提升 RAG 系统效果

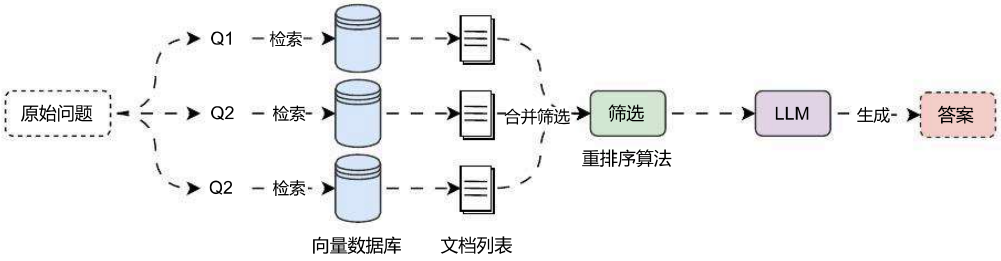
ReRank重排序提升RAG系统效果

01. ReRank 重排序

在完成对问题的改写、不同数据库查询的构建以及路由逻辑、向量数据库索引方面的优化后，我们可以考虑进一步优化`检索阶段`，一般涵盖了`重排序`、`纠正性RAG`两种策略，其中`重排序`是使用频率最高，性价比最高，通常与`混合检索`一起搭配使用，也是目前主流的优化策略（Dify、Coze、智谱、绝大部分开源 Agent 项目都在使用）。

`重排序`的核心思想见字知其意，即对检索到的文档`调整顺序`，除此之外，`重排序`一般还会增加`剔除无关/多余数据`的步骤，在前面的课时中，我们学习的`RRF`算法其实就是重排序中最基础一种。

运行流程如下：



并且`重排序`的逻辑是输入`文档列表`，输出的仍然是`文档列表`，和`DocumentTransformer`类似，不过在 LangChain 中`重排序`是一个`DocumentCompressor`组件（压缩组件），如果需要查找`重排序`组件，可以在`文档转换/检索器集成`列表中查找，一些高频使用的组件还进行了单独的封装，例如：`LongContextReorder`、`Cohere Reranker`等。

在 LangChain 中，使用重排工具很简单，可以单独使用，也可以利用`ContextualCompressionRetriever`检索器进行二次包装合并，并传递检索器+重排工具，这样检索输出得到的结果就是经过重排的了。

02. Cohere 重排序

目前可用的重新排序模型并不多，一种是选择`Cohere`提供的在线模型，可以通过 API 访问，此外还有一些开源模型，如：`bge-rerank-base`和`bge-rerank-large`等，体验与评分最好的是`Cohere`的在线模型，不过它是一项付费服务（注册账号提供免费额度），并且由于 Cohere 服务部署在海外，国内访问速度并没有这么快。

对于学习，我们可以使用`Cohere`的在线模型，如果是部署企业内部的服务，可以使用`bge-rerank-large`，链接如下：

1. Cohere 在线模型：https://cohere.com/
2. bge-rerank-large 开源模型：https://huggingface.co/BAAI/bge-reranker-large

安装依赖：

```
pip install langchain-cohere
```

在 weaviate 向量数据库中，使用`Cohere`在线模型集成重排序示例代码（使用 weaviate 数据库的 DatasetDemo 集合，存储了`项目API文档.md`文档的内容，示例如下：

```
import dotenv
```

```python
import weaviate
from langchain.retrievers import ContextualCompressionRetriever
from langchain_cohere import CohereRerank
from langchain_openai import OpenAIEmbeddings
from langchain_weaviate import WeaviateVectorStore
from weaviate.auth import AuthApiKey

dotenv.load_dotenv()

# 1.创建向量数据库与重排组件
embedding = OpenAIEmbeddings(model="text-embedding-3-small")
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=embedding,
)
rerank = CohereRerank(model="rerank-multilingual-v3.0")

# 2.构建压缩检索器
retriever = ContextualCompressionRetriever(
    base_retriever=db.as_retriever(),
    base_compressor=rerank,
)

# 3.执行搜索并排序
search_docs = retriever.invoke("关于LLMOps应用配置的信息有哪些呢？")
print(search_docs)
print(len(search_docs))
```

输出内容:

```
[Document(metadata={'source': './项目API文档.md', 'start_index': 2324.0,
'relevance_score': 0.9298237}, 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': 0.0, 'relevance_score': 0.7358315}, 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': 675.0,
'relevance_score': 0.098772585}, 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 } }')]
3
```

