# 5.1 链

在许多人第一次接触LangChain的时候，可能会因为其名字误以为它是区块链相关的内容。然而实际上，LangChain的名字源自其框架的核心设计思路：用最简单的链条，将大预言模型开发的各个组件链接起来。这里的“链”，即我们在复杂系统设计中常说的“链式结构”。

5.1.1 链的定义

或许你已经注意到了LangChain的logo——一只鹦鹉和一节链条。鹦鹉学舌寓意着大预言模型像鹦鹉一样预测人类的下一段文本，而由无数节链组合起来的链条，象征着通过各种组件的有序连接，形成强大的应用力量。

如果没有链式结构，那么单独的语言大模型，虽然对于简单的应用可能已经足够，但是对于更复杂的应用，我们需要将多个模型或组件进行“链式”结构的连接和组合，这样才能创造出更强大、更具协同性的应用。

例如，我们可以创建一个链，该链接收用户输入，使用PromptTemplate格式化它，然后将格式化的响应传递给大模型。我们可以通过将多个链结合在一起，或者将链与其他组件结合在一起，来构建更复杂的链。

这种链式结构在创新应用中的价值已经得到了验证。最近，Johei Nakajima在Twitter上分享了一篇名为《使用 GPT-4、Pinecone、LangChain 进行多样化应用的任务驱动自主代理》的论文，其中他介绍了最新的Baby AGI。虽然Baby AGI现在还只是概念代码阶段，但是通过这个概念我们可以看出，链式结构是实现创新应用的非常有价值的工具。

下面是最简单的一个链的示例代码：

先安装库：

!pip -q install openai langchain

设置密钥：

import os  
os.environ['OPENAI\_API\_KEY'] = ''

LLMChain 是最基本的构建块链。它接收一个提示模板，使用用户输入进行格式化，然后返回LLM的响应。

from langchain.llms import OpenAI  
from langchain.prompts import PromptTemplate  
  
llm = OpenAI(temperature=0.9)  
prompt = PromptTemplate(  
 input\_variables=["product"],  
 template="What is a good name for a company that makes {product}?",  
)

要使用LLMChain，首先创建一个提示模板。我们现在可以创建一个非常简单的链，它会接收用户输入，使用输入格式化提示，然后将其发送到LLM。

from langchain.chains import LLMChain  
chain = LLMChain(llm=llm, prompt=prompt)  
  
# Run the chain only specifying the input variable.  
print(chain.run("colorful socks"))

就如我们在模型输入输出所说, 如果有通用语言模型的方法，那么Langchain一定有聊天模型的方法。你也可以在LLMChain中使用聊天模型：

from langchain.chat\_models import ChatOpenAI  
from langchain.prompts.chat import (  
 ChatPromptTemplate,  
 HumanMessagePromptTemplate,  
)  
human\_message\_prompt = HumanMessagePromptTemplate(  
 prompt=PromptTemplate(  
 template="What is a good name for a company that makes {product}?",  
 input\_variables=["product"],  
 )  
 )  
chat\_prompt\_template = ChatPromptTemplate.from\_messages([human\_message\_prompt])  
chat = ChatOpenAI(temperature=0.9)  
chain = LLMChain(llm=chat, prompt=chat\_prompt\_template)  
print(chain.run("colorful socks"))

5.1.2 基础链类型

链的类型分为四种，包括LLMChain、RouterChain、顺序链（Sequential Chains）和转换链（Transformation Chain）。

LLMChain是一种简单的链，它在语言模型周围增加了一些功能。它在LangChain中被广泛应用，包括在其他链和代理中。LLMChain由提示模板和语言模型（可以是LLM或聊天模型）组成。它使用提供的输入键值（如果有，还有记忆键值）格式化提示模板，将格式化的字符串传递给LLM，并返回LLM的输出。

RouterChain是一种使用路由器链创建的链，它可以动态地选择给定输入的下一条链。路由器链由两部分组成：路由器链本身（负责选择要调用的下一条链）和目标链（路由器链可以路由到的链）。

顺序链（Sequential Chains）是在调用语言模型后的下一步，特别是当你希望将一次调用的输出作为另一次调用的输入时。顺序链允许你连接多个链并将它们组成执行特定场景的流水线。顺序链有两种类型：SimpleSequentialChain（最简单形式的顺序链，其中每一步都有一个单一的输入/输出，一个步骤的输出是下一个步骤的输入）和SequentialChain（一种更通用的顺序链，允许多个输入/输出）。

转换链（Transformation Chain）是一种使用通用转换链的方法。作为一个示例，我们将创建一个虚构的转换，它接收一个超长的文本，过滤文本以仅显示前三段，然后将其传递给LLMChain进行总结。

