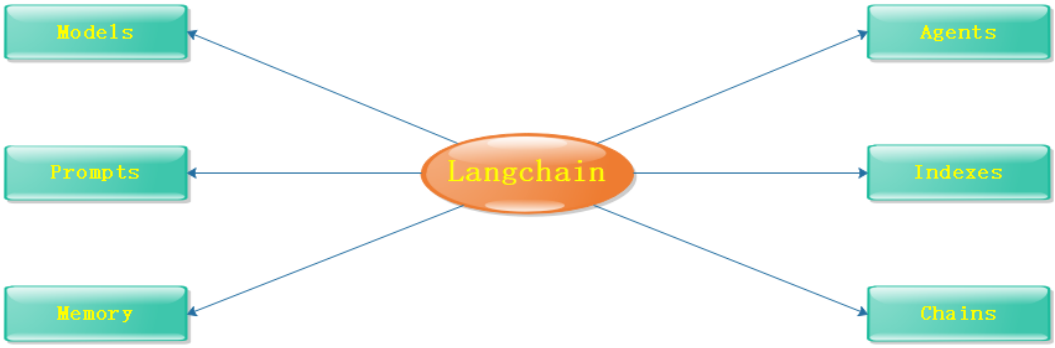
Langchain

- LangChain官网

1. 了解Langchain

- **LangChain** 是一个用于开发由大型语言模型（LLM） 提供支持的应用程序的框架，可简化创建由大型语言模型 (LLM) 和聊天模型提供支持的应用程序的过程
- **LangChain** 这个框架为用户或者开发者带来了两大核心优势或好处
  - 组件：LangChain 设计了一些模块化的组件，它把复杂的任务拆分成更小、更易使用的部分。
  - 链：“链”就像是一个完整的解决方案，它把各个工具（组件）串联起来，形成一套流程，使得即使是初学者也能快速上手解决具体的问题。
- **LangChain** 库本身由几个不同的包组成。
  - `langchain`：构成应用程序认知架构的链、代理和检索策略。
  - `langchain-core`：langchain的核心包。
  - `langchain-community`：第三方集成。

2. Langchain的核心组件



- Models -> 各种类型的模型和模型集成
- Prompts -> prompt模板封装
- Indexes -> 索引，用来结构化文档，以便和模型交互
- Chains -> 链，一系列对各种组件的调用
- Agents -> 代理，决定模型采取哪些行动，执行并且观察流程，直到完成为止
- Memory -> 记忆，用来保存和模型交互时的上下文状态

3. Langchain的组件使用

3.1 Models_module

- LLMs (大语言模型)
- Chat Models (聊天模型)
- Embeddings Models(嵌入模型)
- OutputParser(模型的解析输出)

3.1.1 LLMs

- LLM是将字符串作为输入并返回字符串作为输出的语言模型。

> 示例

```
from langchain_openai import OpenAI
# 创建一个 OpenAI 实例，指定使用的模型为 "gpt-3.5-turbo-instruct"
llm = OpenAI(
    model="gpt-3.5-turbo-instruct",
)
# 定义输入文本
input_text = "请默写鹅鹅鹅"
# 调用 llm 对象的 invoke 方法，传入输入文本并打印响应
print(llm.invoke(input_text))
```

3.1.2 Chat Models

- 聊天模型是使用一系列消息作为输入并返回消息作为输出的语言模型。

> 示例

```
from langchain_openai import ChatOpenAI
# 创建一个 ChatOpenAI 实例，指定使用的模型为 "gpt-4o-mini"
chat = ChatOpenAI(
    model="gpt-4o-mini"    # 默认情况下使用 gpt-3.5-turbo 模型
)
# 输出当前实例的模型名称
print(chat.model_name)
# 打印模型对该消息的响应内容
print(chat.invoke("请默写鹅鹅鹅").content)
```

- 消息类型
  - SystemMessage => 设置LLM模型的行为方式和目标。你可以在这里给出统一的指示
  - AIMessage => 用来保存LLM的响应，以便在下次请求时把这些信息传回给LLM
  - HumanMessage => 发送给LLMs的提示信息
  - ChatMessage => ChatMessage可以接收任意形式的值， 但是在大多数时间，我们应该使用上面的三种类型
    - `ChatMessage` 是一种通用的消息类型，它可以用来表示来自任何角色的消息。
    - 在 `langchain` 中，`ChatMessage` 需要指定一个 `role`，如 `"assistant"` 或 `"user"`。

> 示例

```
# 假设已经正确导入了必要的模块
from langchain_openai import ChatOpenAI
from langchain.schema import (
    AIMessage,
    HumanMessage,
    SystemMessage,
    ChatMessage
)
# 创建一个 ChatOpenAI 实例，指定使用的模型为 "gpt-4o-mini"
llm = ChatOpenAI(model="gpt-4o-mini")
# 定义消息列表
messages = [
    SystemMessage(content="描述一个顾客进入饭店并点餐的场景。"),  # 系统消息
    HumanMessage(content="我进入饭店吃饭。"),  # 用户消息
    ChatMessage(role="assistant", content="欢迎光临，请问您几位？"),  # 迎宾
    HumanMessage(content="就我一位，我点菜。"),  # 用户消息
    ChatMessage(role="assistant", content="好的，这边请"),  # 迎宾回应
    HumanMessage(content="我想点一份牛肉炒饭，还有一份糖醋排骨。另外，我注意到糖醋排骨的价格比其他菜品要高一些，为什么呢？"),  # 用户询问
    AIMessage(content="?"),  # AI 生成的服务员回应
]
```

```
# 调用 llm 对象的 invoke 方法，传入消息列表并打印响应内容
print(llm.invoke(messages).content)
```

