1. Langchain-chain链

1.1 什么是chain(链)

在Langchain中，“链”（Chains）指的是一个概念上的组件或模块，它能够处理输入并产生输出。这个概念类似于编程中的函数或者工作流中的步骤，其中每个链都可以执行特定的任务，如文本生成、问答、翻译等。链条的设计目的是为了提供一种方式来组织和连接不同的自然语言处理（NLP）任务，使得这些任务可以以有序的方式相互作用，从而构建出更加复杂的AI应用。

例如，我们可以构造一条简单链接受用户的输入经过格式化（PromptTemplate）后传递给大模型。

1.2 简单链（内置的链）

1.2.1 LLMchain

`LLMchain` 是一个简单的链，`LLMChain` 由 `PromptTemplate` 和 `语言模型（LLM或聊天模型）` 组成。

```
from langchain.chains.llm import LLMChain
from langchain.prompts.chat import SystemMessagePromptTemplate,
HumanMessagePromptTemplate, ChatPromptTemplate
from langchain_openai import ChatOpenAI

# 创建一个系统消息，用于定义机器人的角色
system_message = SystemMessagePromptTemplate.from_template(
    "You are a helpful assistant."
)

# 创建一个人类消息，用于接收用户的输入
human_message = HumanMessagePromptTemplate.from_template(
    "{user_question}"
)

# 将这些模板结合成一个完整的聊天提示
chat_prompt = ChatPromptTemplate.from_messages([
    system_message,
```

```
    human_message,
])

# 使用 OpenAI API 的 ChatOpenAI 模型
chat_model = ChatOpenAI(
    openai_api_key="EMPTY",  # 替换为你的实际API密钥
    base_url="http://192.168.103.173:10259/v1",
    model="Qwen2___5-7B-Instruct"
)

"""旧版本"""
# 创建一条LLMChain链，verbose=True可以显示提示信息。
llm_chain = LLMChain(
    llm=chat_model,
    prompt=chat_prompt,
    verbose=True
)

# 测试LLMChain
response = llm_chain("你好")
print(response["text"])


"""新版本"""
# 创建一个 RunnableSequence 链
chain = chat_prompt | chat_model

# 测试链
response = chain.invoke({"user_question": "你好"})
print(response.content)
```

我们可以观察调试信息了解LLMChain的过程，输入的提示词先被格式化为完整的提示词然后输入LLM结束链得到答案。

因此链的作用就是将Prompt和LLM链接起来，构成一个完整的应用。

在旧版本的调用中，使用了 `verbose=True` 所以可以打印链的信息：

```
> Entering new LLMChain chain...
Prompt after formatting:
System: You are a helpful assistant.
Human: 你好
> Finished chain.
你好！很高兴能为你提供帮助。请问有什么我可以帮你的事情吗？
```

1.2.2 检索链 RetrievalQA

检索链是Langchain中的一种特殊类型的链，主要用于从大量的文档数据集中检索相关信息，并且通常与向量数据库（如Chroma、Pinecone、Faiss等）结合使用。

检索链可以帮助我们在处理如知识库查询、文档搜索等场景时，更有效地找到相关的文档片段，并且利用这些文档片段来生成准确的回答。

1.2.2.1 检索增强（RAG）

检索增强是指将检索技术与生成式模型相结合的一种方法。

在传统的生成式模型中，所有的知识都是基于模型在训练阶段学到的信息。而在RAG中，当模型接收到一个查询时，它不仅依赖于自身的知识，还会从外部数据源（如文档、数据库或其他形式的知识库）检索相关信息来辅助生成答案。

这种方法特别适用于需要提供精确信息的场景，比如法律咨询、医疗诊断等领域。

1.2.2.2 词嵌入Embedding

词嵌入是自然语言处理（NLP）领域的一个重要概念，指的是将文本中的词汇或短语映射到多维向量空间的技术。

这些向量不仅捕捉了词语的意义，还反映了词语之间的语义关系。在检索增强的上下文中，词嵌入用于将文档转换成向量形式，以便能够进行高效的相似度比较。

向量数据库存储这些向量，并且可以快速地根据新的查询向量来检索出最相似的文档。

下面是一个简单的检索链的例子：

安装依赖：

```
pip install sentence-transformers==3.3.0
pip install faiss-cpu==1.9.0
pip install langchain-huggingface==0.1.2
```

sentence-transformers

`sentence-transformers`是一个Python库，它提供了预训练的模型来计算句子、段落或短文本的嵌入向量（embeddings）。这些嵌入向量可以用来比较文本之间的相似度，进行聚类分析，或者作为其他机器学习任务的输入特征。这个库基于Transformers架构，如BERT、RoBERTa等，并且已经针对特定任务进行了微调，使得用户能够轻松地获取高质量的语义表示。

faiss-cpu

`faiss-cpu`是Facebook Research开发的一个高效相似度搜索库的CPU版本。Faiss旨在帮助开发者快速地从大量的向量中找到与查询向量最相似的一组向量。这在推荐系统、图像检索、文档相似度匹配等领域有着广泛的应用。Faiss支持多种距离度量方法，包括余弦相似度、欧氏距离等，并且优化了索引结构以提高搜索速度。

这两个库经常一起使用，首先，使用`sentence-transformers`将文本转换为嵌入向量；然后，利用`faiss-cpu`构建高效的索引结构，以便快速地在大量文本嵌入中进行相似性搜索。这样的组合不仅提高了处理效率，也简化了开发流程。例如，在构建一个问答系统时，可以先将所有可能的问题转化为嵌入，再通过Faiss建立索引，当用户提出新问题时，系统可以通过查找最相似的问题来提供答案。