通过 LangSmith 观察该检索器的运行流程，可以发现，原始检索器找到了 4 条数据，但是经过重排序后，只返回了相关性高的 3 条（可设置），有一条被剔除了，这也是 `压缩器` 和 `文档转换器` 的区别（不过在旧版本的 LangChain 中，两个没有区别，所以在很多文档转换器的文档中都可以看到压缩器的存在）。

纠正性索引增强生成 CRAG 优化策略

纠正性索引增强生成CRAG优化策略

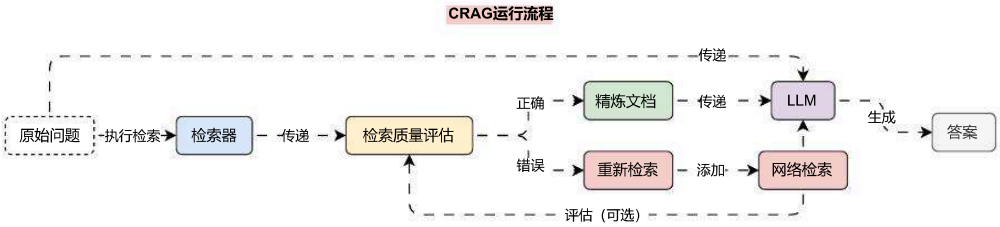
01. 纠正性检索增强生成简介

纠正性检索增强生成（Corrective Retrieval-Augmented Generation，CRAG）是一种先进的自然语言处理技术，旨在提高检索的生成方法的鲁棒性和准确性。在 CRAG 中**引入了一个轻量级的检索评估器来评估检索到的文档的质量，并根据评估结果触发不同的知识检索动作**，以确保生成结果的准确性。

CRAG 的工作流程包含了以下几个步骤：

1. **检索文档**：首先，基于用户的查询，系统执行检索操作以获取相关的文档或信息。
2. **评估检索质量**：CRAG 使用一个轻量级的检索评估器对检索到的每个文档进行质量评估，计算出一个量化的置信度分数。
3. **触发知识检索动作**：根据置信度分数，CRAG 将触发以下三个动作之一：
    - **正确**：如果评估器认为文档与查询高度相关，将采用该文档进行知识精炼。
    - **错误**：如果文档被评估为不相关或误导性，CRAG将利用网络搜索寻找更多知识来源。
4. **知识精炼**：对于评估为正确的文档，CRAG将进行知识精炼，抽取关键信息并过滤掉无关信息。
5. **网络搜索**：在需要时，CRAG会执行网络搜索以寻找更多高质量的知识来源，以纠正或补充检索结果。
6. **分解-重组**：CRAG采用一种分解-重组算法，将检索到的文档解构为关键信息块，筛选重要信息，并重新组织成结构化知识。
7. **生成文本**：最后，利用经过优化和校正的知识，传递给 LLM，生成对应文本。

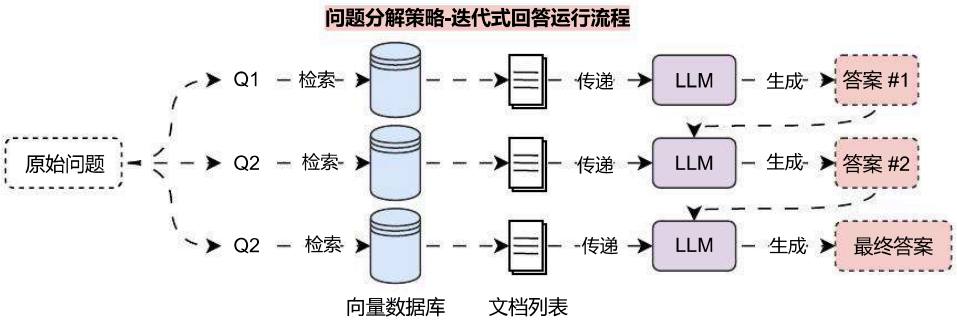
运行流程如下：



02. 组件构成与 LCEL 表达式缺陷

在上述的 CRAG 运行流程中，需要的组件涵盖了：`检索器`、`评估组件`、`判断组件(路由组件)`、`重检索`、`网络搜索工具` 等，并且该链的运行并不是线性运行的，而是存在 `条件判断` 与 `循环(可选)` 迭代多步的可能，对于 LCEL 表达式构建的单链应用来说，要实现这个功能难度非常大，亦或者说代码会非常臃肿。

