1.Langchain

Langchain 是一个开源框架，它允许开发人员将大语言模型与外部的计算和数据源结合起来。例如让聊天机器人不仅回答通用问题，还能从本地数据库或文件中提取信息，并根据这些信息执行具体操作，比如发邮件，订票

LangChain 是一个用于 开发 由 **语言模型驱动** 的 应用程序的框架。它有三个核心组件

Compents组件，为LLMs提供接口封装、模板提示和信息检索索引；

- 模型 Models:包装器允许你连接到大型语言模型，如 GPT-4 或 Hugging Face 也包括GLM 提供的模型。
- Prompt Templates:这些模板让你避免硬编码文本输入。你可以动态地将用户输入插入到模板中，并发送给语言模型。
- Embedding 嵌入与向量存储 VectorStore 是数据表示和检索的手段，为模型提供必要的语言理解基础。
- Indexes: 索引帮助你从语言模型中提取相关信息。

Chains链，它将不同的组件组合起来解决特定的任务；

Agents代理，它们使得LLMs能够与外部API交互。

2. 翻译器(demo1)

2.1 调用语言模型

```
chatOpenAI_client = ChatOpenAI(model='glm-4-plus',
                               api_key="",
                               base_url="https://open.bigmodel.cn/api/paas/v4/")
msg = [
    SystemMessage(content='请将以下的内容翻译成意大利语'),
    HumanMessage(content='你好，请问你要去哪里？')
]
result = chatOpenAI_client.invoke(msg)
```

2.2 使用OutputParsers(输出解析器)

```
parser = StrOutputParser()
print(parser.invoke(result))
```

2.3 使用PromptTemplate(提示模板)

```
prompt_template = ChatPromptTemplate.from_messages([
    ('system', '请将下面的内容翻译成{language}'),
    ('user', "{text}")
])
chain = prompt_template | chatOpenAI_client | parser
#使用chain来调用
print(chain.invoke({'language': 'English', 'text': '我下午还有一节课，不能去打球了。'}))
```

2.4 使用LangServe部署应用程序

```
app = FastAPI(title='使用LangServe部署应用程序', version='V1.0', description='翻译器')
# 添加路由
add_routes(
    app,
    chain,
    path="/translation",
)

if __name__ == "__main__":
    import uvicorn
    # 启动服务
    uvicorn.run(app, host="localhost", port=8000)
```

三种调用方式

- localhost:8000/translation/invoke
- http://localhost:8000/translation/playground/
-
```
from langserve import RemoteRunnable

if __name__ == '__main__':
    client = RemoteRunnable('http://127.0.0.1:8000/translation/')
    print(client.invoke({'language': 'italian', 'text': '你好！'}))
```

2.5 使用LangSmish追踪应用程序

```
os.environ["LANGSMITH_TRACING"] = "true"
os.environ["LANGSMITH_ENDPOINT"] ="https://api.smith.langchain.com"
os.environ["LANGSMITH_API_KEY"] = 'lsv2_pt_288d95'  # 必需配置正确
os.environ["LANGCHAIN_PROJECT"] = "langsmith_tracing"
```

3. 聊天机器人(demo2)

3.1 保存聊天的历史记录

MessagesPlaceholder(variable_name='my_msg')

这是一个特殊的占位符，用于插入一组聊天记录（如历史对话）。它要求传入的上下文中有一个键为 'my_msg' 的变量，且该变量是一个 BaseMessage 列表（例如 [HumanMessage(...), AIMessage(...)]）

```
prompt_template = ChatPromptTemplate.from_messages([
    ('system', '你是一个乐于助人的助手。用{language}尽你所能回答所有问题。'),
    MessagesPlaceholder(variable_name='my_msg')
])

store = {}  # 所有用户的聊天记录都保存到store。key: sessionId,value: 历史聊天记录对象ChatMessageHistory

# 此函数预期将接收一个session_id并返回一个消息历史记录对象。
def get_session_history(session_id: str):
    if session_id not in store:
        store[session_id] = ChatMessageHistory()
    return store[session_id]
```

