递归字符文本分割器的使用与运行流程

递归字符文本分割器的使用与运行流程

01. 递归字符文本分割器

普通的字符文本分割器只能使用单个分隔符对文本内容进行划分，在划分的过程中，可能会出现文档块`过小`或者`过大`的情况，这会让 RAG 变得不可控，例如：

1. **文档块可能会变得非常大**，极端的情况下某个块的内容长度可能就超过了 LLM 的上下文长度限制，这样这个文本块永远不会被引用到，相当于存储了数据，但是数据又丢失了。
2. **文档块可能会远远小于窗口大小**，导致文档块的信息密度太低，块内容即使填充到 Prompt 中，LLM 也无法提取出有用的信息。

那么有没有一种分割方案，可以解决这个问题呢？按照`分隔符`初次分割的时候，去检测块内容，如果太大就按照提供的`备选分隔符`二次分割，如果太小则合并前后的块，最后让所有的块内容长度都控制在指定的大小并尽可能接近呢？

在 LangChain 中就为这种方案提供了一个分割器组件——`RecursiveCharacterTextSplitter`，即**递归字符串分割**，这个分割器可以传递`一组分隔符`和`设定块内容大小`，根据分隔符的优先顺序对文本进行预分割，然后将小块进行合并，将大块进行递归分割，直到获得所需块的大小，最终这些文档块的大小并不能完全相同，但是仍然会逼近指定长度。

RecursiveCharacterTextSplitter 的分隔符参数默认为`["\n\n", "\n", " ", ""]`，即优先使用换两行的数据进行分割，然后在使用单个换行符，如果块内容还是太大，则使用空格，最后再拆分成单个字符。

所以如果使用默认参数，这个字符文本分割器最后得到的文档块长度一定不会超过预设的大小，但是仍然会有小概率出现远小于的情况（目前也没有很好的解决方案）。

例如使用递归字符文本分割器修改上节课的需求，示例代码如下：

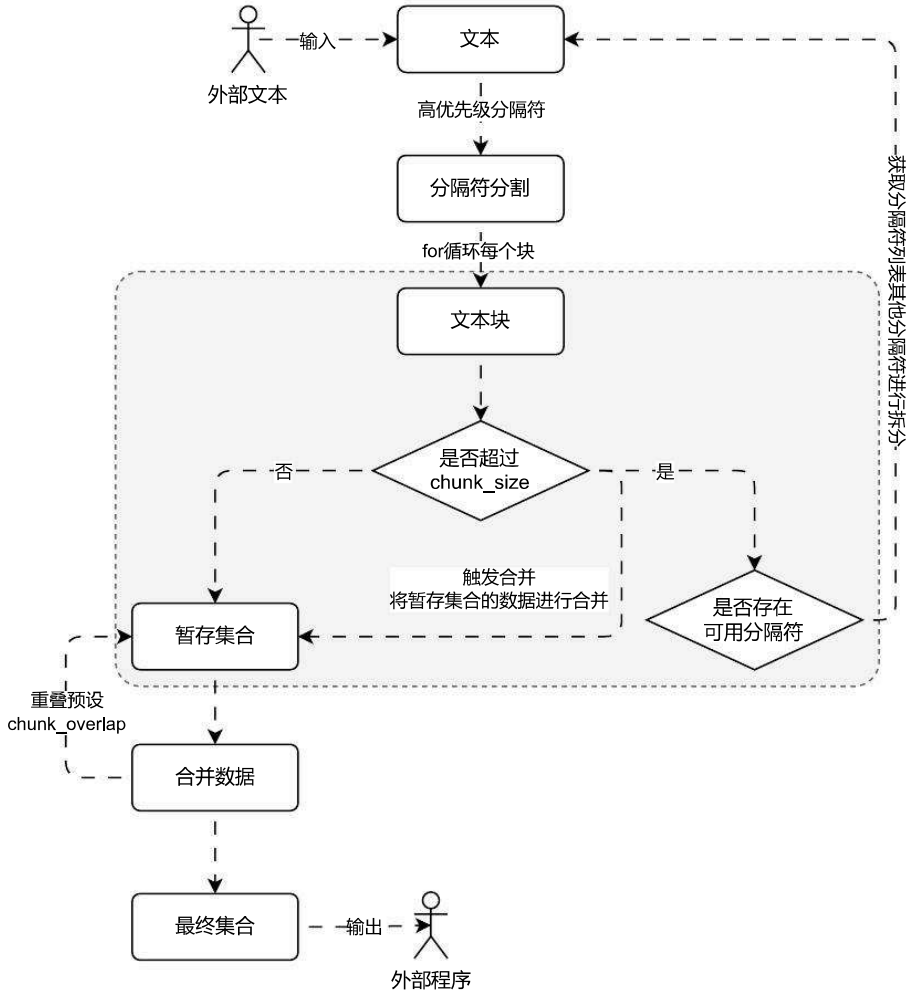
```
from langchain_community.document_loaders import UnstructuredMarkdownLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter

loader = UnstructuredMarkdownLoader("./项目API文档.md")
documents = loader.load()

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=500,
    chunk_overlap=50,
    add_start_index=True,
)
chunks = text_splitter.split_documents(documents)

for chunk in chunks:
    print(f"块大小:{len(chunk.page_content)}, 块元数据:{chunk.metadata}")
```

输出内容：

```
块大小:251, 块元数据:{'source': './项目API文档.md', 'start_index': 0}
块大小:451, 块元数据:{'source': './项目API文档.md', 'start_index': 246}
块大小:490, 块元数据:{'source': './项目API文档.md', 'start_index': 699}
块大小:305, 块元数据:{'source': './项目API文档.md', 'start_index': 1165}
块大小:435, 块元数据:{'source': './项目API文档.md', 'start_index': 1472}
```

```
块大小:497, 块元数据:{'source': './项目API文档.md', 'start_index': 1859}
块大小:237, 块元数据:{'source': './项目API文档.md', 'start_index': 2359}
块大小:483, 块元数据:{'source': './项目API文档.md', 'start_index': 2598}
块大小:486, 块元数据:{'source': './项目API文档.md', 'start_index': 3092}
块大小:438, 块元数据:{'source': './项目API文档.md', 'start_index': 3580}
块大小:293, 块元数据:{'source': './项目API文档.md', 'start_index': 4013}
块大小:498, 块元数据:{'source': './项目API文档.md', 'start_index': 4261}
块大小:463, 块元数据:{'source': './项目API文档.md', 'start_index': 4712}
块大小:438, 块元数据:{'source': './项目API文档.md', 'start_index': 5129}
块大小:474, 块元数据:{'source': './项目API文档.md', 'start_index': 5569}
块大小:93, 块元数据:{'source': './项目API文档.md', 'start_index': 6018}
块大小:464, 块元数据:{'source': './项目API文档.md', 'start_index': 6113}
块大小:478, 块元数据:{'source': './项目API文档.md', 'start_index': 6579}
块大小:379, 块元数据:{'source': './项目API文档.md', 'start_index': 7035}
块大小:489, 块元数据:{'source': './项目API文档.md', 'start_index': 7416}
```

`RecursiveCharacterTextSplitter` 底层的运行流程其实也非常简单，可以拆分成 `预分割`、`大文档块递归分割`、`小文档块合并`。运行流程图如下（需掌握，目前 LLM 应用开发中高频提问到的一个问题）：



对比普通的字符文本分割器， 递归字符文本分割器可以传递多个分隔符，并且根据不同分隔符的优先级来执行相应的分割。在 LangChain 中通过 RecursiveCharacterTextSplitter 类实现对文本的递归字符串分割

02. 衍生代码分割器

递归字符文本分割器的核心部分在于传递不同的分割符列表，通过不同的优先级的列表，可以实现一些复杂文件的拆分，例如在该分割器内部预先构建了大量的的分割列表，用于在特定的编程语言中拆分文本。

支持的编程语言类型存储在 `langchain_text_splitters.Language` 枚举中，涵盖如下：

```
class Language(str, Enum):
    """Enum of the programming languages."""

    CPP = "cpp"
    GO = "go"
    JAVA = "java"
    KOTLIN = "kotlin"
    JS = "js"
    TS = "ts"
    PHP = "php"
    PROTO = "proto"
    PYTHON = "python"
    RST = "rst"
    RUBY = "ruby"
    RUST = "rust"
    SCALA = "scala"
    SWIFT = "swift"
    MARKDOWN = "markdown"
    LATEX = "latex"
    HTML = "html"
    SOL = "sol"
    CSHARP = "csharp"
    COBOL = "cobol"
    C = "c"
    LUA = "lua"
    PERL = "perl"
    HASKELL = "haskell"
```

要想查看给定语言的分隔符，可以使用 `RecursiveCharacterTextSplitter.get_separators_for_language()` 函数获取，例如查看 Python 语言的分隔符，如下：

```
from langchain_text_splitters import Language, RecursiveCharacterTextSplitter

separators =
RecursiveCharacterTextSplitter.get_separators_for_language(Language.PYTHON)
print(separators)

# 输出
['\nclass ', '\ndef ', '\n\tdef ', '\n\n', '\n', ' ', '']
```

可以从分隔符列表中看到 Python 文件的分割逻辑，优先将所有类都分割出来，然后分割函数，接下来分割类方法、模块语句等内容。

要想使用这些 `编程语言分隔符` 其实非常简单，在构造分割器的时候传递即可，或者使用 `from_language()` 并传递编程语言枚举数据也可以实现，示例如下：

```python
from langchain_community.document_loaders import UnstructuredFileLoader
from langchain_text_splitters import Language, RecursiveCharacterTextSplitter

loader = UnstructuredFileLoader("./demo.py")
text_splitter = RecursiveCharacterTextSplitter.from_language(
    Language.PYTHON,
    chunk_size=500,
    chunk_overlap=50,
)

documents = loader.load()
chunks = text_splitter.split_documents(documents)

for chunk in chunks:
    print(f"块大小: {len(chunk.page_content)}, 元数据:{chunk.metadata}")
```

输出内容:

```
块大小: 151, 元数据:{'source': './demo.py'}
块大小: 335, 元数据:{'source': './demo.py'}
块大小: 439, 元数据:{'source': './demo.py'}
块大小: 499, 元数据:{'source': './demo.py'}
块大小: 108, 元数据:{'source': './demo.py'}
块大小: 187, 元数据:{'source': './demo.py'}
块大小: 352, 元数据:{'source': './demo.py'}
块大小: 450, 元数据:{'source': './demo.py'}
块大小: 446, 元数据:{'source': './demo.py'}
块大小: 431, 元数据:{'source': './demo.py'}
块大小: 347, 元数据:{'source': './demo.py'}
块大小: 492, 元数据:{'source': './demo.py'}
块大小: 491, 元数据:{'source': './demo.py'}
块大小: 471, 元数据:{'source': './demo.py'}
块大小: 489, 元数据:{'source': './demo.py'}
块大小: 474, 元数据:{'source': './demo.py'}
块大小: 476, 元数据:{'source': './demo.py'}
块大小: 492, 元数据:{'source': './demo.py'}
块大小: 472, 元数据:{'source': './demo.py'}
块大小: 490, 元数据:{'source': './demo.py'}
块大小: 469, 元数据:{'source': './demo.py'}
块大小: 452, 元数据:{'source': './demo.py'}
块大小: 490, 元数据:{'source': './demo.py'}
块大小: 497, 元数据:{'source': './demo.py'}
块大小: 492, 元数据:{'source': './demo.py'}
块大小: 496, 元数据:{'source': './demo.py'}
块大小: 497, 元数据:{'source': './demo.py'}
块大小: 498, 元数据:{'source': './demo.py'}
块大小: 491, 元数据:{'source': './demo.py'}
块大小: 466, 元数据:{'source': './demo.py'}
块大小: 348, 元数据:{'source': './demo.py'}
```

