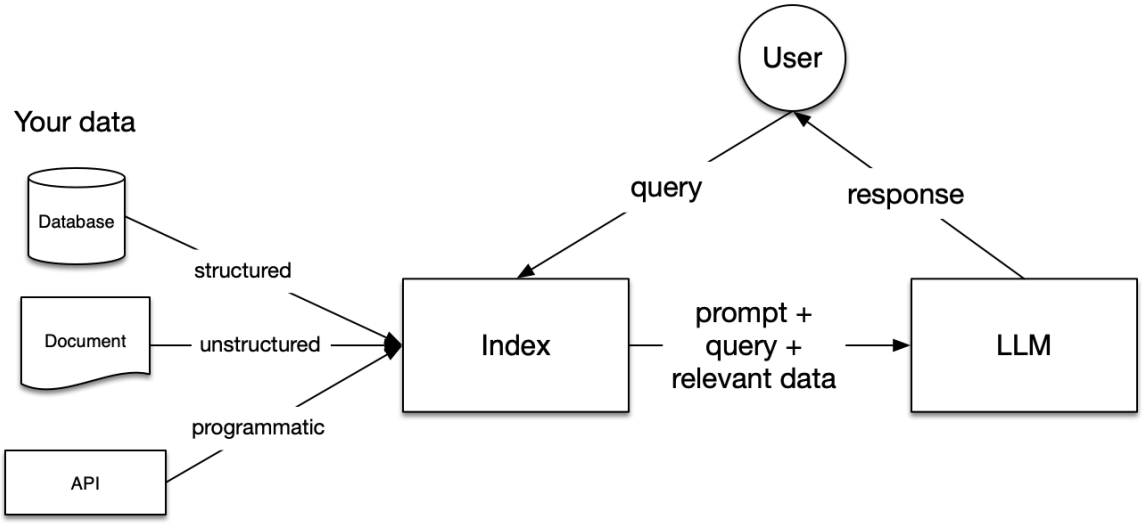
# 基于LlamaIndex/LangChain构建的RAG-ChatGLM大模型知识库

## 检索增强生成 (RAG)

LLM 接受过大量数据的培训，但他们没有接受过**个人定制化**数据的训练。检索增强生成 (RAG) 通过将您的数据添加到LLM 已经有权访问的数据中来解决这个问题。您将在本文档中经常看到对 RAG 的引用。

在 RAG 中，您的数据已加载并准备好用于查询或“索引”。用户查询作用于索引，索引将数据过滤到最相关的上下文。然后，此上下文和您的查询连同提示一起转到LLM ， LLM会提供响应。

即使您正在构建的是聊天机器人或代理，您也会想了解将数据导入应用程序的 RAG 技术。




## RAG 中的阶段

RAG 中有五个关键阶段，这将成为您构建的任何大型应用程序的一部分。这些都是：

- **加载**：这是指将数据从其所在位置（无论是文本文件、PDF、其他网站、数据库还是 API）获取到您的管道中。LlamaHub提供数百种连接器可供选择。
- **索引**：这意味着创建一个允许查询数据的数据结构。对于LLM 来说，这几乎总是意味着创建 `vector embeddings` 数据含义的数字表示，以及许多其他元数据策略，以便轻松准确地找到上下文相关的数据。
- **存储**：一旦数据被索引，您几乎总是希望存储索引以及其他元数据，以避免重新索引。
- **查询**：对于任何给定的索引策略，您可以通过多种方式利用 LLM 和 LlamaIndex 数据结构进行查询，包括子查询、多步查询和混合策略。
- **评估**：任何管道中的关键步骤是检查它相对于其他策略的有效性，或者何时进行更改。评估提供客观衡量您对查询的答复的准确性、忠实度和速度的程度。

# 每个步骤中的重要概念

您还会遇到一些术语，它们指的是每个阶段中的步骤。

## 装载阶段

**节点和文档**：A `Document` 是任何数据源的容器 - 例如 PDF、API 输出或从数据库检索数据。 A `Node` 是 LlamaIndex 中数据的原子单位，表示源的“块” `Document` 。节点具有将它们与它们所在的文档以及其他节点相关联的元数据。

