4数据连接（LEDVR 工作流）

在这一章，我们将主要探讨如何在Langchain框架中连接外部的数据，即数据连接模块（Data Connection）。我们的生活周围充斥着各种各样的数据，例如本地的文档、网页上的知识、企业内部的知识库、各类研究报告、软件数据库以及聊天的历史记录等。这些数据，无论是广泛的互联网数据，还是具有特定价值的企业内部数据，都是我们构建和优化大语言模型的重要资源。

但你可能会问，既然我们已经有了强大的大语言模型，例如OpenAI的gpt-4，为什么我们还需要连接外部的数据呢？原因其实很简单，那就是大语言模型的 “知识 ”是有限的。以OpenAI的gpt-4为例，它的数据集只训练到2021年的9月份，也就是说，这个时间之后的数据并没有被模型学习和理解。所以，到2023年下旬，我们仍会看到ChatGPT在其界面上提示： ChatGPT可能会产生关于人、地点或事实的不准确信息。 这是因为模型在训练数据集之外的知识领域中，其预测能力是受限的。

除此之外，我们还需要个性化的知识，比如企业的内部知识。想象一下，如果你是一个企业，你可能会希望你的聊天机器人能够理解和回答一些关于你的产品或服务的具体问题，这些问题的答案往往需要依赖于你企业内部的专有知识。大语言模型无法直接访问这些知识，因此我们需要将这些知识以某种方式连接到我们的大语言模型中。

连接外部数据不仅可以修复大语言模型的“知识 ”缺失，而且还能让我们开发的应用程序更加“可靠”。当我们的模型需要回答一个问题时，它可以根据真实的外部数据进行回答，而不是仅仅依赖于它在训练时学习的知识。例如，当我们问模型"2023年的新冠病毒疫苗有哪些副作用？”时，模型可以根据最新的医学研究报告来提供答案，而不是依赖于它在两年前学习的可能已经过时的知识。

这些大语言模型不仅需要连接外部的数据，填补缺失的“知识”，同时还受到了提示词的限制。正如我们在模型I/O 的提示词模板数据来源提到的，构建好的提示词模板需要依靠外部数据。然而，这种提示词的字符数量是有限的，这就是我们所说的Max Tokens 概念。

Max Tokens，是指模型在执行任务时能够接收和处理的最大 Token 数量。这个数量包括了我们提交给模型的提示词以及模型返回给我们的答案。也就是说，我们同模型的交互，无论是请求还是响应，都会累积到最大Token数量的计数中。比如我们发出的提示词数量是1000 Tokens，返回的也是1000 Tokens，那么总共Tokens数量就是2000 Tokens，而不仅仅是1000 Tokens。

另外，在大语言模型中，一个Token并不是指一个字符，而是指一个词或者一个词的一部分。对于英文，一个Token可能是一个完整的单词，也可能是一个单词的一部分。对于中文，通常一个汉字就是一个Token。这是由语言模型的编码方式决定的。

让我们以英文为例，英文通常会被分割为子词或者字符。例如，“apple”可能被分割为一个Token，即["apple"]，而“apples”可能被分割为两个Token，即["apple", "s"]。这是因为模型在训练时学习到，“s” 常常用于表示复数。所以，“apples”被分割为两个Token。

对于中文，由于其语言特性，通常每个字符就是一个Token，即每个汉字都是一个Token。但是在某些特殊情况下，如一些复杂的或者不常见的汉字，可能会被编码为两个或者更多的Token。比如，例如，gpt-4模型的Max Tokens是8192个 Tokens 。如果我们将 8192 个 Tokens 全部用于中文，实际能处理的汉字数量可能会比8192少。

以OpenAI的产品为例，我们可以看到，不同的模型型号有着不同的Max Tokens设置。例如，gpt-4模型的Max Tokens是8192个，这意味着它一次能处理8192个Tokens，假设平均每个英文单词对应0.75个Token，那么这个模型大约可以处理约 10922 个英文单词（8192除以0.75）。而gpt-4-32k模型的Max Tokens则是32768个，是gpt-4的四倍，这使得它能处理更长的文本，理解更复杂的上下文。

因此，我们在使用这些大型语言模型时，需要了解其Max Tokens的限制，并根据实际需要选择合适的模型。同时，也需要注意，随着技术的不断更新和迭代，模型的性能和能力也会有所提高，我们需要密切关注这些变化，以确保我们能够最大限度地利用这些强大的工具。

为了解决这些大型语言模型的限制问题，LangChain设计了数据连接模块。这个模块的目的是检索到用户输入的问题相关的外部数据，包括筛选相关问题和相关的文档。然后，这些相关数据会形成提示词模板，提交给LLM或Chat Model类型的模型包装器。这些模型包装器封装了各个大语言模型平台的底层API，使得我们可以方便地与这些平台进行交互，获取大语言模型平台的输出。

然而，一旦我们加载了这些外部的文档数据，我们经常会希望对它们进行转换以更好地适应我们的应用。最简单的例子是我们可能希望将一个长文档分割成较小块，避免超过gpt-4模型的Max Tokens。为了实现这一目标，LangChain框架提供了一系列内置的文档转换器，使得分割、组合、过滤和以其他方式操作文档变得容易。例如，我们可以使用这些转换器将一篇长篇的研究报告分割成一系列的小段落，每个段落都可以作为一个独立的输入提交给我们的模型。

数据连接模块的LEDVR

数据连接模块是一个多功能的数据连接工具集成，可以方便记忆为LEDVR，分别代表加载器 (Loader), 嵌入模型包装器(Text Embedding Model), 文档转换器 (Document Transformers)，向量存储库 (VectorStore) 和检索器 (Retriever) 。

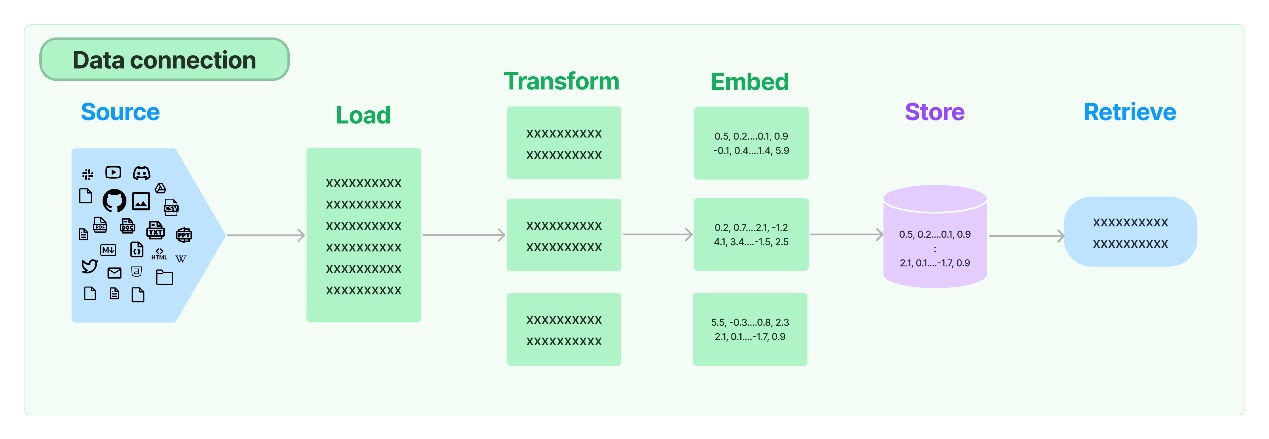
L代表加载器 (Loader),，负责从各种来源加载数据作为Document，其中Document是由文本和相关元数据组成的。无论你的数据是简单的.txt文件，还是任何网页的文本内容，Loader都可以将它们加载为Document。

E代表嵌入模型包装器(Text Embedding Model)，是Langchain 封装了各个模型平台的Text Embedding Model 类型的 API 接口。效果同模型I/O 模块的LLM 和 Chat Model 类型的模型包装器。

D代表文档转换器 (Document Transformers)。一旦你加载了文档，你通常会希望将它们转化以更好地适应你的应用。数据连接模块提供了一系列内置的文档转换器，使得分割、组合、过滤和其他方式的操作文档变得简单。比如OpenAI 平台的主要文本嵌入模型的型号是“text-embedding-ada-002”。

V代表向量存储库 (VectorStore)，是用于存储和检索嵌入向量的工具，处理的数据是模型平台的文本嵌入模型（Text Embedding Model）转换的向量数据，这是处理非结构化数据的一种常见方法。VectorStore负责存储嵌入数据并为你执行向量搜索。在查询时，可以将非结构化查询嵌入，并检索与嵌入查询“最相似”的嵌入向量。

R代表检索器 (Retriever)，它是一个接口，返回给定非结构化查询的文档。它比VectorStore更通用。检索器无需存储文档，只需要返回（或检索）它。



LEDVR 的示例代码

让我们以一个具体的代码示例来解析LangChain数据处理流程中的各个步骤。在这个例子中，我们将使用LangChain的各种工具，包括文档加载器L、文本嵌入模型E、文档转换器D、向量存储库V和检索器R，来完成一个从加载、转换、嵌入、存储到检索的完整流程。