3.1.3 Embedding Models

- 嵌入模型创建一段文本的矢量表示
- IBM watsonx.ai | 🦜🔗 LangChain
- `embed_query`：适用于单个文档
- `embed_documents`：适用于多个文档

> 示例

```
from langchain_openai import OpenAIEmbeddings
# 创建一个 OpenAIEmbeddings 实例，指定使用的模型为 "text-embedding-3-large"
embed = OpenAIEmbeddings(
    model="text-embedding-3-large",
)
# 使用 embed 对象嵌入查询文本并保存结果
result1 = embed.embed_query("我是A文档")
print(result1)  # 打印嵌入结果
print(len(result1))  # 打印嵌入结果的长度

# 使用 embed 对象嵌入多个文档并保存结果
result2 = embed.embed_documents(["我是A文档", "我是B文档"])
print(result2)  # 打印嵌入的文档结果
print(len(result2))  # 打印嵌入文档的数量
```

```
print(len(result2[0]))  # 打印第一个文档嵌入结果的长度
print(len(result2[1]))  # 打印第二个文档嵌入结果的长度
```

3.1.4 OutputParser

- 输出解析器 => 当您想要返回以逗号分隔的项目列表时，可以使用此输出解析器

> 示例

```
# 导入CommaSeparatedListOutputParser包，这个包是用来解析以逗号分隔的字符串，并将其转换为列表
from langchain.output_parsers import CommaSeparatedListOutputParser

# 使用CommaSeparatedListOutputParser实例的parse方法来解析定义好的字符串
print(CommaSeparatedListOutputParser().parse('foo,bar,baz'))
```

- 获取解析格式

> 示例

```
from langchain.output_parsers import CommaSeparatedListOutputParser
# 获取格式说明并打印
print(CommaSeparatedListOutputParser().get_format_instructions())
```

> 示例

```
from langchain.output_parsers import
CommaSeparatedListOutputParser
# 创建 CommaSeparatedListOutputParser 实例
output_parser =
CommaSeparatedListOutputParser()
# 获取格式说明并打印
output_parser.get_format_instructions()
print(output_parser.get_format_instructions
())
# 定义一个包含逗号分隔值的回复
reply ='foo,bar,baz'
# 解析回复并打印解析结果
print("**********************")
print(output_parser.parse(reply))  # 打印解
析后的结果
```

- 指定格式输出

> 示例

```
from langchain.output_parsers import
CommaSeparatedListOutputParser
# 解析输入
output_parser =
CommaSeparatedListOutputParser().parse('foo
,bar,baz')
# 自定义输出格式
formatted_result = '-'.join(output_parser)
print(formatted_result)  # 输出: foo-bar-baz
```

- 解析日期时间格式。

> 示例

```
from langchain_openai import ChatOpenAI
from langchain.output_parsers import
DatetimeOutputParser
# 创建一个 DatetimeOutputParser 实例
output_parser = DatetimeOutputParser()
# 创建一个 ChatOpenAI 实例，使用 gpt-4o-mini
模型
llm = ChatOpenAI(
    model_name='gpt-4o-mini'
)
# 构造请求字符串
request = "中华人民共和国是什么时候成立的"
format_instructions =
output_parser.get_format_instructions()
# 构造消息字典
messages = [{"role": "user", "content": f"
{request}\n{format_instructions}"}]
# 使用构造的消息调用模型
result = llm.invoke(messages)
# 打印模型的响应内容
print("模型响应：", result)
# 假设 result 有 content 属性
response_text = result.content
parsed_result =
output_parser.parse(response_text)
print("解析结果：", parsed_result)
```

3.2 Prompts_module

- prompt基本使用
- prompt使用变量
- prompt外部加载
- prompt zero_shot
- prompt few_shot
- 四个prompt包的区别
  - `from langchain_core.prompts import PromptTemplate`
  - `from langchain_core.prompts import ChatPromptTemplate`
  - `from langchain.prompts import PromptTemplate`
  - `from langchain.prompts import ChatPromptTemplate`
    - `langchain_core.prompts` 与 `langchain.prompts`
      - langchain_core.prompts：提供更底层的接口，在特定情况允许更高效的定制化处理，更加稳定
      - langchain.prompts：提供了更多的便利性和预定义的模板，会稍微增加一些开销。

- `PromptTemplate` 与 `ChatPromptTemplate`
  - PromptTemplate：用于一般提示的模板，通常用于传统的问答或文本生成任务。
  - ChatPromptTemplate：专为聊天或对话系统设计的模板，能够处理多角色对话中的上下文，支持消息格式的更复杂结构。通常用于需要维护对话状态和上下文的应用。