递归分割会尽可能从大分块上保证数据的连续性，打印其中一个分块，示例如下:

```python
print(chunks[2].page_content)
```

输出内容：

```
def split_text(self, text: str) -> List[str]:

"""Split incoming text and return chunks."""

# First we naively split the large input into a bunch of smaller ones.

separator = (

self._separator if self._is_separator_regex else re.escape(self._separator)

)

splits = _split_text_with_regex(text, separator, self._keep_separator)

_separator = "" if self._keep_separator else self._separator

return self._merge_splits(splits, _separator)
```

03. 中文场景下的递归分割

`RecursiveCharacterTextSplitter` 默认配置的分隔符均是英文场合下的，在中文场合下，除了换行/空格，一般还有更加复杂的语句结束判断标识，例如：`。`、`！`、`？`等标识符，如果想更好去切割 `中英文文档`，可以考虑重设分隔符列表（或者继承该类进行重写）。

不同符号的优先级如下：

1. `\n\n`：换行两次优先级最高。
2. `\n`：普通换行符优先级其次，一般切断后都不会导致上下文语义丢失。
3. `。|！|？`：中文中句号、感叹号、问号一般都表示句子结束，也可以尝试切割。
4. `\.\s|\!\s|\?\s`：对应到英文中就是点、感叹号、问号，并且标准的英文写法在这些符号后通常需要添加空格。
5. `；|;\s`：其次就是中英文的分段，在英文分段后一般会添加空格；
6. `，|,\s`：接下来优先级是中英文中的逗号，逗号一般都表示句子语义还未结束，所以一般不切割，除非文本块仍然超过大小。
7. ` `：空格和空字符串是优先级最低的切割符号之一，特别是在英文场合中，有时候两个词才有意义，切割出来意义就不大，而空字符串则是优先级最低的切割符，空字符串会把中文切割成单个汉字，在英文场合下切割成单个字母，几乎完全丢失语义。

更改后的示例如下：

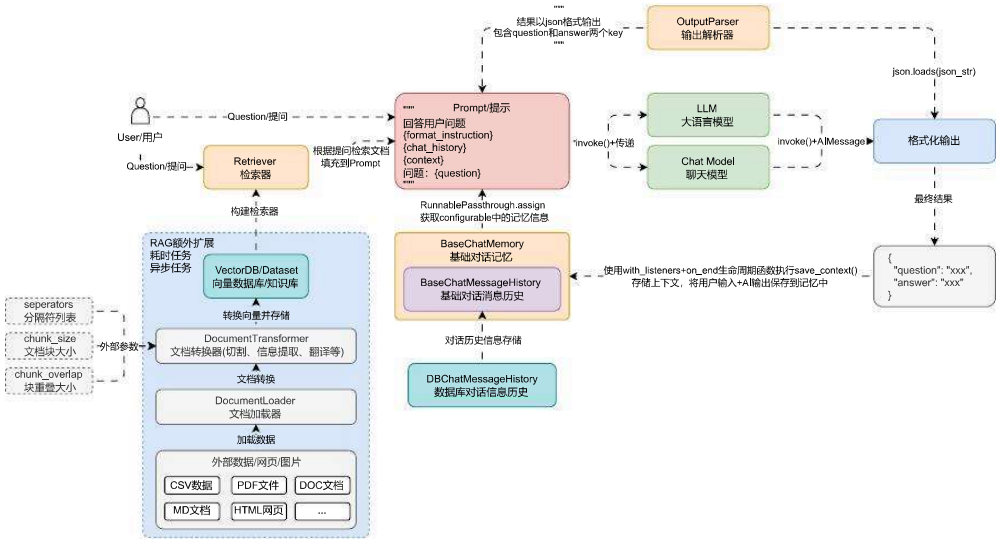
```
from langchain_community.document_loaders import UnstructuredMarkdownLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter

# 1.创建加载器和文本分割器
loader = UnstructuredMarkdownLoader("./项目API文档.md")
text_splitter = RecursiveCharacterTextSplitter(
    separators=[
        "\n\n",
        "\n",
```

```
        "。|！|？",
        "\.\s|\!\s|\?\s",  # 英文标点符号后面通常需要加空格
        "；|;\s",
        "，|,\s",
        " ",
        ""
    ],
    is_separator_regex=True,
    chunk_size=500,
    chunk_overlap=50,
    add_start_index=True,
)

# 2.加载文档与分割
documents = loader.load()
chunks = text_splitter.split_documents(documents)

# 3.输出信息
for chunk in chunks:
    print(f"块大小: {len(chunk.page_content)}, 元数据: {chunk.metadata}")
```

输出示例:

```
块大小: 251, 元数据: {'source': './项目API文档.md', 'start_index': 0}
块大小: 451, 元数据: {'source': './项目API文档.md', 'start_index': 246}
块大小: 490, 元数据: {'source': './项目API文档.md', 'start_index': 699}
块大小: 305, 元数据: {'source': './项目API文档.md', 'start_index': 1165}
块大小: 435, 元数据: {'source': './项目API文档.md', 'start_index': 1472}
块大小: 497, 元数据: {'source': './项目API文档.md', 'start_index': 1859}
块大小: 237, 元数据: {'source': './项目API文档.md', 'start_index': 2359}
块大小: 483, 元数据: {'source': './项目API文档.md', 'start_index': 2598}
块大小: 486, 元数据: {'source': './项目API文档.md', 'start_index': 3092}
块大小: 438, 元数据: {'source': './项目API文档.md', 'start_index': 3580}
块大小: 293, 元数据: {'source': './项目API文档.md', 'start_index': 4013}
块大小: 498, 元数据: {'source': './项目API文档.md', 'start_index': 4261}
块大小: 463, 元数据: {'source': './项目API文档.md', 'start_index': 4712}
块大小: 438, 元数据: {'source': './项目API文档.md', 'start_index': 5129}
块大小: 474, 元数据: {'source': './项目API文档.md', 'start_index': 5569}
块大小: 93, 元数据: {'source': './项目API文档.md', 'start_index': 6018}
块大小: 464, 元数据: {'source': './项目API文档.md', 'start_index': 6113}
块大小: 478, 元数据: {'source': './项目API文档.md', 'start_index': 6579}
块大小: 379, 元数据: {'source': './项目API文档.md', 'start_index': 7035}
块大小: 489, 元数据: {'source': './项目API文档.md', 'start_index': 7416}
```

在 LLMOps 项目中，具体的文本分割逻辑由创建知识库的用户决定，所以 `分隔符`、`文档块大小` 和 `块重叠大小` 均是通过外部传递，然后再生成 `RecursiveCharacterTextSplitter` 分割器，而文档加载器使用 `扩展名` + `通用非结构化文件加载器` 来实现，更新后的聊天机器人的架构运行流程如下:


"""
结果以json格式输出
包含question和answer两个key
"""
OutputParser
输出解析器
json.loads(json_str)
User/用户
Question/提问
Question/提问
Retriever
检索器
根据提问检索文档
填充到Prompt
"""
Prompt/提示
回答用户问题
{format_instruction}
{chat_history}
{context}
问题：{question}
"""
'invoke()+传递
LLM
大语言模型
Chat Model
聊天模型
invoke()+AIMessage
格式化输出
最终结果
{
"question": "xxx",
"answer": "xxx"
}
构建检索器
RunnablePassthrough.assign
获取configurable中的记忆信息
BaseChatMemory
基础对话记忆
BaseChatMessageHistory
基础对话消息历史
使用with_listeners+on_end生命周期函数执行save_context()
存储上下文，将用户输入+AI输出保存到记忆中
对话历史信息存储
DBChatMessageHistory
数据库对话信息历史
RAG额外扩展
耗时任务
异步任务
VectorDB/Dataset
向量数据库/知识库
转换向量并存储
DocumentTransformer
文档转换器(切割、信息提取、翻译等)
文档转换
DocumentLoader
文档加载器
加载数据
外部数据/网页/图片
CSV数据
PDF文件
DOC文档
MD文档
HTML网页
...
seperators
分隔符列表
chunk_size
文档块大小
chunk_overlap
块重叠大小
外部参数

语义文档分割器与其他内容分割器的使用

语义文档分割器与其他文档分割器的使用

01. 语义文档分割器的使用与背景

在前面课时中使用的文档分割器都是使用 `特定字符` 对文本进行拆分，这种拆分模式虽然考虑了文档中的上下文切断的问题，但是并没有考虑句子之间的语义相似性，如果有一篇长文本，需要将其分割成语义相关的块，以便更好地理解和处理，这个时候可以使用 LangChain 中的 `语义相似性分割器(SemanticChunker)` 来实现这个任务。

`语义相似性分割器` 目前仍处于实验性，这个类目前位于 `langchain_experimental` 包中（这个包中的类与方法未来极大概率会发生变更，需要谨慎使用），安装命令：

```
pip install -Uqq langchain_experimental
```

`SemanticChunker` 在使用上和其他的文档分割器存在一些差异，并且该类并没有继承 `TextSplitter`，实例化参数含义如下：

1. `embeddings`：文本嵌入模型，在该分类器底层使用向量的 `余弦相似度` 来识别语句之间的相似性。
2. `buffer_size`：文本缓冲区大小，默认为 1，即在计算相似性时，该文本会叠加前后各 1 条文本，如果不够则不叠加（例如第 1 条和最后 1 条）。
3. `add_start_index`：是否添加起点索引，默认为 False。
4. `breakpoint_threshold_type`：断点阈值类型，默认为 `percentile` 即百分位
5. `breakpoint_threshold_amount`：端点阈值金额/得分。
6. `number_of_chunks`：分割后的文档块个数，默认为 None。
7. `sentence_split_regex`：句子切割正则，默认为 (?<=[.?!])\s+，即以英文的点、问号、感叹号切割语句，不同的文档需要传递不同的切割正则表达式。

例如想要将 `科幻短篇.txt` 按照语义切割成 10 个文档，可以使用如下代码示例：

```
import dotenv
from langchain_community.document_loaders import UnstructuredFileLoader
from langchain_experimental.text_splitter import SemanticChunker
from langchain_openai import OpenAIEmbeddings

dotenv.load_dotenv()

# 1.构建加载器和文本分割器
loader = UnstructuredFileLoader("./科幻短篇.txt")
text_splitter = SemanticChunker(
    embeddings=OpenAIEmbeddings(model="text-embedding-3-small"),
    sentence_split_regex=r"(?<=[。？！])",
    number_of_chunks=10,
)

# 2.加载文本与分割
documents = loader.load()
chunks = text_splitter.split_documents(documents)

for chunk in chunks:
```

