LangChain RAG 应用开发组件深入学习
章节介绍

LangChain 文档组件与文档加载器

- 学习了解 Document 组件在 RAG 应用开发中的作用，并学习 LangChain 文档加载器的配置与使用。
- 学习掌握 LangChain 集成的各类文档加载器，涵盖 CSV 文件加载、 HTML 网页加载、 PDF/ 文件夹 /Markdown 加载器以及通用文件加载器的使用。
- 学习掌握封装 LangChain 自定义文档加载器的使用。

LangChain 文档转换器与分割器

- 学习了解 LangChain 文档转换器的作用与使用场景，涵盖了拆分、合并、过滤、翻译等多个功能。
- 学习掌握 LangChain 集成的各类文本分割器的使用以及封装自定义文本分割器的技巧，不同模式下选择分割器的思路。
- 学习掌握 LangChain 语义分割器的使用及运行流程，以及语义分割器的使用场合。

VectorStore 组件与检索器的使用

- 深入学习 VectorStore 组件，了解多种相似性搜索的几何意义以及在不同场合下的选择策略。
- 学习掌握 LangChain 检索器与第三方检索器的配置与使用。
- 学习掌握自定义 LangChain 检索器的技巧。

Document 组件与文档加载器的使用

Document 组件与文档加载器组件的使用

01. Document 与文档加载器

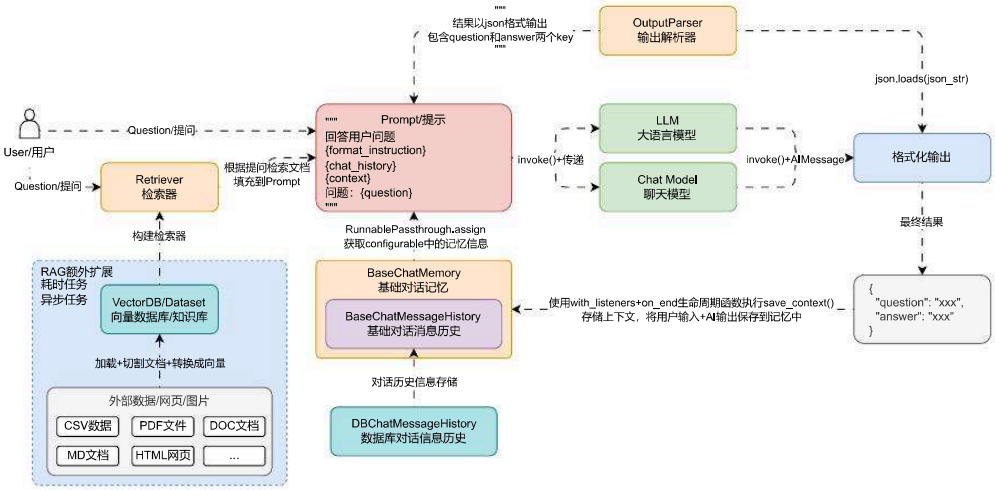
Document 类是 LangChain 中的核心组件，这个类定义了一个文档对象的结构，涵盖了文本内容和相关的元数据，**Document 也是文档加载器、文档分割器、向量数据库、检索器这几个组件之间交互传递的状态数据。**

在 LangChain 旧版本中，Document 还支持 lookup 检索功能，不过新版本下 Document 组件只拥有最基础的记录信息功能：

```
Document = page_content(页面内容) + metadata(元数据)
```

在前面的课时中，我们通过手动输入输入的形式来创建数据，但是**在 RAG 开发中，一般会读取特定来源的数据，而非手动录入数据**，例如：本地 markdown 文件、HTML 网页、PDF 文档、DOC 文档、URL 链接等多种方式来加载数据，然后再将原始文档按照特定切割成特定大小的文档，最后再将数据存储到向量数据库中，很少会手动录入数据。

所以在 RAG 应用外部，一般都会有一个额外的扩展，专门用于处理 `读取数据-切割数据-存储数据` 这个流程，并且这个流程非常耗时，例如上传一个 30M 的文档，需要执行加载/切割/文本嵌入，一般都会使用队列/异步进行处理，架构流程图更新如下：