带有消息历史记录的可运行项

```
do_message = RunnableWithMessageHistory(
    chain,
    get_session_history,
    input_messages_key='my_msg'  # 每次聊天时候发送msg的key
)
```

3.2 流式响应

调用stream方法

```
for resp in do_message.stream(
        {
            'my_msg': [HumanMessage(content='请给我讲一个笑话？')],
            'language': 'English'
        },
        config=config):
    # 每一次resp都是一个token
    print(resp.content, end='-')
```

4. 构建向量数据库和检索器(demo3)

4.1 构建向量空间

pip install langchain-chroma qianfan

```
qianfan_embeddings = QianfanEmbeddingsEndpoint(
    model="bge-large-zh",  # 可选模型名称，例如 bge-large-zh、erine-text-embedding
等
    qianfan_ak=os.getenv("QIANFAN_AK"),
    qianfan_sk=os.getenv("QIANFAN_SK")
)
vector_store = Chroma.from_documents(documents, embedding=qianfan_embeddings)
# vector_store = Chroma.from_documents(documents, embedding=OpenAIEmbeddings())
```

4.2 构建检索器

```
# 检索器: bind(k=1) 返回相似度最高的第一个,分数越低相似度越高
retriever = RunnableLambda(vector_store.similarity_search).bind(k=1)
```

4.3 提示模板

```
# 提示模板
message = """
使用提供的上下文仅回答这个问题:
{question}
上下文:
{context}
"""
prompt_temp = ChatPromptTemplate.from_messages([('human', message)])
```

```
chain = {'question': RunnablePassthrough(), 'context': retriever} | prompt_temp |
qf_client
resp = chain.invoke('请介绍一下猫？')
```

```
RunnablePassthrough将用户的问题之后再传递给prompt和model
```

5. 调用工具(demo4)

5.1 创建搜索引擎

获取TAVILY_API_KEY,

Quickstart - Tavily Docs

```
os.environ["TAVILY_API_KEY"] = ''

# LangChain内置了一个工具，可以使用Tavily搜索引擎。
from langchain_community.tools.tavily_search import TavilySearchResults
search = TavilySearchResults(max_results=2)  #  max_results: 只返回两个结果
tools = [search]
```

5.2 构建代Agent

模型可以自动推理：是调用工具去完成用户的答案，还是模型回答

```
from langgraph.prebuilt import chat_agent_executor
agent_executor =
chat_agent_executor.create_tool_calling_executor(chatOpenAI_client, tools)
resp2 = agent_executor.invoke({'messages': [HumanMessage(content='北京天气怎么
样？')]})
print(resp2['messages'])
print(resp2['messages'][2].content)
```

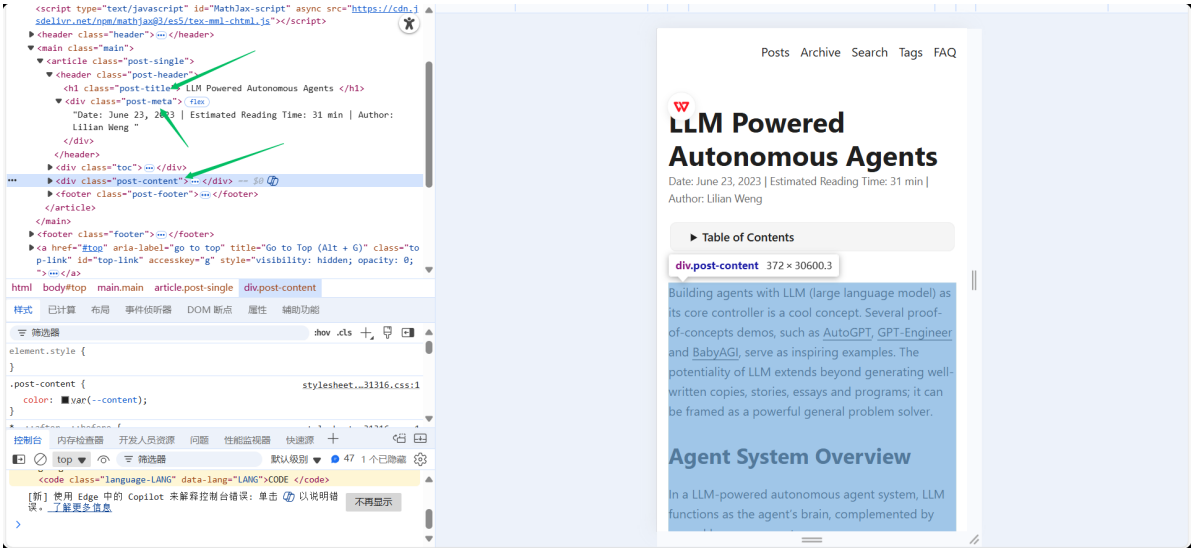
另一种调用方式

```
model_with_tools = model.bind_tools(tools)
resp2 = model_with_tools.invoke([HumanMessage(content='北京天气怎么样？')])
print(f'Model_Result_Content: {resp2.content}')
print(f'Tools_Result_Content: {resp2.tool_calls}')
```

