# LlamaIndex

## 简介

LlamaIndex（以前称为 **GPT Index**）是一个用于构建基于文档的大语言模型（LLM）应用的开源工具。它通过提供灵活的索引结构和简单的接口，帮助开发者将外部数据源（如文档、数据库或API）与大语言模型相结合，从而更高效地查询和分析数据。以下是 LlamaIndex 的核心概念和功能简介：

核心功能：

1. **数据整合与管理**
   - **多种数据源支持**：支持导入多种数据格式，包括文本文件、PDF、CSV、SQL数据库等。
   - **数据预处理**：能够自动分块长文本、清理无关内容，并为后续的模型查询做好准备。
2. **灵活的索引结构**
   - 提供了多种索引类型，例如树状索引（Tree Index）、列表索引（List Index）和图索引（Graph Index）。
   - 开发者可以根据应用场景选择合适的索引，以优化查询效率或结果质量。
3. **智能查询引擎**
   - 基于构建的索引，可以直接使用 LLM 进行语义搜索和自然语言问答。
   - 支持自定义查询模板，以适配不同领域的需求。
4. **与主流 LLM 的集成**
   - 支持与 OpenAI、Anthropic、Cohere 等多种主流语言模型的无缝集成。
   - 可以通过配置文件快速切换不同模型。

## 1. LlamaIndex 使用模板

LlamaIndex 常用使用模版：

1. 读取文档 (手动添加or通过Loader自动添加)；
2. 将文档解析为Nodes；
3. 构建索引（从文档or从Nodes，如果从文档，则对应函数内部会完成第2步的Node解析）

   [可选，进阶] 在其他索引上构建索引，即多级索引结构
4. 查询索引并对话大模型

LlamaIndex的核心是将文档分解为多个Node对象。节点是LlamaIndex中的一等公民。节点表示源文档的“块”，无论是文本块、图像块还是更多。它们还包含元数据以及与其他节点和索引结构的关系信息。当您创建索引时，它抽象了节点的创建，但是，如果您的需求需要，您可以手动为文档定义节点。

安装：

```
!pip install llama-index
!pip install llama-index-core
!pip install llama-index-llms-dashscope
!pip install llama-index-embeddings-dashscope
```

## 1.1 读取文档

1. 使用data loaders读取

```
from llama_index import SimpleDirectoryReader

# 从文件夹读取
documents = SimpleDirectoryReader(input_dir='./data').load_data()

# 从指定文件读取，输入为List
documents = SimpleDirectoryReader(input_files=['./data/file.txt']).load_data()
```

1. 或者直接把自己的text改为document文档

```
from llama_index import Document
# 直接从文本转换
text_list = [text1, text2, ...]
documents = [Document(t) for t in text_list]
```

## 1.2 将文档解析为Nodes

文档是轻量化的数据源容器，可以将文档：

- 解析为 Node 对象
    - ```
      from llama_index.node_parser import SimpleNodeParser
      parser = SimpleNodeParser()
      nodes = parser.get_nodes_from_documents(documents)
      ```
- 直接喂入 Index，函数内部会完成转化Node过程
    - ```
      from llama_index import GPTSimpleVectorIndex
      index = GPTSimpleVectorIndex.from_documents(documents)
      ```
    - 或者从Node构建Index

      ```
      from llama_index import GPTSimpleVectorIndex
      index = GPTSimpleVectorIndex(nodes)
      ```

## 1.3 构建索引

详见使用LlamaIndex索引

当想在多个索引中，复用一个 Node 时，可以通过定义 DocumentStore 结构，并在添加Nodes时指定 DocumentStore

```
from gpt_index.docstore import SimpleDocumentStore

docstore = SimpleDocumentStore()
docstore.add_documents(nodes)

index1 = GPTSimpleVectorIndex(nodes, docstore=docstore)
index2 = GPTListIndex(nodes, docstore=docstore)
```

也可以将文档插入到索引

```
from llama_index import GPTSimpleVectorIndex

index = GPTSimpleVectorIndex([])
for doc in documents:
    index.insert(doc)
```

存储 Index 下次用

```
import os.path as osp
index_file = "data/indices/index.json"
if not osp.isfile(index_file):
    # 判断是否存在，不存在则创建
    index = GPTSimpleVectorIndex.from_documents(documents)
    index.save_to_disk(index_file, encoding='utf-8')
else:
    # 存在则 load
    index = GPTSimpleVectorIndex.load_from_disk(index_file)
```

## 1.4 查询索引并对话大模型

在默认情况下，查询索引通常采用问答形式，无需指定额外参数：

Python

```
# 简单查询
response = index.query("作者成长过程中做了什么？")
print(response)

response = index.query("请根据用户的背景信息撰写一封邮件。")
print(response)
```

您也可以根据所使用的索引类型，通过添加额外参数来优化查询。

1. 设置模式（`mode`）

`mode` 参数允许您指定底层模型的行为。以 `ListIndex` 为例，主要有两种选项：

- `default`：此模式采用“创建并完善”的方法，顺序遍历每个 `Node` 以构建答案。
- `embedding`：此模式根据与查询最相似的 top-k 个节点（通过嵌入向量确定）来合成回复。

```
index = GPTListIndex.from_documents(documents)

# 使用默认模式
response = index.query("作者成长过程中做了什么？", mode="default")

# 使用嵌入模式
response = index.query("作者成长过程中做了什么？", mode="embedding")
```

2. 设置回复模式（`response_mode`）

**注意：** 此选项不适用于 `GPTTreeIndex`。

`response_mode` 参数影响答案是如何从检索到的节点中构建的：

- `default`：对于给定的索引，此模式通过顺序浏览每个节点来“创建并完善”答案。每个节点都涉及一次独立的 LLM 调用。这对于获取更详细的答案很有益。
- `compact`：在每次 LLM 调用过程中，此模式通过填充尽可能多的节点文本块（以适应最大提示大小）来“紧凑”提示。如果一个提示中塞满了太多块，则通过多个提示来“创建并完善”答案。
- `tree_summarize`：给定一组节点和查询，此模式递归地构建一棵树并将根节点作为响应返回。这对于摘要目的很有用。

```
index = GPTListIndex.from_documents(documents)

# 默认回复模式
response = index.query("作者成长过程中做了什么？", response_mode="default")

# 紧凑回复模式
response = index.query("作者成长过程中做了什么？", response_mode="compact")

# 树状总结回复模式
response = index.query("作者成长过程中做了什么？", response_mode="tree_summarize")
```

3. 设置 `required_keywords` 和 `exclude_keywords`

您可以在大多数索引上设置 `required_keywords` 和 `exclude_keywords`（`GPTTreeIndex` 除外）。这可以预先过滤掉不包含 `required_keywords` 或包含 `exclude_keywords` 的节点，从而减少搜索空间，进而减少 LLM 调用/成本的时间/数量。

Python

```
response = index.query(
    "作者在 Y Combinator 之后做了什么？",
    required_keywords=["Combinator"],
    exclude_keywords=["Italy"]
)
```

4. 解析回复

查询的回复通常包含答案文本和用于生成回复的来源节点。

Python

```
response = index.query("<query_str>")

# 获取回复文本
print(response.response) # 注意：根据库版本，通常直接通过 .response 或 .response_str 访问

# 获取来源节点
source_nodes = response.source_nodes
print(source_nodes)

# 获取格式化的来源（如果可用）
formatted_sources = response.get_formatted_sources()
print(formatted_sources)
```