首先，我们使用文档加载器L，创建一个WebBaseLoader实例，用于从网络加载数据。在这个例子中，我们加载的是一个博客文章。加载器读取该网址的内容，并将其转换为一份文档数据。

from langchain.document\_loaders import WebBaseLoader

loader = WebBaseLoader("http://developers.mini1.cn/wiki/luawh.html")

data = loader.load()

随后，我们使用文本嵌入模型E，将这些分割后的文本数据转换为向量数据。我们创建一个OpenAIEmbeddings实例，用于将文本转换为向量。

from langchain.embeddings.openai import OpenAIEmbeddings

embedding = OpenAIEmbeddings(openai\_api\_key="填入你的OpenAI密钥")

接下来，文档转换器D，我们需要将这份文档数据进行分割，转换为 Document 数据。这是为了让数据更好地适应数据连接模块的工作流程。我们创建一个RecursiveCharacterTextSplitter实例，并指定每个片段的大小为500个字符。然后，我们使用这个分割器将文档数据分割成多个片段。

from langchain.text\_splitter import RecursiveCharacterTextSplitter

text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=500, chunk\_overlap=0)

splits = text\_splitter.split\_documents(data)

然后，向量存储库V，我们创建一个向量存储库：FAISS实例，用于存储这些向量数据。

from langchain.vectorstores import FAISS

vectordb = FAISS.from\_documents(documents=splits, embedding=embedding)

最后，检索器R，我们需要在这些数据中进行检索。我们创建一个ChatOpenAI实例和一个MultiQueryRetriever实例，用于执行检索操作。在这个例子中，我们检索的问题是“Task Decomposition的方法有哪些？”。

from langchain.chat\_models import ChatOpenAI

from langchain.retrievers.multi\_query import MultiQueryRetriever

question = "LUA的宿主语言是什么?"

llm = ChatOpenAI(openai\_api\_key="填入你的OpenAI密钥")

retriever\_from\_llm = MultiQueryRetriever.from\_llm(

retriever=vectordb.as\_retriever(), llm=llm

)

docs = retriever\_from\_llm.get\_relevant\_documents(question)

通过这个例子，最后我们获得了四个与相关的Document数据。打印第一条我们找到了“LUA的宿主语言是什么?”这个问题的答案，“Lua提供了非常易于使用的扩展接口和机制：由宿主语言(通常是C或C++)提供这些功能”。

print(docs[0])

<langchain.vectorstores.faiss.FAISS at 0x228dfa4b050>

Document(page\_content='Lua提供了非常易于使用的扩展接口和机制：由宿主语言(通常是C或C++)提供这些功能，Lua可以使用它们，就像是本来就内置的功能一样。 其它特性:', metadata={'source': '<http://developers.mini1.cn/wiki/luawh.html>', 'title': '什么是Lua编程 | 开发者脚本帮助文档', 'description': '迷你世界开发者接口文档', 'language': 'zh-CN'})

我们可以看到LangChain如何将文档加载器L、文本嵌入模型E、文档转换器D、向量存储库V和检索器R有机地组合在一起，形成了一个从加载、转换、嵌入、存储到检索的完整流程。

两种数据类型

在数据连接模块中，操作的主要是两种类型的数据：文档数据和向量数据。这两种数据类型是LangChain数据连接模块中处理流程的类型，它们支持数据在这个模块中的自由流通和转换。

文档数据是数据连接模块的起点。这些数据主要通过加载器从各种不同的源加载进来。无论是简单的文本文件，还是网页内容，甚至是YouTube视频的转录，都可以被加载为文档数据。在这个过程中，每一份文档都被视为一个包含文本和相关元数据的单元。

一旦文档数据被加载进来，它们就可以被传递给文档转换器进行处理。这些处理最主要的功能是分割文档，其他功能还有压缩文档、过滤文档。这些转换过程使得文档数据更好地适应我们的应用需求。

这些经过处理的文档数据会被传递给文本嵌入模型，在这里，它们会被转化为向量数据。向量数据实际上是一种将非结构化的文本转化为一列浮点数的表示形式。这些向量数据会被存储在向量存储库中，并在需要时进行检索。检索过程由检索器完成，它根据用户的查询返回相应的数据。

总的来说，数据在LangChain的数据连接模块中，通过文档加载器L、文本嵌入模型E、文档转换器D、向量存储库V和检索器R，形成了一个从加载、转换、嵌入、存储到检索的完整流程。在这个流程中，数据被规范化为文档数据和向量数据两种类型，使得数据能够自由地在各个组件之间流通和转换。

## 4.1 L文档加载器

在LangChain的数据处理流程中，Document Loaders起着至关重要的作用。它们被用来从各种来源加载数据，并将其转换为“文档”（Document）的形式。

一个“文档”可以理解为一段文本及其相关元数据。例如，我们可以有专门用于加载简单.txt文件的文档加载器，也可以有加载任何网页的文本内容的文档加载器，甚至还可以有加载YouTube视频转录文本的文档加载器。不同类型的文档加载器，使得LangChain可以从各种各样的数据源中抽取并处理数据。

这些文档加载器都会暴露出一个名为”load”的方法，用于从配置的数据源加载数据作为文档。这个”load”方法可以从指定的数据源中读取数据，并将其转换成一份或多份文档。这使得LangChain能够处理各种形式的输入数据，不仅仅限于文本文件，还可以是网页、视频字幕等等。

值得注意的是，文档加载器还可以选择性地实现一个名为”lazy load”的方法，这个方法的作用是实现数据的懒加载，即在需要时才将数据加载到内存中。这种方式可以有效减少内存占用，并提高数据处理的效率。

总的来说，通过Document Loaders，LangChain可以将各种各样的数据源无缝地转换为标准的文档形式，为后续的数据处理和分析提供了坚实的基础。

下面是最简单的文档加载器的代码示例：

加载简单.txt文件的文档加载器。

from langchain.document\_loaders import TextLoader  
# 代码仓库中有这个文件，也可以加入自己的测试文件。如果是文件中包含中文，请指定encoding="utf-8"  
loader = TextLoader(file\_path="./index.md",encoding="utf-8")  
loader.load()

文件中的所有内容都被加载到了Document 数据中。

[Document(page\_content='在语言模型中，一个Token并不是指一个字符，而是指一个词或者一个词的一部分。对于英文，一个Token可能是一个完整的单词，也可能是一个单词的一部分。对于中文，通常一个汉字就是一个Token。这是由语言模型的编码方式决定的。\n\n让我们以英文为例。在许多NLP任务和一些语言模型中，英文通常会被分割为子词或者字符。例如，“apple”可能被分割为一个Token，即["apple"]，而“apples”可能被分割为两个Token，即["apple", "s"]。这是因为模型在训练时学习到，“s”常常用于表示复数。所以，“apples”被分割为两个Token。\n\n对于中文，由于其语言特性，通常每个字符就是一个Token，即每个汉字都是一个Token。但是在某些特殊情况下，如一些复杂的或者不常见的汉字，可能会被编码为两个或者更多的Token。这通常发生在使用子词编码方法的模型中，如Byte Pair Encoding（BPE）或Unigram Language Model（ULM）。\n\n至于每个英文单词对应0.75个Token的例子，这是一个假设的平均值，用于说明如果一个英文单词被分割为多个Token，那么模型能处理的单词数量可能会比Token数量多。实际情况下，这个比值可能会根据具体的文本和模型的编码方式有所不同。\n\n这里需要明确的是，无论英文还是中文，一个Token并不一定等同于一个字符或一个单词，而是取决于具体的编码方式。在理解和使用语言模型时，我们需要考虑到这一点。', metadata={'source': './index.md'})]

### 4.1.1 常见的加载器

LangChain的数据加载能力并不限于单一的数据源或格式，它可以处理各种常见的数据格式，例如CSV、文件目录、HTML、JSON、Markdown以及PDF等。下面，我们将分别解析一下这些不同格式数据的加载方法。

CSV 加载器

逗号分隔值（Comma-Separated Values，简称CSV）文件是一种使用逗号来分隔值的文本文件。每一行都是一条数据记录，每条记录包含一个或多个用逗号分隔的字段。LangChain可以加载CSV数据，其中每一行都被视为一个独立的文档。

这个加载器的典型用法是创建一个 CSVLoader 实例，然后调用其 load 方法来加载文件。

CSVLoader是BaseLoader的子类，主要用于从CSV文件加载数据，将其转换为一系列Document对象。每个Document代表CSV文件中的一行，其中每行被转换为键值对，并输出到Document的page\_content中。对于从CSV文件加载的每个文档，默认情况下源都被设置为file\_path参数的值。如果你设置了source\_column参数为CSV文件中的列名，那么每个文档的源将被设置为指定source\_column的列的值。

它的主要方法是：

- load: 从CSV文件加载数据，转化为Document对象的列表。

你可以设置的主要参数包括：

- file\_path：CSV文件的路径。

- source\_column：可选参数，用于指定作为文档源的列的名称。

- encoding：可选参数，用于指定打开文件的编码方式。

- csv\_args：可选参数，传递给csv.DictReader的参数。

这个类的典型用法是创建一个CSVLoader实例，然后调用其load方法来加载CSV文件的数据。例如：

loader = CSVLoader(file\_path='data.csv', encoding='utf-8')

documents = loader.load()

