Chatbot: The "Hello World"

Course Code: 4E02-2

Author: Lloyd Sun

Date: Sep.23.2024

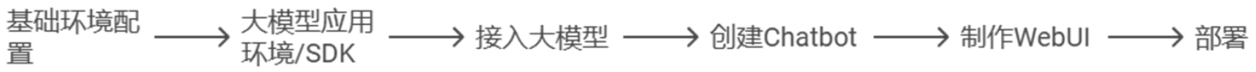
Version: V1.0

---

1 Chatbot Overview & Env Setup

动手构建应用是学习CS/AI最好的方法，正如物理学家Richard Feynman所说，"What I cannot create, I do not understand"，在编写代码构建应用的过程中，才能真正理解技术的细节。

本次介绍LLM应用领域的Hello World项目--Chatbot。简单理解，就是类似ChatGPT刚发布时的形态，一个可以对话的Web APP。在构建Chatbot的框架选择上，主要使用最适合入门学习的langchain。构建过程整体描述如下：



表面上看，实践本项目的前提条件是对python有基本的了解，或者能够正确使用自动编程工具；实际上真正的前提条件是不要害怕编写代码。

2 Env Setup

2.1 Python Environment Setup

如果已经有比较熟悉的IDE，使用原有环境，安装相应的Python工具即可；若希望从零开始搭建环境，直接按照下面的推荐操作即可，并没有纠结与折腾的必要。

1. 安装miniconda：miniconda是anaconda推出的轻量级环境管理工具，安装之后也顺便把python安装好了。在conda环境中怎么操作都没关系，大不了删了再建一个。关于Miniconda的安装，直接到官网上下载就可以了。

2. 使用Conda创建/切换环境：在命令行，使用`conda create -n <name> python=3.11`即可创建环境。Python版本可以不指定，但一般来说，3.8-3.11版本的兼容性最好。环境创建好之后，可以通过`conda env list`查看已经装好的环境，使用`conda activate <name>`进入相应环境。

3. 使用Poetry管理SDK：Python自带的pip是SDK/包管理的工具，但pip的问题是迁移、分享环境比较麻烦，它一般使用requirements.txt来记录环境中依赖的信息，非常难用，因此这里推荐使用poetry进行依赖管理（但坚持使用pip也没关系），可以在相应环境中，命令行输入`pip install poetry`来安装poetry.

4. Poetry的操作与建议：先切换到对应的目录下`cd <project_folder>`，对于新建的项目，习惯使用`poetry init`初始化（命令行中会要求填一些项目基本信息），之后会生成.toml文件，这个文件就是管理环境的核心了；如果已经拿到了别的项目的.toml文件，直接使用`poetry install --no-root`即可配置好相应环境。Poetry的问题是，安装某些环境时，会比较麻烦，比如Pytorch的GPU版本；同时Poetry也可以替代conda做环境管理，但似乎没有必要。总之，poetry的基本操作列在下面，可以尝试下（反正在conda环境中尝试也不会有什么影响）：

   (a) `poetry new <project_name>`: 基于内置的模板/目录结构，生成Python项目

(b) `poetry init`: 初始化

(c) `poetry install --no-root`: 安装依赖

(d) `poetry add <package_name>`: 添加依赖

(e) `poetry remove <package_name>`: 移除依赖

(f) `poetry show`: 查看依赖

5. Poetry换源：poetry（以及pip）默认从pypi网站下载对应的SDK。由于网速问题，很可能下载很慢或连接失败（90%的环境配置问题都是网络问题），这个时候就可以进行换源。poetry的换源方法是在.toml文件最后加上(此处切换为阿里云，也可以换成其他国内源)：

```
'''
[[tool.poetry.source]]
name = "aliyun"
url = "http://mirrors.aliyun.com/pypi/simple"
priority = "primary"
'''
```

> 提示：若使用pip而非poetry进行环境管理，可以忽略上述步骤3-5，直接使用pip安装环境即可(pip换源方法很简单，搜索可得)。

接下来是配置Chatbot应用需要用到的SDK了，主要是langchain、streamlit以及其他的辅助工具：

```
poetry add jupyter
poetry add langchain langchain-openai langchain-groq langchain-community langchain-ollama
poetry add python-dotenv
peotry add streamlit
```

当然，使用pip安装也是一样的，用`pip install <package_name>`即可；如果拿到了本项目Github Repo中的pyproject.toml文件，一键使用`poetry install --no-root`就可以了。