# 2. 使用 LlamaIndex 索引

设置阿里dash scope 模型：

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

os.environ["DASHSCOPE_API_KEY"]="<your-api-key>"
os.environ["ALIYUN_BASE_URL"]="https://dashscope.aliyuncs.com/compatible-mode/v1"
#设置 llm
Settings.llm = DashScope( # <--- Changed this line
    api_key=os.getenv("DASHSCOPE_API_KEY"),
    model="qwen-plus"
)
# 配置嵌入模型 ⭐ 新增设置
Settings.embed_model = DashScopeEmbedding(
    model_name="text-embedding-v3",  # 百炼嵌入模型
    api_key=os.getenv("DASHSCOPE_API_KEY"),
)
```

加载华为2024年度财报PDF文档：

```
import logging
import sys

## showing logs
logging.basicConfig(stream=sys.stdout, level=logging.INFO)
logging.getLogger().addHandler(logging.StreamHandler(stream=sys.stdout))


## load the PDF
from langchain.text_splitter import RecursiveCharacterTextSplitter
from llama_index.core import download_loader

# define loader
UnstructuredReader = download_loader('UnstructuredReader', refresh_cache=True)
loader = UnstructuredReader()

# load the data
documents =
loader.load_data('data/annual_report_2024_cn.pdf',split_documents=False)
```

## 2.1 List Index

列表索引是一种简单的数据结构，其中节点按顺序存储。在索引构建期间，文档文本被分块、转换为节点并存储在列表中。

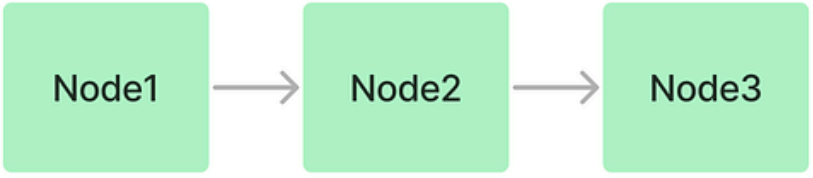
索引构建流程：

· 文档切分：原始文档被分割成更小的文本块（如按段落、固定长度）。

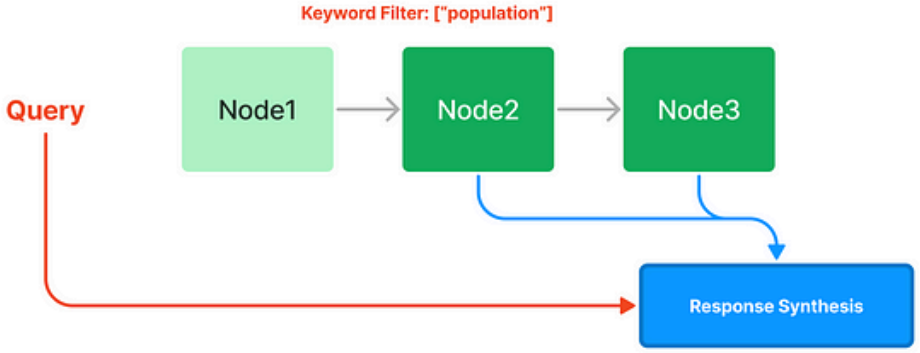
· 节点生成：每个文本块转换为一个 Node 对象，生成对应的向量嵌入。

· 列表存储：所有节点按顺序存入列表，形成索引结构。


在查询期间，如果没有指定其他查询参数，LlamaIndex只是将列表中的所有node加载到Response Synthesis模块中。适用于需全局信息的查询（如摘要生成、全局分析）




列表索引提供了许多查询列表索引的方法，如从基于嵌入的查询中获取前k个邻居，或者添加一个关键字过滤器，如下所示:




创建 List Index并查询：

```
from llama_index.core import GPTListIndex
from IPython.display import Markdown, display

# Example documents (replace with your own)


# Build the index from documents
index = GPTListIndex.from_documents(documents)

# Create a query engine from the index
query_engine = index.as_query_engine()

# Query the index
response = query_engine.query("净收入为多少？")
display(Markdown(f"<b>{response}</b>"))

# Alternative query engine setup with retriever_mode specified
query_engine_with_embedding = index.as_query_engine(
    retriever_mode="embedding",
    verbose=True
)

# Query with the alternative query engine
response_with_embedding = query_engine_with_embedding.query("净收入为多少?")
display(Markdown(f"<b>{response_with_embedding}</b>"))
```

输出：

新提供的上下文中没有提及具体的净收入数值。因此，根据给定的信息，无法直接得出新的净收入数额。基于此，对于“净收入为多少？”这一问题的回答保持不变。
2024年，华为的净利润为人民币62,574百万元。请注意，您所询问的是净收入，而给出的数据是净利润，这两个概念虽然相关但并不相同。在当前提供的信息中，并没有直接提供关于净收入的具体数据。

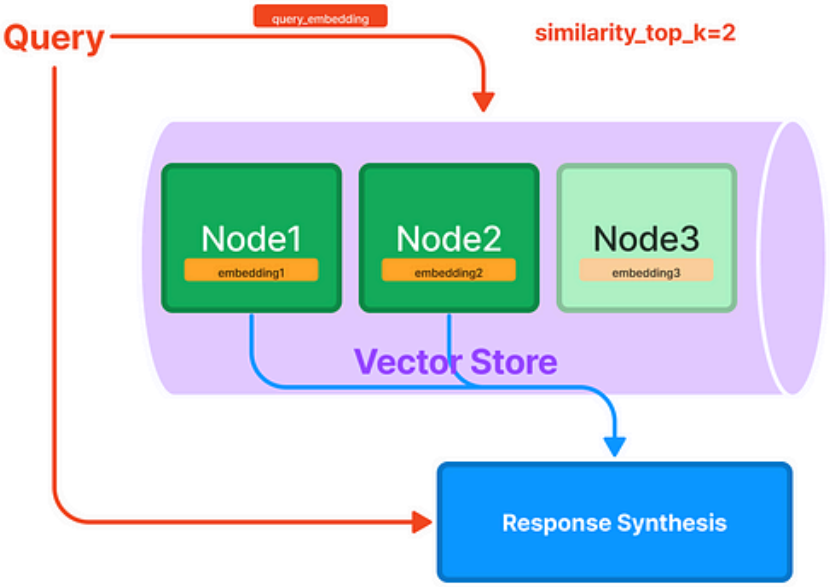
2024年的净收入为861,335百万元人民币，2023年的净收入为703,246百万元人民币。这里提到的净收入指的是客户合同收入

LlamaIndex 能够为列表索引提供 Embedding 支持。

```
# Alternative query engine setup with retriever_mode specified
query_engine_with_embedding = index.as_query_engine(
    retriever_mode="embedding",
    verbose=True
)