6. 构建RAG(demo5)

6.1 加载数据

```
from langchain_community.document_loaders import WebBaseLoader
# WebBaseLoader：LangChain 提供的一个用于从网页加载内容的工具。
# web_paths：要爬取的网页地址列表。可以是一个或多个 URL。
# bs_kwargs：传递给 BeautifulSoup 的参数，用于控制解析行为。
# SoupStrainer：仅解析指定的部分，提升性能。这里指定了三个样式名'post-header' 'post-
title' 'post-content'
loader = WebBaseLoader(
    # 一次爬一个，或一次爬多个网页数据
    web_paths=['https://lilianweng.github.io/posts/2023-06-23-agent/'],
    bs_kwargs=dict(
        parse_only=bs4.SoupStrainer(class_=('post-header', 'post-title', 'post-
content'))
    )
)
```



docs的结构

```
docs = loader.load()
'''
Document(
    page_content="猫是独立的宠物，通常喜欢自己的空间。",
    metadata={"source": "哺乳动物宠物文档"},
)
'''
```

6.2 大文档的切割

```
# 每1000个按字符切分成一个片段，允许片段间有200个重复字符
splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
splits = splitter.split_documents(docs)
```

6.3 文档存入向量空间

```
vectorstore = Chroma.from_documents(documents=splits,
embedding=OpenAIEmbeddings())
retriever = vectorstore.as_retriever()
```

```
prompt = ChatPromptTemplate.from_messages(  # 提问和回答的 历史记录  模板
    [
        ("system", system_prompt),
        MessagesPlaceholder("chat_history"),  #
        ("human", "{input}"),
    ]
)
chain1 = create_stuff_documents_chain(chatOpenAI_client, prompt)
```

6.4 创建子链

```
contextualize_q_system_prompt = """Given a chat history and the latest user
question
which might reference context in the chat history,
formulate a standalone question which can be understood
without the chat history. Do NOT answer the question,
just reformulate it if needed and otherwise return it as is."""

retriever_history_temp = ChatPromptTemplate.from_messages(
    [
        ('system', contextualize_q_system_prompt),
        MessagesPlaceholder('chat_history'),
        ("human", "{input}"),
    ]
)

# 创建一个子链
```

```
history_chain = create_history_aware_retriever(chatOpenAI_client, retriever,
retriever_history_temp)
```

6.5 创建父链

```
chain = create_retrieval_chain(history_chain, chain1)
result_chain = RunnableWithMessageHistory(
    chain,
    get_session_history,
    input_messages_key='input',
    history_messages_key='chat_history',
    output_messages_key='answer'
)
```

7. 整合关系型数据库(demo6)

7.1 sqlalchemy 初始化MySQL数据库连接

```
MYSQL_URI = 'mysql+mysqldb://{}:{}@{}:{}/{}?charset=utf8mb4'.format(USERNAME,
PASSWORD, HOSTNAME, PORT, DATABASE)
db = SQLDatabase.from_uri(MYSQL_URI)
```