在这个例子中，documents就是从data.csv文件中加载的Document对象的列表。每个Document对象代表文件中的一行数据。

文件目录的加载

对于文件目录，LangChain提供了一种方法来加载目录中的所有文档。在底层，它默认使用UnstructuredLoader来实现这个功能。这意味着，只要将文档存放在同一目录下，无论数量多少，LangChain都能够将它们全部加载进来。

这个加载器的典型用法是创建一个DirectoryLoader 实例，然后调用其 load 方法来加载文件。

DirectoryLoader 是 BaseLoader 的子类，主要用于从一个指定的目录加载文档。每个从目录中加载的文件都将被处理为一个 Document 对象。

它的主要方法是：

- load ：加载目录中的所有文件，并将它们作为文档返回。这个方法不接受任何参数。

你可以设置的主要参数包括：

- loader\_cls ：用于加载文件的加载器类，它需要是 BaseLoader 的子类。

- loader\_kwargs ：传递给加载器类的参数。

- recursive ：是否递归加载子目录中的文件。

- show\_progress ：是否显示加载进度。

这个类的典型用法是创建一个 DirectoryLoader 实例，然后调用其 load 方法来加载目录中的文件。例如：

loader = DirectoryLoader(path='data\_directory')

documents = loader.load()

在这个例子中， documents 是从 data\_directory 目录中加载的 Document 对象的列表。每个 Document 对象代表目录中的一个文件。

HTML文件的加载

HTML是用于设计在Web浏览器中显示的文档的标准标记语言。LangChain可以将HTML文档加载为我们后续使用的文档格式。这就意味着，我们可以直接从网页上提取并处理数据。

这个加载器的典型用法是创建一个UnstructuredHTMLLoader或者BSHTMLLoader 实例，然后调用其 load 方法来加载文件。它们都可以将HTML文件加载为我们可以在后续过程中使用的文档格式。

我们可以利用 BSHTMLLoader 类来完成这项工作。这个加载器会从HTML文件中提取文本，并将其存储在 page\_content 中。同时，它还会提取网页标题，并将其作为 title 存储在元数据 metadata 中。

这种方法的优点在于它可以从HTML文档中提取出结构化的信息，比如段落、标题等，这些信息在后续的处理中可能会很有用。

使用 BSHTMLLoader 加载HTML文档的例子如下：

from langchain.document\_loaders import BSHTMLLoader

loader = BSHTMLLoader(file\_path='example.html')

documents = loader.load()

在这个例子中， documents 是从 example.html 文件中加载的 Document`对象的列表，其中每个`Document`对象都代表HTML文件中的一部分内容。

JSON加载器

JSON是一种使用人类可读文本来存储和传输数据对象的开放标准文件格式和数据交换格式，这些对象由属性-值对和数组（或其他可序列化值）组成。LangChain的JSONLoader使用指定的jq模式来解析JSON文件。jq是一种适用于Python的软件包。JSON文件的每一行都被视为一个独立的文档。

这个加载器的典型用法是创建一个JSONLoader 实例，然后调用其 load 方法来加载文件。

JSONLoader 类是用于加载JSON文件并将其转换为 Document 对象的加载器。这个类可以通过引用一个jq schema（一种用于处理JSON数据的查询语言）来提取文本并加载到文档中。

这个类的主要方法是 load ，它会加载并返回来自JSON文件的文档。在加载过程中，会使用到 jq\_schema 参数指定的jq模式来处理JSON数据。

JSONLoader 类的构造函数接受以下参数：

- file\_path ：JSON文件的路径。

- jq\_schema ：用于从JSON中提取数据或文本的jq模式。

- content\_key ：如果jq模式的结果是对象（字典）的列表，则使用此键来从JSON中提取内容。

- metadata\_func ：一个函数，接受由jq模式提取的JSON对象和默认的元数据，返回更新后的元数据的字典。

下面是一个使用 JSONLoader 加载JSON文件的例子：

from langchain.document\_loaders import JSONLoader

loader = JSONLoader(file\_path='example.json', jq\_schema='.[]')

documents = loader.load()

在这个例子中， documents 是从 example.json 文件中加载的 Document`对象的列表，其中每个 Document 对象都代表JSON文件中的一部分内容。

下面的列表提供了一些可能的 jq\_schema 参考值，用户可以根据 JSON 数据的结构使用这些值来提取内容。

JSON -> [{"text": ...}, {"text": ...}, {"text": ...}]

jq\_schema -> ".[].text"

JSON -> {"key": [{"text": ...}, {"text": ...}, {"text": ...}]}

jq\_schema -> ".key[].text"

JSON -> ["...", "...", "..."]

jq\_schema -> ".[]"

Markdown加载器

Markdown是一种使用纯文本编辑器创建格式化文本的轻量级标记语言。LangChain可以将Markdown文档加载为我们后续使用的文档格式。你可以通过指定 mode="elements" 来分离Unstructured 会为不同的文本块创建不同的元素。

这个加载器的典型用法是创建一个UnstructuredMarkdownLoader实例，然后调用其 load 方法来加载文件。

下面是一个使用 UnstructuredMarkdownLoader 加载JSON文件的例子：

markdown\_path = "../../../../../README.md"

loader = UnstructuredMarkdownLoader(markdown\_path, mode="elements")

documents = loader.load()

PDF文件加载器

PDF（Portable Document Format）是Adobe在1992年开发的一种文件格式，用于以独立于应用软件、硬件和操作系统的方式呈现文档，包括文本格式化和图像。LangChain可以将PDF文档加载为我们后续使用的文档格式。

PDF是目前应用最广泛的一种文档类型，包括但不限于金融研究报告、学术论文、市场研究报告等。LangChain 的数据模块设计了多种文档加载器以加载 PDF 文档。以下我们将介绍一些主要的 PDF 文档加载器及其用法。

1. PyPDF 文档加载器 ：它可以将 PDF 使用 PyPDF 加载为文档数组，每个文档包含页面内容和包含页码的元数据。示例如下：

from langchain.document\_loaders import MathpixPDFLoader

loader = MathpixPDFLoader("example\_data/layout-parser-paper.pdf")

或者使用 UnstructuredPDFLoader：

from langchain.document\_loaders import UnstructuredPDFLoader

loader = UnstructuredPDFLoader("example\_data/layout-parser-paper.pdf")

在底层，Unstructured 会为不同的文本块创建不同的元素。默认情况下，我们会将这些元素合并在一起，但你可以通过指定 mode="elements" 来轻松地保持这种分离。

2. 在线 PDF 加载器 ：我们也可以加载在线 PDF 文件，将其转化为我们可以在下游使用的文档格式。这对于各种在线 PDF 网站（例如 https://open.umn.edu/opentextbooks/textbooks/ 和 https://arxiv.org/archive/）都非常有用。注意，所有其他 PDF 加载器也可以用于获取远程 PDF，但 OnlinePDFLoader 是一个遗留功能，专门与 UnstructuredPDFLoader 一起使用。

from langchain.document\_loaders import OnlinePDFLoader

loader = OnlinePDFLoader("https://arxiv.org/pdf/2302.03803.pdf")

3. PyPDFium2 文档加载器 ：使用 PyPDFium2 加载 PDF 文件。

from langchain.document\_loaders import PyPDFium2Loader

loader = PyPDFium2Loader("example\_data/layout-parser-paper.pdf")

data = loader.load()

4. PDFMiner 文档加载器 ：使用 PDFMiner 加载 PDF 文件。

from langchain.document\_loaders import PDFMinerLoader

loader = PDFMinerLoader("example\_data/layout-parser-paper.pdf")

5. 使用 PDFMiner 生成 HTML 文本 ：这对于将文本语义地划分为各个部分非常有帮助，因为输出的 HTML 内容可以通过 BeautifulSoup 进行解析，以获取关于字体大小、页码、PDF 头/页脚等的更多结构化和丰富的信息。

6. PyMuPDF 文档加载器 ：这是 PDF 解析选项中最快的一种，它包含关于 PDF 及其页面的详细元数据，且每页返回一个文档。

from langchain.document\_loaders import PyMuPDFLoader

7. 从目录加载 PDF ：

from langchain.document\_loaders import PyPDFDirectoryLoader

8. pdfplumber 文档加载器 ：与 PyMuPDF 类似，输出的文档包含关于 PDF 及其页面的详细元数据，且每页返回一个文档。

以上就是 LangChain 支持的 PDF 文档加载器及其使用示例。

### 4.1.2加载器的分类

LangChain通过实现各种文档加载器，实现了对多元化数据源的处理。这些加载器的设计使得LangChain可以从各种服务提供商加载数据，进一步扩大了其在不同业务场景的应用范围。下面，我们将对这些加载器进行分类并简要描述。

这类加载器文件和目录加载器主要处理存储在本地或云端的文件和目录，例如CSVLoader，DirectoryLoader，JSONLoader，S3DirectoryLoader，S3FileLoader等。

文本和标记语言文件加载器，这类加载器负责处理各种文本和标记语言格式的文件。

比如BSHTMLLoader，MarkdownLoader，TextLoader，UnstructuredHTMLLoader，UnstructuredMarkdownLoader等。