# Query with the alternative query engine
response_with_embedding = query_engine_with_embedding.query("What is net
operating income?")
display(Markdown(f"<b>{response_with_embedding}</b>"))
```

## 2.2 向量存储索引

**向量存储索引（Vector Store Index）** 是 LlamaIndex 中的一种索引类型，基于嵌入向量实现高效的语义检索。通过将文本表示为向量并存储在专门的数据库中，查询时可以快速找到与查询语句语义最相关的内容。




索引构建流程：

- 文档分割：长文档被拆分为文本块（chunks），例如每块 500 字符。
- 节点生成：每个文本块转化为 Node 对象。
- 向量化：调用嵌入模型 API为每个 Node 生成向量。
- 向量存储：所有向量存入专用数据库。

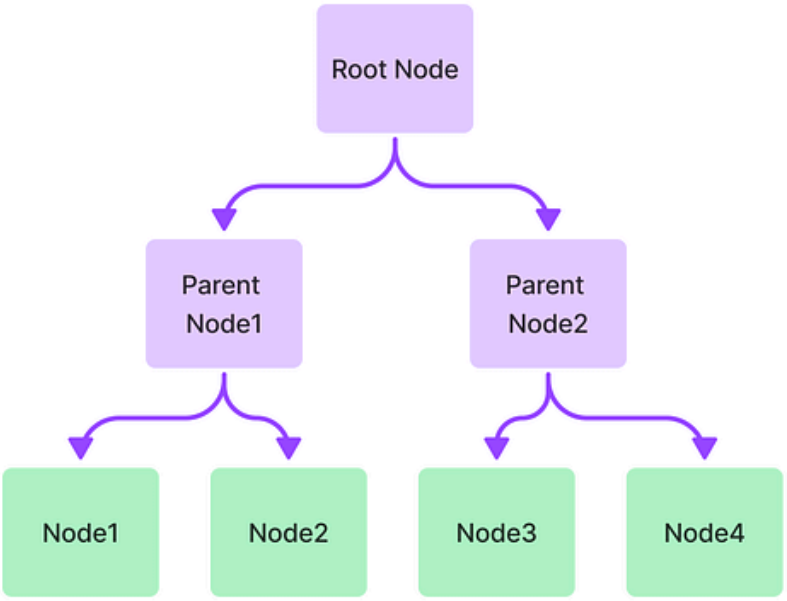
查询流程：

- 查询向量化：用户输入（如 "气候变化的影响"）通过相同的嵌入模型转换为向量。
- 相似度搜索：向量数据库计算查询向量与所有存储向量的相似度，返回 Top-K 最相似节点（如 K=5）。
- 响应合成：LLM（如 GPT-4）根据这些节点生成自然语言响应。

```
from llama_index.core import Document, GPTVectorStoreIndex

# Example documents (replace with your actual data)
documents = [
    Document(text="The author spent their childhood exploring forests, climbing
trees, and reading books.")
]

# Build the index from the documents
index = GPTVectorStoreIndex.from_documents(documents)

# Create a query engine from the index
query_engine = index.as_query_engine()
```

```
# Query the index
response = query_engine.query("What did the author do growing up?")
print(response)
```

## 2.3 树状索引

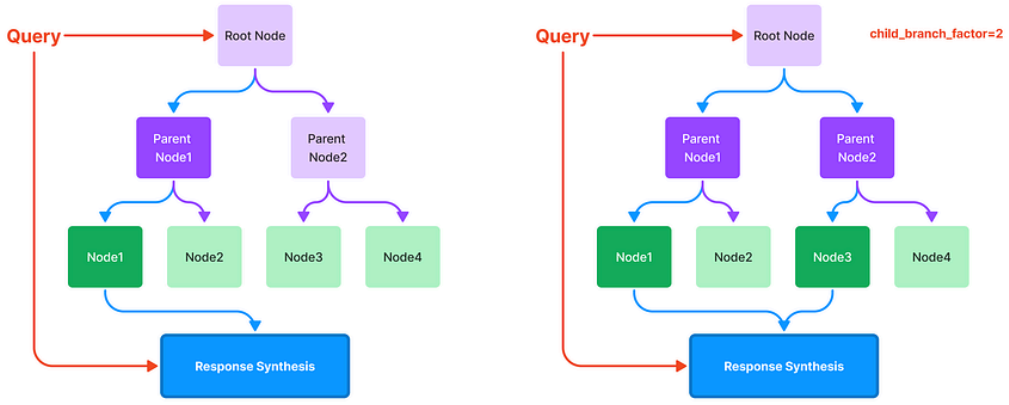



树状索引是树结构索引，**其中每个节点是子节点的摘要**。在索引构建期间，树以自下而上的方式构建，直到我们最终得到一组根节点。

树索引的构建过程是一个自下而上的分层聚合流程：

- **叶节点层**：
  - 每个文本块被作为一个叶节点。
  - 每个叶节点存储对应文本块及其嵌入向量。
- **中间节点层**：
  - 将叶节点按照一定数量（如 5 个或 10 个）分组。
  - 对每组的文本块嵌入进行聚合（如取平均值或加权平均），生成一个代表这一组的嵌入向量。
  - 为每组生成一个中间节点，存储代表性嵌入向量和摘要信息。
- **根节点**：
  - 对中间节点再次聚合，直到只剩一个根节点。
  - 根节点的嵌入向量是整个索引的全局表示。

查询树状索引涉及从根节点向下遍历到叶节点。默认情况下(`child_branch_factor=1`)，查询在给定父节点的情况下选择一个子节点。如果`child_branch_factor=2`，则查询在每个层级选择两个子节点。




示例代码：

```
from llama_index.core import GPTTreeIndex

new_index = GPTTreeIndex.from_documents(documents)
response = query_engine.query("净收入为多少?")
display(Markdown(f"<b>{response}</b>"))

## if you want to have more content from the answer,
# you can add the parameters child_branch_factor
# let's try using branching factor 2
query_engine = new_index.as_query_engine(
    child_branch_factor=2
)
response = query_engine.query("净收入为多少?")
display(Markdown(f"<b>{response}</b>"))
```

**在查询期间构建树状索引**

在 LlamaIndex 中，「查询时构建树状索引」是一种**按需计算**的优化技术，核心思想是**延迟索引构建成本**，直到真正需要时才根据查询动态创建树状结构。

**工作原理对比**

| 索引构建方式 | 构建时机 | 存储成本 | 响应延迟 | 适用场景 |
| --- | --- | --- | --- | --- |
| **预先构建索引** | 数据加载时 | 高（完整树） | 低（查询快） | 高频查询的静态数据 |
| **查询时动态构建** | 首次查询时 | 低（仅叶节点） | 高（需计算） | 低频查询/动态数据 |

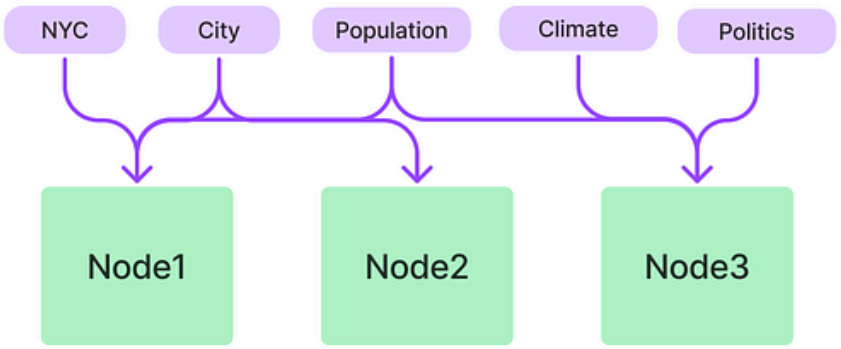
为此，可以设置以下参数：

- `retriever_mode`：指定检索模式，例如 "all_leaf" 表示使用所有叶节点。
- `response_mode`：指定响应模式，例如 "tree_summarize" 表示以树状摘要模式生成响应。
- 在构建索引时，将 `build_tree=False`。

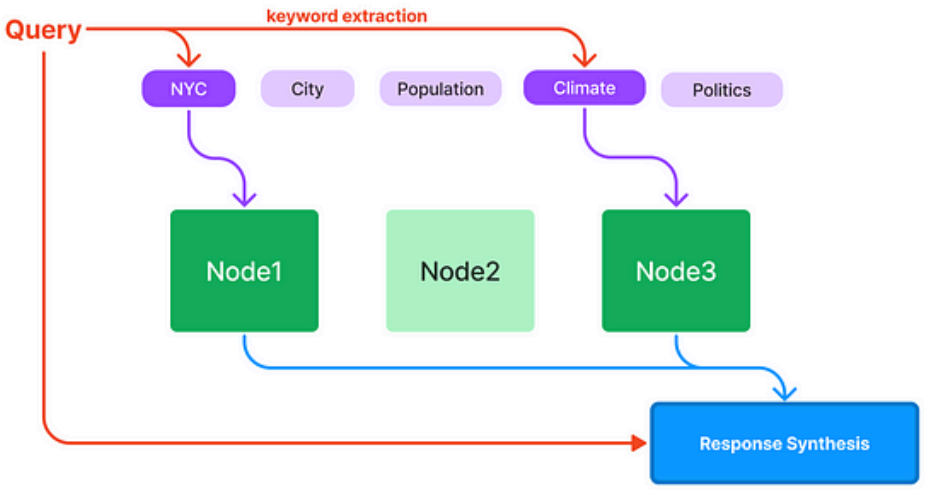
```
from llama_index import GPTKeywordTableIndex, Document