经过以上配置，完整的.toml文件如下图所示:

```
[tool.poetry]
name = "chatbot"
version = "1.0"
description = "A Toy Chatbot Built with Langchain for Educational Purpose"
authors = ["Lloyd Sun"]
readme = "README.md"

[tool.poetry.dependencies]
python = "^3.11"
jupyter = "^1.1.1"
langchain = "^0.3.0"
python-dotenv = "^1.0.1"
langchain-openai = "^0.2.0"
langchain-groq = "^0.2.0"
langchain-community = "^0.3.0"
langchain-ollama = "^0.2.0"
streamlit = "^1.38.0"


[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"

[[tool.poetry.source]]
name = "aliyun"
url = "http://mirrors.aliyun.com/pypi/simple"
priority = "primary"
```

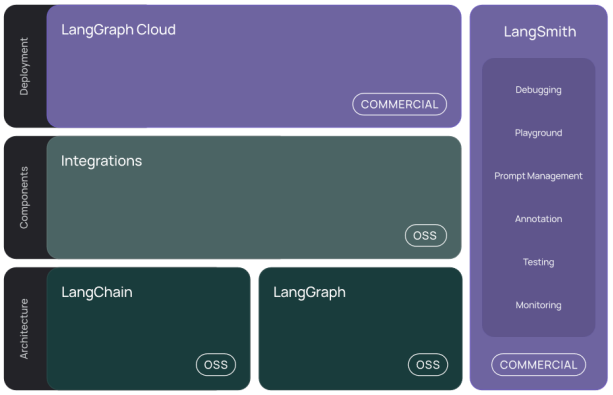
最后，再推荐一下IDE。若没有特别的倾向，推荐使用VS Code即可，当然目前更推荐CursorAI；对于Python项目，PyCharm也非常推荐，只是有点重；另外对于演示、实验、数据分析等场景，Jupyter Notebook也很好用，可以在浏览器或IDE中运行（本项目在VS Code中运行Jupyter，这也是安装依赖时安装Jupyter的原因）。

2.2 Langchain Overview

Langchain是否适合生产环境仍有争议，但无可争议的是，Langchain是目前最适合入门大模型应用的工具。Langchain集成了大量第三方服务、大模型、工具，官方也发布了大量的文档、教程，可以说了解Langchain集成了什么、有哪些功能，就能够掌握领域内好用的工具了。

在版本方面，Langchain在24年初（经过一年以上迭代）才发布了V0.1版本，在5月发布了V0.2版本，近期刚刚发布0.3版本。总体而言V0.1到V0.2改动很大，模块的设计、接口的调用方式等等都不太一样；但V0.3版本主要针对Pydantic 2的支持，langchain本身改动不大。另外，虽然关于langchain的教程有很多（尤其是入门教程），但绝大部分都是针对V0.1的，很多地方并不适用于新版。

目前，Langchain已经发展成由Langchain、LangGraph、LangSmith、LangGraph Cloud等等核心组件组成生态系统了。推荐大家阅读官方给出的“Conceptual Guide”，对于理解LLM应用很有帮助。



需要说明的是，在本项目的依赖安装过程中，也安装了多个langchain相关package，主要是因为langchain将主流的LLM服务提供商，分别使用不同的SDK进行管理，具体可以参考：Langchain Chat models。

2.3 API_KEYs

本项目中会接入三类模型：1）国外闭源模型，Groq；2）国内闭源模型，Deepseek-chat；3）本地模型，使用ollama接入。

对于闭源模型，需要到模型提供商官网注册账号，获得API_KEY之后才能调用。目前，一般的模型提供商在新用户注册时会赠送一些额度，再加上近期大模型推理成本大大降低，所以即使不充值，也足够初期的使用了。本项目选择Groq与Deepseek的原因主要是因为速度快和便宜，对于LLM能力排行，依旧推荐LMSYS Chatbot Arena排行榜。

对于接入闭源模型的API方式，大多模型供应商会提供API与自己研发的SDK（支持python）两种方式，同时也会支持OpenAI SDK的调用方式，也正是因为这一点，可以方便的通过langchain集成进来。

2.4 Ollama

对于LLM本地部署/推理，推荐使用Ollama这个模型接入工具。Ollama安装非常简单，到官网上下载后一键安装即可；同时对新模型的支持速度也非常快，主流的模型发布一天之内，基本就可以在ollama上找到了。

常用的命令行指令如下：

```
ollama pull <model:tag> # 下载模型
ollama run <model:tag> # 命令行中使用模型
ollama list # 查看已安装模型
```