3.2.1 prompt基本使用

- 基本目标的使用

> 示例

```
from langchain_openai import ChatOpenAI
from langchain.prompts.chat import
HumanMessagePromptTemplate
from langchain_core.prompts import
ChatPromptTemplate
from langchain.output_parsers import
DatetimeOutputParser

# 创建一个 DatetimeOutputParser 实例
output_parser = DatetimeOutputParser()

# 创建一个 ChatOpenAI 实例，使用 gpt-4o-mini
模型
llm = ChatOpenAI(
    model_name='gpt-4o-mini'
)
```

```
# 创建聊天提示模板，包含用户消息的模板
chat_prompt =
ChatPromptTemplate.from_messages(
    [

 HumanMessagePromptTemplate.from_template(
            "
{request}\n{format_instructions}"),  # 用户
请求和格式说明
    ]
)
# 格式化聊天提示，填充请求和格式说明
model_request = chat_prompt.format_prompt(
    request="中华人民共和国是什么时候成立的",  #
用户请求的内容

 format_instructions=output_parser.get_form
at_instructions()  # 获取输出解析器的格式说明
)
# print(model_request)  # 可以打印
model_request 以查看格式化后的请求
result = llm.invoke(model_request)  # 调用模
型处理请求
print(result.content)  # 打印模型的响应内容
print("------")  # 分隔符
print(output_parser.parse(result.content))
 # 解析模型的响应内容并打印结果
```

3.2.2 prompt使用变量

- 单变量prompt

> 示例

```
from langchain.prompts import
PromptTemplate

# 创建一个包含{name}变量占位符的提示模板
template = PromptTemplate.from_template("给
我讲个关于{name}的笑话")

print(template)  # 打印提示模板对象
print("-------")
print(template.format(name=input("请输入一个
名字")))  # 使用格式化方法替换占位符，打印提示词
```

- 多角色自定义变量的prompt

> 示例

```
from langchain_openai import ChatOpenAI
from langchain.prompts import
ChatPromptTemplate
# 导入系统消息和用户消息模板类
from langchain.prompts.chat import
SystemMessagePromptTemplate,
HumanMessagePromptTemplate

# 创建一个 ChatOpenAI 实例，使用 gpt-4o-mini
模型
llm = ChatOpenAI(
```

```
    model="gpt-4o-mini"
)

# 创建聊天提示模板，包含系统消息和用户消息的模板
template =
ChatPromptTemplate.from_messages(
    [
        # 系统消息模板

 SystemMessagePromptTemplate.from_template(
"你是{product}的客服助手。你的名字叫{name}"),
        # 用户消息模板

 HumanMessagePromptTemplate.from_template("
{query}"),
    ]
)

# 格式化提示消息，填充产品名、助手名和用户查询
prompt = template.format_messages(
        product="AGI课堂",  # 填入产品名
        name="先知",  # 填入助手名
        query="你是谁"  # 用户查询内容
    )

# 打印格式化后的提示消息
print(prompt)
print("---------------")
# 调用模型处理提示消息并打印响应
print(llm.invoke(prompt))
```

```
print("--------------")
print(llm.invoke(prompt).content)
```

- 解析模型的响应内容

> 示例

```
from langchain.prompts.chat import
HumanMessagePromptTemplate
from langchain_core.prompts import
ChatPromptTemplate
from langchain.output_parsers import
CommaSeparatedListOutputParser
from langchain_openai import ChatOpenAI

# 创建一个逗号分隔列表的输出解析器
output_parser =
CommaSeparatedListOutputParser()
# 创建一个 ChatOpenAI 实例

llm = ChatOpenAI(
    model_name='gpt-4o-mini'
)
# 创建一个聊天提示模板，包含人类消息的格式
chat_prompt =
ChatPromptTemplate.from_messages(
    [

 HumanMessagePromptTemplate.from_template(
```

```
            "
{request}\n{format_instructions}")  # 人类消息模板，包含请求和格式说明
    ]
)
# 格式化模型请求，传入具体请求和格式说明
model_request = chat_prompt.format_prompt(
    request="给我5个性格特征",  # 用户请求

 format_instructions=output_parser.get_format_instructions()  # 获取格式说明
)
# 使用 LLM 调用模型并获取结果
result = llm.invoke(model_request)
# 打印模型的响应结果
print(result)
print("-------")
# 解析模型的响应内容并打印解析后的结果
print(output_parser.parse(result.content))
print("-------")
# 自定义输出格式
formatted_result = '-'.join(output_parser.parse(result.content))
print(formatted_result)
```

3.2.3 prompt外部加载

- 加载JSON文件

> JSON内容

```
{
    "_type": "prompt",
    "input_variables": ["adjective",
"content"],
    "template": "Tell me a {adjective} joke
about {content}."
}
```

> 示例

```
# 从文档当中加载prompt
from langchain.prompts import load_prompt
prompt = load_prompt("simple_prompt1.json")
print(prompt.format(adjective="funny",
content="James"))
```

- 中文版的解决方案
  - 转码操作

> JSON内容

```
{
    "_type": "prompt",
    "input_variables": ["讲述者", "听众", "形
容词", "内容", "时间段"],
    "template": "{讲述者} 给 {听众} 讲述了一个
{形容词} 的关于 {内容} 的故事，在 {时间段} 期间。"
}
```