**连接器**：数据连接器（通常称为 a `Reader` ）将来自不同数据源和数据格式的数据摄取到 `Documents` 和 `Nodes` 。

## 索引阶段

**索引**：获取数据后，LlamaIndex 将帮助您将数据索引到易于检索的结构中。这通常涉及生成 `vector embeddings` 存储在称为 `vector store` .索引还可以存储有关数据的各种元数据。

**嵌入**LLM 生成称为 的数据的数字表示 `embeddings` 。在过滤数据的相关性时，LlamaIndex 会将查询转换为嵌入，并且您的向量存储将查找在数字上与查询的嵌入相似的数据。

## 查询阶段

**检索器**：检索器定义在给定查询时如何从索引有效检索相关上下文。您的检索策略对于检索数据的相关性和检索效率至关重要。

**路由器**：路由器确定将使用哪个检索器从知识库中检索相关上下文。更具体地说，该类 `RouterRetriever` 负责选择一个或多个候选检索器来执行查询。他们使用选择器根据每个候选人的元数据和查询来选择最佳选项。

**节点后处理器**：节点后处理器接收一组检索到的节点并对它们应用转换、过滤或重新排序逻辑。

**响应合成器**：响应合成器使用用户查询和给定的一组检索到的文本块从 LLM 生成响应。

## 组合

有数据支持的 LLM 应用程序有无数的用例，但它们可以大致分为三类：

**查询引擎**：查询引擎是一个端到端管道，允许您对数据提出问题。它接受自然语言查询，并返回响应，以及检索并传递给LLM 的参考上下文。

**聊天引擎**：聊天引擎是一个端到端的管道，用于与您的数据进行对话（多次来回而不是单个问答）。

**代理**：代理是由LLM 提供支持的自动化决策者，通过一组工具与世界进行交互。代理可以采取任意数量的步骤来完成给定的任务，动态地决定最佳的行动方案，而不是遵循预先确定的步骤。这赋予它额外的灵活性来处理更复杂的任务。

# 核心技术分析

## LlamaIndex pipeline

#### 一、载入环节

1. 在载入过程中的"document"不是通用的理解的文档，它更是一种“容器”的代指，它可以是API、CSV、PDF、数据库记录等多种形式。在llamaIndex里有一个LlamaHub的组件，作用就是充当多种数据源的连接器，这个组件库也是开源的，现在已经有了不少connector还在持续更新中。

2. document里的主要属性有两个，一个是用来描述document的"metadata"（元数据）,一个是用来描述document与其他document和node关系的relationships。
3. "node"也不是老IT术语里的“节点”的意思，而是指document中的切分出的一块，我觉得它的同义词应该是“shard”（分片）。

调用llamaindex载入数据源成为document的基础代码是：

```
from llama_index.core import SimpleDirectoryReader
documents = SimpleDirectoryReader("./包含查询文档的文件夹名称").load_data()
```

### 二、建立索引

通过llama_index.core的另一个组件VectorStoreIndex来实现，从名字就可以看到，Index是按向量的方式进行存储的。

```
from llama_index.core import VectorStoreIndex
index = VectorStoreIndex.from_documents(documents)
```

### 三、查询环节

1. llamaindex是通过一个叫“query_engine"的组件来进行查询的，通过在索引上建立的一个或多个retriever来实现。用户可以在提示中输入自然语言查询。普通查询的代码为：

```
query_engine = index.as_query_engine()
response = query_engine.query("要搜索的内容")
```

2. query engine里最重要的组成部分就是retriever, retriever可以被进一步配置完成更加精细化的查询设置。在llamaindex的简化案例中，没有动用retriever，应该采用的就是query engine的默认设置。
3. 查询里还有个离不开的部分是把LLM大模型结合进来，负责LLM的部分在llamaIndex的query engine里叫做“Synthesizer”（合成器），它登场的时机在node从retriever中被提取出，以及进行完处理转换后。默认情况下不调整synthesizer也能得到LLM参与的结果，但是进行配置后可以得到更佳的结果。

llama的文档中给了个配置synthesizer的例子，这个response_mode里的参数“compact"可以把数据块提前压缩，减少对大模型token的调用量，起到节约开销的效果。

```
from llama_index.core.data_structs import Node
from llama_index.core.schema import NodeWithScore
from llama_index.core import get_response_synthesizer

response_synthesizer = get_response_synthesizer(response_mode="compact")

response = response_synthesizer.synthesize(
    "query text", nodes=[NodeWithScore(node=Node(text="需搜索的问题"),
score=1.0)]
)
```