2.5 Repo Structure

本次分享的代码地址是：ToyChatbot。简要介绍每个文件的含义，方便使用：

文件	说明
01_chatbot.ipynb	jupyter notebook文件。包含本次内容主要代码（除WebUI外）
02_webui_st.py	使用Streamlit构建的Web UI
03_langchainic_chatbot.ipynb	使用更Langchain的方式构建Chatbot
04_lc_st.py	Langchain+Streamlit官方webui示例，有少量修改
pyproject.toml & poetry.lock	poetry相关依赖文件
.env.example	.env的示例文件，使用是修改文件名为.env
requirements.txt	使用Streamlit Cloud部署时需要的环境依赖文件

3 Chatbot

> 本节对应代码文件：01_chatbot.ipynb （https://github.com/LloydS827/ToyChabot）

环境配置好之后，可以准备要接入的模型了。本项目将会接入Groq(国外闭源模型)、DeepSeek(国内闭源模型)和Ollama/Qwen2.5(3b, 开源模型)。对于Groq和Deepseek，注册好账号后将API_KEY保存至.env文件即可（或者设置对应环境变量），并注意账户安全；对于开源模型，安装好ollama之后，在命令行输入`ollama pull qwen2.5:3b`即可开始下载。

3.1 Connecting to LLMs

首先是闭源模型接入。通过dotenv，将.env中的API_KEY以环境变量方式导入进来：

```
from langchain.chat_models import init_chat_model
import os
from dotenv import load_dotenv
load_dotenv()
```

接着就可以使用langchain.chat_models中的init_chat_model模块，接入模型了：

```
# connect to deepseek model
llm = init_chat_model(
    model="deepseek-chat",
    model_provider="openai",
    api_key=os.environ.get('DS_API_KEY'),
    base_url="https://api.deepseek.com",
    temperature = 0.2,
    top_p=0.9,
    max_tokens=1024,
    stop=["<|endoftext|>", "<|endofcode|>"],
    verbose=True
)
# invoke the model
response = llm.invoke("Hello, how are you?")
```

init_chat_model方法是langchain V0.2.X版本新加入的功能，对原本的第三方LLM库，如langchain-openai, langchain-groq等加了封装，使用户能够通过同一行代码接入不同模型（原本的第三方库仍需安装）。由于Deepseek模型没有langchain第三方库（大部分国产闭源模型都没有），利用其支持openai接口的特点（大部分闭源模型都支持），可以使用langchain-openai进行接入，因此在model_provider处填入openai。另外，base_url要查看模型服务商的接口说明，如果直接接入openai模型，则可以省略。除model、model_provider、api_key、base_url等四个参数外，其他几个变量非必须，通过名字即可判断含义。模型接入后，使用invoke方法，就可以调用服务了。

同样，使用init_chat_model方法，可以接入groq的llama3-70b模型（这时调用langchain-groq库）：

```
llm = init_chat_model(
    temperature=0.2,
    model="llama3-70b-8192",
    model_provider="groq",
    api_key = os.environ.get('GROQ_API_KEY'),
)
llm.invoke("Hello, how are you?").content
```

最后，同样的方法调用ollama中已下载的本地模型（这时调用langchain-ollama库）：

```
llm = init_chat_model(
    model="qwen2.5:3b",
    model_provider="ollama",
)
llm.invoke("Hello, how are you?").content
```

由于Groq/国外闭源模型一般需要梯子才可以调用，Ollama/开源模型对计算机性能有要求，因此本项目后面的内容将主要基于Deepseek-chat/国内闭源模型进行演示。

3.2 Real-time Chatbot

3.2.1 Basis Chat History

对话历史是Chatbot场景中的重要组成部分。正如在4E02-1中介绍的，需要管理的信息包括系统级提示词、用户输入、LLM回复三类。Langchain中也有相应的方法：

```
from langchain_core.messages import SystemMessage, HumanMessage, AIMessage
# same as 'from langchain.schema import SystemMessage, HumanMessage, AIMessage'

# setup system prompt
system_message = SystemMessage(content='You are a helpful assistant. Try to think step by
step whenever possible')
```

下面，使用列表结构，建立一个最基本的对话历史：

```
chat_history = []
chat_history.append(system_message)
while True:
    query = input('User:')
    if query.lower() == "exit":
        break
    print(query)

    chat_history.append(HumanMessage(content=query))
    response = llm.invoke(chat_history)

    print(response.content)
    chat_history.append(AIMessage(content=response.content))

print("---- Message History ----")
print(chat_history)
```