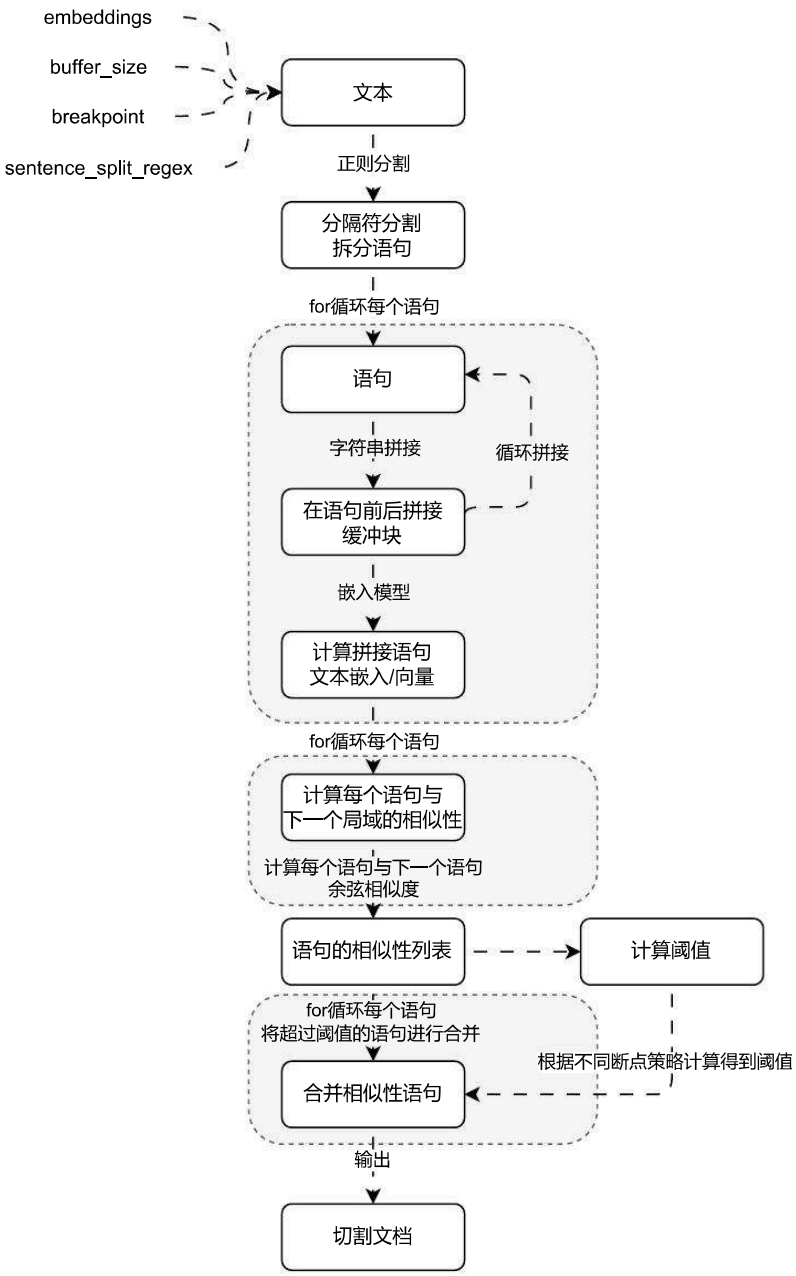
```
    print(f"块大小: {len(chunk.page_content)}, 元数据: {chunk.metadata}")
```

输出内容:

```
块大小: 201, 元数据: {'source': './科幻短篇.txt'}
块大小: 25, 元数据: {'source': './科幻短篇.txt'}
块大小: 31, 元数据: {'source': './科幻短篇.txt'}
块大小: 46, 元数据: {'source': './科幻短篇.txt'}
块大小: 203, 元数据: {'source': './科幻短篇.txt'}
块大小: 19, 元数据: {'source': './科幻短篇.txt'}
块大小: 91, 元数据: {'source': './科幻短篇.txt'}
块大小: 466, 元数据: {'source': './科幻短篇.txt'}
块大小: 116, 元数据: {'source': './科幻短篇.txt'}
块大小: 0, 元数据: {'source': './科幻短篇.txt'}
```

`SemanticChunker` 的原理其实非常简单，核心思想是将文档拆分成独立的每一句，接下来根据传递的缓冲大小前后拼接字符串，然后计算拼接后的新字符串的文本嵌入/向量，然后计算这些文本的相似度，并根据传入的分块数+断点类型计算得到一个阈值，最后将相似度超过某个阈值的合并到一起，从而实现相似度分割。

目前在 `SemanticChunker` 底层检测相似度阈值的方法有 4 种：`百分位数(默认)`、`标准差`、`四分位数`、`梯度`。



02. 其他文档分割器的使用

除了上述的文档分割器，在 LangChain 中还封装了一些其他场合下的分割器（使用频率不高），涵盖了：**基于 HTML 标题/段的分割器、Markdown 标题分割器、递归 JSON 分割器、基于 Token 计数的分割器**等，使用起来和字符文本分割器非常接近。

LangChain 文档分割器翻译文档：https://imooc-langchain.shortvar.com/docs/how_to/#检索器

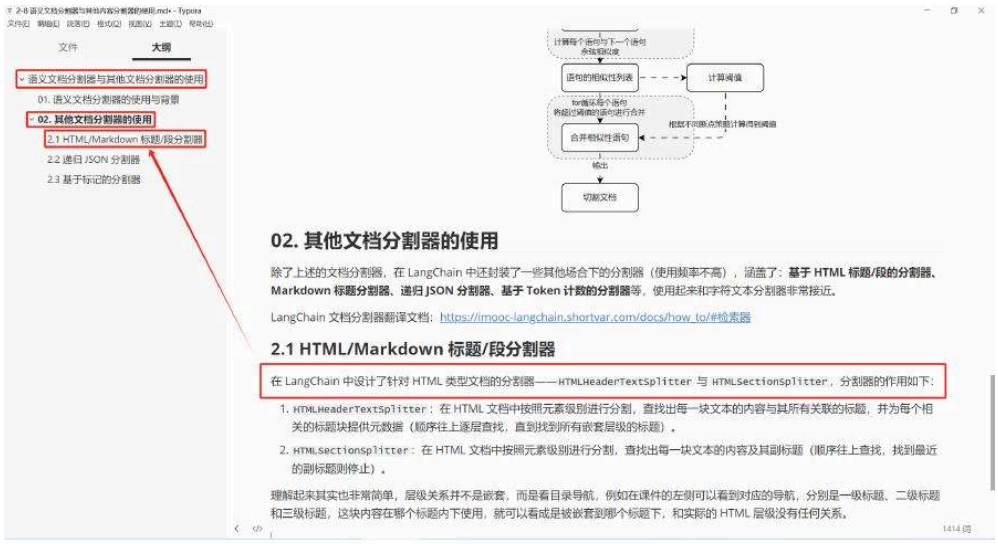
2.1 HTML/Markdown 标题/段分割器

在 LangChain 中设计了针对 HTML 类型文档的分割器——`HTMLHeaderTextSplitter` 与 `HTMLSectionSplitter`，分割器的作用如下：

1. `HTMLHeaderTextSplitter`：在 HTML 文档中按照元素级别进行分割，查找出每一块文本的内容与其所有关联的标题，并为每个相关的标题块提供元数据（顺序往上逐层查找，直到找到所有嵌套层级的标题）。
2. `HTMLSectionSplitter`：在 HTML 文档中按照元素级别进行分割，查找出每一块文本的内容及其副标题（顺序往上查找，找到最近的副标题则停止）。

理解起来其实也非常简单，层级关系并不是嵌套，而是看目录导航，例如在课件的左侧可以看到对应的导航，分别是一级标题、二级标题和三级标题，这块内容在哪个标题内下使用，就可以看成是被嵌套到哪个标题下，和实际的 HTML 层级没有任何关系。

如下，红圈的部分被嵌套的 `语义文档分割器与其他文档分割器的使用-02. 其他文档分割器的使用-2.1 HTML/Markdown 标题/段分割器` 下。



使用示例：

```
from langchain_text_splitters import HTMLHeaderTextSplitter

html_string = """
<!DOCTYPE html>
<html>
<body>
    <div>
        <h1>标题1</h1>
        <p>关于标题1的一些介绍文本。</p>
        <div>
            <h2>子标题1</h2>
            <p>关于子标题1的一些介绍文本。</p>
            <h3>子子标题1</h3>
            <p>关于子子标题1的一些文本。</p>
            <h3>子子标题2</h3>
            <p>关于子子标题2的一些文本。</p>
        </div>
```

```
        <div>
            <h3>子标题2</h2>
            <p>关于子标题2的一些文本。</p>
        </div>
        <br>
        <p>关于标题1的一些结束文本。</p>
    </div>
</body>
</html>
"""
headers_to_split_on = [
    ("h1", "一级标题"),
    ("h2", "二级标题"),
    ("h3", "三级标题"),
]
html_splitter = HTMLHeaderTextSplitter(headers_to_split_on)
html_header_splits = html_splitter.split_text(html_string)

print(html_header_splits)
```

输出内容：

```
[
Document(page_content='标题1'),
Document(metadata={'一级标题': '标题1'}, page_content='关于标题1的一些介绍文本。  \n子标题1 子子标题1 子子标题2'),
Document(metadata={'一级标题': '标题1', '二级标题': '子标题1'}, page_content='关于子标题1的一些介绍文本。'),
Document(metadata={'一级标题': '标题1', '二级标题': '子标题1', '三级标题': '子子标题1'}, page_content='关于子子标题1的一些文本。'),
Document(metadata={'一级标题': '标题1', '二级标题': '子标题1', '三级标题': '子子标题2'}, page_content='关于子子标题2的一些文本。'),
Document(metadata={'一级标题': '标题1'}, page_content='子标题2'),
Document(metadata={'一级标题': '标题1', '三级标题': '子标题2'}, page_content='关于子标题2的一些文本。'),
Document(metadata={'一级标题': '标题1'}, page_content='关于标题1的一些结束文本。')
]
```

另外在 LangChain 中除了 HTML 类型的文档可以使用这套分割规则，Markdown 类的文件也有类似的分割规则，可以使用 Markdown 标题分割器——`MarkdownHeaderTextSplitter` 完成同样的文档分割。

详细文档：https://imooc-langchain.shortvar.com/docs/how_to/markdown_header_metadata_splitter/

2.2 递归 JSON 分割器

对于 JSON 类的数据，在 LangChain 中也封装了一个递归 JSON 分割器——`RecursiveJsonSplitter`，这个分割器会按照深度优先的方式遍历 JSON 数据，并构建较小的 JSON 块，而且尽可能保持嵌套 JSON 对象完整，但如果需要保持文档块大小在最小块大小和最大块大小之间，则会将它们拆分。

在 JSON 数据中，如果值不是嵌套的 JSON，而是一个非常大的字典，则不会对该字符串进行拆分，可以配合 `递归字符文本分割器` 强制性拆分字符串，确保块大小在限制的范围内。