> 示例

```
import codecs
import json
# 定义文件路径
file_path = "simple_prompt2.json"
# 读取文件并确保使用 UTF-8 编码
with codecs.open(file_path, "r",
encoding="utf-8") as f:
    content = f.read()
# 输出转码之后的内容
print(content)
# 解析 JSON 内容
print("---------")
prompt_data = json.loads(content)
# 获取模板字符串
template = prompt_data["template"]
print(template)
print("---------")
# 格式化输出
formatted_prompt = template.format(
    讲述者="张三",
    听众="李四",
    形容词="有趣",
    内容="猫",
    时间段="文艺复兴时期"
)
# 打印格式化后的提示
print(formatted_prompt)
# 输出: "张三 给 李四 讲述了一个 有趣的关于 猫 的
故事，在 文艺复兴时期 期间。"
```

3.2.4 prompt zero_shot

> 示例

```
# 导入必要的类库
from langchain_core.prompts import
PromptTemplate
from langchain_openai import ChatOpenAI

# 创建一个提示模板，其中包含一个输入变量
"sample"，它将被替换为实际值。
template = "请说一下{sample}的概念"

# 使用PromptTemplate类初始化一个提示对象，指定输
入变量名称为"sample"
prompt = PromptTemplate(input_variables=
["sample"], template=template)
# 使用format方法来填充模板中的 "sample" 变量，并
将其设置为 "零样本"
prompt_text = prompt.format(sample="零样本")

# 输出格式化的提示文本
print(prompt_text)  # 输出: "请说一下零样本的概
念"

# 创建一个ChatOpenAI模型实例，并指定使用"gpt-4o-
mini"模型
llm = ChatOpenAI(
    model="gpt-4o-mini"
)
```

```
# 调用模型的invoke方法来处理我们之前创建的提示文本
result = llm.invoke(prompt_text)
# 输出模型返回的内容
print(result.content)  # 输出模型对提示文本的回答
```

3.2.5 prompt few_shot

> 示例

```
from langchain_core.prompts import PromptTemplate, FewShotPromptTemplate
from langchain_openai import ChatOpenAI

# 1.实例化模型
llm = ChatOpenAI(
    model="gpt-4o-mini"
)

# 2.给出部分示例
examples = [
    {"word": "明亮", "antonym": "黑暗"},
    {"word": "新", "antonym": "旧"}
]

# 3.设置example_prompt
example_template = """
单词: {word}
反义词: {antonym}
"""
```

```
# 4.实例化example_prompt
example_prompt =
PromptTemplate(input_variables=["word",
"antonym"],

 template=example_template)

# 5.实例化few-shot-prompt
few_shot_prompt =
FewShotPromptTemplate(examples=examples,

 example_prompt=example_prompt,

 prefix="给出每个单词的反义词",

 suffix="单词:{input}反义词",

 input_variables=["input"])
# 6.指定模型的输入
prompt_text =
few_shot_prompt.format(input="漂亮")
print(prompt_text)
print("---------------")
#7.将prompt_text输入模型
result = llm.invoke(prompt_text)
print(result.content)
```

3.3 Indexes_module

3.3.1 Loaders

- document-loading
- loader txt

```
from langchain_community.document_loaders import TextLoader
# 创建一个TextLoader实例，指定要加载的文本文件路径及编码格式为utf-8
loader = TextLoader("./医疗数据.txt", encoding='utf8')
# 使用loader加载文档内容
doc = loader.load()
# 打印加载的文档内容
print(doc)
print("---------")
# 输出文档内容的长度
print(len(doc))
print("---------")
# 打印文档第一页内容的前10个字符
print(doc[0].page_content[:10])
```

- loader pdf
    - 自动处理文件编码的问题

> 示例

```
from langchain_community.document_loaders import PyPDFLoader
# 创建一个PyPDFLoader实例，指定要加载的PDF文件路径
loader = PyPDFLoader("中国人工智能系列白皮书.pdf")
# 加载并PDF
pages = loader.load()
# 加载并拆分PDF文件内容为多个页面对象
# pages = loader.load_and_split()
print(pages)
print("----------")
# 打印第14页的内容（因为列表索引从0开始，所以pages[13]对应的是第14页）
print(pages[13].page_content)
```

- loader csv

> 示例

```
from langchain_community.document_loaders
import CSVLoader
# 创建一个CSVLoader实例，指定要加载的CSV文件路径
及编码格式为utf-8
loader = CSVLoader("data.csv",
encoding="utf-8")
# 加载CSV文件内容到一个包含多个记录的对象列表中
pages = loader.load()
# pages = loader.load_and_split()
# 打印pages的类型和长度
print(type(pages), len(pages))
# 打印第一个记录的类型
print(type(pages[0]))
# 打印第一个记录的内容
print(pages[0].page_content)
```