3.2.2 Managing Chat History

现在需要对上述基本的对话历史进行管理，期待的管理方式是，能够设置需要保存的历史问答“轮数”（每一轮包括一问一答），来控制输入给LLM的对话长度。针对这样的管理需求，只需要对对话历史列表的长度进行判断，超过限制删除对应的对话即可。需要注意的是，要始终保持SystemMessage在第一位：

```
def manage_chat_history(chat_history, chat_length):
    # Ensure system message is always present
    system_message = chat_history[0]

    # Calculate the number of complete conversation rounds
    conversation_rounds = min(chat_length, (len(chat_history) - 1) // 2)

    # Keep only the specified number of conversation rounds
    managed_history = [system_message] + chat_history[-(conversation_rounds * 2):]

    return managed_history
```

将该方法加入主流程中，即可得到能够限定对话历史长度的Chatbot了：

```
chat_length = 3  # Set the desired number of conversation rounds to keep
chat_history = [system_message]

while True:
    query = input('User: ')
    if query.lower() == "exit":
        break
    print(query)

    chat_history.append(HumanMessage(content=query))
```

```
    chat_history = manage_chat_history(chat_history, chat_length)

    response = llm.invoke(chat_history)
    print("AI:", response.content)
    chat_history.append(AIMessage(content=response.content))
    chat_history = manage_chat_history(chat_history, chat_length)

print("---- Message History ----")
print(chat_history)
```

4 Webui & Streamlit

本节对应代码文件：02_webui_st.py （https://github.com/LloydS827/ToyChabot）

Streamlit是机器学习常用的开源WebUI架构，也是最适合制作原型、Demo的方法，类似的还有Gradio、Chainlit等。Streamlit语法简洁，非常容易上手，这里暂不展开叙述，可以通过chatbot webui代码，进行st的代码练习。

4.1 Streamlit Chat History

上述对话历史管理功能，实质上是对列表进行相应增删操作。对话历史信息，可以通过数据库进行存储与管理，并通过session_id进行区分。Langchain提供接口的数据库参考官方文档的Memory模块（如Elasticsearch、Kafka、Redis、MongoDB等），这里介绍streamlit中的对话历史管理功能：

```
from langchain_community.chat_message_histories import StreamlitChatMessageHistory
history = StreamlitChatMessageHistory(key="chat_messages")
```

StreamlitChatMessageHistory实质上是用来存储st.session_state中的信息，因此可以用在Streamlit APP运行时管理对话历史。当应用更新或页面刷新时，对话历史也会重置。

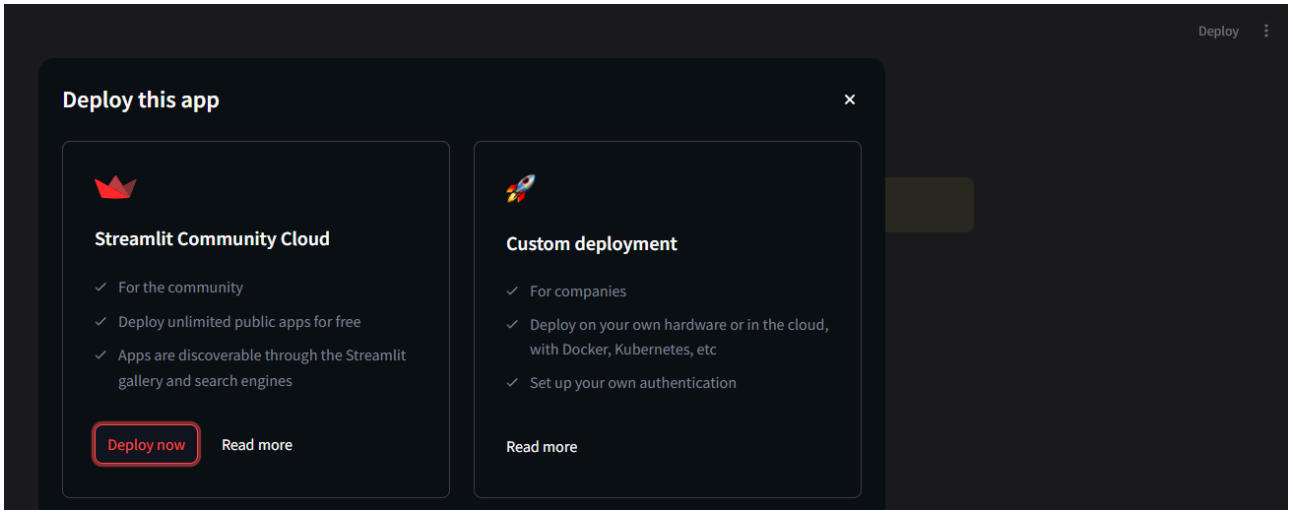
4.2 WebUI

保持以上的逻辑不变，即：1）使用init_chat_model接入Deepseek模型；2）使用StreamlitChatMessageHistory管理对话历史，并对最大对话轮数进行限制；3）使用st制作简单webui，即可得到webui代码（主要由cursorAI基于之前代码生成，并让其在最后进行封装、重构）。这里就不再复制完整代码了。