# 1. 初始只创建叶节点层（不构建树）
documents = [Document(text="段落1..."), Document(text="段落2...")]
index = GPTKeywordTableIndex.from_documents(
    documents,
    build_tree=False  # 关键设置：仅生成叶节点
)

# 2. 查询时动态构建树并生成响应
query_engine = index.as_query_engine(
    retriever_mode="all_leaf",    # 使用所有叶节点为基础
    response_mode="tree_summarize" # 树状摘要模式
)

# 首次查询触发树构建
response = query_engine.query("气候变化的主要影响？")
```

## 2.4 关键词索引

关键字表索引从每个Node提取关键字，并构建从每个关键字到该关键字对应的Node的映射。




在查询时，我们从查询中提取相关关键字，并将其与预提取的Node关键字进行匹配，获取相应的Node。提取的节点被传递到响应合成模块。




示例代码：

```
from llama_index.core import GPTKeywordTableIndex
index = GPTKeywordTableIndex.from_documents(documents)
query_engine = index.as_query_engine()
response = query_engine.query("净收入为多少?")
```

源码解析：

1. 调用llm 生成关键词：

```
    def _extract_keywords(self, text: str) -> Set[str]:
        """Extract keywords from text."""
        response = self._llm.predict(
            self.keyword_extract_template,
            text=text,
        )
        return extract_keywords_given_response(response, start_token="KEYWORDS:")
```

```
DEFAULT_KEYWORD_EXTRACT_TEMPLATE_TMPL = (
    "Some text is provided below. Given the text, extract up to {max_keywords} "
    "keywords from the text. Avoid stopwords."
    "---------------------\n"
    "{text}\n"
    "---------------------\n"
    "Provide keywords in the following comma-separated format: 'KEYWORDS:
<keywords>'\n"
)
```

> Max_keywords 默认 10

2. 关键字提取：

```
def extract_keywords_given_response(
    response: str, lowercase: bool = True, start_token: str = ""
) -> Set[str]:
    """
    在从给定的文本字符串中提取关键词应。
    将<start_token>: <word1>, <word2>, ...解析为[word1, word2, ...]
    如果响应未以<start_token>开头，则抛出异常
    """

    ...

    # 如果关键字由多个单词组成拆分为子词
    # (去除停用词)
    return expand_tokens_with_subtokens(set(results))
```

## 2.5 可组合性图索引

可组合性图索引（Composable Graph Index）是 LlamaIndex 提供的高级索引结构，用于整合多个子索引的信息并进行跨文档查询。这种索引特别适用于需要对多个相关但独立的数据源进行联合分析的场景，比如：

1. 分析不同年份的财务报告
2. 比较多篇学术论文的技术细节
3. 整合多个知识库的专家系统
4. 结合产品说明和用户反馈的分析系统

**核心概念**

- 子索引（Subindexes）
    - 每个数据源建立独立的索引（如 VectorStoreIndex）
    - 保留各数据源的上下文和特性
    - 支持个性化配置（解析器、参数等）
- 组合索引（Composed Graph）
    - 将子索引整合成一个有向图
    - 添加元信息描述各子索引内容
    - 实现跨索引的信息路由和综合

通过一个场景编写代码:我们将执行以下步骤来演示可组合性图索引的能力:

- 加载多篇PDF格式的学术论文。
- 为每篇论文创建独立的“知识库”（索引）。
- 为每篇论文生成一个简短的摘要。
- 将所有论文的知识库组合成一个更大的、可相互关联的“总知识图谱”。
- 查询

```
from llama_index.core import Settings, StorageContext, VectorStoreIndex
from llama_index.core.indices.composability import ComposableGraph
from llama_index.llms.dashscope import DashScope
from llama_index.embeddings.dashscope import DashScopeEmbedding
from llama_index.readers.file import UnstructuredReader
```

```
from llama_index.core.node_parser import SentenceSplitter
import os

# 设置API密钥和环境变量
os.environ["DASHSCOPE_API_KEY"] = "<your-api-key>"

# 配置模型设置 - 统一全局设置
Settings.llm = DashScope(
    api_key=os.getenv("DASHSCOPE_API_KEY"),
    model="qwen-plus"
)

# 关键修改1: 设置较小的批处理大小
Settings.embed_model = DashScopeEmbedding(
    model_name="text-embedding-v3",
    api_key=os.getenv("DASHSCOPE_API_KEY"),
    embed_batch_size=8  # 设置为安全值 (小于10)
)

# 关键修改2: 降低分块大小以加速处理
Settings.node_parser = SentenceSplitter(
    chunk_size=384,  # 减小分块大小以降低复杂度
    chunk_overlap=15,
    include_prev_next_rel=True
)

# 关键修改3: 限制输出长度避免token超限
Settings.num_output = 384

# 准备两篇论文的文档
paper_files = {
    "transformer": "data/1706.03762v7.pdf",  # "Attention is All You Need"
    "vit": "data/2010.11929v2.pdf"           # "An Image is Worth 16x16 Words"
}

# 自定义PDF解析器
reader = UnstructuredReader()

# 为每篇论文创建索引
index_set = {}
doc_set = {}

print("开始创建单篇论文索引...")
for paper_name, file_path in paper_files.items():
    print(f"正在处理: {paper_name} 论文...")