7.2 生成SQL

```
from langchain.chains.sql_database.query import create_sql_query_chain
test_chain = create_sql_query_chain(devagi_client, db)
resp = test_chain.invoke({'question': '请问：chip_dict表中有多少条数据？'})
print(resp)
```

7.3 执行SQL

```
from langchain_community.tools.sql_database.tool import QuerySQLDataBaseTool
execute_sql_tool = QuerySQLDataBaseTool(db=db)
```

```
answer_prompt = PromptTemplate.from_template(
    """给定以下用户问题、SQL语句和SQL执行后的结果，回答用户问题。
    Question: {question}
    SQL Query: {query}
    SQL Result: {result}
    回答: """
)
```

```
def clean_sql(raw_sql: str) -> str:
    # 去除 Markdown 标记
    cleaned = raw_sql.replace("```sql", "").replace("```", "")
    # 去除前后空白
    cleaned = cleaned.strip()
    return cleaned
```

```
chain = (
    # 第一步：使用 RunnablePassthrough.assign 创建一个新的键 "query"
    # 它的值是通过 test_chain 生成的 SQL 语句，并经过 clean_sql 函数清洗
    RunnablePassthrough.assign(query=test_chain | clean_sql)

    # 第二步：继续添加一个键 "result"
    # 它的值是上一步输出中的 "query" 键对应的 SQL 语句，
    # 传给 execute_sql_tool 工具执行后得到的结果
    .assign(result=itemgetter('query') | execute_sql_tool)

    # 第三步：将包含 question、query 和 result 的字典输入 answer_prompt，
    # 根据模板生成最终发送给 LLM 的提示词（Prompt）
    | answer_prompt

    # 第四步：将构造好的 Prompt 发送给大模型（devagi_client）进行推理，
    # 得到一个自然语言的回答（字符串形式）
    | devagi_client

    # 第五步：使用 StrOutputParser() 解析模型输出，
    # 确保最终返回的是一个干净的字符串，而不是带有 metadata 的对象
    | StrOutputParser()
)

rep = chain.invoke(input={'question': 'chip_dict表中有多少条数据'})
print(rep)
```

7.5 使用Agent整合关系型数据库

```
from langchain_community.agent_toolkits import SQLDatabaseToolkit
toolkit = SQLDatabaseToolkit(db=db, llm=chatOpenAI_client)
tools = toolkit.get_tools()
```

```
system_prompt = """
您是一个被设计用来与SQL数据库交互的代理。
给定一个输入问题，创建一个语法正确的SQL语句并执行，然后查看查询结果并返回答案。
除非用户指定了他们想要获得的示例的具体数量，否则始终将SQL查询限制为最多10个结果。
你可以按相关列对结果进行排序，以返回MySQL数据库中最匹配的数据。
您可以使用与数据库交互的工具。在执行查询之前，你必须仔细检查。如果在执行查询时出现错误，请重写查询SQL并重试。
不要对数据库做任何DML语句(插入，更新，删除，删除等)。

首先，你应该查看数据库中的表，看看可以查询什么。
不要跳过这一步。
然后查询最相关的表的模式。
"""
# 构建 PromptTemplate
prompt = ChatPromptTemplate.from_messages([
    ("system", system_prompt),
    MessagesPlaceholder(variable_name="messages"),
])
```

```
from langgraph.prebuilt.chat_agent_executor import create_tool_calling_executor
# 创建代理
agent_executor = create_tool_calling_executor(
    model=chatOpenAI_client,  # 你的大模型客户端
    tools=tools,  # SQL 工具列表
    prompt=prompt  # 系统提示
)
```

```
resp = agent_executor.invoke({'messages': [HumanMessage(content='chip_dict表有哪些label？')]})
```

8. 检索YouTube视频字幕(demo7)

8.1 向量空间持久化

```
embeddings = OpenAIEmbeddings(model='text-embedding-3-small')
persist_dir = '../chroma_data_dir'  # 存放向量数据库的目录
vectorstore = Chroma.from_documents(split_doc, embeddings,
persist_directory=persist_dir)
```