3.3.2 Splitters

- create_documents

> 示例

```
from langchain.text_splitter import
CharacterTextSplitter

# 分割器实例化对象
# separator => 分割文本的字符或字符串
# chunk_size => 每个文本块的最大长度
# chunk_overlap => 文本块之间的重叠字符数
text_spliter = CharacterTextSplitter(
```

```
    separator='',

    chunk_size=10,

    chunk_overlap=2,
)

# 对一句话进行分割
result = text_spliter.split_text("今天天气好晴朗,处处好风光啊好风光蝴蝶儿忙啊,蜜蜂也忙,小鸟儿忙着,白云也忙")
print(result)
print("**********")
# 对多个句子也就是文档切分
texts = text_spliter.create_documents(
    [
        "今天天气好晴朗,处处好风光啊好风光蝴蝶儿忙啊,蜜蜂也忙,小鸟儿忙着,白云也忙",
        "分割文本的字符或字符串,每个文本块的最大长度,文本块之间的重叠字符数"
    ]
)
print(texts)
```

> 示例

```
from langchain.text_splitter import RecursiveCharacterTextSplitter # 分割文本
from langchain_community.document_loaders import PyPDFLoader # 加载PDF文档
```

```
# 创建一个PyPDFLoader实例，参数是PDF文件的路径
loader = PyPDFLoader("中国人工智能系列白皮书.pdf")
pages = loader.load()
# 创建一个RecursiveCharacterTextSplitter实例，定义文本分割规则
"""
separators =>
    "\n\n":  表示两个连续的换行符，分割
    "\n":    表示单个换行符，分割。
    " ":     表示单个空格，分割。
    "":      表示如果没有其他更好的分割点，可以在任意位置分割。
"""
text_splitter = RecursiveCharacterTextSplitter(
    separators=["\n\n", "\n", " ", ""],
    # 每个文本块的最大字符数
    chunk_size=200,
    # 文本块之间的重叠字符数
    chunk_overlap=50,
    # 是否在分割后的每个块中添加起始索引，便于后续追踪原文位置
    add_start_index=True,
)
print([pages[13].page_content])
print("---------")
# 使用创建的文本分割器，对特定页的内容进行分割，此处以第13页为例
```

```
paragraphs =
text_splitter.create_documents([pages[13].p
age_content])
# 遍历分割后的所有段落，并打印它们的内容
for para in paragraphs:
    print(para.page_content)
```

- 按Token进行切割(了解)

> 示例

```
from langchain_community.document_loaders
import PyPDFLoader
from langchain.text_splitter import
RecursiveCharacterTextSplitter
# 创建一个PyPDFLoader实例，参数是PDF文件的路径
loader = PyPDFLoader("中国人工智能系列白皮
书.pdf")
pages = loader.load()
# 创建一个RecursiveCharacterTextSplitter实
例，定义文本分割规则，使用tiktoken编码器进行分割
text_splitter =
RecursiveCharacterTextSplitter.from_tiktoke
n_encoder(
    # 每个文本块的最大字符数
    chunk_size=200,
    # 文本块之间的重叠字符数
    chunk_overlap=50,
)
# 打印第13页的内容（实际上是第14页，因为索引从0开
始）
```

```
print(pages[13].page_content)
print("------------")
# 使用创建的文本分割器，对特定页的内容进行分割，此处以第13页为例
texts = text_splitter.split_text(pages[13].page_content)
# 打印分割后的所有段落及其数量
print(texts)
print(len(texts))
```

3.3.3 Embedding

- txt

> 示例

```
from langchain_openai import OpenAIEmbeddings
# 将一个文本字符串转换为其嵌入表示形式
text = 'this is a text'
# 使用embeddings实例将文本转换为嵌入向量
embeddings = OpenAIEmbeddings()
embedding_text = embeddings.embed_query(text)
# 打印嵌入向量及其长度
print(embedding_text)
print(len(embedding_text))
```

- csv

> 示例

```
from langchain_openai import
OpenAIEmbeddings
from langchain_community.document_loaders
import CSVLoader
# 创建一个CSVLoader实例来加载指定路径的CSV文件，
并指定编码为utf-8
loader = CSVLoader("data.csv",
encoding="utf-8")
# 加载
pages = loader.load()
print(pages)
embeddings = OpenAIEmbeddings()
# 存放的是每一个trunk的embeding。
embeded_docs =
embeddings.embed_documents([i.page_content
for i in pages])
print(embeded_docs)
# 表示的是每一个trunk的embeding的维度
print(len(embeded_docs[0]))
```

- demo

> 示例

```
from langchain_community.document_loaders
import TextLoader
from langchain.text_splitter import
CharacterTextSplitter
```

```
from langchain_openai import
OpenAIEmbeddings
from langchain_community.vectorstores
import Chroma

# 创建一个TextLoader实例来加载指定路径的文本文件，
并指定编码为utf-8
loader = TextLoader("./医疗数据.txt",
encoding='utf-8')
# 加载文本数据
pku_str = loader.load()
print(pku_str)
print("----1111111111----")

# 如果加载的数据是一个列表，则将其转换为字符串
if isinstance(pku_str, list):
    # 使用列表推导式从Document对象中提取
page_content属性，并用换行符连接成一个字符串
    pku_str = "\n".join([doc.page_content
for doc in pku_str])
# 打印处理后的文本内容
print(pku_str)
print("----2222222222----")

# 创建一个CharacterTextSplitter实例来分割文本
text_splitter =
CharacterTextSplitter(chunk_size=100,
chunk_overlap=5)
# 使用text_splitter来分割文本
texts = text_splitter.split_text(pku_str)
```

```
# 创建OpenAIEmbeddings实例用于文本嵌入
embedd = OpenAIEmbeddings()

# 使用Chroma将分割后的文本向量化，并使用之前创建的embedd实例
docsearch = Chroma.from_texts(texts, embedd)
# 设置查询条件
query = "咽喉症状：咽干,打喷嚏：频繁打喷嚏,头部症状：头痛,是什么症状,需要吃什么药"
# 使用相似度搜索方法在向量数据库中查找与查询条件最相似的文档
result = docsearch.similarity_search(query)
# 打印搜索结果
print(result)
print("-----33333333333---")
# 打印搜索结果的数量
print(len(result))
```