文档和电子书加载器，处理电子书和文档的加载器包括Docx2txtLoader，PDFMinerLoader，UnstructuredEPubLoader，UnstructuredPDFLoader，UnstructuredWordDocumentLoader等。

社交媒体和在线论坛加载器，这类加载器主要用于处理来自社交媒体和在线论坛的数据，如FacebookChatLoader，GitHubIssuesLoader，RedditPostsLoader，TwitterTweetLoader等。

云服务提供商加载器，对于云服务提供商的数据源，LangChain提供了相应的加载器，如AirbyteJSONLoader，AirtableLoader，AzureBlobStorageContainerLoader，AzureBlobStorageFileLoader，GoogleDriveLoader，OneDriveLoader等。

文本聊天和消息服务加载器，这类加载器主要处理各种聊天和消息服务的数据，

比如DiscordChatLoader，SlackDirectoryLoader，TelegramChatApiLoader，TelegramChatFileLoader，WhatsAppChatLoader等。

网页和网站数据加载器，对于网页和网站的数据，如SitemapLoader，UnstructuredURLLoader，WebBaseLoader等加载器提供了处理方法。

其他特殊类型的加载器，这类加载器主要处理一些特定的或者特殊的数据源，如HuggingFaceDatasetLoader，UnstructuredImageLoader，WeatherDataLoader，YoutubeAudioLoader等。

通过实现这些加载器，LangChain可以从各种各样的服务提供商处加载数据，进一步提升了它的多样化数据处理能力。这对于需要处理大量、多样性数据的机器学习、自然语言处理等领域具有非常重要的意义。

## 4.2 E嵌入模型包装器

在深度学习和自然语言处理领域，嵌入（Embedding）是一种将文本数据转化为浮点数值表示形式的技术，它能够衡量两段文本之间的相关性。嵌入的一个典型例子是词嵌入，它将每个词映射到一个多维空间中的点，使得语义上相似的词在空间中的距离更近。词嵌入是将词语映射到向量空间中的一种技术，它通过对大量文本数据的训练，为每个词语生成一个高维向量。这个向量能够捕获词语的语义信息，例如，相似的词语（如“男”和“国王”，“女”和“女王”）在向量空间中的位置会非常接近。这是因为嵌入模型在训练过程中学习到了词语之间的语义关系。

例如，我们可以使用预训练的Word2Vec或GloVe等模型得到每个词的向量表示。假设“国王”的向量表示为[1.2, 0.7, -0.3]，“男”的向量表示为[1.1, 0.6, -0.2]，“女王”的向量表示为[-0.9, -0.8, 0.2]，“女”的向量表示为[-0.8, -0.7, 0.3]。我们会发现，相同性别的词语（如“国王”和“男”）在向量空间中的距离更近，这就反映了他们之间的语义关系。这种关系可以通过计算向量之间的余弦相似度来量化。

词嵌入的一个重要应用就是在自然语言处理任务中，例如文本分类、命名实体识别、情感分析等。通过将词语转换为向量，我们可以利用深度学习模型来处理文本数据，实现对语言的理解。

Langchain框架提供了一个名为Embeddings的类，它为多种文本嵌入模型（如OpenAI、Cohere、Hugging Face等）提供了统一的接口。该类可以将一段文本转化为一个向量表示，这使得我们可以在向量空间中处理文本，执行如语义搜索这样的操作，寻找在向量空间中最相似的文本。我们把实例化的Embeddings类称为嵌入模型包装器。同 Model I/O 模块的LLM 类和 Chta Model 类型模型包装器并列为三大模型包装器。OpenAI 平台的嵌入模型，使用大量的文本数据进行训练，以尽可能地捕捉和理解人类语言的复杂性。这使得OpenAI的嵌入模型可以生成高质量的向量表示，有效地捕捉文本中的语义关系和模式。在Langchain框架中最常用的是OpenAI平台的嵌入模型，第二代嵌入模型，text-embedding-ada-002。这种嵌入模型对于搜索、聚类、推荐、异常检测和分类任务等都有很好的效果。

嵌入模型包装器同其他两个模型包装器一样使用，需要导入 Embedding 类，设置密钥。嵌入模型包装器提供了两个主要的方法，分别是 embed\_documents 和 embed\_query 。前者接受一组文本作为输入并返回他们的嵌入向量，而后者接受一个文本并返回其嵌入向量。这两个方法之所以分开，是因为模型平台的嵌入模型对于待搜索的文档和搜索查询本身有不同的嵌入方法。

例如，在使用OpenAI的嵌入模型时，我们可以通过以下代码示例来嵌入一组文档和一个查询：

通过简单的代码示例，展示这个嵌入模型包装器的使用过程。

from langchain.embeddings import OpenAIEmbeddings

embeddings\_model = OpenAIEmbeddings(openai\_api\_key="填入你的密钥")

接下来，我们可以使用 embed\_documents 方法将一系列文本嵌入为向量。例如，我们可以将以下五句话嵌入为向量：

embeddings = embeddings\_model.embed\_documents(

[

"Hi there!",

"Oh, hello!",

"What's your name?",

"My friends call me World",

"Hello World!"

]

)

len(embeddings), len(embeddings[0])

这将返回一个嵌入向量列表，其中每个嵌入向量的长度为1536。

最后，我们可以使用 embed\_query方法将单个查询嵌入为向量。这在你想要将一个查询和其他已嵌入的文本进行比较时非常有用：

embedded\_query = embeddings\_model.embed\_query("What was the name mentioned in the conversation?")

embedded\_query[:5]

这将返回该查询的嵌入向量，我们在这里只展示了向量的前5个元素。

[0.0053587136790156364,

-0.0004999046213924885,

0.038883671164512634,

-0.003001077566295862,

-0.00900818221271038]

### 4.2.1 嵌入模型包装器的类型

在LangChain中，我们为各种大语言模型平台提供了嵌入模型接口的封装。这其中，OpenAI平台提供的接口封装为“OpenAIEmbeddings”。这类嵌入方式的特点是能够充分利用大规模预训练模型的语义理解能力，其中包括OpenAI、Hugging Face等提供的自然语言处理模型。以下是一些具体的类型：

1. 自然语言模型嵌入：这类嵌入包括“OpenAIEmbeddings”、“HuggingFaceEmbeddings”、“HuggingFaceHubEmbeddings”、“HuggingFaceInstructEmbeddings”、“SelfHostedHuggingFaceEmbeddings”和“SelfHostedHuggingFaceInstructEmbeddings”。这些嵌入方法主要利用诸如OpenAI、Hugging Face等自然语言处理模型进行文本嵌入。

2. AI平台或云服务嵌入：这类嵌入主要依托AI平台或云服务的能力进行文本嵌入，典型的包括Elasticsearch、SagemakerEndpoint、DeepInfra等。这些嵌入方式的主要特点是能够利用云计算的优势，处理大规模的文本数据。

3. 专门的嵌入模型：这类嵌入主要是专门用于处理特定结构文本的嵌入模型，例如AlephAlpha的AsymmetricSemanticEmbedding和SymmetricSemanticEmbedding，适用于结构不同或相似的文本。

4. 自托管嵌入：这类嵌入一般适用于用户自行部署和管理的场景，如SelfHostedEmbeddings，给予用户更大的灵活性和控制权。

5. 仿真或测试用嵌入：FakeEmbeddings一般用于测试或模拟场景，不涉及实际的嵌入计算。

6. 其他类型：此外，LangChain还支持一些其他类型的嵌入方式，如Cohere、LlamaCpp、ModelScope、TensorflowHub、MosaicMLInstructor、MiniMax、Bedrock、DashScope和Embaas等。这些嵌入方式各有特点，能够满足不同的文本处理需求。

在以上这些类型中，用户可以根据自己的具体需求，选择最合适的文本嵌入类型。同时，LangChain也将持续引入更多的嵌入类型，以进一步提升其处理文本的能力。

## 4.3 D文档转换器

在大语言模型开发时代，处理海量文档成为了一个常见且重要的任务。LangChain 框架的数据连接模块为此提供了一系列强大的包装器，其中文档转换器（Document Transformers）就是我们解决这个问题的关键工具之一。

文档转换器的主要任务分为两个步骤：第一步是对文档进行分割，主要由切割器完成，第二步是将分割后的文档转换为Document数据格式。尽管从名称上看，文档转换器主要进行的是转换操作，但实际上，这是从结果出发来定义的。在数据连接模块中，数据以Document对象和向量形式在各个包装器中流通。向量形式的数据由向量存储库管理，而转为向量之前，数据以Document对象的形式存在。

文档转换器的主要任务是将文档数据进行切割和转换为 Document 对象的数据格式。然后，这些数据会被传递给文本嵌入模型包装器进行处理，转换为嵌入向量，最后存储在向量存储库中，检索器从中检索与用户输入的问题相关的文档内容。