    # 加载单篇论文
    loader = SimpleDirectoryReader(
        input_files=[file_path],
        file_extractor={".pdf": reader}
    )

    # 关键修改4: 分批处理文档避免大请求
    try:
        documents = loader.load_data()
    except Exception as e:
```

```
        print(f"文档加载错误，尝试分批加载: {str(e)}")
        documents = []
        for part in range(5):  # 分5批加载
            partial_docs = loader.load_data(show_progress=True,
num_docs_per_page=10)
            documents.extend(partial_docs)

    doc_set[paper_name] = documents

    # 创建存储上下文
    storage_context = StorageContext.from_defaults()

    # 创建针对学术论文优化的索引
    try:
        cur_index = VectorStoreIndex.from_documents(
            documents=documents,
            storage_context=storage_context,
            show_progress=True
        )
    except Exception as e:
        print(f"索引创建错误，尝试减小批处理大小: {str(e)}")
        # 如果还不行，尝试使用更小的批处理
        from llama_index.core import ServiceContext
        service_context = ServiceContext.from_defaults(embed_batch_size=5)
        cur_index = VectorStoreIndex.from_documents(
            documents=documents,
            storage_context=storage_context,
            service_context=service_context,
            show_progress=True
        )

    index_set[paper_name] = cur_index

    # 保存索引以备后用
    storage_context.persist(f'./paper_indexes/{paper_name}')
    print(f"已完成 {paper_name} 论文索引创建!\n")

# 为每篇论文生成摘要
index_summaries = {}
for paper_name in paper_files:
    print(f"正在为 {paper_name} 论文生成摘要...")
    query_engine = index_set[paper_name].as_query_engine(similarity_top_k=3)  #
减少检索量

    # 关键修改5: 使用较短提示减少响应长度
    response = query_engine.query(
        "总结这篇论文的核心贡献和技术创新点 (100字)"
    )
    index_summaries[paper_name] = response.response
    print(f"{paper_name.upper()} 论文摘要:\n{response.response}\n")

# 创建组合索引（知识图谱）
print("正在创建组合索引...")
try:
    graph = ComposableGraph.from_indices(
        base_class=VectorStoreIndex,
```

```
        indices=[index_set[name] for name in paper_files],
        index_summaries=[index_summaries[name] for name in paper_files]
    )
except Exception as e:
    print(f"创建组合索引出错: {str(e)}")
    # 如果出错，降级为简单索引
    from llama_index.core import SummaryIndex
    all_nodes = []
    for paper_name in paper_files:

all_nodes.extend(index_set[paper_name].docstore.get_nodes(list(index_set[paper_na
me].index_struct.nodes_dict.keys())))

    graph = SummaryIndex(all_nodes)

# 保存组合索引
graph.storage_context.persist(f'./paper_indexes/combined')
print("组合索引已保存!\n")

# 配置查询引擎
custom_query_engines = {
    index_set[paper_name].index_id: index_set[paper_name].as_query_engine(
        similarity_top_k=3  # 减少子索引检索量
    ) for paper_name in paper_files
}

# 关键修改6: 确保最终查询批处理不超限
query_engine = graph.as_query_engine(
    custom_query_engines=custom_query_engines,
    verbose=True,
    similarity_top_k=3  # 减少检索结果数量
)

# 执行跨论文查询
print("执行跨论文比较查询...")
try:
    # 关键修改7: 使用更短的查询
    response = query_engine.query(
        "比较Transformer和ViT的核心异同点:架构设计、注意力机制、数据处理方式"
    )
except Exception as e:
    print(f"查询出错: {str(e)}")
    print("尝试简化查询...")
    response = query_engine.query(
        "比较Transformer和ViT的架构差异"
    )

print("\n跨论文查询结果:")
print(response.response)

try:
    # 关键修改8: 检查源节点存在性
    print("\n响应来源:")
    if hasattr(response, 'source_nodes') and response.source_nodes:
        for source_node in response.source_nodes:
            file_path = source_node.node.metadata.get('file_path', '未知来源')
```

```
            preview = source_node.node.text[:40].replace('\n', ' ') + "..."
            print(f"- {preview} (来自 {file_path})")
    else:
        print("未找到来源信息")
except Exception as e:
    print(f"无法获取响应来源: {str(e)}")
```

输出:

```
开始创建单篇论文索引...
正在处理: transformer 论文...
WARNING:root:'doc_id' is deprecated and 'id_' will be used instead
'doc_id' is deprecated and 'id_' will be used instead
'doc_id' is deprecated and 'id_' will be used instead
'doc_id' is deprecated and 'id_' will be used instead

Parsing nodes:   0%|          | 0/1 [00:00<?, ?it/s]
Generating embeddings:   0%|          | 0/15 [00:00<?, ?it/s]
已完成 transformer 论文索引创建!

正在处理: vit 论文...
WARNING:root:'doc_id' is deprecated and 'id_' will be used instead
'doc_id' is deprecated and 'id_' will be used instead
'doc_id' is deprecated and 'id_' will be used instead
'doc_id' is deprecated and 'id_' will be used instead

Parsing nodes:   0%|          | 0/1 [00:00<?, ?it/s]
Generating embeddings:   0%|          | 0/29 [00:00<?, ?it/s]
已完成 vit 论文索引创建!

正在为 transformer 论文生成摘要...
TRANSFORMER 论文摘要:
这篇论文提出了Transformer模型，完全基于注意力机制，摒弃了传统的循环和卷积结构。该模型在机器翻译任务上表现出色，不仅质量更优，而且具有更好的并行化能力，训练时间也大幅减少。此外，Transformer还成功应用于英语句法分析任务，展示了其良好的泛化性能。

正在为 vit 论文生成摘要...
VIT 论文摘要:
该论文的核心贡献在于提出了视觉变换器（ViT），这是一种将自然语言处理中的变换器模型应用于计算机视觉任务的新方法。技术创新点包括使用自注意力机制来处理图像数据，以及在大规模数据集上进行预训练以提高模型性能。此外，论文还展示了ViT在多个基准测试中达到了与最先进的卷积神经网络相媲美的结果。

正在创建组合索引...
创建组合索引出错: ComposableGraph.from_indices() missing 2 required positional arguments: 'root_index_cls' and 'children_indices'
组合索引已保存!

执行跨论文比较查询...