5.1.3 工具链的理解与应用

在Langchain 中，“链”的概念是最经常使用的。这些”链”其实就是由一系列工具链构成的，每一个工具都可以视为整个链中的一个环节。这些环节可能非常简单，例如将一个提示模板和一个大型语言模型链接起来，形成一个大型语言模型链（LLMChains）。然而，也可能更加复杂，例如在整个流程中，通过多个环节进行多个步骤的链接。这可能还包括多个大型语言模型以及各种不同的实用工具等。在工具链中，一个链的输出将成为下一个链的输入，这就形成了一个输入输出的链式流程。例如，你可能会从大型语言模型的输出中提取某些内容，将其作为Wolfram Alpha查询的输入，然后将查询结果带回，并再次通过大型模型生成将返回给用户的响应。这就是一个典型的工具链的示例。

常见工具链的功能与应用

在实际的应用中，一些常见的工具链如APIChain、ConversationalRetrievalQA等已经被封装好了。

APIChain使得大型语言模型可以与API进行交互，以获取相关的信息。构建该链时，需要提供一个与所提供的API文档相关的问题。

ConversationalRetrievalQA链在检索问答链的基础上提供了一个聊天历史组件。它首先将聊天历史（要么明确传入，要么从提供的内存中检索）和问题合并成一个独立的问题，然后从检索器中查找相关的文档，最后将这些文档和问题传递给一个问答链，以返回响应。

对于需要对多个文档进行文档合并的任务，我们可以使用文档合并链，如MapReduceDocumentsChain或StuffDocumentsChain等。

对于需要从同一段落中提取多个实体及其属性的任务，我们可以使用提取链。

还有一些专门设计用来满足特定需求的链，如ConstitutionalChain，这是一个保证大型语言模型输出遵循一定宪法原则的链，通过设定特定的规则和指导方针，使得生成的内容符合这些原则，从而提供更受控、符合伦理和上下文适当的回应。

工具链的使用方法

这些工具链的使用方法通常是先使用类方法实例化，然后通过run方法调用，输出结果是一个字符串，然后将这个字符串传递给下一个链。类方法通常以”from”和下划线开始，比较常见的有from\_llm()和from\_chain\_type()，他们都接受外部的数据来源作为参数。

下面以SQLDatabaseChain为例子，看看如何使用它。SQLDatabaseChain就是一个通过from\_llm()方法实例化的链，它用于回答SQL数据库上的问题。

from langchain import OpenAI, SQLDatabase, SQLDatabaseChain  
  
db = SQLDatabase.from\_uri("sqlite:///../../../../notebooks/Chinook.db")  
llm = OpenAI(temperature=0, verbose=True)  
  
db\_chain = SQLDatabaseChain.from\_llm(llm, db, verbose=True)  
  
db\_chain.run("How many employees are there?")

运行的结果是：

> Entering new SQLDatabaseChain chain...  
 How many employees are there?  
 SQLQuery:  
  
 /workspace/langchain/langchain/sql\_database.py:191: SAWarning: Dialect sqlite+pysqlite does \*not\* support Decimal objects natively, and SQLAlchemy must convert from floating point - rounding errors and other issues may occur. Please consider storing Decimal numbers as strings or integers on this platform for lossless storage.  
 sample\_rows = connection.execute(command)  
  
  
 SELECT COUNT(\*) FROM "Employee";  
 SQLResult: [(8,)]  
 Answer:There are 8 employees.  
 > Finished chain.  
  
  
  
  
  
 'There are 8 employees.'

5.1.4 链的使用

异步支持

LangChain通过利用asyncio库为链（Chain）提供了异步支持。

目前在LLMChain（通过arun, apredict, acall）、LLMMathChain（通过arun和acall）、ChatVectorDBChain以及QA链中支持异步方法。其他链的异步支持正在规划中。

使用方法

所有的链都可以像函数一样被调用。当链对象只有一个输出键（也就是说，它的output\_keys中只有一个元素）的时候，我们预期的结果只需要一个字符串，可以使用run方法。

在LangChain中，所有继承自Chain类的对象，提供了一些用于执行链逻辑的方式。其中一种比较直接的方式就是使用\_\_call\_\_方法。\_\_call\_\_ 方法是 Chain 类的一个方法，它让 Chain 类的实例可以像函数一样被调用，比如 result = chain(inputs, return\_only\_outputs=True)就完成了调用链。