你可能会问，为什么我们需要切割文档呢？我们先看看主要用于切割的转换器和加载器之间的区别。

### 4.3.1 加载器和转换器的区别

文档加载器的主要任务是从各种源加载数据，并将这些数据转换为 Document 对象。Document 是一个包含文本及其相关元数据的对象。例如，文档加载器可以加载简单的.txt文件，也可以加载网页的文本内容，甚至可以加载 YouTube 视频的转录内容。这是处理数据的第一步，将不同格式、不同来源的数据统一为 Document 对象，为后续的嵌入向量存储打下基础。

然而，加载后的文档可能非常长，可能包含几十页甚至几百页的内容。处理这样长的文档可能会带来一些问题。一方面，大语言模型平台处理长文本的能力是有限的，例如，某些模型平台有最大 Max Tokens 的限制。另一方面，将整个文档作为一个整体处理可能无法充分发挥模型的效果，因为文档中不同部分的内容可能在语义上存在较大的差异。因此，我们需要将长文档切割为较小的文本块，使得每个文本块在语义上尽可能的一致，这就是转换器的文本切割任务，由文本切割器完成。

文本切割器按照一定的策略将文档切割为多个小块。这些策略可能包括如何切割文本（例如，按照句子切割）、如何确定每个小块的大小（例如，按照一定的字符数切割）等。通过合理的切割，我们可以保证每个小块的内容在语义上尽可能的一致，并且可以被模型平台处理。

因此，文档加载器和文本转换器是处理文档数据的两个重要步骤。文档加载器负责将各种源的数据转换为 Document 对象，而转换器则负责将长文档切割为可以被模型平台处理的小块。通过这两步，我们可以将各种各样的数据转换为适合模型平台处理的形式。

切割的需求

从加载器和转换器的区别我们看到，切割的需求源于对大量文档数据的处理效率、适应模型平台的限制。如果数据太大，增删改查等操作的效率就会受到影响。特别是当我们处理几十页、几百页的文档时，我们需要考虑在何处进行切割，是在使用加载器加载完文档后进行切割，还是在加载之前就进行切割。如果我们在加载大型文件时遇到中断，那么我们就需要重新开始。为了避免这种情况，我们可以将大型文件切割为多个数据块。在 LangChain 框架中，提供了多种内置的转换器完成在加载之前进行切割的任务，满足开发者的切割文档的需求。

转换的需求

当然，文档转换器不仅仅提供了切割功能，它还能根据开发者的需要，提供转换的方法。比如，如果我们已经使用加载器将文档加载并转换为 Document 对象的格式，那么我们可以调用文档转换器的 split\_documents 方法进行切割。

from langchain.document\_loaders import WebBaseLoader

from langchain.text\_splitter import RecursiveCharacterTextSplitter

# 这里我们使用的是WebBaseLoader加载器加载了网络文档

loader = WebBaseLoader("https://lilianweng.github.io/posts/2023-06-23-agent/")

data = loader.load()

# 此时我们切割的Documents 对象的数据

text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=500, chunk\_overlap=0)

splits = text\_splitter.split\_documents(data)

如果我们并没有使用加载器，而是直接使用 Python 的内置方法打开和读取文件，那么我们可以将读取到的结果传递给文档转换器的 create\_documents 方法进行转换。

# 这里我们没有使用加载器.

with open('../../../state\_of\_the\_union.txt') as f:

state\_of\_the\_union = f.read()

from langchain.text\_splitter import CharacterTextSplitter

text\_splitter = CharacterTextSplitter(

separator = "\n\n",

chunk\_size = 1000,

chunk\_overlap = 200,

length\_function = len,

)

texts = text\_splitter.create\_documents([state\_of\_the\_union])

print(texts[0])

此外，如果我们只需要字符串，还可以使用 split\_text 方法进行转换。

with open('../../../state\_of\_the\_union.txt') as f:

state\_of\_the\_union = f.read()

text\_splitter = CharacterTextSplitter(

separator = "\n\n",

chunk\_size = 1000,

chunk\_overlap = 200,

length\_function = len,

)

text=text\_splitter.split\_text(state\_of\_the\_union)[0]

无论开发者使用什么方式读取文件，这些内置的文档转换器都可以帮助我们处理数据，进行切割和转换，满足开发者的转换文档的需求。加载器和转换器的目的都是适应大语言模型平台的合适大小的“块” 数据，这些“块”像快递包裹一样，通过数据连接模块的LEDVR工作流，或被链组件包装，或被代理组件包装，最后提交给模型平台，获取模型平台的响应回答。

### 4.3.2 文本切割器

当处理长文本时，往往需要将文本分割成块。尽管这看起来简单，但实际上可能涉及很多复杂性。理想情况下，您会希望将语义相关的文本部分保持在一起。而“语义相关”的含义可能取决于文本的类型。下面将介绍几种实现这一目标的方法。

文本分割器的工作原理是：将文本分割成小的、语义上有意义的块（通常是句子）。将这些小块开始组合成一个大的块，直到达到某个大小（通过某种函数进行测量）。一旦达到该大小，将该块作为自己的文本片段，然后开始创建新的文本块，新的文本块和前一个文本块会有一些重叠（以保持块与块之间的上下文）。这意味着，您可以沿着两个不同的轴来定制你的文本分割器：文本如何被分割块的大小以及如何被测量。

默认推荐的文本分割器是RecursiveCharacterTextSplitter。这个文本分割器接受一个字符列表，它尝试基于第一个字符进行分割，但如果任何块太大，它就会移动到下一个字符，依此类推。默认情况下，它尝试分割的字符是["\n\n", "\n", " ", ""] 。

除了可以控制分割的字符，您还可以控制以下几点：

length\_function：如何计算块的长度。默认只计算字符数量，但是在此通常会传入一个标记计数器。

chunk\_size：你的块的最大大小（由长度函数测量）。

chunk\_overlap：块之间的最大重叠。有一些重叠可以在块之间保持连续性（例如采用滑动窗口的方式）。

add\_start\_index：是否在元数据中包含每个块在原始文档中的起始位置。

4.3.3文本分割器的使用在处理大规模文本数据时，LangChain提供了多种文本分割方法，以满足各种类型的应用需求。下面我们通过示例代码，展示如何使用不同的文本分割器方法。

1. 按字符分割

这是最简单的方法。它基于字符（默认为"\n\n"）进行分割，并通过字符数量来衡量块的大小。 chunk\_size 属性值设置块大小， chunk\_overlap 属性值设置块之间的最大重叠。

# This is a long document we can split up.

with open('../../../state\_of\_the\_union.txt') as f:

state\_of\_the\_union = f.read()

from langchain.text\_splitter import CharacterTextSplitter

text\_splitter = CharacterTextSplitter(

chunk\_size = 1000,

chunk\_overlap = 200,

)

texts = text\_splitter.create\_documents([state\_of\_the\_union])

print(texts[0])

2. 代码分割

CodeTextSplitter允许你对多种语言的代码进行分割。导入枚举Language并指定语言即可。

我们通过代码主要展示了如何使用 RecursiveCharacterTextSplitter 切割器来处理编程语言的代码。在这个例子中，我们处理的是 JavaScript 代码。我们定义了一段 JavaScript 代码，然后使用 RecursiveCharacterTextSplitter 的 from\_language 类方法创建了一个适用于 JavaScript 语言的切割器。这个方法接受一个 language 参数，它的类型是枚举类型 Language，可以表示多种编程语言。除了 JavaScript，目前还支持 'cpp', 'go', 'java', 'js', 'php', 'proto', 'python', 'rst', 'ruby', 'rust', 'scala', 'swift', 'markdown', 'latex', 'html', 'sol' 等多种编程语言。

from langchain.text\_splitter import (

RecursiveCharacterTextSplitter,

Language,

)

JS\_CODE = """

function helloWorld() {

console.log("Hello, World!");

}

// Call the function

helloWorld();

"""

js\_splitter = RecursiveCharacterTextSplitter.from\_language(

language=Language.JS, chunk\_size=60, chunk\_overlap=0

)

js\_docs = js\_splitter.create\_documents([JS\_CODE])

js\_docs

3. Markdown标题文本分割器

在处理各种聊天或问答应用时，我们常常需要在嵌入和向量存储之前将输入文档切分。这是因为当我们将整个段落或文档进行嵌入时，嵌入过程会考虑到文本内部的整体上下文和句子、短语之间的关系。这样可以得到一个更全面的向量表示，捕捉到文本的广义主题和主旨。

这些场景中，我们切分的目标通常是将具有共同上下文的文本保持在一起。因此，我们可能希望特别尊重文档本身的结构。例如，一个 Markdown 文件是按照标题进行组织的。在特定的标题组内创建块是一种直观的想法。实现这样的切分想法，我们可以使用 MarkdownHeaderTextSplitter。这个工具可以根据指定的一组标题来分割一个 Markdown 文件。例如，如果我们希望分割以下 Markdown：

# Markdown的一级标题

## Markdown的二级标题

Markdown的段落。Markdown的段落Markdown的段落Markdown的段落。

Markdown的段落。

Markdown的段落。

Markdown的段落。

Markdown的段落。

我们可以设置切分的标题为：

headers\_to\_split\_on = [

("#", "Header 1"),

("##", "Header 2"),

]