跨论文查询结果:
Transformer和ViT（Vision Transformer）在架构设计、注意力机制以及数据处理方式上有一些核心的异同点。

1. **架构设计**:
```

- **相同点**：两者都基于Transformer架构，利用了自注意力机制来捕捉输入序列中的长距离依赖关系。它们都采用了编码器结构，并且每一层都包含了多头自注意力机制和前馈神经网络。
- **不同点**：原始的Transformer主要用于自然语言处理任务，如机器翻译，其输入是文本序列；而ViT则是将Transformer应用于计算机视觉领域，它首先将图像分割成一系列的patches，并将这些patches线性嵌入到一维序列中作为模型的输入。此外，ViT通常不包含解码器部分，而是直接使用编码器输出进行分类或其他视觉任务。

2. **注意力机制**：
- **相同点**：两者都采用了多头自注意力机制，通过计算query，key，value之间的相似度来为每个位置分配权重，从而允许模型关注输入序列中最重要的部分。
- **不同点**：虽然基本原理一致，但具体实现细节可能有所不同。例如，在处理图像时，ViT需要考虑如何有效地将二维空间

响应来源:
- 3 2 0 2 g u A 2 ] L C . s c [ 7 v 2 6... (来自 data/1706.03762v7.pdf)
- Numerous efforts have since continued to... (来自 data/1706.03762v7.pdf)
- Here, the encoder maps an input sequence... (来自 data/1706.03762v7.pdf)
- Dot-product attention is identical to ou... (来自 data/1706.03762v7.pdf)
- Similarly, self-attention layers in the ... (来自 data/1706.03762v7.pdf)
- In this work, we use sine and cosine fun... (来自 data/1706.03762v7.pdf)
- Convolutional layers are generally more ... (来自 data/1706.03762v7.pdf)
- 5.3 Optimizer We used the Adam optimize... (来自 data/1706.03762v7.pdf)
- The configuration of this model is liste... (来自 data/1706.03762v7.pdf)
- We used beam search as described in the ... (来自 data/1706.03762v7.pdf)
- (2016) [8] WSJ 23 F1 88.3 90.4 90.4 91.... (来自 data/1706.03762v7.pdf)
- Learning phrase representations using rn... (来自 data/1706.03762v7.pdf)
- Neural machine translation in linear tim... (来自 data/1706.03762v7.pdf)
- Neural machine translation of rare words... (来自 data/1706.03762v7.pdf)
- 12 Attention Visualizations making re... (来自 data/1706.03762v7.pdf)
- 1 2 0 2 n u J 3 ] V C . s c [ 2 v 9... (来自 data/2010.11929v2.pdf)
- Therefore, in large-scale image recognit... (来自 data/2010.11929v2.pdf)
- In a different line of work, Sparse Tran... (来自 data/2010.11929v2.pdf)
- 2 Published as a conference paper at IC... (来自 data/2010.11929v2.pdf)
- 3 Published as a conference paper at IC... (来自 data/2010.11929v2.pdf)
- We therefore perform 2D interpolation of... (来自 data/2010.11929v2.pdf)
- Note that the Transformer's sequence len... (来自 data/2010.11929v2.pdf)
- The first comparison point is Big Transfe... (来自 data/2010.11929v2.pdf)
- We re- port mean and standard deviation ... (来自 data/2010.11929v2.pdf)
- To boost the performance on the smaller ... (来自 data/2010.11929v2.pdf)
- This way, we assess the intrinsic model ... (来自 data/2010.11929v2.pdf)
- The first layer of the Vision Transformer... (来自 data/2010.11929v2.pdf)
- Each dot shows the mean attention distan... (来自 data/2010.11929v2.pdf)
- Attention augmented convolutional networ... (来自 data/2010.11929v2.pdf)
- Momentum contrast for unsupervised visu... (来自 data/2010.11929v2.pdf)
- Object-centric learning with slot atten-... (来自 data/2010.11929v2.pdf)
- 11 Published as a conference paper at I... (来自 data/2010.11929v2.pdf)
- In CVPR, 2020. Xiaohua Zhai, Avital Oli... (来自 data/2010.11929v2.pdf)
- (2017)) is a popular building block for ... (来自 data/2010.11929v2.pdf)
- All models are fine-tuned with cosine lea... (来自 data/2010.11929v2.pdf)
- We also experimented with 15% corruption... (来自 data/2010.11929v2.pdf)
- These values correspond to Figure 3 in t... (来自 data/2010.11929v2.pdf)
- These correspond to Figure 5 in the main... (来自 data/2010.11929v2.pdf)
- Interestingly, scaling the width of the ... (来自 data/2010.11929v2.pdf)
- Relative positional embeddings: Consider... (来自 data/2010.11929v2.pdf)
- Attention distance was computed for 128 ... (来自 data/2010.11929v2.pdf)
- Figure 13, present the performance of Ax... (来自 data/2010.11929v2.pdf)

```
- D.8 ATTENTION MAPS  To compute maps of t... (来自 data/2010.11929v2.pdf)
- 21  Published as a conference paper at I... (来自 data/2010.11929v2.pdf)
```

## 2.6 Pandas索引和SQL索引

> 它对结构化数据很有用

```
from llama_index.indices.struct_store import GPTPandasIndex
import pandas as pd

df = pd.read_csv("titanic_train.csv")

index = GPTPandasIndex(df=df)

query_engine = index.as_query_engine(
    verbose=True
)
response = query_engine.query(
    "What is the correlation between survival and age?",
)
```

## 2.7 文档摘要索引

> 这是一个全新的LlamaIndex数据结构，它是为了问答而制作的。

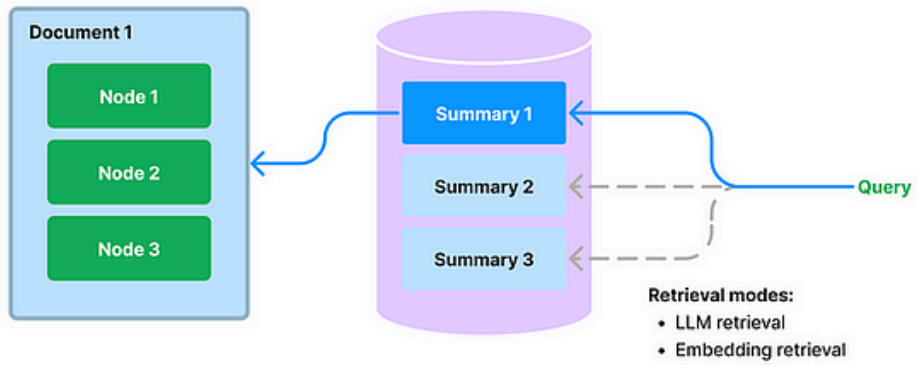
通常，大多数用户以以下方式开发基于LLM的QA系统:

1. 获取源文档并将其分成文本块。
2. 然后将文本块存储在矢量数据库中。
3. 在查询期间，通过使用相似度和/或关键字过滤器进行Embedding来检索文本块。
4. 执行整合后的响应。

然而，这种方法存在一些影响检索性能的局限性：

1. 文本块没有完整的全局上下文，这通常限制了问答过程的有效性。
2. 需要仔细调优top-k /相似性分数阈值，因为过小的值可能会导致错过相关上下文，而过大的值可能会增加不相关上下文的成本和延迟。
3. Embeddings可能并不总是为一个问题选择最合适的上下文，因为这个过程本质上是分别决定文本和上下文的。

为了增强检索结果，一些开发人员添加了关键字过滤器。然而，这种方法有其自身的挑战，例如通过手工或使用NLP关键字提取/主题标记模型为每个文档确定适当的关键字，以及从查询中推断正确的关键字。




这就是 LlamaIndex 引入文档摘要索引的原因，它可以为每份文档提取非结构化文本摘要并编制索引，从而提高检索性能，超越现有方法。该索引比单一文本块包含更多信息，比关键字标签具有更多语义。它还允许灵活的检索，包括基于 LLM 和嵌入的方法。在构建期间，该索引接收文档并使用 LLM 从每个文档中提取摘要。在查询时，它会根据摘要使用以下方法检索相关文档：

- 基于 LLM 的检索：获取文档摘要集合并请求 LLM 识别相关文档+相关性得分
- 基于嵌入的检索：利用摘要嵌入相似性来检索相关文档，并对检索结果的数量施加顶k限制。

文档摘要索引的检索类为任何选定的文档检索所有节点，而不是在节点级返回相关块。

示例代码：

```
from llama_index.core import (
    Document,
    VectorStoreIndex,
    DocumentSummaryIndex,
    StorageContext,
    Settings,
    SimpleDirectoryReader,
    ServiceContext,
)
from llama_index.core.schema import TextNode
from llama_index.llms.dashscope import DashScope
from llama_index.embeddings.dashscope import DashScopeEmbedding
from llama_index.core.node_parser import SentenceSplitter
from llama_index.core.extractors import SummaryExtractor
from llama_index.core.ingestion import IngestionPipeline
import os

# 1. 配置环境
os.environ["DASHSCOPE_API_KEY"] = "<your-api-key>" # Ensure this is correct and
active

# 全局设置
llm = DashScope(model="qwen-plus", api_key=os.getenv("DASHSCOPE_API_KEY"))
embed_model = DashScopeEmbedding(
    model_name="text-embedding-v3",
    api_key=os.getenv("DASHSCOPE_API_KEY"),
```

```
    embed_batch_size=8
)

# Set global LLM and Embed Model for LlamaIndex
Settings.llm = llm
Settings.embed_model = embed_model
Settings.node_parser = SentenceSplitter(
    chunk_size=512,
    chunk_overlap=20,
    include_prev_next_rel=True
)