8.2 加载向量数据库

```
vectorstore = Chroma(persist_directory=persist_dir,
embedding_function=embeddings)
```

8.3 pydantic

通过继承BaseModel 类，定义结构化的数据模型，并对输入的数据进行自动类型转换和验证

支持必填字段、可选字段、字段验证、读取.env文件的值

sdudy_langchain/pydantic_test

9.提取结构化数据(demo8)

构建一个链从：非结构化的文本中提取结构化信息。

9.1 模板参数

```
prompt = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            "你是一个专业的提取算法。只从未结构化文本中提取相关信息。如果你不知道要提取的属性的值，返回该属性的值为null。",
        ),
        # 请参阅有关如何使用参考记录消息历史的案例
        # MessagesPlaceholder('examples'),
        ("human", "{text}"),
    ]
)
```

```
chain = {'text': RunnablePassthrough()} | prompt |
chatOpenAI_client.with_structured_output(schema=ManyPerson)
```

9.2 输出结构化信息

```
class Person(BaseModel):
    """
    关于一个人的数据模型
    """
    name: Optional[str] = Field(default=None, description='表示人的名字')

    hair_color: Optional[str] = Field(
        default=None, description="如果知道的话，这个人的头发颜色"
    )
    height_in_meters: Optional[str] = Field(
        default=None, description="以米为单位测量的高度"
    )
```

9.3 提取多个信息

```
class ManyPerson(BaseModel):
    """
    数据模型类： 代表多个人
    """
    people: List[Person]
```

```
text = "My name is Jeff, my hair is black and i am 6 feet tall. Anna has the same
color hair as me."
resp = chain.invoke(text)
```

10. AI自动生成数据(demo9)

pip install langchain-experimental

10.1 创建训练型数据生成器

```
from langchain_experimental.synthetic_data import create_data_generation_chain
```

10.2 创建表格型数据生成器

```
from langchain_experimental.tabular_synthetic_data.openai import
create_openai_data_generator
```

10.2.1 pydantic定义数据模型

10.2.2 提供样例数据

10.2.3 FewShotPromptTemplate

```
examples = [
    {
        "example": "Patient ID: 123456, Patient Name: 张娜, Diagnosis Code:
J20.9, Procedure Code: 99203, Total Charge: $500, Insurance Claim Amount: $350"
    },
    {
        "example": "Patient ID: 789012, Patient Name: 王兴鹏, Diagnosis Code:
M54.5, Procedure Code: 99213, Total Charge: $150, Insurance Claim Amount: $120"
    },
    {
        "example": "Patient ID: 345678, Patient Name: 刘晓辉, Diagnosis Code:
E11.9, Procedure Code: 99214, Total Charge: $300, Insurance Claim Amount: $250"
    },
]
```

```
openai_template = PromptTemplate(input_variables=['struc_example'], template="
{example}")
```

input_variables=['example']，表示这个模板接受一个输入变量，名字叫 `"example"`。每个 example 是一段包含病人信息的文本字符串（比如 "Patient ID: 123456, Patient Name: 张娜, ..."）

template="{example}" 是一个占位符，表示这个地方会被你提供的某个示例替代。就是把 example 的内容直接放进 prompt 中。

```
prompt_template = FewShotPromptTemplate(
    prefix=SYNTHETIC_FEW_SHOT_PREFIX,
    suffix=SYNTHETIC_FEW_SHOT_SUFFIX,
    examples=examples,
    example_prompt=openai_template,
    input_variables=['subject', 'extra']
)
```

`FewShotPromptTemplate` 来构建一个带有“少量样本（few-shot examples）”的提示模板

10.2.4 创建结构化数据生成器

```
generator = create_openai_data_generator(
    output_schema=MedicalBilling,  # 指定输出数据的格式
    llm=devagi_client,
    prompt=prompt_template
)
```