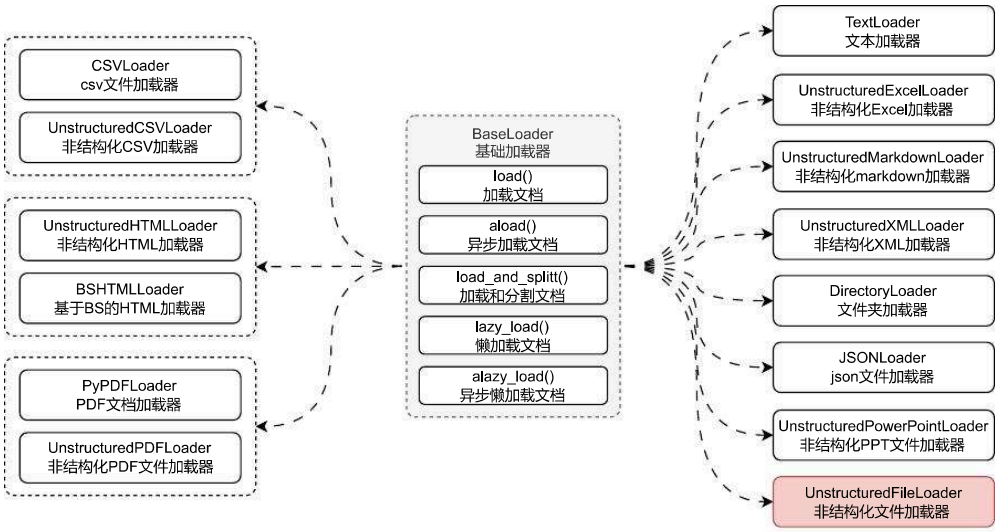


在新的架构流程中，文档加载器起到的作用就是从各式各样的数据中提取出相应的信息，并转换成标准的 Document 组件，从而屏蔽不同类型文件的读取差异。

在 LangChain 中所有文档加载器的基类为 `BaseLoader`，封装了统一的 5 个方法：

1. `load()/aload()`：加载和异步加载文档，返回的数据为文档列表。
2. `load_and_split()`：传递分割器，加载并将大文档按照传入的分割器进行切割，返回的数据为分割后的文档列表。
3. `lazy_load()/alazy_load()`：懒加载和异步懒加载文档，返回的是一个迭代器，适用于传递的数据源有多份文档的情况，例如文件夹加载器，可以每次获得最新的加载文档，不需要等到所有文档都加载完毕。

在 LangChain 中封装了几十种文档加载器，几乎所有的文件都可以使用这些加载器完成数据的读取，而不需要手动去封装：



LangChain 文档加载器文档：https://imooc-langchain.shortvar.com/docs/integrations/document_loaders/

02. TextLoader 使用技巧与源码解析

在 LangChain 中最简单的加载器组件就是 **TextLoader**，这个加载器可以加载一个文本文件（源码、markdown、text 等存储成文本结构的文件，DOC 并不是文本文件），并把整个文件的内容读入到一个 Document 对象中，同时为文档对象的 `metadata` 添加 `source` 字段用于记录源数据的来源信息。

TextLoader 使用起来非常简单，传递对应的文本路径即可：

示例代码：

```
from langchain_community.document_loaders import TextLoader

loader = TextLoader("./电商产品数据.txt", encoding="utf-8")
documents = loader.load()

print(documents)
print(len(documents))
print(documents[0].metadata)
```

输出内容：

```
[Document(page_content='xxx', metadata={'source': './电商产品数据.txt'})]
1
{'source': './电商产品数据.txt'}
```

TextLoader 源码底层主要通过 `open` 函数与对应的编码方式打开对应的文件，获取其内容，并将传递的路径信息复制到生成的文档示例中的 `metadata` 字段中，从而实现数据的快速加载。