3.3.4 VectorStore

> 示例

```
from langchain_openai import OpenAIEmbeddings
from langchain_community.document_loaders import CSVLoader
from langchain.text_splitter import CharacterTextSplitter
```

```
from langchain_community.vectorstores
import Chroma
# 加载文档
loader = CSVLoader("data.csv",
encoding="utf-8")
pages = loader.load()
# 加载文档------>文本拆分
text_spliter
=CharacterTextSplitter.from_tiktoken_encode
r(chunk_size=500)
docs = text_spliter.split_documents(pages)
# 文本嵌入
embeddings = OpenAIEmbeddings()
# 向量存储存储
db = Chroma.from_documents(docs,
embeddings,persist_directory='./new_db')
# 将向量化的文档数据持久化保存到指定的目录中
db.persist()
# 从磁盘加载 Database
db_new_connection =
Chroma(persist_directory='./new_db',embeddi
ng_function=embeddings)
# 相似性搜索：创建一个新的Chroma对象，使用之前存储
的向量数据库进行相似性搜索。
new_doc = '嘉柏湾的房子有那一些'
similar_docs=
db_new_connection.similarity_search(new_doc
)
# 遍历获取
for doc in similar_docs:
```

```
    content = doc.page_content
    print(content)
```

3.3.4 Retrievers

- **检索器 (Retrievers)**:
  - 主要功能：负责将查询转换为向量，并利用已有的索引找到最相关的数据。
  - 角色：充当用户和向量数据库之间的中介，确保高效的查询处理。
- **向量数据库**:
  - 主要功能：存储高维向量，并提供高效的相似性搜索能力。
  - 角色：作为底层数据存储，支持检索器的工作。

> 示例

```
# 数据库连接器当中创建一个检索器对象 (retriever)
retriever =
db_new_connection.as_retriever()
# 使用检索器对象的 get_relevant_documents 方法来获取与查询字符串相关的文档列表
sim_docs =
retriever.get_relevant_documents('xxx')
print(sim_docs)
```

> 示例

```

```

```
from langchain_openai import
OpenAIEmbeddings
from langchain_community.document_loaders
import TextLoader
from langchain_community.vectorstores
import FAISS
from langchain.text_splitter import
CharacterTextSplitter
# 加载文档
loader = TextLoader('./医疗数据.txt',
encoding='utf8')
docments = loader.load()
# 切分文档
text_spliter =
CharacterTextSplitter(chunk_size=100,chunk_
overlap=5)
texts =
text_spliter.split_documents(docments)
print(texts)
# 实例化embedding模型
embed = OpenAIEmbeddings()
db = FAISS.from_documents(texts, embed)
# 文档检索器
retriever = db.as_retriever(search_kwargs=
{"k": 1})
result =
retriever.get_relevant_documents("过敏性鼻炎
用什么药物好？")
# 假设 result 是你的文档列表
print("结果")
```

```
for doc in result:
    content = doc.page_content  # 获取文档内
容
    print(content)  # 打印文档内容
```

3.4 Memory_module

3.4.1 Chat Message

- 基本记忆管理

> 示例

```
from langchain.memory import
ChatMessageHistory

# 1.实例对象
history = ChatMessageHistory()
# 2. 添加历史信息
history.add_user_message("在吗")
history.add_ai_message("有什么事吗？")

print(history.messages)
```

- 记忆存储

> 示例

```
from langchain.memory import
ChatMessageHistory
```

```
from langchain.schema import
messages_from_dict, messages_to_dict
# 实例化对象
history = ChatMessageHistory()
# 添加历史信息
history.add_user_message("吃完了吗?")
history.add_ai_message("你说啥?")
# 保存历史信息到字典里
dicts = messages_to_dict(history.messages)
print(dicts)

print("*****")
# 从字典转换成列表
new_message = messages_from_dict(dicts)
print(new_message)
```

3.4.2 Memory classes

- Memory Classes 是一个更高级的概念，提供了一种更灵活和强大的方式来管理对话中的记忆和状态。它们不仅包括Chat Messages的功能，还提供了更多高级特性，如长期记忆、短期记忆、不同记忆类型的管理等。
- 基础的内存模块，用于存储历史的信息