`RecursiveJsonSplitter` 的参数非常简单，只需传递 `max_chunk_size` 和 `min_chunk_size(可选)` 即可。

例如这里有一个很大的 json 文本（https://api.smith.langchain.com/openapi.json），使用递归 JSON 分割器对其进行拆分，如下：

```
import requests
from langchain_text_splitters import RecursiveJsonSplitter

# 1.获取并加载json
url = "https://api.smith.langchain.com/openapi.json"
json_data = requests.get(url).json()

# 2.递归JSON分割器
text_splitter = RecursiveJsonSplitter(max_chunk_size=300)

# 3.分割json数据并创建文档
json_chunks = text_splitter.split_json(json_data=json_data)
chunks = text_splitter.create_documents(json_chunks)

for chunk in chunks[:3]:
    print(chunk)
```

输出内容：

```
page_content='{"openapi": "3.1.0", "info": {"title": "LangSmith", "version":
"0.1.0"}, "paths": {"/api/v1/sessions/{session_id}": {"get": {"tags": ["tracer-
sessions"], "summary": "Read Tracer Session", "description": "Get a specific
session."}}}}'
page_content='{"paths": {"/api/v1/sessions/{session_id}": {"get": {"operationId":
"read_tracer_session_api_v1_sessions__session_id__get", "security": [{"API Key":
[]}, {"Tenant ID": []}, {"Bearer Auth": []}]}}}}'
page_content='{"paths": {"/api/v1/sessions/{session_id}": {"get": {"parameters":
[{"name": "session_id", "in": "path", "required": true, "schema": {"type":
"string", "format": "uuid", "title": "Session Id"}}, {"name": "include_stats",
"in": "query", "required": false, "schema": {"type": "boolean", "default": false,
"title": "Include Stats"}}, {"name": "accept", "in": "header", "required": false,
"schema": {"anyOf": [{"type": "string"}, {"type": "null"}], "title":
"Accept"}}]}}}}'
```

`RecursiveJsonSplitter` 分割器的运行流程其实也非常简单，这个分割器会按照 `深度优先` 的方式遍历整个 JSON，即一层一层往下读取数据，然后将对应的数据提取生成一个新的 JSON，直到数据大小接近块大小（极端情况下还是会超过预设的块大小，例如 JSON 数据中的 Key 很长，亦或者 Value 很长，甚至出现单条数据就超过了预设大小）。

所以如果要使用该分割器，一般会结合 `RecursiveCharacterTextSplitter` 降低单条数据超过预设大小的风险，思路就是将递归 JSON 分割器生成的文档列表进行二次分割。

2.3 基于标记的分割器

对于大语言模型来说，上下文的长度计算应该通过 token 进行计算，而不是通过字符长度 `len()` 函数，在 OpenAI 的 GPT 模型中，一个汉字大约等于 1.5 个 Token，一个单词为 1 个 Token，所以使用 `len()` 函数可能会导致很大的误差。

在 LLM 应用开发中，不同的模型对于 Token 的计算并不相同，但是可以使用 `tiktoken` 这个包来大致计算文本的 `token` 数，误差也相对较小，首先安装 `tiktoken` 包，命令如下：

```
pip install -U tiktoken
```

接下来定义一个基于 `tiktoken` 的长度计算函数，如下：

```
def calculate_token_count(query: str) -> int:
    """计算传入文本的token数"""
    encoding = tiktoken.encoding_for_model("text-embedding-3-large")
    return len(encoding.encode(query))
```

然后将该函数传递给分割器的 `length_function`，例如前几节课时的案例，修改优化后，代码如下：

```
import tiktoken
from langchain_community.document_loaders import UnstructuredFileLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter

# 1.定义加载器和文本分割器
loader = UnstructuredFileLoader("./科幻短篇.txt")
text_splitter = RecursiveCharacterTextSplitter(
    separators=[
        "\n\n",
        "\n",
        "。|！|？",
        "\.\s|\!\s|\?\s",  # 英文标点符号后面通常需要加空格
        "；|;\s",
        "，|,\s",
        " ",
        ""
    ],
    is_separator_regex=True,
    chunk_size=500,
    chunk_overlap=50,
    length_function=calculate_token_count,
)

# 2.加载文档并执行分割
documents = loader.load()
chunks = text_splitter.split_documents(documents)

# 3.循环打印分块内容
for chunk in chunks:
    print(f"块大小: {len(chunk.page_content)}, 元数据: {chunk.metadata}")
```

输出内容：

```
块大小: 334, 元数据: {'source': './科幻短篇.txt'}
块大小: 409, 元数据: {'source': './科幻短篇.txt'}
块大小: 372, 元数据: {'source': './科幻短篇.txt'}
块大小: 95, 元数据: {'source': './科幻短篇.txt'}
```

在 LangChain 中，除了传递 `length_function` 方法，还可以直接调用分割器的类方法 `from_tiktoken_encoder()` 来快速创建基于 `tiktoken` 分词器的文本分割器（确保分词器使用的模型和开发的 LLM 保持一致即可），例如：

```
RecursiveCharacterTextSplitter.from_tiktoken_encoder(
    model_name="gpt-4",
    chunk_size=500,
    chunk_overlap=50,
    separators=[
        "\n\n",
        "\n",
        "。|！|？",
        "\.\s|\!\s|\?\s",  # 英文标点符号后面通常需要加空格
        "；|;\s",
        "，|,\s",
        " ",
        ""
    ],
    is_separator_regex=True,
)
```

自定义 LangChain 文档分割器技巧

自定义 LangChain 文档分割器技巧

01. 自定义文档分割器

在 LangChain 中，如果内置的文档分割器均没办法完成需求，还可以根据特定的需求实现自定义文档分割器（一般极少），实现的方法也非常简单，继承文本分割器基类 `TextSplitter`，在构造函数中传递相关参数，然后实现 `split_text()` 方法即可。

例如，实现一个根据传递的分隔符实现对文档进行片段划分，并且将分割出来的文档片段转换成 N 个关键词的分割器，代码示例：

```
from typing import List

import jieba.analyse
from langchain_community.document_loaders import UnstructuredFileLoader
from langchain_text_splitters import TextSplitter


class CustomTextSplitter(TextSplitter):
    """自定义文本分割器"""

    def __init__(self, seperator: str, top_k: int = 10, **kwargs) -> None:
        super().__init__(**kwargs)
        self._seperator = seperator
        self._top_k = top_k

    def split_text(self, text: str) -> List[str]:
        """分割传入的文本为字符串列表"""
        split_texts = text.split(self._seperator)
        text_keywords = []
        for split_text in split_texts:
            text_keywords.append(
                jieba.analyse.extract_tags(split_text, self._top_k)
            )

        return [",".join(keywords) for keywords in text_keywords]


# 1.创建加载器与分割器
loader = UnstructuredFileLoader("./科幻短篇.txt")
text_splitter = CustomTextSplitter("\n\n")

# 2.加载文档并分割
documents = loader.load()
chunks = text_splitter.split_documents(documents)

# 3.循环遍历文档信息
for chunk in chunks:
    print(f"文档库块内容:{chunk.page_content}")
```

生成内容：

```
文档库块内容:星际,穿越,标题
```

```
文档库块内容:概要
文档库块内容:星系,李昂,文明,这个,高度发达,宇航员,星际,灭绝,一个,穿越
文档库块内容:迷航,星际,第一章
文档库块内容:飞船,李昂,观察窗,浩瀚,星空,宇航员,经验丰富,星系,与众不同,穿越
文档库块内容:李昂,星空,睁开眼,眩晕,飞船,星系,扭曲,闪过,发现自己,陌生
文档库块内容:异星,第二章,文明
文档库块内容:李昂,文明,行星,一颗,高度发达,小行星,星球,灭绝,星系,这颗
文档库块内容:赛跑,第三章,时间
文档库块内容:寻找,李昂,科技知识,小行星,飞船,星系,撞击,紧迫,自己,阻止
文档库块内容:火花,第四章,智慧
文档库块内容:能量,李昂,场来,计划,小行星,解决方案,撞击,这个,轨道,科学家
文档库块内容:第五章,危机,希望
文档库块内容:星球,矿物,能量,李昂,夜以继日,小行星,蕴藏,罕见,深处,科学家
文档库块内容:第六章,星际,救援
文档库块内容:小行星,成功,矿物,能量,李昂,撞击,带回,采集,勇敢,实验室
文档库块内容:归途,第七章
文档库块内容:李昂,遗迹,热烈欢迎,星系,隐藏,感激,解除,通往,这个,探索
文档库块内容:第八章,重逢,星际
文档库块内容:李昂,归属感,归途,星系,穿越,踏上,冒险,告别,前所未有,平静
文档库块内容:第九章,开始
文档库块内容:李昂,星球,星际,之间,一个,开启,激励,全新,无数,英雄
文档库块内容:第十章,星际,使者
文档库块内容:文明,李昂,旅程,使者,美好,宇宙,探索,交流,挑战,面对
文档库块内容:慕课
文档库块内容:梦工厂,程序员
```

02. RAG 文档分割/分块总结

在前面的课时中，我们学习过 `字符文本分割器`、`递归字符文本分割器`、`Html标题/段分割器`、`语义分割器` 等多种文本分割器类型，这也是目前 RAG 分块 Chunk 的 4 种策略：

1. `固定大小分块`：这是最常见的分块方法，通过设定块的大小和是否有重叠来决定分块。这种方法简单直接，不需要使用任何NLP库，因此计算成本低且易于使用，例如 `CharacterTextSplitter`，亦或者直接循环遍历固定大小拆分。
2. `基于结构的分块`：常见的 HTML、MARKDOWN 格式，或者其他可以有明确结构格式的文档。这种可以借助“结构感知”对文档分块，充分利用文档文本意外的信息，类似 LangChain 中的 `HTMLHeaderTextSplitter` 等。
3. `基于语义的分块`：这种策略旨在确保每个分块包含尽可能多的语义独立信息。可以采用不同的方法，如标点符号、自然段落、或者NLTK、Spicy 等工具包来实现语义分块，或者 Embedding-based 方法，例如 LangChain 中的 `SemanticChunker` 等。
4. `递归分块`：递归分块使用一组分隔符，以分层和迭代的方式将输入文本划分为更小的块。如果最初分割文本没有产生所需大小或结构的块，则该方法会继续递归地分割直到满足条件，例如 LangChain 中的 `RecursiveCharacterTextSplitter` 等。

这些策略各有优势和适用场景，选择合适的分块策略取决于具体的应用需求和数据特性。很遗憾，到目前为止还没有什么是最优的策略，但这也是很难有一个产品一统天下的原因。同时策略可以组合使用，并不是一类文档只能用一种策略。

> **Important**
>
> 对于一个 RAG 场景，分成四个主要阶段：`预检索`、`检索`、`后检索` 和 `生成`，其中 `分块` 是 `预检索` 阶段的策略，如果在 `分块` 阶段尝试了上述 4 种策略均没有很好的效果，或许就不应该采用 RAG 的策略，而是使用 `微调` 的方式，让这部分知识成为模型永久的记忆，效果可能会更好！