核心源码：

```
# langchain_community/document_loaders/text.py->TextLoader::lazy_load
def lazy_load(self) -> Iterator[Document]:
    """Load from file path."""
    text = ""
```

```
    try:
        with open(self.file_path, encoding=self.encoding) as f:
            text = f.read()
    except UnicodeDecodeError as e:
        if self.autodetect_encoding:
            detected_encodings = detect_file_encodings(self.file_path)
            for encoding in detected_encodings:
                logger.debug(f"Trying encoding: {encoding.encoding}")
                try:
                    with open(self.file_path, encoding=encoding.encoding) as f:
                        text = f.read()
                    break
                except UnicodeDecodeError:
                    continue
        else:
            raise RuntimeError(f"Error loading {self.file_path}") from e
    except Exception as e:
        raise RuntimeError(f"Error loading {self.file_path}") from e

    metadata = {"source": str(self.file_path)}
    yield Document(page_content=text, metadata=metadata)
```

以 TextLoader 为例，扩展到 LangChain 封装的其他文档加载器，使用技巧都是一模一样的，**在实例化加载器的时候，传递对应的信息（文件路径、网址、目录等），然后调用加载器的 `load()` 方法即可一键加载文档。**

自定义 LangChain 文档加载器使用技巧

自定义 LangChain 文档加载器使用技巧

01. 自定义加载器使用技巧

对于一些企业的内部数据，例如**数据库、API接口等定制化非常强的数据，如果使用通用的文档加载器进行提取，虽然可以提取记录到相应的信息，但是加载的数据格式或者样式大概率没法满足我们的需求**，这个时候就可以考虑实现自定义文档加载器。

例如上节课使用的 `WebBaseLoader` 文档加载器加载慕课网首页的信息，会提取得到很多空白数据（空格、换行、Tab 等），将这类数据通过分割存储到向量数据库中，会极大降低检索与生成的效率和正确性。

```
D:\Code\llmops\llmops-api\venv\Scripts\python.exe D:\Code\llmops\llmops-api\study\29-LangChain内置文档加载器使用技巧\3.URL网页加载器
[Document(page_content='\n\n\n\n\n慕课网-程序员的梦工厂\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n \n\n\n\n\n\n\n\n\n'
1
{'source': 'https://imooc.com', 'title': '慕课网-程序员的梦工厂', 'language': 'No language found.'}

Process finished with exit code 0
```

在 LangChain 中实现自定义文档加载器非常简单，只需要继承 `BaseLoader` 基类，然后实现 `lazy_load()` 方法即可，如果该文档加载器有异步使用的场景，还需要实现 `alazy_load()` 方法。

假设有一个这样的需求，加载对应的文本信息，其中每行数据都作为一个 `Document` 组件，该文档加载器实现示例：

```python
from typing import Iterator, AsyncIterator

from langchain_core.document_loaders import BaseLoader
from langchain_core.documents import Document


class CustomDocumentLoader(BaseLoader):
    """自定义文档加载器"""

    def __init__(self, file_path: str) -> None:
        self.file_path = file_path

    def lazy_load(self) -> Iterator[Document]:
        """逐行读取文件的数据并使用yield返回文档"""
        with open(self.file_path, encoding="utf-8") as f:
            line_number = 0
            for line in f:
                yield Document(
                    page_content=line,
                    metadata={"line_number": line_number, "source":
self.file_path}
                )
                line_number += 1

    async def alazy_load(self) -> AsyncIterator[Document]:
        """lazy_load的异步方法，如果不实现，将委托lazy_load实现"""
        import aiofiles
        async with aiofiles.open(self.file_path, encoding="utf-8") as f:
            line_number = 0
            async for line in f:
                yield Document(
```

```
                    page_content=line,
                    metadata={"line_number": line_number, "source":
self.file_path}
                )
                line_number += 1


loader = CustomDocumentLoader("./喵喵.txt")
documents = loader.load()

print(documents)
print(len(documents))
print(documents[0].metadata)
```

输出示例：

```
[Document(page_content='喵喵🐱\n', metadata={'line_number': 0, 'source': './喵
喵.txt'}), Document(page_content='喵喵🐱\n', metadata={'line_number': 1, 'source':
'./喵喵.txt'}), Document(page_content='喵😻😻', metadata={'line_number': 2,
'source': './喵喵.txt'})]
3
{'line_number': 0, 'source': './喵喵.txt'}
```

