

RAG+知识库篇

https://www.langchain.asia/modules/data_connection/indexing?utm_source=chatgpt.com

书籍云盘链接：

<https://pan.baidu.com/s/1KgTvOfzzoHoLFxxS0le0gg>

为什么需要 RAG?

LLM 应用的局限性：无法访问**私有数据** (Private data) 和缺乏**当前事件** (Current events) 的知识（知识截止日期）。直接使用这些受限的 LLM 会导致**幻觉** (hallucination)，即生成不准确或错误的信息。

因此解决思路就是在 LLM 给出答案之前，从外部知识库检索相关信息，并把这些信息作为

上下文(context),从而生成准确的信息。

RAG 的挑战

外部数据量通常远超 LLM 的**上下文窗口 (context window)** 大小限制，无法将所有信息一次性放入提示 (Prompt) 中。

因此，关键目标是在每次用户提问时，能够智能地从大量文档中

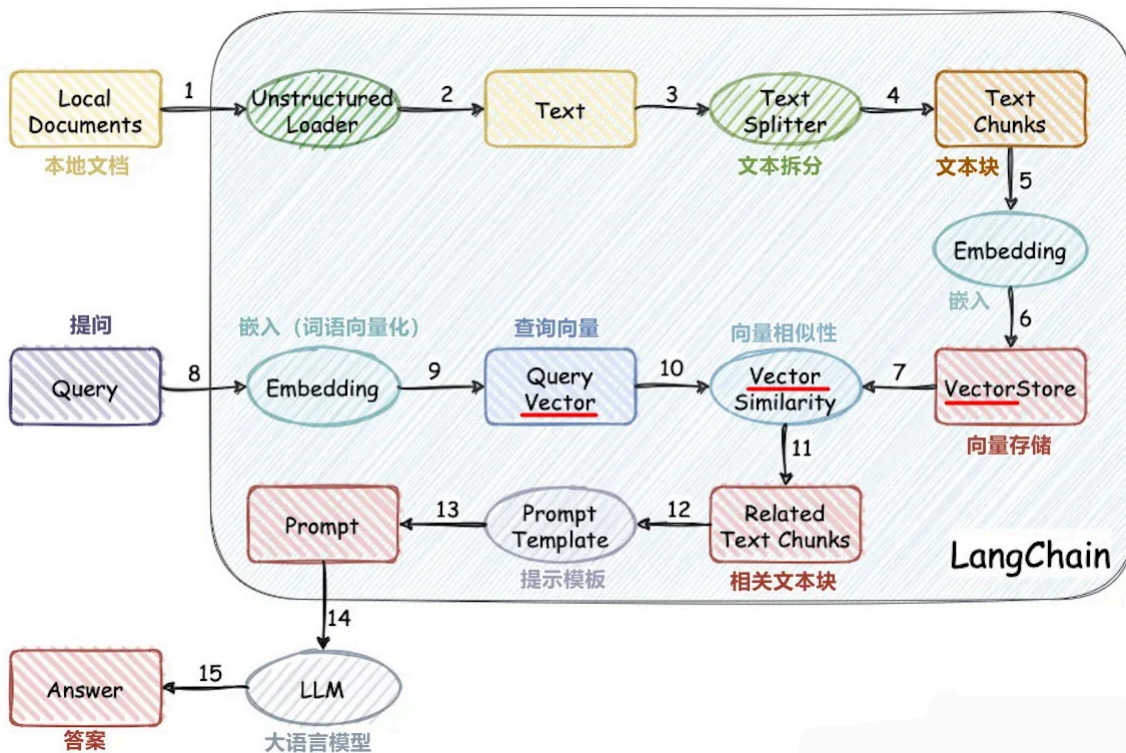
挑选出最相关的一小部分文本作为上下文提供给 LLM。

RAG 可分解为两个主要步骤

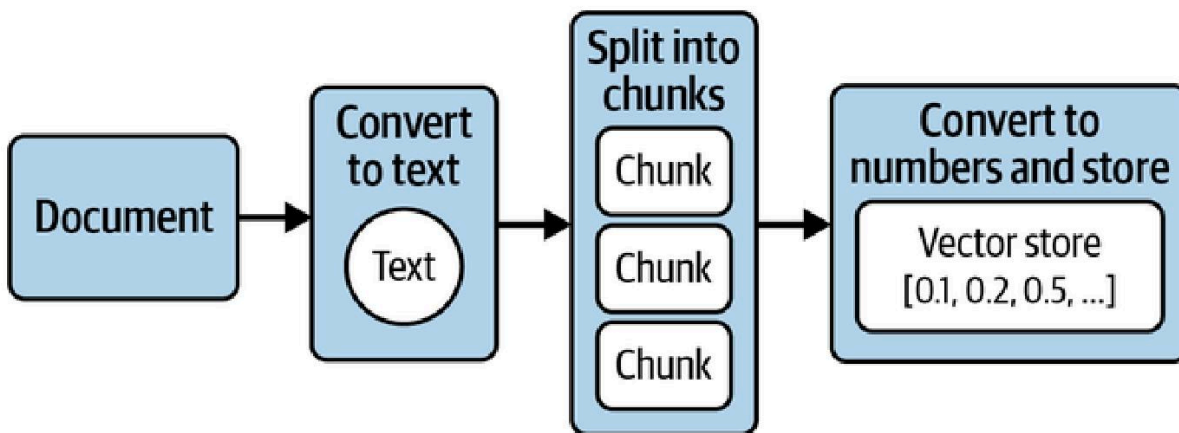
：**索引 (Indexing)** 和 **检索 (Retrieving)**。

索引

主要流程



1. **提取文本 (Extract Text):** 从原始文档（如 PDF、网页）中解析并提取纯文本内容。
2. **分割文本 (Split Text):** 将长文本分割成更小、更易于处理的**块 (chunks)**。
3. **转换为数字 (Convert to Numbers):** 将文本块转换成计算机可以理解的**嵌入向量 (embeddings)**。