> 示例

```
from langchain_openai import ChatOpenAI
# 导入ConversationChain模块，这个模块用于创建一个对话链，可以用于持续的多轮对话
```

```
from langchain.chains import
ConversationChain
# 初始化ChatOpenAI模型实例
llm = ChatOpenAI()
# 创建一个对话链实例，使用上面初始化的ChatOpenAI模型，并设置verbose为True，这样可以在运行时输出更多的调试信息
conversation = ConversationChain(
    llm=llm,
    verbose=True # 设置了verbose=True以便查看提示。
)
# 使用对话链的predict方法来获取对输入"小明有4个苹果"的回答
result1 = conversation.predict(input="小明有4个苹果")
# 使用对话链的predict方法来获取对输入"小张有5个苹果"的回答
result2 = conversation.predict(input="小张有5个苹果")
# 使用对话链的predict方法来获取对输入"小明和小张一共有多少个苹果"的回答
result3 = conversation.predict(input="小明和小张一共有多少个苹果")

conversation.invoke(input="请通俗一点进行回答")
```

- 保存历史信息

> 示例

```
from langchain_openai import ChatOpenAI
from langchain.schema import
messages_from_dict, messages_to_dict
# 导入ConversationChain模块，这个模块用于创建一个对话链，可以用于持续的多轮对话
from langchain.chains import
ConversationChain
# 初始化ChatOpenAI模型实例
llm = ChatOpenAI()
# 创建一个对话链实例，使用上面初始化的ChatOpenAI模型，并设置verbose为True，这样可以在运行时输出更多的调试信息
conversation = ConversationChain(
    llm=llm
)
# 使用对话链的predict方法来获取对输入"小明有4个苹果"的回答
result1 = conversation.predict(input="小明有4个苹果")
# 使用对话链的predict方法来获取对输入"小张有5个苹果"的回答
result2 = conversation.predict(input="小张有5个苹果")
# 使用对话链的predict方法来获取对输入"小明和小张一共有多少个苹果"的回答
result3 = conversation.predict(input="小明和小张一共有多少个苹果")

conversation.invoke(input="请通俗一点进行回答")
```

```
# 保存历史对话记录
print(conversation.memory.chat_memory.messages)
# 转换成字典
dicts = messages_to_dict(conversation.memory.chat_memory.messages)
print("--------------------")
print(dicts)
```

- 序列号存储到文件
  - pickle是专门用于把数据写入二进制文件当中，pickle模块是Python专用的持久化模块
  - pickle.dump() => 序列化保存
  - pickle.load() => 反序列号读取

> 示例

```
import pickle
from langchain.chains import ConversationChain
from langchain_openai import ChatOpenAI
from langchain.schema import messages_to_dict

llm = ChatOpenAI()
conversation = ConversationChain(
    llm=llm,
)
```

```
result1 = conversation.predict(input="小明有4个苹果")
result2 =conversation.predict(input="小张有5个苹果")
result3 =conversation.predict(input="小明和小张一共有多少个苹果")
conversation.invoke(input="请通俗一点进行回答")
# 保存历史对话记录
conversation.memory.chat_memory.messages
# 转换成字典
dicts = messages_to_dict(conversation.memory.chat_memory.messages)
print(dicts)
# 存储到文件
f = open("./memory", 'wb')
pickle.dump(dicts,f)
f.close()
```

- 再次添加对话

> 示例

```
import pickle
from langchain.chains import ConversationChain
from langchain.schema import messages_from_dict
from langchain.memory import ChatMessageHistory
```

```
from langchain.memory import
ConversationBufferMemory
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(
    model="gpt-4o-mini"
)

# 使用pickle加载之前保存的内存数据
dicts_load = pickle.load(open("./memory",
"rb"))
print(dicts_load)
print("-----------")

# 将加载的字典转换成消息列表
new_messages =
messages_from_dict(dicts_load)
# 创建一个包含历史消息的聊天历史记录对象
retrieved_chat_history =
ChatMessageHistory(messages=new_messages)
# 创建一个对话缓冲内存对象，并传入聊天历史记录
retrieved_memory =
ConversationBufferMemory(chat_memory=retrie
ved_chat_history)
# 重新加载之前的对话链，并设置语言模型和对话内存
conversation_reload = ConversationChain(
    llm=llm,
    # verbose=True,  # 如果设置为True，将会输
出更多的调试信息
    memory=retrieved_memory
```

```
)
# 向对话链输入新的话语
conversation_reload.predict(input="我回来
了")
# 打印出对话链中的当前对话内存
print(conversation_reload.memory.buffer)
```

3.4.3 ConversationBufferWindowMemory

- 保存一段时间内对话的互动列表。它只使用**最后** K 次互动

> 示例

```
from langchain.memory import
ConversationBufferWindowMemory
# 将历史记录作为消息列表获取
window =
ConversationBufferWindowMemory(k=2)
window.save_context({"input": "第一轮问"},
{"output": "第一轮答"})
window.save_context({"input": "第二轮问"},
{"output": "第二轮答"})
window.save_context({"input": "第三轮问"},
{"output": "第三轮答"})
# 调用load_memory_variables方法，加载当前存储的
记忆变量，并打印出来。
# 需要传递一个空字典 {}
print(window.load_memory_variables({}))
```

3.4.4 ConversationSummaryMemory

- 汇总对话：每次新的互动发生时对其进行汇总，然后将其添加到之前所有互动的“对话汇总”中。
- 优点 => 是对于长对话，可以减少使用的 Token 数量，因此可以记录更多轮的对话信息，使用起来也直观易懂。
- 缺点 => 是对于较短的对话，可能会导致更高的 Token 使用。