这就是 LCEL 表达式构建链应用的缺陷，其实在 `问题分解策略` 时，就已经可以感受到，例如 `问题分解策略-迭代式回答运行流程` 中，每一次子问题生成的回复要传递给下一次子问题作为上下文，如下：



该功能本身是 `链应用` 的一部分，但是在代码中，我们只能通过循环来完成该过程，核心代码如下：

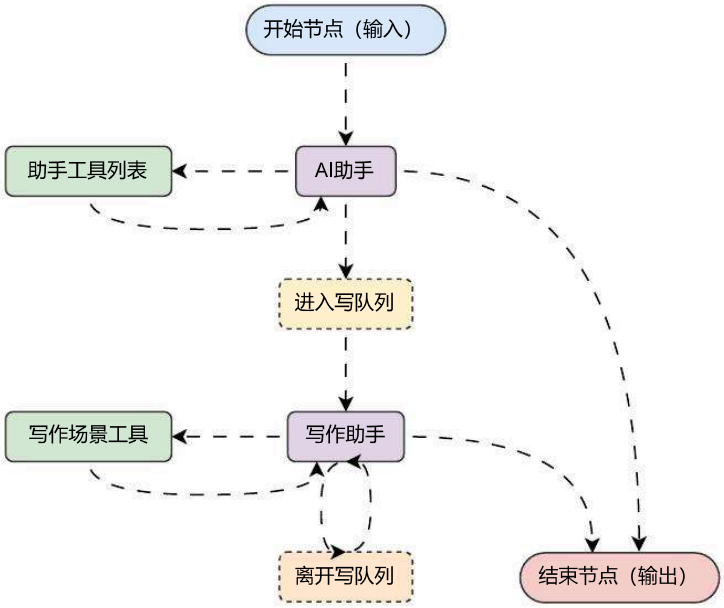
```
qa_pairs = ""
for sub_question in sub_questions:
    answer = chain.invoke({"question": sub_question, "qa_pairs": qa_pairs})
    qa_pair = format_qa_pair(sub_question, answer)
    qa_pairs += "\n---\n" + qa_pair
    print(f"问题: {sub_question}")
    print(f"答案: {answer}")
```

而且涉及到工具类的调用，也没法在 LCEL 表达式构建的链应用中判断是否需要执行工具，并自动将工具的处理结果返回给 LLM（无法回退，只能从前往后流动），对于这类涵盖了**单代理、多代理、多LLM集成、多智能体、分层、顺序、循环**等复杂场景的应用，一般的 LCEL 表达式就无法实现了，这个时候可以考虑使用 LangGraph 来构建 `循环图结构` 类型的应用结构。

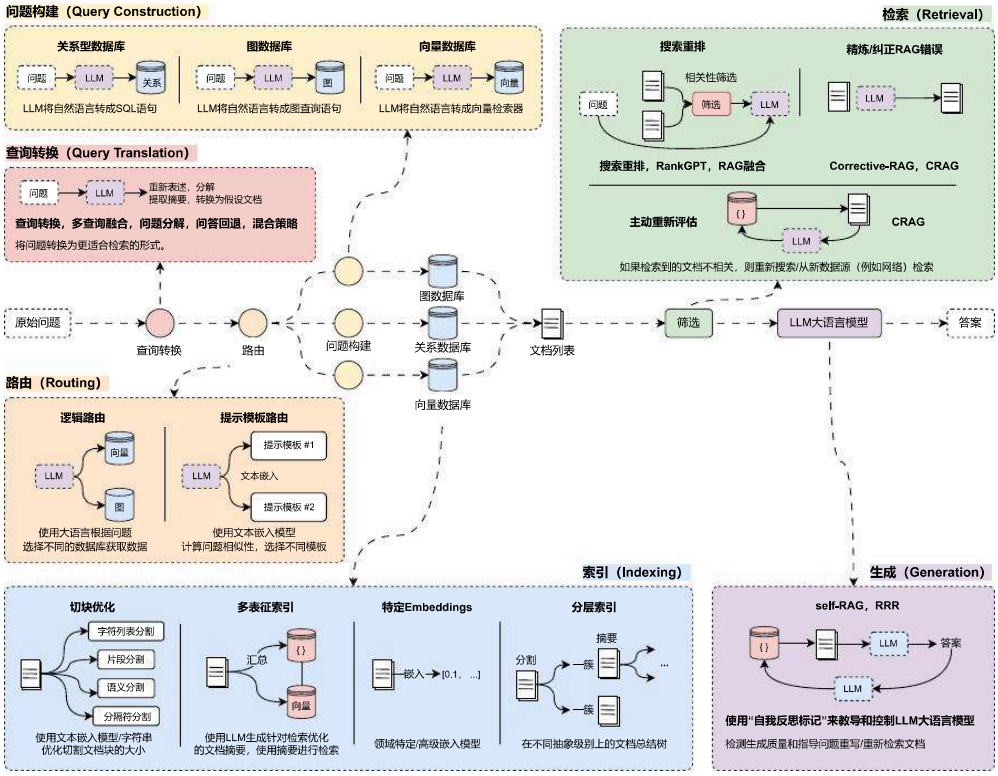
LangGraph 官网：https://www.langchain.com/langgraph。