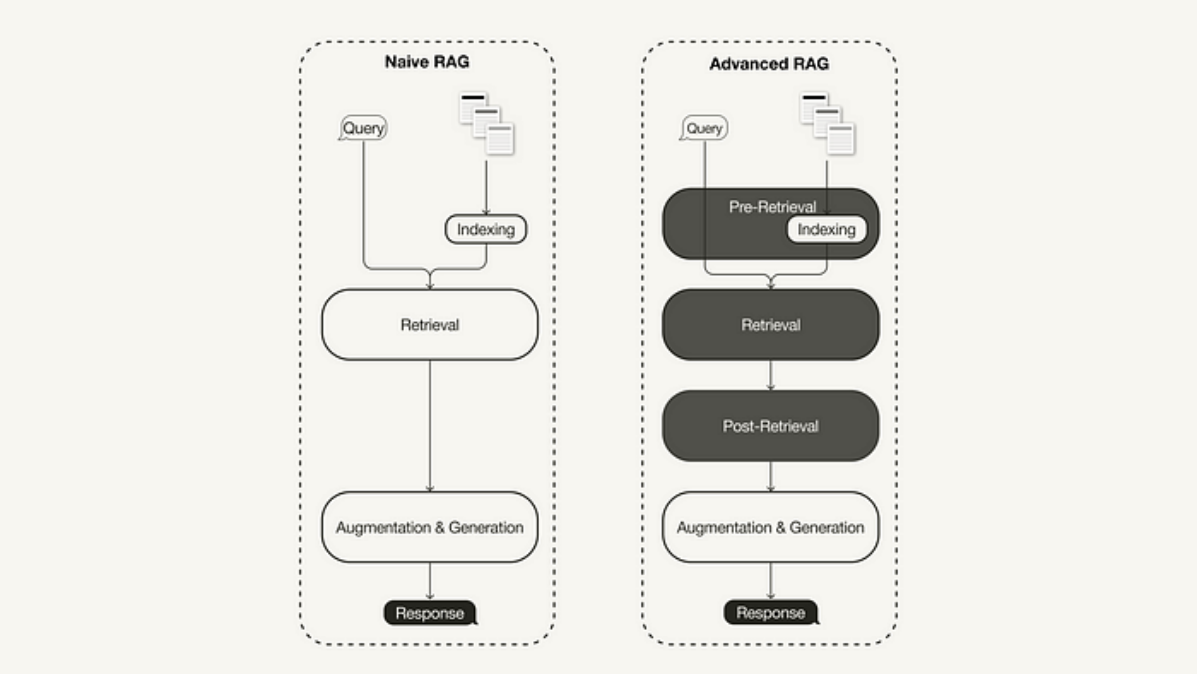
我用了llamaIndex教程文档里的示例文章提问，查询问题为“What are the two things the author worked on?”采用普通查询模式时，得到的结果为：

```
The author worked on writing and programming.
```

在采用配置后的synthesizer后，得到的查询结果为：

```
The author worked on developing a new software application and conducting
research on the impact of
```

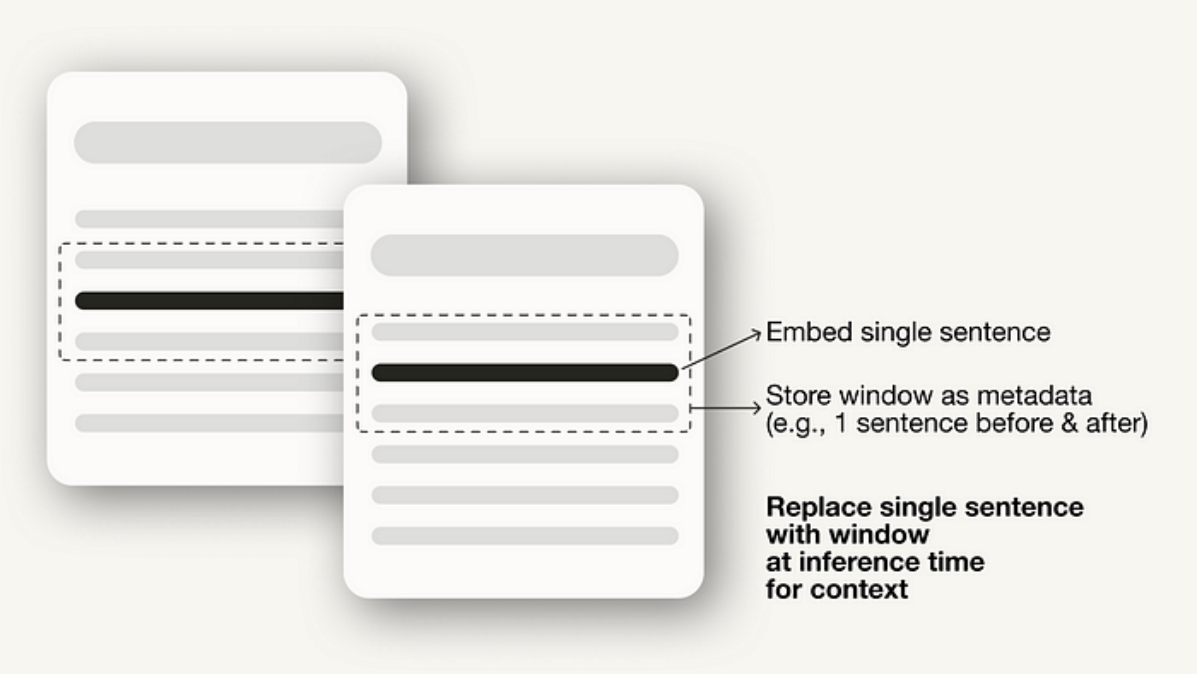
## Advanced-RAG(高级检索增强生成)