02. 文档加载器扩展思考

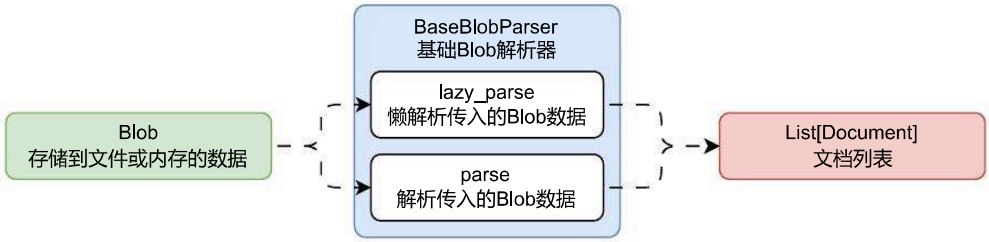
在上面的自定义文档加载器中，可以看到 `lazy_load()` 方法的两个核心步骤就是：`读取文件数据`、`将文件数据解析成Document`，并且绝大部分文档加载器都有着两个核心步骤，而且 `读取文件数据` 这个步骤大家都大差不差。

就像 `*.md`、`*.txt`、`*.py` 这类文本文件，甚至是 `*.pdf`、`*.doc` 等这类非文本文件，都可以使用同一个 `读取文件数据` 步骤将文件读取为 `二进制内容`，然后在使用不同的解析逻辑来解析对应的二进制内容，所以很容易可以得出：

```
文档加载器 = 二进制数据读取 + 解析逻辑
```

因此，在项目开发中，如果大量配置自定义文档解析器的话，将解析逻辑与加载逻辑分离，维护起来会更容易，而且也更容易复用相应的逻辑（具体使用哪种方式取决于开发）。

这样原先的 DocumentLoader 运行流程就变成了如下：



这样所有 DocumentLoader 就变成了共用 `Blob(数据读取)`，每个加载器内部只实现不同的 `parse` 即可，这也是 LangChain 目前正在设计的新方案，由于目前 LangChain 在这块尚不完善，并且篇幅较长，我们留到下一节课来讲解。

Blob 与 BlobParser 代替文档加载器

Blob与BlobParser代替文档加载器

01. LangChain 中的 Blob 方案

许多文档加载器都涉及到解析文件，此类加载器之间的差异通常源于文件解析方式，而不是文件加载方式。例如，你可以使用 `open()` 函数来读取 PDF 或 Markdown 文件的二进制内容，但是需要不同的解析逻辑来将二进制数据转换为文本。

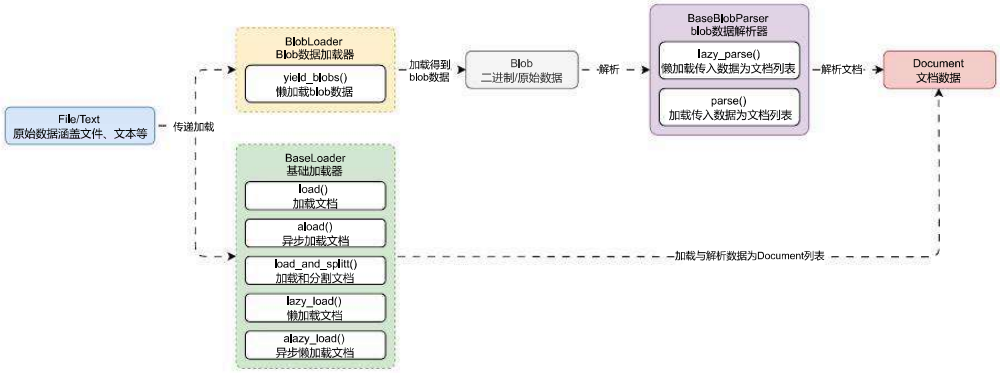
在 LangChain 中也提供了一个类似的解决方案 `Blob`，其灵感来源于 `Blob WebAPI规范`（这是前端 Web 浏览器中定义的相关规范）。