接着，我们可以使用 MarkdownHeaderTextSplitter 来进行切分，实例化分割器后，我们调用实例的split\_text 方法，这个方法接收了Markdown 文件的内容，返回Document格式的数据，一旦转换为这种数据格式，就可以使用其他的切割器的 split\_documents 方法进行再切割：

# MD splits

markdown\_splitter = MarkdownHeaderTextSplitter(headers\_to\_split\_on=headers\_to\_split\_on)

md\_header\_splits = markdown\_splitter.split\_text(markdown\_document)

这样，我们就得到了按标题切分的文档。然而，这仍然可能不够。如果某个标题下的内容非常长，我们可能需要进一步切分。这时，我们可以使用 RecursiveCharacterTextSplitter 来进行字符级别的切分：

# Char-level splits

from langchain.text\_splitter import RecursiveCharacterTextSplitter

chunk\_size = 250

chunk\_overlap = 30

text\_splitter = RecursiveCharacterTextSplitter(

chunk\_size=chunk\_size, chunk\_overlap=chunk\_overlap

)

# Split

splits = text\_splitter.split\_documents(md\_header\_splits)

这样，我们就可以得到更小的、便于处理的文本块了。

4. 递归按字符分割

这是通用文本推荐的文本分割器。它由一系列字符参数化。直到块足够小，它会尝试按顺序分割它们。默认列表是["\n\n", "\n", " ", ""] 。这样做的效果是尽可能地将所有段落（然后是句子，然后是单词）保持在一起，因为这些通常看起来是最强的语义相关的文本部分。

5. 按标记（Token）分割

在处理自然语言处理任务时，我们经常需要将长文本分割成小块以便于模型处理。这就需要使用标记切割器。标记切割器的主要任务是按照一定的规则将文本切割成小块，这些小块的长度通常由模型的输入限制决定。以下就是一些常用的标记切割器：

Tiktoken 标记切割器：tiktoken 是由 OpenAI 创建的一种快速的字节对编码（BPE）标记器。我们可以使用它来估计使用的标记数量。对于 OpenAI 的模型来说，它的准确度可能会更高。tiktoken 的文本切割方式是按照传入的字符进行切割，块大小的计算方式是通过 tiktoken 标记器进行计算。这个标记切割器使用的方式更复杂一点，我们通过代码展示它的使用方式。先要安装tiktoken python包。然后导入 CharacterTextSplitter 类，使用类方法 from\_tiktoken\_encoder 实例化这个类。跟SpacyTextSplitter 等其他内置的切割器不一样的是，Tiktoken 标记切割器是由 CharacterTextSplitter 类的类方法实例而来。

#!pip install tiktoken

# This is a long document we can split up.

with open("../../../state\_of\_the\_union.txt") as f:

state\_of\_the\_union = f.read()

from langchain.text\_splitter import CharacterTextSplitter

text\_splitter = CharacterTextSplitter.from\_tiktoken\_encoder(

chunk\_size=100, chunk\_overlap=0

)

texts = text\_splitter.split\_text(state\_of\_the\_union)

SpacyTextSplitter 标记切割器：spaCy 是一种用于高级自然语言处理的开源软件库，用 Python 和 Cython 编写。作为 NLTK 的另一种替代方案，我们可以使用 spaCy 的标记器。spaCy 的文本切割方式是通过 spaCy 标记器进行切割，块大小的计算方式是通过字符数量进行计算。

SentenceTransformersTokenTextSplitter标记切割器：是专门用于处理句子转换模型的专用文本切割器。默认行为是将文本切割成适合所要使用的句子转换器模型的标记窗口的块。

NLTKTextSplitter 标记切割器：Natural Language Toolkit（NLTK）是一套用于符号和统计自然语言处理的库和程序，用 Python 编程语言编写。与仅仅在 "\n\n" 处切割不同，我们可以使用 NLTK 基于 NLTK 标记器进行切割。NLTK 的文本切割方式是通过 NLTK 标记器进行切割，块大小的计算方式是通过字符数量进行计算。

Hugging Face 标记切割器：Hugging Face 提供了许多标记器。我们使用 Hugging Face 的标记器 GPT2TokenizerFast 来计算文本长度（以标记为单位）。这个文本切割方式是按照传入的字符进行切割，块大小的计算方式是通过 Hugging Face 标记器计算出的标记数量进行计算。这个标记切割器使用的方式更复杂一点，我们通过代码展示它的使用方式。先要安装 transformers 库，导入 GPT2TokenizerFast 包，调用这个类的 from\_pretrained 方法，实例化一个 tokenizer 。然后 CharacterTextSplitter 类的tokenizer 属性为这个标志器实例，使用类方法 from\_huggingface\_tokenizer 实例化这个类。

from transformers import GPT2TokenizerFast

tokenizer = GPT2TokenizerFast.from\_pretrained("gpt2")

# This is a long document we can split up.

with open("../../../state\_of\_the\_union.txt") as f:

state\_of\_the\_union = f.read()

from langchain.text\_splitter import CharacterTextSplitter

text\_splitter = CharacterTextSplitter.from\_huggingface\_tokenizer(

tokenizer, chunk\_size=100, chunk\_overlap=0

)

texts = text\_splitter.split\_text(state\_of\_the\_union)

这些标记切割器是框架内置的切割器。它们的使用方式我们通过选择哪种标记切割器主要取决于我们的任务需求和所使用的模型。在选择标记切割器时，我们需要考虑到模型的输入限制、我们希望保留的上下文信息以及我们希望如何切割文本等因素。

## 4.4 V向量存储库

在嵌入模型包装器（LEDVR工作流的 E）这一部分，我们了解到嵌入模型包装器提供了两个主要的方法，分别是 embed\_documents 和 embed\_query 。前者接受一组文本作为输入并返回他们的嵌入向量，而后者接受一个文本并返回其嵌入向量。我们也展示了如何利用这个包装器将查询语句转换成浮点数列表，也就是向量。但是，当我们得到这个向量后，我们应该如何使用它呢？

这就是向量存储库（LEDVR工作流的 V）要解决的问题。

向量存储库可以看作是一个大的包装器，它负责处理数据连接模块中LEDVR工作流的LED三个环节的输出结果。对于开发者来说，使用向量存储库可以极大地简化工作。开发者不需要关心如何与各个模型平台进行交互，也不需要将数据处理成其他形式。比如，我们LEDVR工作流一直都在处理文档对象格式的数据，开发者只需要专注于这个格式，然后将数据交给向量存储库就可以了。向量存储库会在底层处理数据格式的转换，解析模型包装器的返回数据等各种复杂的工作。

举个简单例子，如果我们单独将查询语句转向量，做法是实例化嵌入模型包装器后，调用embed\_documents 方法，可是这个方法接收的是字符串列表。如果我们不使用向量存储库这个包装器，我们要先把文本分割器处理过的文档数据格式的结果，转换为字符串列表，最终得到嵌入模型包装器的字符串结果后，我们又要考虑如何转换为向量存储库的格式，否则使用不了向量存储库的查询功能。

相比之下，如果我们使用向量存储库这个包装器，只需要将原始的文档对象格式的数据交给向量存储库，向量存储库会负责将文档转换成字符串，然后将字符串转换成向量，最后将向量存储起来。当我们需要查询时，我们只需要提供查询语句，向量存储库会自动将查询语句转换成向量，然后进行查询。这样一来，我们就可以把所有复杂的数据处理工作都交给向量存储库。简单粗暴一点便是我们可以忘掉embed\_documents 方法了。因为向量存储库帮我们做好了，这也正是 Langchain 的设计理念，让Langchain 为开发者做更多。

### 4.4.1 向量存储库的使用

在数据连接模块中，数据以Document对象和向量形式在各个包装器中流通。向量形式的数据由向量存储库管理，我们为什么要使用向量这种数据格式？这是因为传统的数据库是结构化的，而如今很多数据都是非结构化数据。

非结构化数据是指在日常操作中并不遵循固定格式或者不容易被数据库系统识别的数据。例如，电子邮件、博客、社交媒体帖子、音频和视频等等。这些数据无法通过预定义的数据模型进行分类，或者不适合通过常规的关系数据库进行处理。

我们对非结构化数据的需求主要是存储和搜索。存储是为了保留这些数据以供日后分析和使用，而搜索则是为了从海量数据中找到我们所需要的信息。例如，当我们在互联网上搜索关键词时，搜索引擎会从非结构化的网页数据中找到与关键词相关的信息。而在大数据和人工智能领域，非结构化数据也被广泛用于情感分析、文本分类、语义理解等任务。

处理非结构化数据的一种常见方法是将其嵌入并存储生成的嵌入向量，然后在查询时将非结构化查询嵌入，并检索与嵌入查询“最相似”的嵌入向量。这种方法将复杂的非结构化数据转化为了结构化的向量，大大简化了数据的处理和分析。向量存储库 (VectorStore) 就是实现这个功能的工具，它负责存储嵌入的数据并执行向量搜索。