# 2. 文档预处理
# --- MODIFIED: This function will now return Document objects ---
def load_documents_from_files(file_paths: list):
    """加载文档并返回Document对象列表"""
    all_documents = []

    for file_path in file_paths:
        if not os.path.exists(file_path):
            print(f"文件不存在: {file_path}")
            continue

        print(f"正在加载: {os.path.basename(file_path)}")
        try:
            # SimpleDirectoryReader directly loads into Document objects
            docs = SimpleDirectoryReader(
                input_files=[file_path]
            ).load_data()

            # Add custom metadata to each document
            for doc in docs:
                doc.metadata["source"] = os.path.basename(file_path)
                doc.metadata["doc_type"] = "academic"
            all_documents.extend(docs) # Extend the list with new documents

        except Exception as e:
            print(f"文件 {os.path.basename(file_path)} 加载错误: {str(e)}")

    print(f"已加载 {len(all_documents)} 个文档")
    return all_documents

# 3. 创建摘要索引管道
def create_summary_index(documents, storage_path):
    """创建文档摘要索引"""
    if not documents:
        raise ValueError("没有可用的文档")

    # 创建摘要提取器
    summary_extractor = SummaryExtractor(
        llm=llm,  # Explicitly pass the LLM to the extractor
        summaries=["self"],
        prompt="生成此文档的技术摘要，突出核心贡献、创新点和关键技术",
    )

    # 创建解析管道
```

```
    # --- MODIFIED: Add SentenceSplitter to the pipeline explicitly ---
    transformations = [
        Settings.node_parser, # The global node parser (SentenceSplitter) will
split documents into nodes
        summary_extractor     # Then, SummaryExtractor will process these nodes
    ]

    # 创建处理管道
    pipeline = IngestionPipeline(
        transformations=transformations,
    )

    # 运行管道处理文档
    # --- MODIFIED: Pass documents to the pipeline instead of pre-created nodes -
--
    print("正在通过管道处理文档并提取摘要...")
    transformed_nodes = pipeline.run(documents=documents, show_progress=True)

    # Filter out any nodes that might somehow end up with None ref_doc_id (safety
check)
    initial_node_count = len(transformed_nodes)
    transformed_nodes = [node for node in transformed_nodes if node.ref_doc_id is
not None]
    if len(transformed_nodes) < initial_node_count:
        print(f"警告: {initial_node_count - len(transformed_nodes)} 个节点因
ref_doc_id 为 None 而被移除。")



    # 创建存储上下文
    storage_context = StorageContext.from_defaults()

    # 创建摘要索引
    summary_index = DocumentSummaryIndex(
        nodes=transformed_nodes,  # These nodes should now have proper ref_doc_id
        storage_context=storage_context,
        llm=llm,  # Pass LLM to the index as well
    )

    # 保存索引
    os.makedirs(storage_path, exist_ok=True)  # Ensure directory exists
    storage_context.persist(persist_dir=storage_path)
    print(f"摘要索引已保存到: {storage_path}")
    return summary_index

# 4. 创建查询引擎 (No change)
def create_query_engine(index):
    """创建摘要查询引擎"""
    return index.as_query_engine(
        response_mode="tree_summarize",
    )

# 5. 使用示例
if __name__ == "__main__":
    # 文档路径配置
    PAPER_PATHS = [
        "data/academic/1706.03762v7.pdf",  # Transformer论文
```

```
        "data/academic/2010.11929v2.pdf"   # ViT论文
    ]

    INDEX_PATH = "./paper_summary_index"

    # 步骤1: 加载文档 (不再创建TextNode，而是Document)
    documents = load_documents_from_files(PAPER_PATHS)

    if not documents:
        print("没有可处理的文档，程序退出")
        exit()

    # 步骤2: 创建摘要索引
    # Pass the list of Document objects to the create_summary_index function
    summary_index = create_summary_index(documents, INDEX_PATH)

    # 步骤3: 创建查询引擎
    query_engine = create_query_engine(summary_index)

    # 执行查询
    queries = [
        "总结Transformer的核心创新点",
        "ViT是如何处理图像的?",
        "Transformer和ViT在注意力机制应用上有何异同"
    ]

    for i, q in enumerate(queries):
        print(f"\n{'='*50}\n查询 #{i+1}: {q}\n{'-'*50}")

        try:
            response = query_engine.query(q)
            print(f"结果:\n{response.response}\n")
        except Exception as e:
            print(f"查询出错: {str(e)}")
```

输出:

```
正在加载: 1706.03762v7.pdf
正在加载: 2010.11929v2.pdf
已加载 37 个文档
正在通过管道处理文档并提取摘要...

Parsing nodes:   0%|          | 0/37 [00:00<?, ?it/s]
100%|██████████| 82/82 [04:52<00:00,  3.56s/it]

摘要索引已保存到: ./paper_summary_index

==================================================
查询 #1: 总结Transformer的核心创新点
--------------------------------------------------
结果:
Transformer的核心创新点主要包括以下几个方面：

1. **自注意力机制**：Transformer引入了多头自注意力机制，这使得模型能够并行处理输入序列，并且在处理长距离依赖关系时更加有效。
```

2. **编码器-解码器架构**：模型由编码器和解码器两部分组成，每部分都包含多个相同的层。这种结构允许模型有效地捕捉输入序列的信息，并生成相应的输出序列。

3. **位置前馈网络**：每个编码器和解码器层中都包含一个位置前馈网络，它对序列中的每个位置进行独立的线性变换，从而增强了模型的表达能力。

4. **残差连接与层归一化**：为了提高训练的稳定性和效率，Transformer在每个子层之后使用了残差连接和层归一化技术。

5. **位置编码**：由于Transformer没有递归或卷积结构，因此通过添加位置编码来为模型提供序列中元素的位置信息。

这些创新点共同使得Transformer在处理序列数据时具有高效、并行化和强大的表示能力。

==========
查询 #2: ViT是如何处理图像的?
----------
结果:
ViT处理图像的过程包括以下几个步骤：

1. **图像分割**：首先，将输入的图像分割成固定大小的块（patches）。

2. **线性嵌入**：每个块被展平并进行线性投影，生成patch embeddings。这些嵌入向量形成一个序列。

3. **位置嵌入**：为了保留位置信息，向patch embeddings中添加可学习的位置嵌入。

4. **分类标记**：在patch embeddings序列的前面添加一个额外的可学习的“分类标记”（[class] token）。这个标记在通过Transformer编码器后，作为图像的表示用于分类任务。

5. **Transformer编码器**：将包含位置嵌入和分类标记的patch embeddings序列输入到标准的Transformer编码器中。编码器由多头自注意力机制（MSA）和多层感知机（MLP）交替组成，并且在每个块之前应用层归一化（LN），之后添加残差连接。

6. **分类头**：最后，将分类标记的输出通过一个分类头进行处理。在预训练阶段，分类头是一个具有一个隐藏层的多层感知机（MLP），而在微调阶段，它

==========
查询 #3: Transformer和ViT在注意力机制应用上有何异同
----------
结果:
Transformer 和 Vision Transformer (ViT) 在注意力机制的应用上有一些共同点和不同点。

**共同点：**
- 两者都使用自注意力机制（self-attention）来处理输入数据。自注意力机制允许模型在处理每个元素时，能够关注到序列中的其他所有元素，从而捕捉长距离依赖关系。
- 自注意力机制的核心组成部分是查询（Query）、键（Key）和值（Value），这些组件在两种模型中都是通过线性变换从输入数据中生成的。

**不同点：**
- **输入数据的处理方式：**
  - Transformer 主要用于处理一维序列数据，如文本。它将输入序列分割成固定长度的片段，并对每个片段应用自注意力机制。