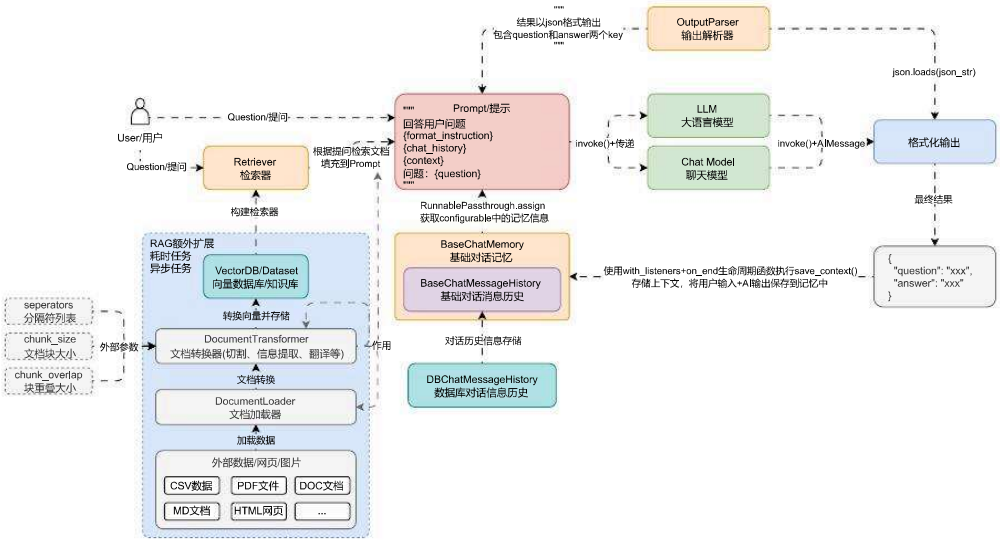
非分割类型的文档转换器使用技巧

非分割类型的文档转换器使用技巧

01. DocumentTransformer 组件

在 LangChain 中，另一种非分割类型的文档转换器，这类转换器也是传递 `文档列表` 并返回 `文档列表`，一般是将某种文档按照需求转换成另外一种格式（例如：**翻译文档、文档重排、HTML 转文本、文档元数据提取、文档转问答**等）。

这类文档转换器由于接收 `文档列表`，返回的也是 `文档列表`，所以可以在 LLM 应用中任何存在 `文档列表` 的地方使用，例如下方的 LLM 应用架构流程图中的 `文档加载`、`文档切割`、`检索器检索` 的环节交互数据都是 `文档列表`，所以这几个环节都可以添加文档转换器组件。



在 LangChain 中，使用文档转换器技巧非常简单，按照对应组件的构造函数进行传参，然后调用 `transformer_documents` 函数即可完成对文档的快速转换（每个转换器生成的格式不一样，需查看文档了解生成内容详情）。

LangChain 封装的文档转换器组件：https://imooc-langchain.shortvar.com/docs/integrations/document_transformers/。

02. 问答转换器

在 RAG 的外挂知识库中，向量存储知识库中使用的文档通常以叙述或对话格式存储。但是，绝大部分用户的查询都是问题格式，所以如果我们在对文档进行向量化之前先将其转换为 `问答格式`，可以在一定程度上增加检索相关文档的可能性，降低检索不相关文档的可能性。

这个技巧也是 RAG 应用开发中常见的一种优化策略，即将原始数据转换成 `QA` 数据后进行存储，除此之外，对于绝大部分 LLM 的微调，使用的也是 `QA问答数据` 也可以考虑使用该问答转换器进行转换。

在 LangChain 中封装了 `Doctran` 库并实现了 `DoctranQATransformer` 类可以快捷实现该功能，这个库底层使用 OpenAI 的函数回调来实现对问答数据的提取，首先安装该库：

```
pip install -U doctran
```

假设有一段如下的文本，需要转换成 `QA数据`：

```
机密文件 - 仅供内部使用
```

日期：2023年7月1日
主题：各种话题的更新和讨论
亲爱的团队，
希望这封邮件能找到你们一切安好。在这份文件中，我想向你们提供一些重要的更新，并讨论需要我们关注的各种话题。请将此处包含的信息视为高度机密。
安全和隐私措施
作为我们不断致力于确保客户数据安全和隐私的一部分，我们已在所有系统中实施了强有力的措施。我们要赞扬IT部门的John Doe（电子邮件：john.doe@example.com）在增强我们网络安全方面的勤奋工作。未来，我们提醒每个人严格遵守我们的数据保护政策和准则。此外，如果您发现任何潜在的安全风险或事件，请立即向我们专门的团队报告，联系邮箱为security@example.com。
人力资源更新和员工福利
最近，我们迎来了几位为各自部门做出重大贡献的新团队成员。我要表扬Jane Smith（社保号：049-45-5928）在客户服务方面的出色表现。Jane一直受到客户的积极反馈。此外，请记住我们的员工福利计划的开放报名期即将到来。如果您有任何问题或需要帮助，请联系我们的人力资源代表Michael Johnson（电话：418-492-3850，电子邮件：michael.johnson@example.com）。
营销倡议和活动
我们的营销团队一直在积极制定新策略，以提高品牌知名度并推动客户参与。我们要感谢Sarah Thompson（电话：415-555-1234）在管理我们的社交媒体平台方面的杰出努力。Sarah在过去一个月内成功将我们的关注者基数增加了20%。此外，请记住7月15日即将举行的产品发布活动。我们鼓励所有团队成员参加并支持我们公司的这一重要里程碑。
研发项目
在追求创新的过程中，我们的研发部门一直在为各种项目不懈努力。我要赞扬David Rodriguez（电子邮件：david.rodriguez@example.com）在项目负责人角色中的杰出工作。David对我们尖端技术的发展做出了重要贡献。此外，我们希望每个人在7月10日定期举行的研发头脑风暴会议上分享他们的想法和建议，以开展潜在的新项目。
请将此文档中的信息视为最机密，并确保不与未经授权的人员分享。如果您对讨论的话题有任何疑问或顾虑，请随时直接联系我。
感谢您的关注，让我们继续共同努力实现我们的目标。
此致，
Jason Fan
联合创始人兼首席执行官
Psychic
jason@psychic.dev

使用示例：

```python
import dotenv
from langchain_community.document_transformers import DoctranQATransformer
from langchain_core.documents import Document

dotenv.load_dotenv()

# 1.构建文档列表
page_content = """..."""
documents = [Document(page_content=page_content)]

# 2.构建问答转换器并转换
qa_transformer = DoctranQATransformer(openai_api_model="gpt-3.5-turbo-16k")
transformer_documents = qa_transformer.transform_documents(documents)

# 3.输出内容
for qa in transformer_documents[0].metadata.get("questions_and_answers"):
    print(qa)
```

输出内容：

```
{'question': '文件日期是什么？', 'answer': '2023年7月1日'}
{'question': '文件主题是什么？', 'answer': '各种话题的更新和讨论'}
{'question': '谁是IT部门的网络安全负责人？', 'answer': 'John Doe（电子邮件：john.doe@example.com）'}
{'question': '如果发现安全风险或事件，应该向谁报告？', 'answer': '专门的团队，联系邮箱为security@example.com'}
{'question': '谁在客户服务方面表现出色？', 'answer': 'Jane Smith（社保号：049-45-5928）'}
{'question': '员工福利计划的开放报名期是什么时候？', 'answer': '即将到来'}
{'question': '人力资源代表的联系信息是什么？', 'answer': 'Michael Johnson（电话：418-492-3850，电子邮件：michael.johnson@example.com）'}
{'question': '谁在管理社交媒体平台方面做出了杰出努力？', 'answer': 'Sarah Thompson（电话：415-555-1234）'}
{'question': '产品发布活动的日期是什么时候？', 'answer': '7月15日'}
{'question': '谁在研发部门担任项目负责人角色？', 'answer': 'David Rodriguez（电子邮件：david.rodriguez@example.com）'}
{'question': '研发头脑风暴会议的日期是什么时候？', 'answer': '7月10日'}
```

03. 翻译转换器

在 RAG 应用开发中，将文档通过嵌入/向量的方式进行比较的好处在于能跨语言工作，例如：`你好，世界！`、`Hello, World!` 和 `こんにちは、世界！` 分别是 `中英日` 三国的语言，但是因为语义相近，所以在向量空间中的位置也是非常接近的。

当一个 RAG 应用需要跨语言工作时，一般有两种策略：

1. 在将文档切块并嵌入存储到向量数据库时，同时将文档翻译成多国语言并进行相同的操作。
2. 在进行检索操作时，将检索出来的文档执行翻译功能，然后使用翻译后的文档。

这两种策略都涉及到一个功能，就是 `文档的翻译`，或者是说将 `文档` 转换成另外一种形式的 `文档`，这类操作其实和 `文档转换器` 的作用一模一样，所以可以考虑使用该组件来实现这个功能，LangChain 中针对翻译的转换器就提供了不少，例如 `Doctran`。

首先执行命令安装 `doctran`，这是一个文本翻译库，底层使用 OpenAI 的函数调用功能来在不同语言之间实现翻译：

```
pip install -U doctran
```

接下来就可以使用 `DoctranTextTranslator` 组件来实现对文档的快速中英文转换，示例如下：

```
import dotenv
from langchain_community.document_transformers import DoctranTextTranslator
from langchain_core.documents import Document

dotenv.load_dotenv()

# 1.构建文档列表
page_content = """..."""
documents = [Document(page_content=page_content)]

# 2.构建翻译转换器并翻译
text_translator = DoctranTextTranslator(openai_api_model="gpt-3.5-turbo-16k")
```

```
translator_documents = text_translator.transform_documents(documents)

# 3.输出翻译内容
print(translator_documents[0].page_content)
```

输出内容:

```
Confidential Document - For Internal Use Only
Date: July 1, 2023
Subject: Updates and Discussions on Various Topics

Dear Team,

I hope this email finds you all well. In this document, I would like to provide
you with some important updates and discuss various topics that require our
attention. Please consider the information included here as highly confidential.

Security and Privacy Measures

As part of our ongoing commitment to ensuring customer data security and privacy,
we have implemented strong measures across all systems. We commend John Doe from
the IT department (email: john.doe@example.com) for his diligent work in
enhancing our network security. Going forward, we remind everyone to strictly
adhere to our data protection policies and guidelines. Additionally, if you come
across any potential security risks or incidents, please report them immediately
to our dedicated team at security@example.com.

Human Resources Updates and Employee Benefits

Recently, we have welcomed several new team members who have made significant
contributions to their respective departments. I would like to commend Jane Smith
(Social Security Number: 049-45-5928) for her outstanding performance in customer
service. Jane has consistently received positive feedback from clients.
Furthermore, please note that the open enrollment period for our employee
benefits program is approaching. If you have any questions or need assistance,
please contact our HR representative, Michael Johnson (phone: 418-492-3850,
email: michael.johnson@example.com).

Marketing Initiatives and Events

Our marketing team has been actively developing new strategies to increase brand
awareness and drive customer engagement. We would like to thank Sarah Thompson
(phone: 415-555-1234) for her exceptional efforts in managing our social media
platforms. Sarah has successfully increased our follower base by 20% in the past
month. Additionally, please remember the product launch event scheduled for July
15th. We encourage all team members to attend and support this important
milestone for our company.

Research and Development Projects
```

Our R&D department has been tirelessly working on various projects in pursuit of innovation. I would like to commend David Rodriguez (email: david.rodriguez@example.com) for his outstanding work in the role of project lead. David has made significant contributions to the development of our cutting-edge technology. Furthermore, we encourage everyone to share their ideas and suggestions at the R&D brainstorming meeting scheduled for July 10th to explore potential new projects.