LangGraph 是 LCEL 表达式的扩充/超集，旨在克服传统 LangChain 链条运行时无法循环的限制，LangChain 以 `循环图(一个数学概念)` 作为 Agent 应用的框架基础，对比单条链顺序执行，在 `图结构` 中可以轻松引入循环。

例如下方就是一张 LangGraph 循环图示例（有向无环/有环图）：



最后，CRAG 这个优化策略会涉及到 `循环调用` 与 `工具回调` 两个我们目前还没有学习到的 LangChain 组件，在下一章我们讲解并深入学习 LangGraph 时，再回过头来一起实现。



使用 Self-RAG 纠正低质量的检索生成

使用Self-RAG纠正低质量的检索生成

01. 开卷考试与 Self-RAG

这里我们从一个常见的生活场景入手——参加开卷考试，一般来说我们通常会采用以下两种作答策略：

- `方法一`：对于熟悉的题目，直接快速作答；对于不熟悉的题目，快速翻阅参考书，找到相关部分，在脑海中整理分类和总结后，再在试卷上作答。
- `方法二`：每一个题目都需要参考书本进行解答。先找到相关不分，在脑海中进行整合和总结后，再到试卷上书写答案。

显然，`方法一` 大家用的更多一些，是首选方法。方法二不仅耗时，还有可能引入无关的或错误的信息，导致出现混淆和错误，甚至在考生原本擅长的领域也不例外。

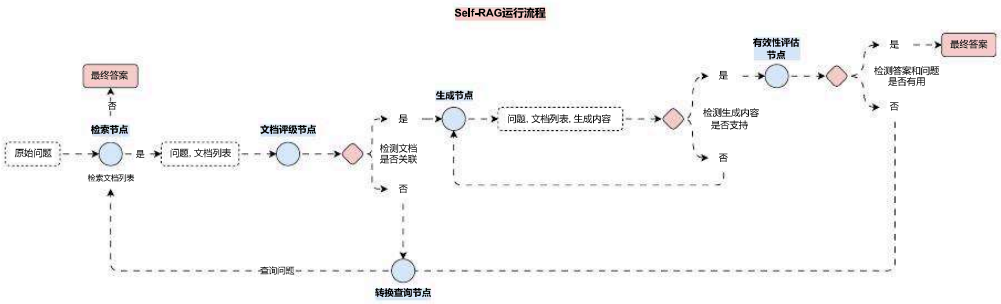
映射到 RAG 应用开发中，很容易可以发现，`方法二` 是典型的 `RAG`，即（检索->整合->生成）流程，而方法一就是 `Self-RAG` 的流程。

`Self-RAG` 全程为自我反思 RAG，见名知其意，即对原始查询、检索的内容、生成的内容进行自我反思，根据反思的结果执行不同的操作，例如：**直接输出答案、重新检索、剔除不相关的内容、检测生成内容是否存在幻觉、检测生成内容是否有帮助**等，可以把 `Self-RAG` 看成是一个拥有自我反思能力的智能体，这个智能体主要用来依据相关知识库回复用户问题，自我迭代，直到输出满意的结果。

一个 `Self-RAG` 应用主要有三大步骤组成：

1. `按需检索（Retrieval as Needed）`：使用需要检索时，例如查询“慕小课是谁时？”，模型会输出一个 `检索query`，表示需要检索与 `query` 相关的内容；相反，当模型被要求写“写一篇关于Python依赖注入的文章”时，大模型会直接生成答案，无需进行检索。
2. `以并行方式生成内容（Parallel Generation）`：模型会同时使用 `prompt` 和检索到的内容来生成模型输出，在整个过程中，会触发多种类型的反思（Reflection），涵盖了：反思文档是否有关联、反思生成内容是否存在幻觉，如果不关联则重新检索，如果存在幻觉/支持度不够，则重新生成。
3. `内容的评估和选择`：对步骤 2 中生成的内容进行评估，并选择最佳文档段落作为输出。