### 检索前优化

检索前优化集中在数据索引优化和查询优化上。数据索引优化技术旨在以有助于提高检索效率的方式存储数据，例如 [1]：

1. 滑动窗口使用片段之间的重叠，是最简单的技术之一。
2. 提高数据粒度应用数据清洗技术，例如删除无关信息、确认事实准确性、更新过时信息等。
3. 添加元数据，如日期、目的或章节，用于过滤目的。
4. 优化索引结构涉及不同的策略来索引数据，例如调整片段大小或使用多索引策略。本文将实现的一项技术是句子窗口检索，它将单个句子嵌入到检索中，并在推断时用更大的文本窗口替换它们。




此外，检索前技术不仅限于数据索引，还可以涉及推理时的技术，如查询路由、查询重写和查询扩展。

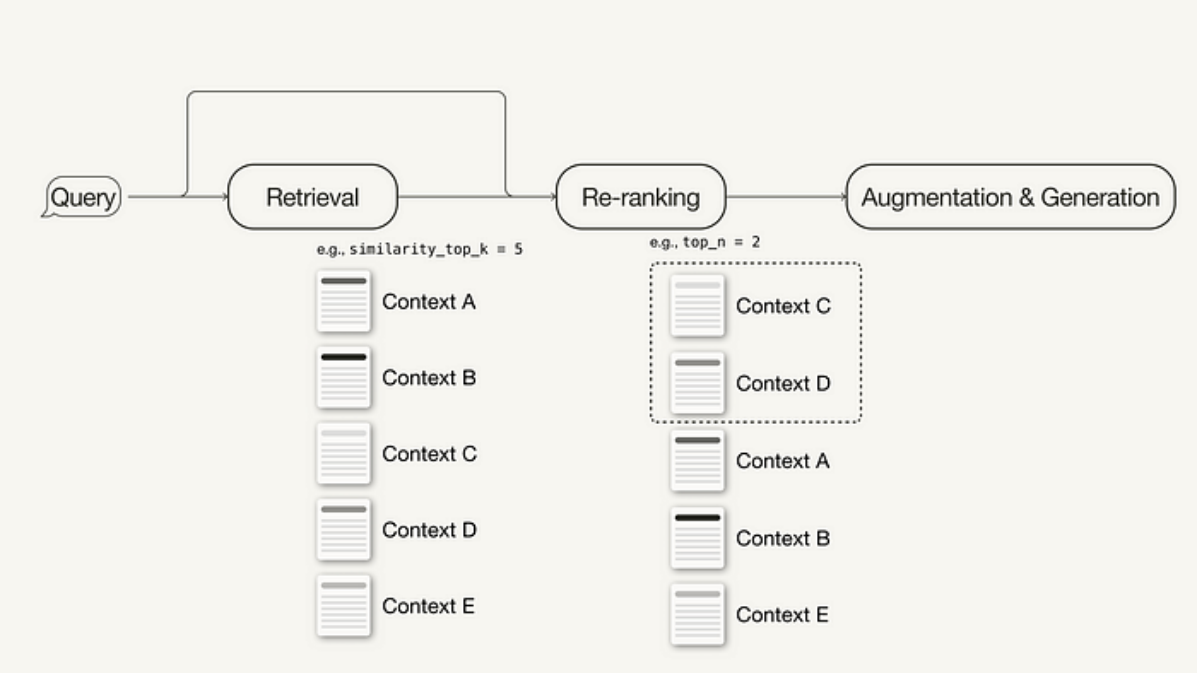
## 检索优化

检索阶段的目标是确定最相关的上下文。通常，检索基于向量搜索，它计算查询与索引数据之间的语义相似性。因此，大多数检索优化技术都围绕嵌入模型展开 [1]：

1. 微调嵌入模型，将嵌入模型定制为特定领域的上下文，特别是对于术语不断演化或罕见的领域。例如，BAAI/bge-small-en是一个高性能的嵌入模型，可以进行微调（请参阅微调指南）。
2. 动态嵌入根据单词的上下文进行调整，而静态嵌入则为每个单词使用单一向量。例如，OpenAI的embeddings-ada-02是一个复杂的动态嵌入模型，可以捕获上下文理解。[1]

除了向量搜索之外，还有其他检索技术，例如混合搜索，通常是指将向量搜索与基于关键字的搜索相结合的概念。如果您的检索需要精确的关键字匹配，则此检索技术非常有益。

## 检索后优化




对检索到的上下文进行额外处理可以帮助解决一些问题，例如超出上下文窗口限制或引入噪声，从而阻碍对关键信息的关注。在RAG调查中总结的检索后优化技术包括：

1. 提示压缩：通过删除无关内容并突出重要上下文，减少整体提示长度。
2. 重新排序：使用机器学习模型重新计算检索到的上下文的相关性得分。

## 实现高级RAG代码部分

- 检索前优化：句子窗口检索
- 检索优化：混合搜索
- 检索后优化：重新排序

**索引优化示例：句子窗口检索**

对于句子窗口检索技术，您需要进行两个调整：首先，您必须调整如何存储和后处理您的数据。我们将使用 SentenceWindowNodeParser，而不是 SimpleNodeParser。

```
from llama_index.core.node_parser import SentenceWindowNodeParser

node_parser = SentenceWindowNodeParser.from_defaults(
    window_size=3,
    window_metadata_key="window",
    original_text_metadata_key="original_text",
)
```

SentenceWindowNodeParser 做了两件事情：

- 它将文档分成单个句子，这些句子将被嵌入。
- 对于每个句子，它创建一个上下文窗口。如果您指定 window_size = 3，则生成的窗口将为三个句子长，从嵌入句子的前一个句子开始，并跨越后一个句子。窗口将存储为元数据。

在检索过程中，返回与查询最接近的句子。检索后，您需要通过定义 MetadataReplacementPostProcessor 并在节点后处理器列表中使用它来用元数据替换句子。

```