案例

首先，我们需要创建一个向量存储，这通常涉及到将文档转换成嵌入向量，并存储到数据库中。

> 需要在魔搭下载Embedding模型：
>
> ```
> from modelscope.hub.snapshot_download import snapshot_download
> emb_model_dir = snapshot_download('AI-ModelScope/bge-large-zh-v1.5',cache_dir='models')
> ```

代码：

```
from langchain_community.vectorstores import FAISS  # 导入FAISS向量存储库
from langchain_huggingface import HuggingFaceEmbeddings  # 导入Hugging Face嵌入模型
from langchain_community.document_loaders import TextLoader  # 导入文本加载器
from langchain.text_splitter import RecursiveCharacterTextSplitter  # 导入递归字符文本分割器
```

```
from langchain_openai import ChatOpenAI  # 导入ChatOpenAI模型

# 使用 OpenAI API 的 ChatOpenAI 模型
chat_model = ChatOpenAI(
    openai_api_key="EMPTY",  # 替换为你的实际API密钥
    base_url="http://192.168.103.173:10259/v1",
    model="Qwen2___5-7B-Instruct"
)

# 加载文本文件 "黑悟空.txt"，编码格式为 'utf-8'
loader = TextLoader("黑悟空.txt", encoding='utf-8')
docs = loader.load()  # 将文件内容加载到变量 docs 中

# 把文本分割成 200 字一组的切片，每组之间有 20 字重叠
text_splitter = RecursiveCharacterTextSplitter(chunk_size=200,
chunk_overlap=20)
chunks = text_splitter.split_documents(docs)  # 将文档分割成多个小块
print(chunks)
# 初始化嵌入模型，使用预训练的语言模型 'bge-large-zh-v1___5'
embedding = HuggingFaceEmbeddings(model_name='models/AI-
ModelScope/bge-large-zh-v1___5')

# 构建 FAISS 向量存储和对应的 retriever
vs = FAISS.from_documents(chunks, embedding)  # 将文本块转换为向量并存
储在FAISS中
```

接下来，我们创建一个检索器（retriever），它可以从向量数据库中获取相关文档：

```
retriever = vs.as_retriever()  # 创建一个检索器用于从向量存储中获取相关信
息
```

然后，我们可以创建一个检索链，它将使用检索器来查找相关文档，并利用这些文档来生成回答：

```
from langchain.chains import RetrievalQA  # 导入RetrievalQA链

from langchain.prompts import (
    ChatPromptTemplate,
    SystemMessagePromptTemplate,
    HumanMessagePromptTemplate,
)
```

```
# 创建一个系统消息，用于定义机器人的角色
system_message = SystemMessagePromptTemplate.from_template(
    "根据以下已知信息回答用户问题。\n 已知信息{context}"
)

# 创建一个人类消息，用于接收用户的输入
human_message = HumanMessagePromptTemplate.from_template(
    "用户问题：{question}"
)

# 将这些模板结合成一个完整的聊天提示
chat_prompt = ChatPromptTemplate.from_messages([
    system_message,
    human_message,
])

# 定义链的类型参数，包括使用的提示模板
chain_type_kwargs = {"prompt": chat_prompt}

# 创建一个问答链，将语言模型、检索器和提示模板结合起来
# chat_model:生成回答的语言模型，  stuff:所有检索到的文档内容合并成一个大文本块，然后传递给语言模型。
# retriever: 之前创建的一个 FAISS 检索器实例。它的作用是从 FAISS 向量存储中找到与用户问题最相关的文档或文本块。这些相关的文档会被传递给语言模型以生成回答。
# chain_type_kwargs 是一个字典，包含了用于配置问答链的一些关键参数。
qa = RetrievalQA.from_chain_type(llm=chat_model,
chain_type="stuff", retriever=retriever,
chain_type_kwargs=chain_type_kwargs)


# 用户的问题
user_question = "黑熊精自称为？"
# 使用检索器获取与问题相关的文档
related_docs = retriever.invoke(user_question)

# 打印相关文档的内容
print("相关文档:")
for i, doc in enumerate(related_docs):
    print(f"文档 {i+1}:")
    print(doc.page_content)
    print("-" * 40)
```

```
# 使用问答链来回答问题 "黑熊精自称为？" 并打印结果
print(qa.invoke(user_question))
```

查看 `retriever.invoke` 包含了检索器搜索到的相关信息。

1.3 链的调用方式

1.3.1 call

一种最直接的方法就是使用**call**，默认情况下**call**返回输入'user_question'和输出'text'的键值。

1.3.2 run

使用run方法会输出一个字符串而不是一个字典。

1.4 自定义链

通过Langchain LCEL表达式可以轻松自定义链。
例如我们构造一条简单链接受用户的输入并给出回答。

```
# 使用LCEL表达式构造自定义链
from langchain_core.output_parsers import StrOutputParser  # 导入字符串输出解析器
from langchain_core.prompts import ChatPromptTemplate  # 导入聊天提示模板
from langchain_openai import ChatOpenAI  # 导入ChatOpenAI模型

# 使用 OpenAI API 的 ChatOpenAI 模型
chat_model = ChatOpenAI(
    openai_api_key="EMPTY",  # 替换为你的实际API密钥
    base_url="http://192.168.103.173:10259/v1",
    model="Qwen2___5-7B-Instruct"
)

# 创建一个聊天提示模板，其中包含占位符 {topic}
prompt = ChatPromptTemplate.from_template("说出一句包含{topic}的诗句。")

# 创建一个字符串输出解析器，用于将语言模型的输出解析为字符串
```

```
output_parser = StrOutputParser()

# 构造一个链，依次包含提示模板、语言模型和输出解析器
chain = prompt | chat_model | output_parser

# 使用链来生成回答，并打印结果
print(chain.invoke({"topic": "花"}))
```

1.5 作业：完成一个成语接龙