10.2.5 调用结构化数据生成器

```
result = generator.generate(
    subject='医疗账单',  # 指定生成数据的主题
    extra='医疗总费用呈现正态分布，最小的总费用为1000，名字可以是随机的，最好使用比较生僻的人名',  # 额外的一些指导信息
    runs=1  # 指定生成数据的数量
)
```

11. 文本分类(demo10)

```
temperature=0
```

情感分析：sentiment analysis (SA)

话题标记：topic labeling(TL)

新闻分类：news classification (NC)

对话行为分类：dialog act classification (DAC)

11.1 创建分类模型

```
class Classification2(BaseModel):
    """
        定义一个Pydantic的数据模型，未来需要根据该类型，完成文本的分类
    """
    # 文本的情感倾向，预期为字符串类型
    sentiment: str = Field(..., enum=["happy", "neutral", "sad"], description="文本的情感")


    # 文本的攻击性，预期为1到5的整数
    aggressiveness: int = Field(..., enum=[1, 2, 3, 4, 5], description="描述文本的攻击性，数字越大表示越攻击性")


    # 文本使用的语言，预期为字符串类型
    language: str = Field(..., enum=["spanish", "english", "french", "中文", "italian"], description="文本使用的语言")
```

11.2 创建提示模板

```
tagging_prompt = ChatPromptTemplate.from_template(
    """
    从以下段落中提取所需信息。
    只提取'Classification2'类中提到的属性。
    段落:
    {input}
    """
)
```

11.3 结构化输出

```
chain = tagging_prompt | devagi_client.with_structured_output(Classification2)
input_text = "Estoy increiblemente contento de haberte conocido! Creo que seremos
muy buenos amigos!"
result: Classification2 = chain.invoke({'input': input_text})
```

12. 文本摘要(demo11)

Map-reduce：是否超过大模型指定的Token上限？Stuff

还有一种Refine摘要

pip install tiktoken chromadb

```
from langchain_community.document_loaders import WebBaseLoader
loader = WebBaseLoader('https://lilianweng.github.io/posts/2023-06-23-agent/')
docs = loader.load()  # 加载网页上的一篇文章
```

12.1 模板传参

```
prompt_template = """针对下面的内容，写一个简洁的总结摘要:
"{text}"
简洁的总结摘要:"""
prompt = PromptTemplate.from_template(prompt_template)
llm_chain = LLMChain(llm=chatOpenAI_client, prompt=prompt)
stuff_chain = StuffDocumentsChain(llm_chain=llm_chain,
document_variable_name='text')
```

12.2 Map-reduce摘要

12.2.1 按token切割文档

```
text_splitter = CharacterTextSplitter.from_tiktoken_encoder(chunk_size=1000,
chunk_overlap=0)
split_docs = text_splitter.split_documents(docs)
```

12.2.2 每个文档片段分别摘要

```
map_template = """以下是一组文档(documents)
"{docs}"
根据这个文档列表，请给出总结摘要:"""
map_prompt = PromptTemplate.from_template(map_template)
map_llm_chain = LLMChain(llm=model, prompt=map_prompt)
```

12.2.3 StuffDocumentsChain

摘要累计token超过4000，分批次传送StuffDocumentsChain

```
reduce_template = """以下是一组总结摘要:
{docs}
将这些内容提炼成一个最终的、统一的总结摘要:"""
reduce_prompt = PromptTemplate.from_template(reduce_template)
reduce_llm_chain = LLMChain(llm=model, prompt=reduce_prompt)
combine_chain = StuffDocumentsChain(llm_chain=reduce_llm_chain,
document_variable_name='docs')
reduce_chain = ReduceDocumentsChain(
    # 这是最终调用的链。
    combine_documents_chain=combine_chain,
    # 中间的汇总的链
    collapse_documents_chain=combine_chain,
    # 将文档分组的最大令牌数。
    token_max=4000
)
```

12.2.4 合并所有链

```
map_reduce_chain = MapReduceDocumentsChain(
    llm_chain=map_llm_chain,
    reduce_documents_chain=reduce_chain,
    document_variable_name='docs',
    return_intermediate_steps=False
)
```