Please consider the information in this document as highly confidential and ensure that it is not shared with unauthorized individuals. If you have any questions or concerns regarding the topics discussed, please feel free to contact me directly.

Thank you for your attention, and let's continue working together towards our goals.

Sincerely,

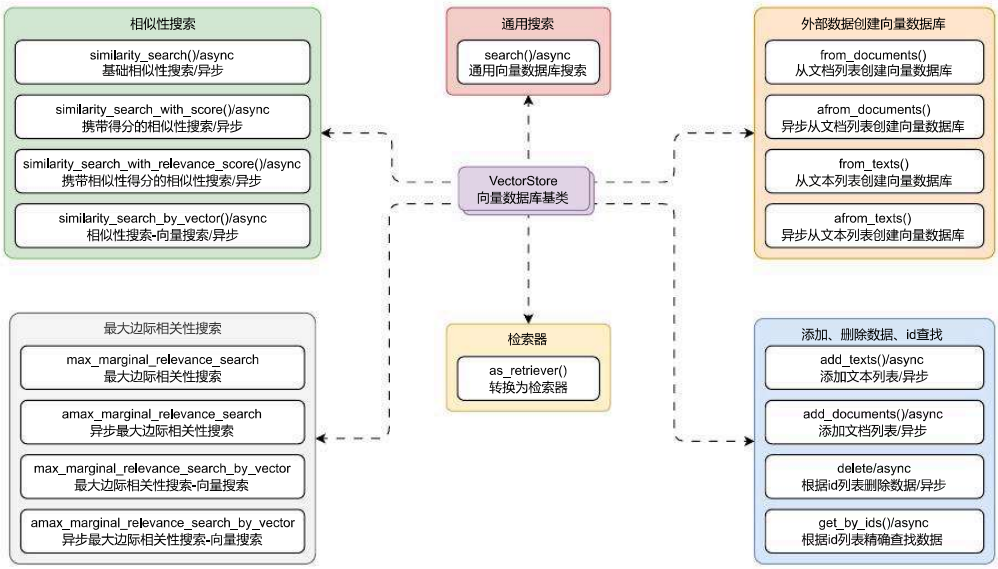
Jason Fan
Co-founder and CEO
Psychic
jason@psychic.dev

VectorStore 组件深入学习与检索方法

VectorStore 组件深入学习与检索方法

01. VectorStore 组件深入学习

考虑到目前市面上的向量数据库众多，每个数据库的操作方式也无统一标准，但是仍然存在着一些公共特征，LangChain 基于这些通用的特征封装了 `VectorStore` 基类，在这个基类下，可以将方法划分成 5 种：`相似性搜索`、`最大边际相关性搜索`、`通用搜索`、`添加删除精确查找数据`、`通用搜索`，类图如下：



1.1 带得分阈值的相似性搜索

在 LangChain 的相似性搜索中，无论结果多不匹配，只要向量数据库中存在数据，一定会查找出相应的结果，在 RAG 应用开发中，一般是将高相似文档插入到 Prompt 中，所以可以考虑添加一个 `相似性得分阈值`，超过该数值的部分才等同于有相似性。

在 `similarity_search_with_relevance_scores()` 函数中，可以传递 `score_threshold` 阈值参数，过滤低于该得分的文档。

例如没有添加阈值检索 `我养了一只猫，叫笨笨`，示例与输出如下：

```
import dotenv
from langchain_community.vectorstores import FAISS
from langchain_core.documents import Document
from langchain_openai import OpenAIEmbeddings

dotenv.load_dotenv()

embedding = OpenAIEmbeddings(model="text-embedding-3-small")

documents = [
    Document(page_content="笨笨是一只很喜欢睡觉的猫咪", metadata={"page": 1}),
    Document(page_content="我喜欢在夜晚听音乐，这让我感到放松。", metadata={"page":
2}),
    Document(page_content="猫咪在窗台上打盹，看起来非常可爱。", metadata={"page": 3}),
    Document(page_content="学习新技能是每个人都应该追求的目标。", metadata={"page":
4}),
```

```
    Document(page_content="我最喜欢的食物是意大利面，尤其是番茄酱的那种。", metadata=
{"page": 5}),
    Document(page_content="昨晚我做了一个奇怪的梦，梦见自己在太空飞行。", metadata=
{"page": 6}),
    Document(page_content="我的手机突然关机了，让我有些焦虑。", metadata={"page": 7}),
    Document(page_content="阅读是我每天都会做的事情，我觉得很充实。", metadata={"page":
8}),
    Document(page_content="他们一起计划了一次周末的野餐，希望天气能好。", metadata=
{"page": 9}),
    Document(page_content="我的狗喜欢追逐球，看起来非常开心。", metadata={"page": 10}),
]
db = FAISS.from_documents(documents, embedding)

print(db.similarity_search_with_relevance_scores("我养了一只猫，叫笨笨",
score_threshold=0.4))

# 输出内容
[(Document(metadata={'page': 1}, page_content='笨笨是一只很喜欢睡觉的猫咪'),
0.4592331743070337), (Document(metadata={'page': 3}, page_content='猫咪在窗台上打
盹，看起来非常可爱。'), 0.22960424668403867), (Document(metadata={'page': 10},
page_content='我的狗喜欢追逐球，看起来非常开心。'), 0.02157827632118159),
(Document(metadata={'page': 7}, page_content='我的手机突然关机了，让我有些焦虑。'),
-0.09838758604956)]
```

添加阈值 `0.4`，搜索输出示例如下：

```
print(db.similarity_search_with_relevance_scores("我养了一只猫，叫笨笨",
score_threshold=0.4))

# 输出

[(Document(metadata={'page': 1}, page_content='笨笨是一只很喜欢睡觉的猫咪'),
0.45919389344422157)]
```

对于 `score_threshold` 的具体数值，要看相似性搜索方法使用的逻辑、计算相似性得分的逻辑进行设置，并没有统一的标准，并且与向量数据库的数据大小也存在间接关系，数据集越大，检索出来的准确度相比少量数据会更准确。

1.2 as_retriever() 检索器

在 LangChain 中，VectorStore 可以通过 `as_retriever()` 方法转换成检索器，在 `as_retriever()` 中可以传递一下参数：

1. `search_type`：搜索类型，支持 `similarity(基础相似性搜索)`、`similarity_score_threshold(携带相似性得分+阈值判断的相似性搜索)`、`mmr(最大边际相关性搜索)`。
2. `search_kwargs`：其他键值对搜索参数，类型为字典，例如：`k`、`filter`、`score_threshold`、`fetch_k`、`lambda_mult` 等，当搜索类型配置为 `similarity_score_threshold` 后，必须添加 `score_threshold` 配置选项，否则会报错，参数的具体信息要看 `search_type` 类型对应的函数配合使用。

并且由于检索器是 Runnable 可运行组件，所以可以使用 Runnable 组件的所有功能（组件替换、参数配置、重试、回退、并行等）。

例如将向量数据库转换成`携带得分+阈值判断的相似性搜索，并设置得分阈值为0.5，数据条数为10条`，代码示例如下：

```
import dotenv
import weaviate
from langchain_community.document_loaders import UnstructuredMarkdownLoader
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.构建加载器与分割器
loader = UnstructuredMarkdownLoader("./项目API文档.md")
text_splitter = RecursiveCharacterTextSplitter(
    separators=["\n\n", "\n", "。|！|？", "\.\s|\!\s|\?\s", "；|;\s", "，|,\s", "
", "", ],
    is_separator_regex=True,
    chunk_size=500,
    chunk_overlap=50,
    add_start_index=True,
)

# 2.加载文档并分割
documents = loader.load()
chunks = text_splitter.split_documents(documents)

# 3.将数据存储到向量数据库
db = WeaviateVectorStore(
    client=weaviate.connect_to_wcs(
        cluster_url="https://eftofnujtxqcsaOsn272jw.c0.us-
west3.gcp.weaviate.cloud",
        auth_credentials=AuthApiKey("21pzYyOorl2dxH9xCoZG1O2bOeuDeKJNEbBO"),
    ),
    index_name="DatasetDemo",
    text_key="text",
    embedding=OpenAIEmbeddings(model="text-embedding-3-small"),
)

# 4.转换检索器
retriever = db.as_retriever(
    search_type="similarity_score_threshold",
    search_kwargs={"k": 10, "score_threshold": 0.5},
)

# 5.检索结果
documents = retriever.invoke("关于配置接口的信息有哪些")

print(list(document.page_content[:50] for document in documents))
print(len(documents))
```

输出内容：

```
['接口说明：用于更新对应应用的调试长记忆内容，如果应用没有开启长记忆功能，则调用接口会发生报错。\n\n接', '如果接口需要授权，需要在 headers 中添加 Authorization ，并附加 access', '接口示例：\n\njson\n{\n    "code": "success",\n    "data": {', '接口信息：授权 +POST:/apps/:app_id/debug\n\n接口参数：\n\n请求参数：\n\nap', '1.2 [todo]更新应用草稿配置信息\n\n接口说明：更新应用的草稿配置信息，涵盖：模型配置、长记忆', '请求参数：\n\napp_id -> uuid：路由参数，必填，需要获取的应用 id。\n\n响应参数：\n\n', 'memory_mode -> string：记忆类型，涵盖长记忆 long_term_memory ', '1.6 [todo]获取应用调试历史对话列表\n\n接口说明：用于获取应用调试历史对话列表信息，该接口支', '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', '响应参数：\n\nsummary -> str：该应用最新调试会话的长记忆内容。\n\n响应示例：\n\njso']
10
```

> **Caution**
>
> 课后练习：检索器返回的数据为 `文档列表`，并没有携带相关性得分信息，如果想携带得分信息，应该如何操作?
>
> 思路：构建一个自定义函数，调用 `similarity_search_with_relevance_scores()` 函数，将检索结果的得分填充到文档的 `元数据` 中，使用 `RunnableLambda` 函数将自定义函数包装成 Runnable 可运行组件/函数。

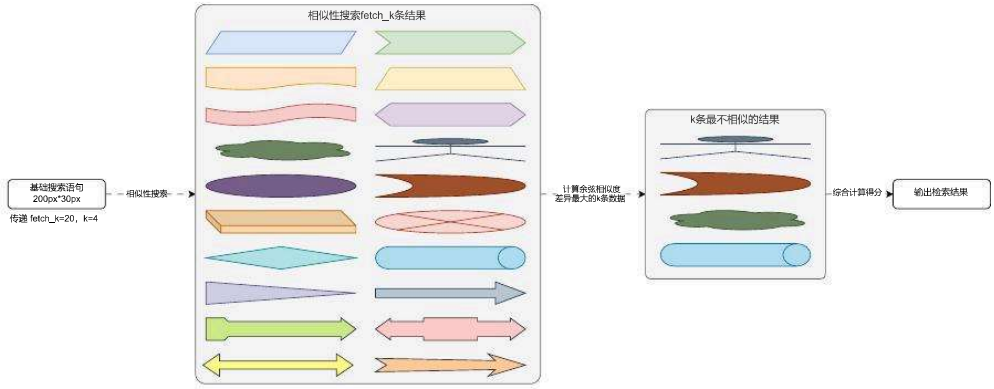
02. MMR 最大边际相关性

最大边际相关性（MMR，max_marginal_relevance_search）的基本思想是同时考量查询与文档的 `相关度`，以及文档之间的 `相似度`。`相关度` 确保返回结果对查询高度相关，`相似度` 则鼓励不同语义的文档被包含进结果集。具体来说，它计算每个候选文档与查询的 `相关度`，并减去与已经入选结果集的文档的最大 `相似度`，这样更不相似的文档会有更高分。