> 示例

```
from langchain.memory import
ConversationSummaryMemory
from langchain_openai import ChatOpenAI
from langchain.chains import
ConversationChain
llm = ChatOpenAI(
    model="gpt-4o-mini"
)
# 初始化对话链
conversation = ConversationChain(
    llm=llm,

 memory=ConversationSummaryMemory(llm=llm,b
uffer="以中文表示"),
)
# 回合1
result = conversation("我明天要去北京，需要买车
票。")
```

```
print(result)
print("1111111111111111111111111")
# 回合2
result = conversation("我是去北京看升旗仪式，听说天安门广场的升旗很有气势。")
print(result)
print("2222222222222222222222222")
# 回合3
result = conversation("我今天来到了售票站,我是来干嘛的呢？")
print(result)
```

3.5 Chains_module

3.5.1 single

> 示例

```
from langchain_core.prompts import PromptTemplate
from langchain.chains import LLMChain
from langchain_openai import ChatOpenAI
# 定义模板
template = "{name}开了一个早餐店, 帮我取一个有吸引力的店名"
# 参数一 => 包含模板中所有变量的名称。
# 参数二 => 表示提示模板的字符串,待填充的变量位置
prompt = PromptTemplate(input_variables=["name"], template=template)
print(prompt)
```

```
# 实例化模型
llm = ChatOpenAI(model_name="gpt-4o-mini")

# 构造Chain，将大模型与prompt组合在一起
chain = LLMChain(llm=llm, prompt=prompt)

# 执行Chain
result = chain.invoke({"name": "先知"})
print(f'result-->{result}')
```

3.5.2 llmchains

> 示例

```
from langchain_core.prompts import
PromptTemplate
from langchain.chains import LLMChain,
SimpleSequentialChain
from langchain_openai import ChatOpenAI
# 创建第一条链
# 定义模板
template = "{name}开了一个早餐店，帮我取一个有吸
引力的点名"
prompt = PromptTemplate(input_variables=
["name"], template=template)

llm = ChatOpenAI(model_name="gpt-4o-mini")

# 构造Chain：第一条链
```

```
first_chain = LLMChain(llm=llm,
prompt=prompt)

# 创建第二条链
# 定义模板
second_prompt =
PromptTemplate(input_variables=
["name"],template="{name}开店赚钱了,然后又开了
一个公司，帮我取一个有吸引力的公司名")

# 创建第二条链
second_chain = LLMChain(llm=llm,
prompt=second_prompt)

# 融合两条链:verbose为True的时候，显示模型推理过
程，否则不显示
overall_chain =
SimpleSequentialChain(chains=[first_chain,
second_chain],verbose=True)

# 使用链
result = overall_chain.invoke("先知")
print(result)
```

3.5.3 LCEL

- LangChain 表达式语言 (LCEL) 是一种声明式方法，可轻松组合链。LCEL 从第一天开始就被设计为支持将原型投入生产，无需更改代码，从最简单的“prompt + LLM”链到最复杂的链（我们已经看到有人

在生产中成功运行了包含 100 多个步骤的 LCEL 链）。参考

> 示例

```
from typing import List
import openai
prompt_template = "Tell me a short joke about {topic}"
client = openai.OpenAI()

def call_chat_model(messages: List[dict]) -> str:
    response = client.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=messages,
    )
    return response.choices[0].message.content

def invoke_chain(topic: str) -> str:
    prompt_value = prompt_template.format(topic=topic)
    messages = [{"role": "user", "content": prompt_value}]
    return call_chat_model(messages)

print(invoke_chain("ice cream"))
```

> 示例

```
from langchain_openai import ChatOpenAI
from langchain_core.prompts import
ChatPromptTemplate

model = ChatOpenAI(model="gpt-3.5-turbo")
prompt =
ChatPromptTemplate.from_template("tell me a
short joke about {topic}")

chain = prompt | model

print(chain.invoke({"topic": "ice
cream"}).content)
```

3.6 Agents_module

> 示例

```
from langchain import PromptTemplate
from langchain_openai import ChatOpenAI
from langchain.agents import load_tools,
initialize_agent, AgentType

llm = ChatOpenAI(model="gpt-3.5-turbo")
# 工具加载函数:利用工具来增强模型: llm-math计算,
wikipedia
# 参数一 => 要加载的工具名称
# 参数二 => 语言模型
tools = load_tools(["llm-math",
"wikipedia"], llm=llm)
```

```
# 初始化一个智能体（agent）
# 参数一 => 之前加载的工具
# 参数二 => 语言模型
# 参数三 => 智能体的类型,不需要针对特定任务进行训练
# 参数四 => 智能体在执行任务时是否输出详细的日志信息
# 参数五 => 智能体在处理输入时是否应该处理解析错误
agent = initialize_agent(tools=tools,
                         llm=llm,

agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
                         verbose=True,

handle_parsing_errors=True)

prompt_template = "明朝建立什么时候,皇帝是谁?"

prompt = PromptTemplate.from_template(prompt_template)

res = agent.run(prompt)
print(res)
```

作业

- 使用langchain实现一个RAG