拆分成流程图后，`Self-RAG` 的运行流程如下：

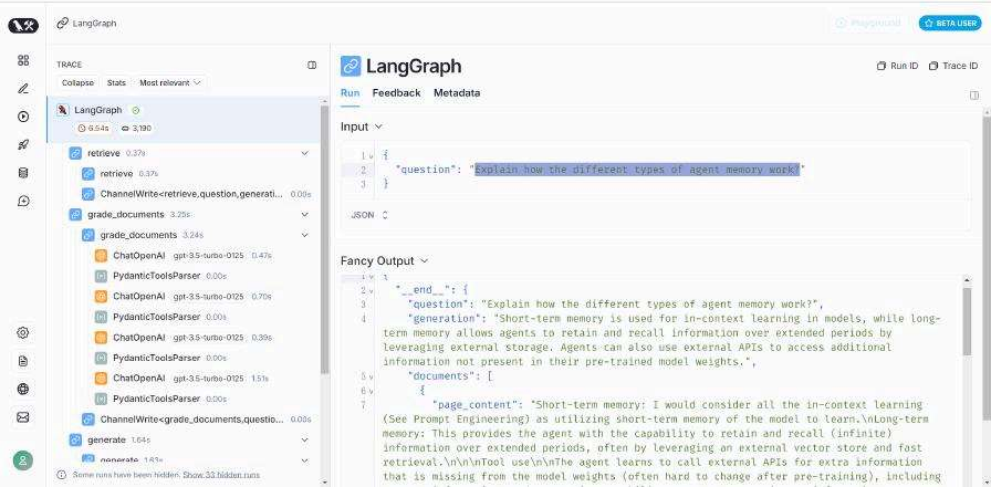


在 `Self-RAG` 架构中，用到了 3 处循环结构，对于该架构的 AI 应用，单纯使用 LCEL 表达式构建的链应用也没法或者很难实现整个过程，同样必须使用到 LangGraph 来构建 `循环图`，所以关于 `Self-RAG` 整个流程的构建，依旧会等到我们学习完 LangGraph 再来尝试。

02. LangSmith 下的 Self-RAG

在 LangChain 官网的示例中，提供了一个 `Self-RAG` 的 LangSmith 日志运行流程，我们可以通过这个日志的运行流程来深入理解。

链接：https://smith.langchain.com/public/55d6180f-aab8-42bc-8799-dadce6247d9b/r




RAG优化策略与知识库模块功能解析
01. 查询转换阶段
01. 学习掌握多查询转换优化策略，扩大检索范围。
02. 学习掌握RAG多查询结果融合策略，并了解RRF算法的作用与应用。
03. 学习掌握问题分解优化策略，并在LangChain中针对迭代式回答的优化策略实现了对应示例。
04. 学习掌握回答回退优化策略，通过提问更抽象的问题来扩大检索范围，解决复杂问题检索文档少的问题。
05. 学习掌握混合策略实现doc-doc对称检索的思想、运行流程以及实现案例。
06. 学习掌握集成检索器实现多种策略的合并，通过集成检索器来合并多个检索策略的优点，也是很常见的一个组件。
02. 路由逻辑阶段
01. 基于函数回调的数据库路由，快速实现检索不同的数据库。
02. 基于语义相似性的Prompt路由，为不同的提问方式选择不同的Prompt。
03. 查询构建阶段
01. Self-Query自查询检索器实现对向量数据库的动态元数据过滤器构建，更加精准地检索数据。
04. 索引阶段
01. MultiVectorRetriever实现多向量检索，使用摘要、总结、子文档检索全文。
02. 使用RAPTOR递归文档树的方式来执行对数据的分层检索，并探讨了该检索策略的优缺点。
05. 筛选阶段
01. 掌握ReRank重排模型提升重要性文档的顺序，并使用Cohere在线模型实现重排序。
02. 学习了解CRAG的思想以及运行流程，并提出了LCEL表达式的相关缺陷及LangGraph的概念。
06. 生成阶段
01. 学习了解了Self-RAG的思想及运行流程，并通过观察LangSmith的运行日志深入了解Self-RAG。
07. 知识库模块功能解析
01. 学习了解LLMOps知识库模块的各个功能，并设计了对应数据库表的ER图及功能API的初步编写。