4. **存储 (Store):** 将嵌入向量及其对应的原始文本块存储在专门的**向量存储 (vector store)** 中，以便后续快速检索。



提取文本

LangChain 的文档加载器从各种数据源提取文本，并将其存储在 `Document` 类中。支持的文件类型包括.txt、.csv、.json、Markdown 等，还可以从 Slack、Notion 等平台加载数据。例如，使用 `TextLoader` 加载.txt 文件，`WebBaseLoader` 加载网页 HTML 并解析为文本，`PDFLoader` 提取 PDF 文档中的文本。

加载后的文档通常是一个包含 `page_content` (文本内容) 和 `metadata` (元数据) 的 `Document` 对象列表。

```
# 递归查找加载目录中的所有txt类型的文件
loader = DirectoryLoader('./知识库/txt_files/', glob='**/*.txt', loader_cls=TextLo
# 将数据转成 document 对象，每个文件会作为一个 document
documents = loader.load()

# 从网页中导入
WebBaseLoader("https://www.langchain.com")
loader.load()
```



分割文本成块

为什么需要分割？

LLM 和嵌入模型有上下文窗口限制；将整个长文档直接嵌入或放入提示是不可行的。

分割文本的目标是将长文档分割成既小又能保持**语义相关性**的块 (chunk)。

Langchain 中包括 `CharacterTextSplitter` 和 `RecursiveCharacterTextSplitter` 两种切分文本成块的方式，两者均可自行选择 `chunk_size` 和 `chunk_overlap`。分别表示成块的大小，和相邻块之间重叠文本的大小。