向量存储库的工作流程可以通过以下的代码示例来说明。首先，我们需要安装faiss-cpu这个Python包，这是一个用于高效相似性搜索和聚类的库。

pip install faiss-cpu

向量存储库的工作原理是由VectorStore类实例化而来，这个类主要提供了一些实例化的类方法。通过理解这些类方法的功能，我们能够更好地使用向量存储库，并能够根据自己的需求进行定制。其中，from\_documents是一个常用的方法，它接受一个文档列表和一个嵌入模型包装器作为输入，返回一个初始化后的向量存储库。这个方法首先从每个文档中提取文本和元数据，然后调用from\_texts方法，将文本、嵌入模型以及元数据作为输入，生成向量存储库。这个方法的异步版本，afrom\_documents，提供了同样的功能，但是它以异步的方式运行。除此之外，from\_texts是一个更基础的方法，它直接接受一组文本和一个嵌入模型包装器，以及可选的元数据，生成向量存储库。这个方法的异步版本，afrom\_texts，也提供了相同的功能。最后，as\_retriever方法返回一个VectorStoreRetriever对象，这个对象包装了向量存储库，并提供了一些用于查询的方法。例如，它可以执行相似性搜索，也可以执行最大边缘相关性搜索。

所以我们实例化一个FAISS向量存储库（Langchain 封装了几十个向量数据库平台的服务，这里我们选的是Facebook AI Similarity Search (FAISS) 库，你可以选择其他库），并将文档块 documents和OpenAI的嵌入模型包装器 OpenAIEmbeddings 一起传递给这个向量存储库，使用from\_documents 方法实例化向量存储库 db 。此时，向量存储库 db 会自动调用嵌入模型将每个文档块转换成一个向量，并将这些向量存储起来。我们已经完成了向量存储库的准备工作，接下来我们就可以通过这个向量存储库来对文档进行高效的相似性搜索了。

from langchain.document\_loaders import TextLoader

from langchain.embeddings.openai import OpenAIEmbeddings

from langchain.text\_splitter import CharacterTextSplitter

from langchain.vectorstores import FAISS

# LEDVR： raw\_documents是L, OpenAIEmbeddings()是 E, documents是D, db 是 V

raw\_documents = TextLoader('../../../state\_of\_the\_union.txt').load()

text\_splitter = CharacterTextSplitter(chunk\_size=1000, chunk\_overlap=0)

documents = text\_splitter.split\_documents(raw\_documents)

db = FAISS.from\_documents(documents, OpenAIEmbeddings())

我们使用similarity\_search方法来嵌入一个查询， “ "What did the president say about Ketanji Brown Jackson" ”。

query = "What did the president say about Ketanji Brown Jackson"

docs = db.similarity\_search(query)

print(docs[0].page\_content)

通过比较查询向量与存储库中向量的相似度，我们就可以找到与查询最相关的文本。

Tonight. I call on the Senate to: Pass the Freedom to Vote Act. Pass the John Lewis Voting Rights Act. And while you’re at it, pass the Disclose Act so Americans can know who is funding our elections.

Tonight, I’d like to honor someone who has dedicated his life to serve this country: Justice Stephen Breyer—an Army veteran, Constitutional scholar, and retiring Justice of the United States Supreme Court. Justice Breyer, thank you for your service.

One of the most serious constitutional responsibilities a President has is nominating someone to serve on the United States Supreme Court.

And I did that 4 days ago, when I nominated Circuit Court of Appeals Judge Ketanji Brown Jackson. One of our nation’s top legal minds, who will continue Justice Breyer’s legacy of excellence.

向量存储库是处理非结构化数据的一个强大工具。它可以将复杂的非结构化数据转化为结构化的向量，使得数据的存储和搜索变得更为简单和高效。通过嵌入和向量相似度的计算，帮助我们在海量的非结构化数据中快速找到我们需要的信息。

### 4.4.2 向量存储库的搜索方法

我们刚刚通过实例代码了解了如何使用向量存储库。现在我们还需要知道，向量存储库主要提供以下几种搜索方法：

1. similarity\_search(query: str, k: int = 4) -> List[Document]: 这个方法接受一个字符串查询和一个整数k作为参数，返回与查询最相似的k个文档的列表。查询是要搜索的字符串，k是要返回的文档数量，默认为4。

2. similarity\_search\_by\_vector(embedding: List[float], k: int = 4) -> List[Document]: 这个方法接受一个嵌入向量和一个整数k作为参数，返回与嵌入向量最相似的k个文档的列表。嵌入向量是由文本嵌入模型生成的查询的向量表示。

3. max\_marginal\_relevance\_search(query: str, k: int = 4, fetch\_k: int = 20, lambda\_mult: float = 0.5) -> List[Document]: 这个方法使用最大边际相关性算法返回选择的文档。最大边际相关性算法优化了查询的相似性和所选择文档之间的多样性。查询是要搜索的字符串，k是要返回的文档数量，默认为4。fetch\_k是要传递给MMR算法的文档数量。lambda\_mult是一个在0和1之间的数字，决定了结果之间的多样性程度，0对应最大的多样性，1对应最小的多样性，默认为0.5。

4. max\_marginal\_relevance\_search\_by\_vector(embedding: List[float], k: int = 4, fetch\_k: int = 20, lambda\_mult: float = 0.5) -> List[Document]: 这个方法与上面的max\_marginal\_relevance\_search方法类似，但是接受的是嵌入向量而不是查询字符串。

以上所有的方法都有对应的异步版本，方法名前加上'a'，比如asimilarity\_search、asimilarity\_search\_by\_vector等。这些异步方法可以在协程中使用，让你的程序在等待结果的同时可以执行其他任务，提高了程序的效率。

我们可以观察到这些方法，返回的结果都是 List[Document] 的数据格式。这也是我们在数据连接模块的最主要数据格式。无论是加载器加载的结果，还是实例化向量存储库的时候，我们都使用的是文档数据类型。而浮点列表的向量数据类型，通常都在包装器的内部流通使用，我们甚至可以不知道到底转成了什么浮点数字，对于我们大部分人来说，浮点数列表只是一堆数字。

### 4.5 R检索器

在我们的LEDVR数据处理流程中，有一个环节可能让你感到疑惑，那就是最后的“检索器”（Retriever）。你可能会问，既然我们已经通过LEDV流程把外部数据转化为了向量形式并保存在向量库中，而且我们还可以对这个库进行查询并获取相关文档，为什么还需要一个检索器呢？实际上，这正是我们想要讨论的重点。此外，我们之前还强调了向量存储库实例的 as\_retriever 方法，这个方法返回一个 VectorStoreRetriever 对象。这个对象甚至还“包装”了向量存储库。为什么要一个检索器？为什么一定是 LEDVR ？

在众多的向量库数据平台中，比如“Chroma”,“FAISS”,“Pinecone”, “Zilliz”等等，我们可以看到向量库的种类繁多。直接和这些数据库如“Qdrant”进行交互，可能需要我们具备深入的数据库操作知识，如了解查询语法，管理数据库连接，处理错误和异常等。这样的操作可能在使用上较为复杂和不便。如果我们有一种方法，能将各种向量库平台统一到一个接口上，那将会非常方便。这就是LangChain做的事情，它封装了 VectorStoreRetriever 类，为我们提供了一个统一的查询接口，无论向量存储库的底层是什么，无论是什么向量存储库，我们将向量存储库的实例调用as\_retriever 方法得到了一个基于向量存储库的检索器，即VectorStoreRetriever 类的实例。所以创建一个检索器很容易，如果你已经创建好了向量存储库，直接调用as\_retriever 方法，就得到了一个检索器实例，这个检索器可以查询，更重要的是可以被传递到 Langchain 框架的其他模块中，让检索器在整个框架中自由流通。

当我们从原理上来理解检索器后，我们可以简化记忆，从向量存储库到检索器，中间只需一个as\_retriever 方法。向量存储库调用它，便创建了一个检索器。这就组合了LEDVR 工作流，用 as\_retriever 方法粘合了V 和 R，整个LEDVR 工作流到此结束。

那么，检索器是什么呢？可以把检索器看作是一个向量存储库的包装器，包装了一套统一的接口，无论底层的向量存储库是什么，都可以使用同样的方式进行查询，而这个包装器的核心就是包装了向量存储库的实例。这使得我们可以轻松地切换不同的向量存储库，而无需修改查询代码。

还有问题没解决的是谁使用检索器？检索器作用是什么？。其实，在LangChain框架中，所有的基础模块都是为了链（Chain）模块的基建工作而设计的。这就像一座大厦的建设，每一块砖，每一捆钢筋，都在为整个建筑的最终成型做准备。从“谁使用了检索器”这个视角来看，我们可以更清晰地理解这个过程。

在处理用户查询时，我们首先需要通过检索器获取相关的文档，这些文档能够帮助我们回答用户的问题。然后，我们需要将这些文档提交给模型平台，利用大型语言模型的能力生成回答。这就好比我们在一个外部数据的大型知识库中寻找答案，然后通过强大的语言理解和生成能力，以一种人类易于理解的方式表达这个答案。

这时，你可能会问，我们是不是需要调用模型平台的能力了？答案是肯定的。然而，LangChain的设计目标是让这一切变得简单和直观。在内置的链组件中，LangChain将所有这些步骤都整合在一起。只需要指定模型包装器和检索器，链组件就能完成所有的功能。