```
  - ViT 则专门设计用于处理二维图像数据。它将图像分割成多个小块（patches），并将每个块展平为一维向量，然后将其作为序列输入到自注意力层中。这样，ViT 能够直接应用于图像分类等任务。

- **位置编码：**
  - Transformer 使用位置编码来保留输入序列中元素的位置信息，因为自注意力机制本身不包含位置信息。
  - ViT 同样需要
```

添加支持加载已生成的 index：

```
if __name__ == "__main__":
    PAPER_PATHS = [
        "data/academic/1706.03762v7.pdf",
        "data/academic/2010.11929v2.pdf"
    ]

    INDEX_PATH = "./paper_summary_index"

    summary_index = None

    # --- 添加加载逻辑 ---
    if os.path.exists(INDEX_PATH) and os.listdir(INDEX_PATH):
        print(f"检测到已存在的索引目录: {INDEX_PATH}，尝试加载索引...")
        try:
            # 创建一个用于加载的 StorageContext
            storage_context =
StorageContext.from_defaults(persist_dir=INDEX_PATH)
            # 使用 load_index_from_storage 加载索引
            # 注意：对于DocumentSummaryIndex，你通常需要指定
type=DocumentSummaryIndex
            # 但如果它能自动推断，也可以省略。这里显式指定更安全。
            summary_index = load_index_from_storage(
                storage_context=storage_context,
                llm=llm, # 加载时也传递LLM和Embed Model
                embed_model=embed_model,
                index_type="document_summary" # 明确指定索引类型
            )
            print("索引加载成功！")
        except Exception as e:
            print(f"加载索引失败: {e}")
            print("将重新生成索引...")
            # 如果加载失败，则回退到生成新索引
            documents = load_documents_from_files(PAPER_PATHS)
            if not documents:
                print("没有可处理的文档，程序退出")
                exit()
            summary_index = create_summary_index(documents, INDEX_PATH)
    else:
        print(f"未检测到索引目录或目录为空: {INDEX_PATH}，正在生成新索引...")
        documents = load_documents_from_files(PAPER_PATHS)
        if not documents:
            print("没有可处理的文档，程序退出")
            exit()
```

```
        summary_index = create_summary_index(documents, INDEX_PATH)

    if summary_index is None:
        print("未能创建或加载索引，程序退出。")
        exit()

    # 步骤3: 创建查询引擎
    query_engine = create_query_engine(summary_index)

    # 执行查询
    queries = [
        "总结Transformer的核心创新点",
        "ViT是如何处理图像的?",
        "Transformer和ViT在注意力机制应用上有何异同"
    ]

    for i, q in enumerate(queries):
        print(f"\n{'='*50}\n查询 #{i+1}: {q}\n{'-'*50}")

        try:
            response = query_engine.query(q)
            print(f"结果:\n{response.response}\n")
        except Exception as e:
            print(f"查询出错: {str(e)}")
```

| 特征 | `ref_doc_id` | `node_id` |
| --- | --- | --- |
| **层级** | 文档级（标识源文档） | 节点级（标识单个数据单元） |
| **生成方式** | 由文档 ingestion 自动生成（或手动指定） | 由节点创建过程自动生成（或手动指定） |
| **唯一性范围** | 全局唯一（每个文档对应一个 `ref_doc_id`） | 全局唯一（每个节点对应一个 `node_id`） |
| **典型操作** | 删除文档关联的所有节点（`delete_ref_doc`） | 删除单个节点（`delete_nodes`） |
| **使用场景** | 溯源、文档级管理 | 细粒度检索、节点关系建模 |
| **无文档场景** | `None`（如 `build_index_from_nodes`） | 必须存在（节点创建时强制生成） |

## 2.8 知识图谱索引

它通过在一组文档中提取知识三元组（主语、谓语、宾语）来建立索引。 在查询时，它既可以只使用知识图谱作为上下文进行查询，也可以利用每个实体的底层文本作为上下文进行查询。通过利用底层文本，我们可以针对文档内容提出更复杂的查询。

参考文档《高级 RAG》”使用知识图谱改进 RAG 检索“章节

## 2.9 总结

| Type | Purpose | LLM Call aka Embeeding Fee |
|---|---|---|
| Vector Store Index | Very simple to use, allows answering a query over a large corpus of data | Yes |
| List Index | List index is useful for synthesizing an answer that combines information across multiple data sources | No, only when perform data retrieval |
| Tree Index | It is useful for summarizing a collection of documents | No, only when perform data retrieval |
| Keyword Table Index | It is useful for routing queries to the disparate data source | Yes |
| Composability Graph Index | It is useful for building a knowledge graph and stack multiple indicies | Yes |
| Pandas Index and SQL Index | It is useful for structured data | No |
| Document Summary Index | Premade for answer-questioning purpose. | Yes |

# 补充

## 1. Retriever_mode

LlamaIndex中不同索引类型对应的**检索器模式（Retriever Modes）**及其映射的检索器类。以下是分索引类型的详细解释：

**1. 向量索引（Vector Index）**

- **特点**：检索器模式配置**无效**（会被忽略）。
- **默认行为**：无论参数如何，调用 `vector_index.as_retriever(...)` 始终返回 `VectorIndexRetriever`，基于向量相似度进行检索。

**2. 摘要索引（Summary Index）**

通过不同模式选择检索方式：

- `default`：使用 `SummaryIndexRetriever`，基于摘要内容直接检索。
- `embedding`：使用 `SummaryIndexEmbeddingRetriever`，将摘要转为嵌入向量后检索（适合语义匹配）。
- `llm`：使用 `SummaryIndexLLMRetriever`，通过LLM生成查询来检索（适合自然语言理解）。

**3. 树索引（Tree Index）**

针对树结构数据的不同检索粒度：

- `select_leaf`：`TreeSelectLeafRetriever`，检索匹配的叶子节点（适合细粒度查询）。
- `select_leaf_embedding`：`TreeSelectLeafEmbeddingRetriever`，结合嵌入向量检索叶子节点。
- `all_leaf`：`TreeAllLeafRetriever`，检索所有叶子节点（适合全面查询）。
- `root`：`TreeRootRetriever`，直接检索根节点（适合获取整体摘要）。

**4. 关键词表索引（Keyword Table Index）**

基于关键词提取方法选择检索器：

- `default`：`KeywordTableGPTRetriever`，使用GPT生成关键词检索（精度较高）。
- `simple`：`KeywordTableSimpleRetriever`，基于简单关键词匹配（轻量级，速度快）。

- `rake`：`KeywordTableRAKERetriever`，使用RAKE算法提取关键词（适合多词短语）。

**5. 知识图谱索引（Knowledge Graph Index）**

不同模式对应知识图谱的查询方式：

- `keyword` / `embedding` / `hybrid`：均使用 `KGTableRetriever`，分别通过关键词匹配、嵌入向量或混合方式检索知识图谱数据。
    - **注**：模式名称可能仅表示输入查询的类型，但检索器类不变。

**6. 文档摘要索引（Document Summary Index）**

针对文档摘要的检索策略：

- `llm`：`DocumentSummaryIndexLLMRetriever`，通过LLM生成查询匹配摘要内容。
- `embedding`：`DocumentSummaryIndexEmbeddingRetrievers`，基于摘要的嵌入向量检索