Blob WebAPI 规范文档链接：https://developer.mozilla.org/en-US/docs/Web/API/Blob。

该方案下有 `Blob`、`BlobLoader` 和 `BaseBlobParser` 三个类，含义如下：

1. `Blob`：LangChain 封装的数据对象，通过引用或值表示原始数据，该类提供一个接口，以表示不同形式具体化的二进制数据，使用该类可以有助于将数据加载器的开发与解析器耦合。
2. `BlobLoader`：Blob 数据加载器，类似 `DocumentLoader`，不过 `BlobLoader` 被设计成可以加载任何数据（未来的规划）。
3. `BlobParser`：Blob 数据解析器，用于将传入的 Blob 数据转换成文档列表。

在这个方案下，文档加载器的运行流程变成如下：



例如上节课的需求（加载对应的文本信息，其中每行数据都作为一个 `Document` 组件），使用 Blob 的方案来实现，只需自定义一个解析器并实现 `lazy_parser()` 方法即可，示例代码如下：

```
from typing import Iterator

from langchain_core.document_loaders import Blob
from langchain_core.document_loaders.base import BaseBlobParser
from langchain_core.documents import Document


class CustomParser(BaseBlobParser):
    """自定义解析器，提取文本文件的每一行为一个Document元素"""

    def lazy_parse(self, blob: Blob) -> Iterator[Document]:
        line_number = 0
        with blob.as_bytes_io() as f:
            for line in f:
                yield Document(
```

```
                    page_content=line,
                    metadata={"source": blob.source, "line_number": line_number}
                )
                line_number += 1


blob = Blob.from_path("./喵喵.txt")
parser = CustomParser()
documents = list(parser.lazy_parse(blob))

print(documents)
print(len(documents))
print(documents[0].metadata)
```

输出内容：

```
[Document(page_content='喵喵🐱\r\n', metadata={'source': './喵喵.txt',
'line_number': 0}), Document(page_content='喵喵🐱\r\n', metadata={'source': './喵
喵.txt', 'line_number': 1}), Document(page_content='喵😻😻', metadata={'source':
'./喵喵.txt', 'line_number': 2})]
3
{'source': './喵喵.txt', 'line_number': 0}
```

除了 `from_path()` 函数，`Blob` 类还可以在构造时传递 `data` 参数，直接从内存中加载内容，而无需从文件中获取，示例如下：

```
blob = Blob(data="喵喵🐱\r\n喵喵🐱\r\n喵😻😻")
```

02. Blob 数据存储类

LangChain 中设计的 Blob 数据存储类和 `Blob WebAPI规范` 定义的类非常接近，拥有以下方法和属性：

1. `data`：原始数据，支持存储字节、字符串数据。
2. `mimetype`：文件的 mimetype 类型。
3. `encoding`：文件的编码，默认值为 utf-8。
4. `path`：文件的原始路径，支持传递字符串路径或者 Path 类。
5. `metadata`：存储的元数据，一般都有 source 字段。
6. `source()`：只读函数/属性，用于返回数据的来源。
7. `as_string()`：将数据转换成字符串。
8. `as_bytes()`：将数据转换成字节数据。
9. `as_bytes_io()`：将数据转换成缓冲流字节数据。
10. `from_path()`：从对应的路径中加载 Blob 数据（文件）。
11. `from_data()`：从对应的原始数据中加载 Blob 数据（非文件）。

03. Blob 加载器

解析器封装了将二进制数据解析为 Document 组件所需的逻辑，而 Blob 加载器则封装了从给定存储位置加载 Blob 所需的逻辑。不过目前在 LangChain 中，只集成了一个 `FileSystemBlobLoader`，即文件系统二进制数据加载器。

这个加载器可以加载传入文件夹下的特定文件，代码示例：

```
from langchain_community.document_loaders.blob_loaders import
FileSystemBlobLoader

loader = FileSystemBlobLoader(".", show_progress=True)
for blob in loader.yield_blobs():
    print(blob.source)
```