比较项	CharacterTextSplitter	RecursiveCharacterTextSplitter
主要作用	把长文本按字符数切分成小块	更智能地、逐层按字符分隔符切分
切分策略	简单暴力 ：到字符数上限就切断	递归分层 ：优先按段落、句子、单词等自然边界切
是否保留语义连贯性	 容易打断句子或段落	 尽可能保持句子和段落完整
常见应用场景	- 非结构化短文本- 不在意语义完整性时	- 书籍、论文、技术文档- 需要高质量上下文片段
适合什么情况	快速、简单地切	要求高质量、有逻辑连贯的小块

```
# 初始化加载器
text_splitter = CharacterTextSplitter(chunk_size=100, chunk_overlap=0)
# 切割加载的 document
split_docs = text_splitter.split_documents(documents)
#另一种方式
splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=200,
)
splitted_docs = splitter.split_documents(docs)
```

文本嵌入 (Embedding)

和 Transformer 中的 embedding 层作用一致，是把文本映射成高维稠密向量的过程。对于语义相似的文本，其嵌入向量在高维空间中距离更近。**余弦相似度 (cosine similarity)** 作为衡量两个向量在语义上相似程度的常用方法。数值越接近 1 表示越相似

LangChain 的 Embeddings 类，它是与各种文本嵌入模型（如 OpenAI, Cohere, Hugging Face）交互的接口。该类主要提供两个方法：

`embed_documents` (接收文本列表，返回嵌入向量列表) 和 `embed_query` (接收单个查询文本，返回其嵌入向量)。

```
#将导入的文档数据转成向量
embeddings = HuggingFaceEmbeddings(model_name='shibing624/text2vec-
```

存储到向量库

向量存储 (Vector Store)：一种专门设计用来存储向量并高效执行相似性搜索（如余弦相似度计算）的数据库。

向量存储库包括很多种，如 **Chroma**、**Qdrant**、**FAISS**、**Pinecone** 等，其中有的是支持长期存储，有的只支持临时。

```
# 将 document 通过 huggingface 的 embeddings 对象计算 embedding 向量信息
docsearch = Chroma.from_documents(split_docs, embeddings)
```

跟踪文档变更

当原始文档发生变化的时候，就需要重新索引和更新向量存储中的对应内容，同时删除过时的信息。

Langchain 中管理文档变更的工具是索引 API 中的 `RecordManager` 类，它更像后端数据库的工作方式。

它会记录文档的**哈希值（基于内容和元数据）、写入时间以及源 ID（标识文档来源）**。当尝试再次索引同一来源的文档时，`RecordManager` 可以通过比较哈希值来判断文档是否发生变化，从而决定是跳过、更新还是删除旧版本。

更具体来说，它有助于：

- 避免将重复内容写入向量存储
- 避免重复写入未更改的内容
- 避免对未更改的内容重新计算嵌入

```
# 初始化记录管理器
record_manager = SQLRecordManager(namespace="my_namespace", db_url=...)
# 创建数据库模式
record_manager.create_schema()

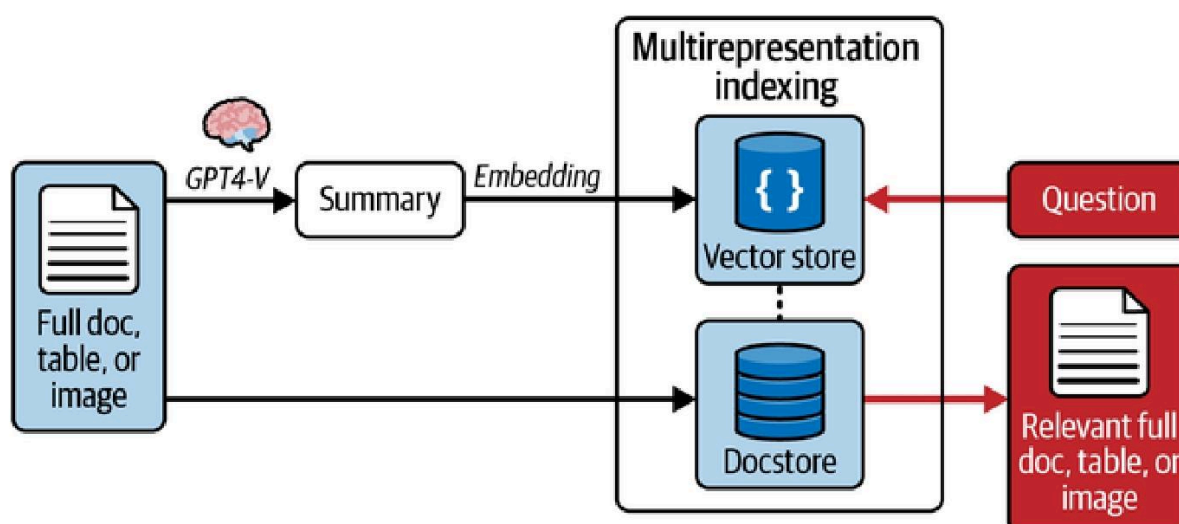
# 索引文档
indexs = index(
    docs_source=split_docs,
    record_manager=record_manager,
    vector_store=vector_store,
    cleanup="incremental", #增量删除模式，还可选择 'full', 'scoped_full'
    source_id_key="source"
)
```