先看看\_\_call\_\_方法的定义：

def \_\_call\_\_(  
 self,  
 inputs: Union[Dict[str, Any], Any],  
 return\_only\_outputs: bool = False,  
 callbacks: Callbacks = None,  
 \*,  
 tags: Optional[List[str]] = None,  
 include\_run\_info: bool = False,  
 ) -> Dict[str, Any]:

这个 \_\_call\_\_ 方法接收的参数，最有用的是以下三个：

inputs：这个参数是要传递给链的输入。它的类型是 Any，这意味着可以接收任何类型的输入。

return\_only\_outputs：这个参数是一个布尔值，如果设为 True，则只返回输出结果。如果设为 False，则可能返回其他额外的信息。

callbacks：这个参数是回调函数的列表，它们将在链执行过程中的某些时刻被调用。

\_\_call\_\_ 方法返回一个字典，这个字典包含了链执行的结果和可能的其他信息。

在 Python 中，如果一个类定义了 \_\_call\_\_ 方法，那么这个类的实例就可以像函数一样被调用。例如，如果 chain 是 Chain 类的一个实例，那么你可以像调用函数一样调用 chain：

result = chain(inputs, return\_only\_outputs=True)

在这个调用中，inputs 是要传递给链的输入，return\_only\_outputs=True 表示只返回输出结果。返回的 result 是一个字典，包含了链执行的结果。

使用的时候，最重要的参数是 inputs:

chat = ChatOpenAI(temperature=0)  
prompt\_template = "Tell me a {adjective} joke"  
llm\_chain = LLMChain(llm=chat, prompt=PromptTemplate.from\_template(prompt\_template))  
  
llm\_chain(inputs={"adjective": "corny"})

返回的结果是：

{'adjective': 'corny',  
 'text': 'Why did the tomato turn red? Because it saw the salad dressing!'}

你可以通过设置return\_only\_outputs为True来配置它只返回输出键值。

llm\_chain("corny", return\_only\_outputs=True)

返回的结果就不包含 "adjective": "corny"：

{'text': 'Why did the tomato turn red? Because it saw the salad dressing!'}

然而，当链对象只有一个输出键（也就是说，它的output\_keys中只有一个元素）的时候，我们可以使用run方法。

# llm\_chain only has one output key, so we can use run  
llm\_chain.output\_keys

['text']

output\_keys中只有一个元素 ['text']，我们可以run方法：

llm\_chain.run({"adjective": "corny"})

如果输入的键值只有一个，预期的输出也是一个字符串，那么输入可以是字符串也可以是对象，可以使用run方法也可以使用\_\_call\_\_方法。

run方法将整个链的输入键值（input key values）进行处理，并返回处理后的结果。需要注意的是，与\_\_call\_\_方法可能返回字典形式的结果不同，run方法总是返回一个字符串。这也是为什么当链对象只有一个输出键的时候，我们倾向于使用run方法，因为这时候处理结果自然只有一个，返回字符串形式更直观也更便于处理。

例如，假设我们有一个链对象，它的任务是根据输入的文本生成摘要，那么在调用run方法的时候，我们可以直接将待摘要的文本作为参数输入，然后得到摘要后的文本。在这种情况下，你可以直接输入字符串，而无需指定输入映射。

另外，你可以很容易地将一个Chain对象作为一个工具，通过它的run方法集成到你的Agent中，这样可以将链的处理能力直接用于你的Agent逻辑中。

支持自定义链

你可以子类化Chain并实现你自己的自定义链。从其输出中仅仅调试链对象可能会比较困难，因为大多数链对象涉及到相当多的输入提示预处理和LLM输出后处理。

链的调试

将verbose设置为True将会在运行链对象时打印出一些链对象的内部状态。

conversation = ConversationChain(  
 llm=chat,  
 memory=ConversationBufferMemory(),  
 verbose=True  
)  
conversation.run("What is ChatGPT?")

加记忆的链

链可以使用Memory对象进行初始化，这将使得在调用链时数据持久化，使得链具有状态。

from langchain.chains import ConversationChain  
from langchain.memory import ConversationBufferMemory  
  
conversation = ConversationChain(  
 llm=chat,  
 memory=ConversationBufferMemory()  
)  
  
conversation.run("Answer briefly. What are the first 3 colors of a rainbow?")  
# -> The first three colors of a rainbow are red, orange, and yellow.  
conversation.run("And the next 4?")  
# -> The next four colors of a rainbow are green, blue, indigo, and violet.

链序列化

我们使用的序列化格式是json或yaml。目前，只有一些链支持这种类型的序列化。我们将随着时间的推移增加支持的链的数量。首先，让我们看看如何将链保存到磁盘。这可以通过.save方法完成，并指定一个带有json或yaml扩展名的文件路径。我们可以使用load\_chain方法从磁盘加载链。