而在 LangChain 中MMR 的实现过程和 `FAISS` 的 `带过滤器的相似性搜索` 非常接近，同样也是先执行相似性搜索，并得到一个远大于 `k` 的结果列表，例如 `fetch_k` 条数据，然后对搜索得到的 `fetch_k` 条数据计算文档之间的相似度，通过加权得分找到最终的 k 条数据。

> **Important**
>
> 简单来说，MMR 就是在一大堆最相似的文档中查找最不相似的，从而保证 `结果多样化`。

所以 `MMR` 在保证查询准确的同时，尽可能提供 `多样化结果`，以增加信息检索的有效性和多样性，MMR 的运行演示图如下：



根据上面的运行流程，执行一个 MMR 最大边际相似性搜索需要的参数为：`搜索语句`、`k条搜索结果数据`、`fetch_k条中间数据`、`多样性系数(0代表最大多样性，1代表最小多样性)`，在 LangChain 中也是基于这个思想进行封装，`max_marginal_relevance_search()` 函数的参数如下：

1. `query`：搜索语句，类型为字符串，必填参数。
2. `k`：搜索的结果条数，类型为整型，默认为 4。
3. `fetch_k`：要传递给 MMR 算法的的文档数，默认为 20。
4. `lambda_mult`：函数系数，数值范围从0-1，底层计算得分 = `lambda_mult` * `相关性` + `(1 - lambda_mult)` * `相似性`，所以 0 代表最大多样性、1 代表最小多样性。
5. `kwargs`：其他传递给搜索方法的参数，例如 `filter` 等，这个参数使用和相似性搜索类似，具体取决于使用的向量数据库。

使用示例：

```
import dotenv
import weaviate
from langchain_community.document_loaders import UnstructuredMarkdownLoader
from langchain_openai import OpenAIEmbeddings
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_weaviate import WeaviateVectorStore

dotenv.load_dotenv()

# 1.构建加载器与分割器
loader = UnstructuredMarkdownLoader("./项目API文档.md")
text_splitter = RecursiveCharacterTextSplitter(
    separators=["\n\n", "\n", "。|！|？", "\.\s|\!\s|\?\s", "；|;\s", "，|,\s", "
", "", ],
    is_separator_regex=True,
    chunk_size=500,
    chunk_overlap=50,
    add_start_index=True,
)

# 2.加载文档并分割
documents = loader.load()
chunks = text_splitter.split_documents(documents)

# 3.将数据存储到向量数据库
db = WeaviateVectorStore(
    client=weaviate.connect_to_local("192.168.2.120", "8080"),
    index_name="DatasetDemo",
    text_key="text",
    embedding=OpenAIEmbeddings(model="text-embedding-3-small"),
)
db.add_documents(chunks)

# 4.执行最大边际相关性搜索
search_documents = db.max_marginal_relevance_search("关于应用配置的接口有哪些？")

# 5.打印搜索的结果
print(list(document.page_content[:100] for document in search_documents))
```

返回结果：

```
['1.2 [todo]更新应用草稿配置信息\n\n接口说明：更新应用的草稿配置信息，涵盖：模型配置、长记忆模式等，该接口会查找该应用原始的草稿配置并进行更新，如果没有原始草稿配置，则创建一个新配置作为草稿配', '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', '如果接口需要授权，需要在 headers 中添加 Authorization ，并附加 access_token 即可完成授权登录，示例：\n\njson\nAuthorization: Bearer ey', 'memory_mode -> string：记忆类型，涵盖长记忆 long_term_memory 和 none 代表无。\nstatus -> string：应用配置的状态，drafted 代表草稿、']
```

在 LangChain 封装的 VectorStore 组件中，内置了两种搜索策略：`相似性搜索`、`最大边际相关性搜索`，这两种策略有不同的使用场景，一般来说 80% 的场合使用相似性搜索都可以得到不错的效果，对于一些追求创新/创意/多样性的 RAG 场景，可以考虑使用 `最大边际相关性搜索`。

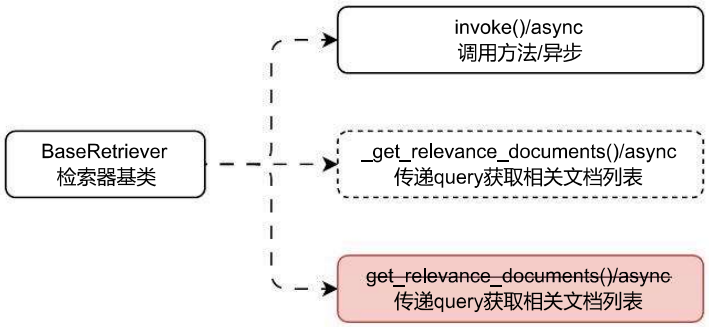
在使用 `相似性搜索` 时，尽可能使用 `similarity_search_with_relevance_scores()` 方法并传递阈值信息，确保在向量数据库数据较少的情况下，不将一些不相关的数据也检索出来，并且着重调试 `得分阈值(score_threshold)`，对于不同的文档/分割策略/向量数据库，得分阈值并不一致，需要经过调试才能得到一个相对比较正确的值（阈值过大检索不到内容，阈值过小容易检索到不相关内容）。

检索器组件深入学习与使用技巧

检索器组件深入学习与使用技巧

01. BaseRetriever 检索器基类

在 LangChain 中，传递一段 `query` 并返回与这段文本相关联文档的组件被称为 `检索器`，并且 LangChain 为所有检索器设计了一个基类——BaseRetriever，该类集成了 `RunnableSerializable`，所以该类是一个 `Runnable` 可运行组件，支持使用 `Runnable` 组件的所有配置，在 `BaseRetriever` 下封装了一些通用的方法，类图如下：



其中 `get_relevance_documents()` 方法将在 `0.3.0` 版本开始被遗弃（老版本非 Runnable 写法），使用检索器的技巧也非常简单，按照特定的规则创建好检索器后（通过 `as_retriever()` 或者 `构造函数`），调用 `invoke()` 方法即可。

并且针对所有 `向量数据库`，LangChain 都配置了 `as_retriever()` 方法，便于快捷将向量数据库转换成检索器，不同的检索器传递的参数会有所差异，需要查看源码或者查看文档搭配使用，例如下方是一个向量数据库检索器的使用示例：

```
import dotenv
import weaviate
from langchain_core.runnables import ConfigurableField
from langchain_openai import OpenAIEmbeddings
from langchain_weaviate import WeaviateVectorStore
from weaviate.auth import AuthApiKey

dotenv.load_dotenv()

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

# 2.转换检索器
retriever = db.as_retriever(
    search_type="similarity_score_threshold",
    search_kwargs={"k": 10, "score_threshold": 0.5},
```

```
)

# 3.执行基础相似性搜索
similarity_documents = retriever.with_config(configurable={
    "search_type": "similarity",
    "search_kwargs": {"k": 10},
}).invoke("关于应用配置的接口有哪些？")
print("相似性搜索：", similarity_documents)
print("内容长度:", len(similarity_documents))
```

输出内容：

```
相似性搜索：  [...]
内容长度: 10
```

02. VectorStoreRetriever 检索器

`VectorStoreRetriever` 是 `BaseRetriever` 的子类，这是一个专门针对向量数据库的基础检索器，在 `VectorStoreRetriever` 的内部实现了 `_get_relevant_documents()` 方法，还定义了单独的属性：

1. `vectorstore`：检索器归属的向量数据库。
2. `search_type`：搜索类型。
3. `search_kwargs`：搜索参数。

这些参数均来源于 `as_retriever()` 或者在实例化类时传递的参数，由于该组件是一个 Runnable 可运行组件，所以可以使用 `.configurable_fields()` 来修改类内部的参数。

例如实现在运行时传递 `search_type` 与 `search_kwargs`，修改成 `mmr` 方法进行检索，并且返回 `8` 条数据， 代码示例：

```
import dotenv
import weaviate
from langchain_core.runnables import ConfigurableField
from langchain_openai import OpenAIEmbeddings
from langchain_weaviate import WeaviateVectorStore
from weaviate.auth import AuthApiKey

dotenv.load_dotenv()

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

# 2.转换检索器
retriever = db.as_retriever(
    search_type="similarity_score_threshold",
```

```
    search_kwargs={"k": 10, "score_threshold": 0.5},
).configurable_fields(
    search_type=ConfigurableField(id="search_type"),
    search_kwargs=ConfigurableField(id="search_kwargs"),
)

# 3.执行基础相似性搜索，并返回4条数据
similarity_documents = retriever.with_config(configurable={
    "search_type": "mmr",
    "search_kwargs": {"k": 10},
}).invoke("关于应用配置的接口有哪些？")
print("相似性搜索：", similarity_documents)
print("内容长度:", len(similarity_documents))
```

输出内容：

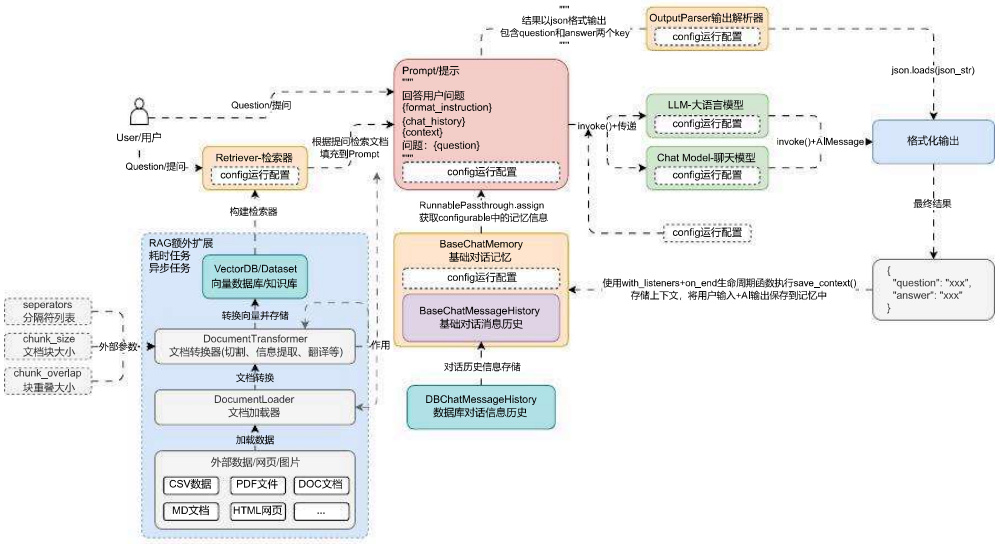
```
相似性搜索：  [...]
内容长度: 10
```

除了使用 `.with_config()` 传递运行时配置，也可以在执行 `.invoke()` 函数时传递 config，效果是一模一样的。

```
similarity_documents = retriever.invoke(
    "关于应用配置的接口有哪些？",
    config={"configurable": {
        "search_type": "mmr",
        "search_kwargs": {"k": 8},
    }}
)
```