from llama_index.core.postprocessor import MetadataReplacementPostProcessor

# The target key defaults to `window` to match the node_parser's default
postproc = MetadataReplacementPostProcessor(
    target_metadata_key="window"
)
...
query_engine = index.as_query_engine(
    node_postprocessors = [postproc],
)
```

**检索优化示例：混合搜索**

在LlamaIndex中实现混合搜索与两个参数更改相同，如果底层向量数据库支持混合搜索查询的话。alpha 参数指定向量搜索和基于关键字的搜索之间的加权，其中 alpha = 0

表示基于关键字的搜索，alpha = 1 表示纯向量搜索。

```
query_engine = index.as_query_engine(
    ...,
    vector_store_query_mode="hybrid",
    alpha=0.5,
    ...
)
```

**检索后优化示例：重新排序**

将 reranker 添加到您的高级RAG管道中只需要三个简单的步骤：

首先，定义一个重新排序模型。在这里，我们使用来自Hugging Face的 BAAI/bge-reranker-base。 在查询引擎中，将重新排序模型添加到节点后处理器列表中。 在查询引擎中增加 similarity_top_k 以检索更多的上下文段落，在重新排序后可以将其减少到 top_n。

```
# !pip install torch sentence-transformers
from llama_index.core.postprocessor import SentenceTransformerRerank

# Define reranker model
rerank = SentenceTransformerRerank(
    top_n = 2,
    model = "BAAI/bge-reranker-base"
)
...
# Add reranker to query engine
query_engine = index.as_query_engine(
        similarity_top_k = 6,
        ...,
                node_postprocessors = [rerank],
        ...,
)
```

# 参考


[1] Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., ... & Wang, H. (2023). Retrieval-augmented generation for large language models: A survey. arXiv preprint arXiv:2312.10997.

[2] https://towardsdatascience.com/advanced-retrieval-augmented-generation-from-theory-to-llamaindex-implementation-4de1464a9930