输出内容：

```
100%|██████████| 3/3 [00:00<00:00, 58.15it/s]
1.Blob解析器示例.py
2.FileSystemBlobLoader示例.py
喵喵.txt
```

如果要实现一个 Blob 加载器，只需要继承 `BlobLoader` 类，并实现 `yield_blobs()` 方法即可，`BlobLoader` 类的代码如下：

```
# langchain_core/document_loaders/blob_loaders::BlobLoader->yield_blobs
class BlobLoader(ABC):
    """Abstract interface for blob loaders implementation.

    Implementer should be able to load raw content from a storage system
according
    to some criteria and return the raw content lazily as a stream of blobs.
    """

    @abstractmethod
    def yield_blobs(
        self,
    ) -> Iterable[Blob]:
        """A lazy loader for raw data represented by LangChain's Blob object.

        Returns:
            A generator over blobs
        """
```

04. Blob 通用加载器

在上面的示例中，Blob 加载器与解析器是分开使用的，其实在 LangChain 中还封装了一个由 `BlobLoader` 与 `BaseBlobParser` 组成的类——GenericLoader，这个类旨在提供标准化的方法，让 `BlobLoader` 使用更简单，不过目前也仅支持 `FileSystemBlobLoader`。

使用示例如下：

```
from langchain_community.document_loaders.generic import GenericLoader

loader = GenericLoader.from_filesystem(".", glob="*.txt", show_progress=True)

for idx, doc in enumerate(loader.lazy_load()):
    print(f"当前加载第{idx + 1}个文件，文件信息:{doc.metadata}")
```

输出内容：

```
100%|██████████| 1/1 [00:00<00:00, 19.76it/s]
当前加载第1个文件，文件信息:{'source': '喵喵.txt'}
```

整体来说，Blob 解决方案目前 LangChain 封装与集成得非常少，如果需要使用 Blob 的形式来加载文件，目前还需要大量编写加载文件与解析数据的逻辑，效率比较低，不过随着未来 LangChain 团队封装的 Blob 解析逻辑越来越多，会逐渐代替 DocumentLoader 的方案。对于目前的版本来说，大家只需要知道有这个东西即可。

文档转换器与字符分割器组件的使用

文档转换器与字符分割器组件的使用

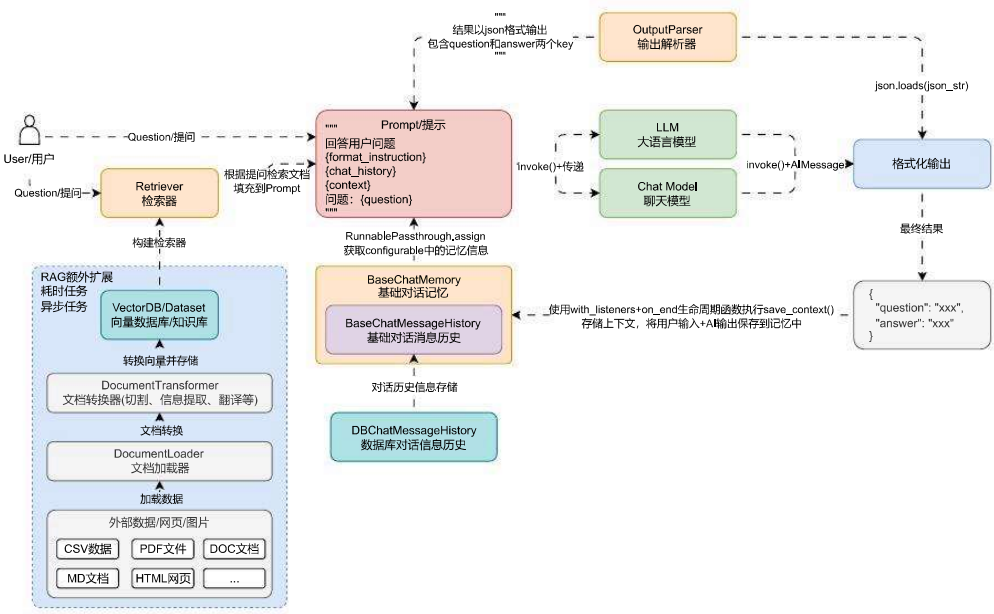
01. DocumentTransformer 组件

在 LangChain 中，使用 `文档加载器` 加载得到的文档一般来说存在着几个问题：**原始文档太大、原始文档的数据格式不符合需求（需要英文但是只有中文）、原始文档的信息没有经过提炼**等问题。