比如，当我们在LangChain中配置好模型包装器和检索器之后，链组件会首先利用检索器找到相关的文档，然后将这些文档送入模型包装器，最后返回模型生成的答案。在这个过程中，我们只需要关心如何配置模型包装器和检索器，而不需要担心这些基础模块之间的交互细节，因为所有这些都由链组件自动完成了。

### 4.5.1 检索器的使用

我们用代码示例展示如何在LangChain中配置模型包装器和检索器，然后使用链组件来实现信息检索和问题回答的全过程。

首先，我们使用 TextLoader 加载文本文件"state\_of\_the\_union.txt"，该文件包含了一系列的文档：

loader = TextLoader("../../state\_of\_the\_union.txt")

documents = loader.load()

接着，我们使用 CharacterTextSplitter 对文档进行分割，将每篇文档分割成一系列的文本块。每个文本块的大小为1000个字符，相邻的文本块之间有重叠部分：

text\_splitter = CharacterTextSplitter(chunk\_size=1000, chunk\_overlap=0)

texts = text\_splitter.split\_documents(documents)

然后，我们使用 OpenAIEmbeddings 为文本块生成嵌入向量，并使用 Chroma 将这些文本块和对应的嵌入向量存储起来，创建了一个向量存储库的实例 docsearch：

embeddings = OpenAIEmbeddings()

docsearch = Chroma.from\_documents(texts, embeddings)

最后，我们使用 RetrievalQA 创建了一个检索式问答系统。这个系统使用了我们之前创建的 docsearch 作为检索器，和OpenAI大型语言模型作为回答生成器。这个系统可以根据用户的问题，找到相关的文本块，然后生成回答：

qa = RetrievalQA.from\_chain\_type(llm=OpenAI(),

chain\_type="stuff", retriever=docsearch.as\_retriever())

在这个过程中，我们只需要关注如何配置模型包装器和检索器，而不需要关心这些组件之间的交互细节，因为链组件已经帮我们自动处理了所有的事情。

### 4.5.2 检索器的类型

在实际的信息检索过程中，我们可能会遇到各种各样的问题和需求，比如需要精确匹配关键词、需要理解语义、需要根据时间排序、需要从网络上获取最新的数据等等。每种类型的检索器都是为了解决这些特定需求而设计的。分类可以让我们更清楚地理解每种检索器的功能和用途，从而能够更有效地选择和使用。

具体来说，以下是每种类型的检索器所解决的问题和适用的场景：自查询检索器：这种检索器适用于需要通过自然语言查询来检索具有一定结构或元数据的文档的场景。比如，在一个电子商务网站中，用户可能会输入“ "最新的iPhone手机" ”，检索器可以将这个查询转化为一个结构化的查询，比如{"category": "手机", "brand": "iPhone", "order": "newest"}，从而能够更精确地获取到用户想要的结果。时间加权向量存储检索器：这种检索器适用于信息的新旧程度对查询结果影响较大的场景。比如，在新闻检索中，用户通常更关心最新的新闻，因此检索器需要根据新闻的发布时间来对结果进行排序。向量存储支持的检索器：这种检索器适用于需要基于语义相似度进行检索的场景。比如，在问答系统中，用户的问题可能会有很多种表达方式，只有理解了问题的语义，才能找到正确的答案。网络研究检索器：这种检索器适用于需要从网络上获取最新数据的场景。比如，用户可能想要获取关于一个热点事件的最新信息，此时检索器可以直接从网络上进行搜索，获取到最新的数据。

目前主流的Retriever检索器主要有以下几种类型：向量存储库支持的检索器、自查询检索器、MultiQueryRetriever、Contextual Compression Retriever和Ensemble Retriever。

首先，Vector store-backed retriever 是最常见的一种检索器，我们的案例多采用这种方式实现检索器。这是一种使用向量存储来检索文档的检索器。它是一个围绕向量存储库包装，使其符合Retriever接口。它使用向量存储实现的搜索方法，如相似性搜索和最大边缘相关性（MMR），来查询向量存储中的文本。一旦你构建了一个向量存储，就很容易构建一个检索器。如果你已经创建好了向量存储库，直接调用as\_retriever 方法，就得到了一个检索器实例。

让我们来看看MultiQueryRetriever。在基于距离的向量数据库检索中，我们将查询语句嵌入到高维空间中，并通过计算"距离"来寻找相似的嵌入文档。但是，检索结果可能因查询语句的微小变化或嵌入并未完全捕获数据语义而有所不同。人们有时通过提示工程或调整来手动解决这些问题，但这是一项繁琐的工作。MultiQueryRetriever（多查询检索器）通过使用大型语言模型（LLM）从不同角度生成多个查询，自动化了提示调整的过程。对于每个查询，它都会检索一组相关文档，并在所有查询中取唯一的并集，从而获得一组可能相关的文档。通过对同一问题的多角度生成，MultiQueryRetriever可能能够克服基于距离的检索的某些限制，并获得更丰富的结果。

然后，我们来讲一下Contextual Compression Retriever。检索的一个挑战在于，当你将数据输入系统时，你通常不知道你的文档存储系统将面临什么样的特定查询。这意味着与查询最相关的信息可能被埋藏在一篇包含大量无关文本的文档中。将整篇文档通过你的应用程序可能会导致LLM调用更昂贵，并导致回应效果更差。为了解决这个问题，提出了Contextual Compression Retriever。这个方法很简单：在直接返回检索到的文档时，可以使用给定查询的上下文来压缩它们，这样只有相关的信息会被返回。“压缩”在这里指的是压缩单个文档的内容，并完全过滤掉一些文档。要使用Contextual Compression Retriever，你需要一个基本的检索器和一个文档压缩器。Contextual Compression Retriever将查询传给基本的检索器，获取初始文档，然后将它们传递给文档压缩器。文档压缩器接收一个文档列表，通过减少文档的内容或整个删除文档来缩短它。

再来看看Ensemble Retriever。EnsembleRetriever接受一个检索器列表作为输入，汇集他们的get\_relevant\_documents()方法的结果，并根据Reciprocal Rank Fusion算法对结果进行重新排序。通过利用不同算法的优势，EnsembleRetriever可以获得比任何单个算法更好的性能。最常见的模式是将稀疏检索器（如BM25）与密集检索器（如Embedding similarity）结合在一起，因为它们的优势是互补的。这也被称为"混合搜索"。稀疏检索器擅长根据关键词找到相关文档，而密集检索器擅长根据语义相似性找到相关文档。

我们再来看看其他一些类型的检索器：Self-querying retriever、Time-weighted vector store retriever、Vector store-backed retriever和WebResearchRetriever。

Self-querying retriever正如其名称所示，具有自我查询的能力。具体而言，对于任何自然语言查询，检索器使用一个查询构造的大型语言模型链来编写一个结构化的查询，然后将这个结构化的查询应用到其底层的向量存储。这使得检索器不仅可以使用用户输入的查询来与存储文档的内容进行语义相似度比较，还可以从用户查询中提取出存储文档元数据的过滤器，并执行这些过滤器。

Time-weighted vector store retriever使用语义相似度和时间衰减的组合。他们的得分算法为：semantic\_similarity + (1.0 - decay\_rate) ^ hours\_passed 。值得注意的是，`hours\_passed`指的是自检索器中的对象上次被访问以来经过的小时数，而不是它被创建以来的小时数。

最后，我们来看看WebResearchRetriever（网络研究检索器）。给定一个查询，这个检索器会：构建一组相关的Google搜索；对每一个进行搜索；加载所有结果的URL；然后将查询与合并的页面内容进行嵌入和相似性搜索。

在上述检索器中，你可以根据实际情况选择最适合你需求的一种，或者根据需要将它们结合使用，以达到更好的检索效果。

### 4.6 本章小结

在本章节中，我们深入探讨了LangChain的核心组成部分：LEDVR（加载器，嵌入模型包装器，文档转换器，向量存储库和检索器）工作流。这五个模块共同构成了一个流程，处理外部数据加入到LangChain框架中，让我们看到了数据是什么形式的，以什么方式流动的，它们又流向了哪里整个过程。

首先，加载器（Loader）是我们从各种来源（如本地文件、数据库、网络等）获取数据的入口。加载器将原始数据加载到系统中，为后续的处理步骤做好准备。

然后，我们有嵌入模型包装器（Text Embedding Model）。它将原始的文本数据转化为高维的向量表示，这使得我们可以利用向量空间中的距离来度量文本之间的语义相似性。

接着，文档转换器（Document Transformers）对嵌入向量进行处理，以适应特定的检索需求。例如，我们可能需要将长文档分割成多个小块。

在文档经过嵌入模型包装器和文档转换器的处理后，我们将得到的向量表示存储在向量存储库（VectorStore）中。向量存储库内置了很多搜索方法，这些搜索方法适用于不同的应用程序适用场景，向量存储库还内置了各大向量数据库平台的包装器，这些包装器可以快速检索出与给定查询最相关的文档。

最后，我们有检索器（Retriever）。检索器利用向量存储库提供的检索能力，对用户的查询进行响应。

总结来说，LEDVR工作流是一种强大的信息检索和处理工具，我们在本章节中详细介绍了每个模块的定义和使用方法、以及分类方法，希望能帮助读者更好地理解和使用LangChain框架中的数据连接模块的功能。