模式	删除时机	删除范围	是否需要 <code>source_id_key</code>	适用场景
<code>incremental</code>	索引过程中	当前批次中未更新的文档	是	实时更新，避免重复写入
<code>full</code>	索引后	所有未在当前批次中返回的文档	否	完全替换向量存储内容
<code>scoped_full</code>	索引后	当前批次中未更新的文档	是	分批处理，保留未处理的数据

索引优化

简单的文本分割和嵌入（朴素 RAG）可能导致检索结果不一致和幻觉，尤其是在处理包含表格和图像的复杂文档时。需要更高级的索引策略来提升准确性和性能。

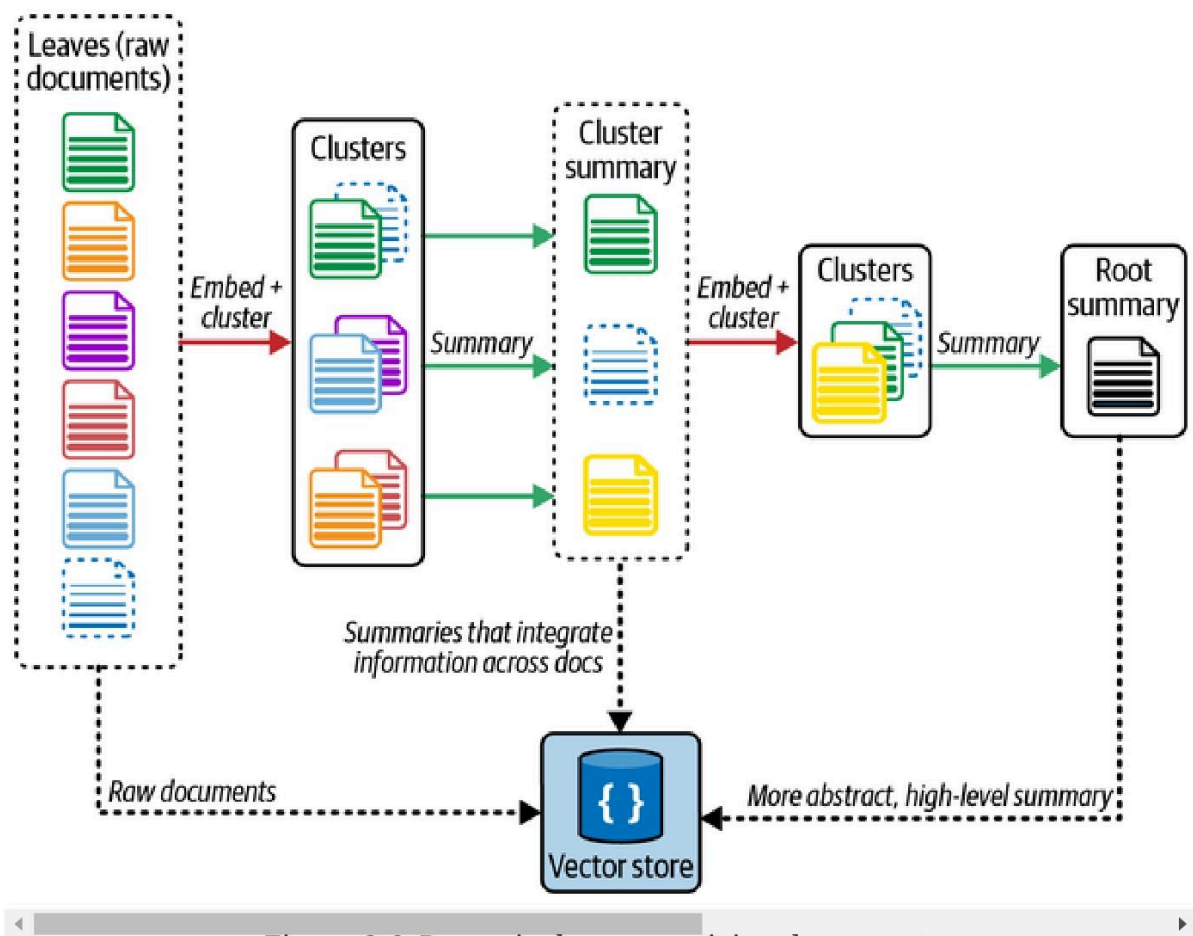
MultiVectorRetriever（多向量检索器）：



- 解决问题：如何处理文本和表格混合的文档，避免分割时丢失表格信息。
- 核心思想：**解耦**用于检索的文档（通常是摘要）和用于最终答案合成的文档（原始文档或表格）。
- 流程：为表格/文档块生成摘要 → 嵌入并存储这些摘要（包含指向原始数据的 ID）→ 将原始数据存储在单独的文档存储 (docstore) 中 → 检索时，先搜索摘要 → 根据摘要中的 ID 从 docstore 取回完整的原始数据 → 将原始数据作为上下文给 LLM。

```
# 配置 MultiVectorRetriever 指向摘要的向量存储和原始文档块的 Docstore
multivector_retriever = MultiVectorRetriever(