如果将这类数据直接转换成向量并存储到数据库中，会导致在执行相似性搜索和 RAG 的过程中，错误率大大提升。所以在 LLM 应用开发中，在加载完数据后，一般会执行多一步 `转换` 的过程，即将加载得到的 `文档列表` 进行转换，得到符合需求的 `文档列表`。

转换涵盖的操作就非常多，例如：**文档切割、文档属性提取、文档翻译、HTML 转文本、重排、元数据标记**等都属于转换。

在前面的聊天机器人架构中添加上转换步骤，更新后的 `聊天机器人架构/运行流程` 如下所示：



在 LangChain 中针对文档的转换也统一封装了一个基类 `BaseDocumentTransformer`，**所有涉及到文档的转换的类均是该类的子类**，将大块文档切割成 chunk 分块的文档分割器也是 `BaseDocumentTransformer` 的子类实现。

`BaseDocumentTransformer` 基类封装了两个方法：

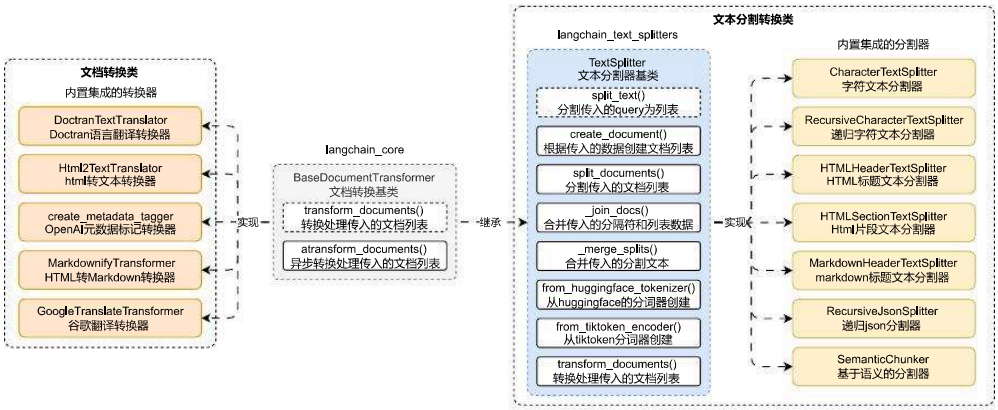
1. `transform_documents()`：抽象方法，传递文档列表，返回转换后的文档列表。
2. `atransform_documents()`：转换文档列表函数的异步实现，如果没有实现，则会委托 `transform_documents()` 函数实现。

在 LangChain 中，文档转换组件分成了两类：`文档分割器(使用频率高)`、`文档处理转换器(使用频率低，老版本写法)`。

并且目前 LangChain 团队已经将 `文档分割器` 这个高频使用的部分单独拆分成一个 Python 包，哪怕不使用 LangChain 框架本身进行开发，也可以使用其文本分割包，快速分割数据，在使用前必须执行以下命令安装：

```
pip install -qU langchain-text-splitters
```

对于文本分割器来说，除了继承 `BaseDocumentTransformer`，还单独设置了文本分割器基类 `TextSplitter`，从而去实现更加丰富的功能，`BaseDocumentTransformer` 衍生出来的类图：



02. 字符分割器基础使用技巧

在文档分割器中，最简单的分割器就是——**字符串分割器**，这个组件会基于给定的字符串进行分割，默认为 `\n\n`，并且在分割时会尽可能保证数据的连续性。分割出来每一块的长度是通过字符数来衡量的，使用起来也非常简单，实例化 `CharacterTextSplitter` 需传递多个参数，信息如下：