在 LCEL 表达式构建的链应用中，`.with_config()` 可以通过链一起传递，或者是调用 `.invoke()` 函数是传递 `config+configurable` 属性完成对配置信息的替换，所以在 RAG 应用开发中，可以对检索器配置好相应的选项，如果需要特定信息时传递运行配置即可，否则会运行默认配置信息。

为 `聊天机器人架构运行流程图` 添加上 `动态配置` 选项，更新后如下：



除此之外，`Runnable` 可运行组件的其他配置也可以轻松配置使用：

1. 使用 `.configurable_alternatives()` 来实现对 `向量数据库检索器` 的替换；
2. 使用 `.with_retry()` 来实现对出现错误的重试；
3. 使用 `.with_fallbacks()` 来实现对出现错误时的回退；
4. 使用 `.with_listeners()` 来实现对执行生命周期的监听；
5. 使用 `.bind()` 来实现动态传递绑定时参数（不过对于 weaviate 向量数据库来说，目前没有效果）。

`.bind()` 函数会将相应的数据传递给 `.invoke()` 的 `kwargs` 参数，但是在 `weaviate` 的 `invoke()` 中并没有用到 `kwargs` 参数，所以 `.bind()` 函数不会起到效果，要想传递搜索参数给 `weaviate` 的 `.invoke()` 方法只能通过 `search_kwargs` 进行传递。

核心代码：

```
def _get_relevant_documents(
    self, query: str, *, run_manager: CallbackManagerForRetrieverRun
) -> List[Document]:
    if self.search_type == "similarity":
        docs = self.vectorstore.similarity_search(query, **self.search_kwargs)
    elif self.search_type == "similarity_score_threshold":
        docs_and_similarities = (
            self.vectorstore.similarity_search_with_relevance_scores(
                query, **self.search_kwargs
            )
        )
        docs = [doc for doc, _ in docs_and_similarities]
    elif self.search_type == "mmr":
        docs = self.vectorstore.max_marginal_relevance_search(
            query, **self.search_kwargs
        )
    else:
        raise ValueError(f"search_type of {self.search_type} not allowed.")
    return docs
```

内置的检索器组件与自定义检索器技巧

内置的检索器组件与自定义检索器技巧

01. 内置检索与自定义

除了向量数据库检索器，在 LangChain 内部还集成了大量的第三方检索器，这些检索器功能格式，涵盖：维基百科搜索、Weaviate 混合搜索、Zep 检索器、ES 搜索 等，链接：https://imooc-langchain.shortvar.com/docs/integrations/retrievers/。

不过由于这些内置检索器开发的时间较早，绝大部分都是在 Runnable 与 LCEL 表达式出来之前发布的，部分检索器可能并不是 Runnable 可运行组件，使用方法也有差异，使用前需要查阅文档进行使用。

如果内置的检索器没法完成需求，可以考虑使用 `自定义检索器`，比方接入一个可以快速检索 `百度网站`，或者 `慕课网` 课程的检索器。

在 LangChain 中实现自定义检索器的技巧其实非常简单，只需要继承 `BaseRetriever` 类，然后实现 `_get_relevant_documents()` 方法即可，从 `query` 到 `list[document]` 的逻辑全部都在这个函数内部实现，异步的方法也可以不需要实现，底层会委托同步方法来执行。

例如构建一个最基础的 `通过内容进行查找` 的自定义检索器，示例如下：

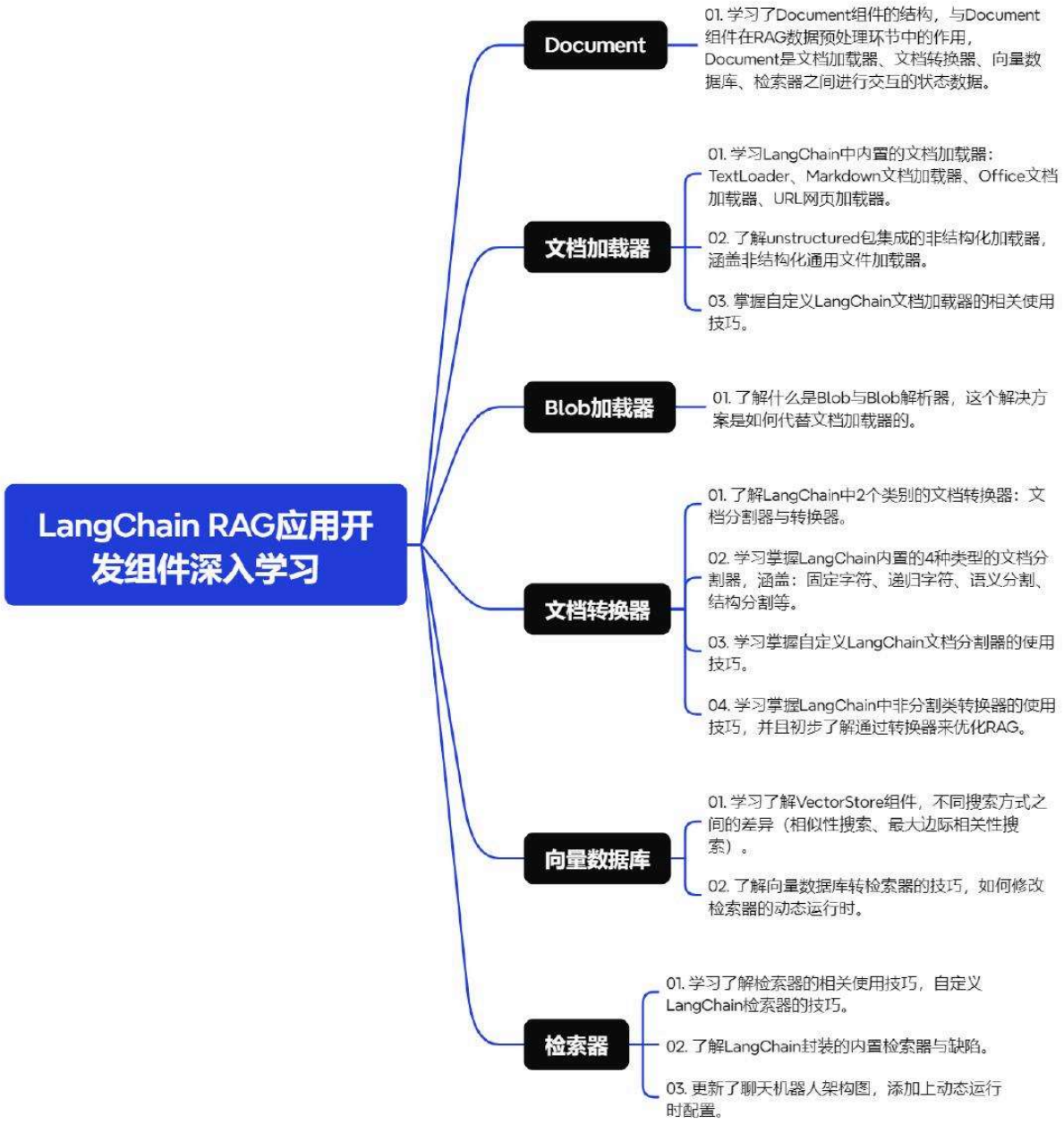
```
from typing import List

from langchain_core.callbacks import CallbackManagerForRetrieverRun
from langchain_core.documents import Document
from langchain_core.retrievers import BaseRetriever


class CustomRetriever(BaseRetriever):
    """自定义检索器"""
    documents: list[Document]
    k: int

    def _get_relevant_documents(self, query: str, *, run_manager:
CallbackManagerForRetrieverRun) -> List[Document]:
        """传递query获取关联的文档"""
        matching_documents = []
        for document in self.documents:
            if len(matching_documents) > self.k:
                return matching_documents
            if query.lower() in document.page_content.lower():
                matching_documents.append(document)
        return matching_documents


# 1.定义预设文档
documents = [
    Document(page_content="笨笨是一只很喜欢睡觉的猫咪", metadata={"page": 1}),
    Document(page_content="我喜欢在夜晚听音乐，这让我感到放松。", metadata={"page":
2}),
    Document(page_content="猫咪在窗台上打盹，看起来非常可爱。", metadata={"page": 3}),
    Document(page_content="学习新技能是每个人都应该追求的目标。", metadata={"page":
4}),
    Document(page_content="我最喜欢的食物是意大利面，尤其是番茄酱的那种。", metadata=
{"page": 5}),
```

```
    Document(page_content="昨晚我做了一个奇怪的梦，梦见自己在太空飞行。", metadata=
{"page": 6}),
    Document(page_content="我的手机突然关机了，让我有些焦虑。", metadata={"page": 7}),
    Document(page_content="阅读是我每天都会做的事情，我觉得很充实。", metadata={"page":
8}),
    Document(page_content="他们一起计划了一次周末的野餐，希望天气能好。", metadata=
{"page": 9}),
    Document(page_content="我的狗喜欢追逐球，看起来非常开心。", metadata={"page": 10}),
]

# 2.创建检索器
retriever = CustomRetriever(documents=documents, k=3)

# 3.调用检索器获取搜索结果并打印
retriever_documents = retriever.invoke("猫")
print(retriever_documents)
print(len(retriever_documents))
```

输出内容：

```
[Document(metadata={'page': 1}, page_content='笨笨是一只很喜欢睡觉的猫咪'),
Document(metadata={'page': 3}, page_content='猫咪在窗台上打盹，看起来非常可爱。')]
2
```

对于检索器这个模块来说，正常只需要了解 `向量数据库检索器` 的相关使用技巧，对于其他的一些检索器直到怎么根据文档使用即可，特别是针对网络/爬虫的检索，目前被更好的方案逐步代替——`工具回调`（下一周学习的内容）。


LangChain RAG应用开发组件深入学习
Document
01. 学习了Document组件的结构，与Document组件在RAG数据预处理环节中的作用，Document是文档加载器、文档转换器、向量数据库、检索器之间进行交互的状态数据。
文档加载器
01. 学习LangChain中内置的文档加载器：TextLoader、Markdown文档加载器、Office文档加载器、URL网页加载器。
02. 了解unstructured包集成的非结构化加载器，涵盖非结构化通用文件加载器。
03. 掌握自定义LangChain文档加载器的相关使用技巧。
Blob加载器
01. 了解什么是Blob与Blob解析器，这个解决方案是如何代替文档加载器的。
文档转换器
01. 了解LangChain中2个类别的文档转换器：文档分割器与转换器。
02. 学习掌握LangChain内置的4种类型的文档分割器，涵盖：固定字符、递归字符、语义分割、结构分割等。
03. 学习掌握自定义LangChain文档分割器的使用技巧。
04. 学习掌握LangChain中非分割类转换器的使用技巧，并且初步了解通过转换器来优化RAG。
向量数据库
01. 学习了解VectorStore组件，不同搜索方式之间的差异（相似性搜索、最大边际相关性搜索）。
02. 了解向量数据库转检索器的技巧，如何修改检索器的动态运行时。
检索器
01. 学习了解检索器的相关使用技巧，自定义LangChain检索器的技巧。
02. 了解LangChain封装的内置检索器与缺陷。
03. 更新了聊天机器人架构图，添加上动态运行时配置。