    vectorstore=summary_vectorstore, # 指向存储摘要嵌入的向量存储
    docstore=docstore,               # 指向存储原始文档块的 Docstore
    id_key="doc_id"                  # 指明在向量存储的 metadata 中使用哪个 key 来查
)
```

RAPTOR (Recursive Abstractive Processing for Tree-Organized Retrieval):



- 解决问题：如何处理既需要细节信息（来自单个文档块）又需要高层概括信息（跨多个文档）的问题。
- 核心思想：通过**递归聚类**和**摘要**构建一个文档信息的**树状结构**。
- 流程：对原始文档块进行嵌入和聚类 → 对每个簇生成摘要 → 对摘要再进行嵌入、聚类、生成更高层摘要，如此递归 → 将所有层级的摘要和原始文档块一起索引。

ColBERT (Optimizing Embeddings):

- 解决问题：传统定长嵌入向量会压缩信息，可能丢失细节或包含无关信息，导致幻觉。
- 核心思想：为文档和查询中的**每个 token** 生成上下文相关的嵌入向量（**token-level embeddings**），在检索时进行**后期交互 (late interaction)**。
- 流程：计算查询中每个 token 与文档中所有 token 的最大相似度 → 将这些最大相似度分数相加得到文档的总分。这种方法更细粒度，效果通常更好。
- 提供了使用 RAGatouille 库（实现了 ColBERT）进行索引和搜索的 Python 示例。