1. `separator`：分隔符，默认为 `\n\n`。
2. `is_separator_regex`：是否正则表达式，默认为 `False`。
3. `chunk_size`：每块文档的内容大小，默认为 `4000`。
4. `chunk_overlap`：块与块之间重叠的内容大小，默认为 `200`。
5. `length_function`：计算文本长度的函数，默认为 `len`。
6. `keep_separator`：是否将分隔符保留到分割的块中，默认为 `False`。
7. `add_start_index`：是否添加开始索引，默认为 `False`，如果是的话会在元数据中添加该切块的起点。
8. `strip_whitespace`：是否删除文档头尾的空白，默认为 `True`。

如果想将文档切割为不超过 500 字符，并且每块之间文本重叠 50 个字符，可以使用 `CharacterTextSplitter` 来实现，代码如下：

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

# 1.构建Markdown文档加载器并获取文档列表
loader = UnstructuredMarkdownLoader("./项目API文档.md")
documents = loader.load()

# 2.构建分割器
text_splitter = CharacterTextSplitter(
    separator="\n\n",
    chunk_size=500,
    chunk_overlap=50,
    add_start_index=True,
)

# 3.分割文档列表
```

```
chunks = text_splitter.split_documents(documents)

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

输出内容:

```
Created a chunk of size 771, which is longer than the specified 500
Created a chunk of size 980, which is longer than the specified 500
Created a chunk of size 542, which is longer than the specified 500
Created a chunk of size 835, which is longer than the specified 500
块内容大小:251,元数据:{'source': './项目API文档.md', 'start_index': 0}
块内容大小:451,元数据:{'source': './项目API文档.md', 'start_index': 246}
块内容大小:771,元数据:{'source': './项目API文档.md', 'start_index': 699}
块内容大小:435,元数据:{'source': './项目API文档.md', 'start_index': 1472}
块内容大小:497,元数据:{'source': './项目API文档.md', 'start_index': 1859}
块内容大小:237,元数据:{'source': './项目API文档.md', 'start_index': 2359}
块内容大小:980,元数据:{'source': './项目API文档.md', 'start_index': 2598}
块内容大小:438,元数据:{'source': './项目API文档.md', 'start_index': 3580}
块内容大小:293,元数据:{'source': './项目API文档.md', 'start_index': 4013}
块内容大小:498,元数据:{'source': './项目API文档.md', 'start_index': 4261}
块内容大小:463,元数据:{'source': './项目API文档.md', 'start_index': 4712}
块内容大小:438,元数据:{'source': './项目API文档.md', 'start_index': 5129}
块内容大小:542,元数据:{'source': './项目API文档.md', 'start_index': 5569}
块内容大小:464,元数据:{'source': './项目API文档.md', 'start_index': 6113}
块内容大小:835,元数据:{'source': './项目API文档.md', 'start_index': 6579}
块内容大小:489,元数据:{'source': './项目API文档.md', 'start_index': 7416}
```

使用 `CharacterTextSplitter` 进行分割时，虽然传递了 `chunk_size` 为 500，但是仍然没法确保分割出来的文档一直保持在这个范围内，这是因为在底层 `CharacterTextSplitter` 是先按照分割符号拆分整个文档，然后循环遍历拆分得到的列表，将每个列表逐个相加，直到最接近 `chunk_size` 窗口大小时则完成一个 Document 的组装。

但是如果基于分割符号得到的文本，本身长度已经超过了 `chunk_size`，则会直接进行警告，并且将对应的文本单独变成一个块。

核心代码如下:

```
# langchain_text_splitters/character->CharacterTextSplitter::split_text
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)

def _split_text_with_regex(
    text: str, separator: str, keep_separator: bool
) -> List[str]:
```

```
    # Now that we have the separator, split the text
    if separator:
        if keep_separator:
            # The parentheses in the pattern keep the delimiters in the result.
            _splits = re.split(f"({separator})", text)
            splits = [_splits[i] + _splits[i + 1] for i in range(1, len(_splits),
2)]
            if len(_splits) % 2 == 0:
                splits += _splits[-1:]
            splits = [_splits[0]] + splits
        else:
            splits = re.split(separator, text)
    else:
        splits = list(text)
    return [s for s in splits if s != ""]
```