关于本地webui部署，只需在同目录下，命令行输入`streamlit run 02_webui_st.py`，即可运行程序。默认使用本地8501端口。

4.3 Deploy via Streamlit Cloud

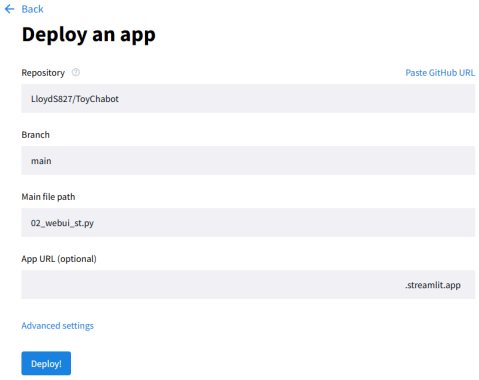
使用免费的Streamlit Cloud服务，即可将上述WebUI进行外网部署。在本地运行webui后，在右上角可以看到“deploy”字样，点击即可进入Cloud界面：



使用Streamlit Cloud部署，需要先将项目上传至Github，在Github建立好repo之后，需要的操作列举如下

```
git init
git remote add origin <http.git>
git branch -M main
git add .
git commit -m "<your commit message>"
git push -u origin main
# add -> commit -> push
```

需要注意的是项目中要包含requirements.txt文件，里面写入对应的依赖。然后切换到streamlit中的部署应用页面，填入对应的项目地址、分支、主程序等信息即可：



5 More Langchainic Way

> 本节对应代码文件：03_langchainic_chatbot.ipynb & 04_lc_st.py（https://github.com/LloydS827/ToyChabot）

以上的步骤完成了从大模型接入，到Webui开发/部署的全流程，但实现方式上只是接入了langchain的一些模块，并没有使用langchain的编程风格，而Langchain架构的核心就是所谓的Langchain Expression Language（LCEL）。

5.1 LCEL

基础的LCEL如下所示：

```
from langchain.chat_models import init_chat_model
import os
from dotenv import load_dotenv
load_dotenv()
```

```
llm = init_chat_model(
    model="deepseek-chat",
    model_provider="openai",
    api_key=os.environ.get('DS_API_KEY'),
    base_url="https://api.deepseek.com",
    temperature = 0.2,
)

from langchain.prompts import ChatPromptTemplate
from langchain.schema.output_parser import StrOutputParser

prompt_template = ChatPromptTemplate.from_messages(
    [
        ("system", "You are a comedian who tells jokes about {topic}."),
        ("human", "Tell me {joke_count} jokes in {language}."),
    ]
)
chain = prompt_template | llm | StrOutputParser()
result = chain.invoke({"topic": "dogs", "joke_count": 3, "language": "English"})
```

具体来说，LCEL是指这样的编程方式：`chain = prompt_template | llm | StrOutputParser()`，即用“|”连接起不同的模块，整体形成可以端到端运行的chain。 在本例中，chain由三个部分组成：1）prompt_template：本质上是对字符串str的一系列操作，并没有太复杂的内容；2）LLM：与之前相同，使用deepseek-chat模型；3）StrOutputParser：对结果做解析，这里只是把结果中的content拿出来，等同于`result.content`。

LCEL是langchain最核心的内容，甚至为了能够使不同模块，能够通过“|”链接起来，对易用性、实用性做了很多牺牲（这也是langchain被诟病的原因之一）。LCEL的实现依赖于runnable类，即将所有模块都用runnable类打包，再由runnable类实现LCEL操作。例如上面的例子中，使用底层的runnable类进行编写，则代码如下：

```
from langchain.schema.runnable import RunnableLambda, RunnableSequence
# Create individual runnables (steps in the chain)
format_prompt = RunnableLambda(lambda x: prompt_template.format_prompt(**x))
invoke_model = RunnableLambda(lambda x: llm.invoke(x.to_messages()))
parse_output = RunnableLambda(lambda x: x.content)

chain = RunnableSequence(first=format_prompt, middle=[invoke_model], last=parse_output)
```

上述代码的含义是，使用RunnableSequence将三个RunnableLambda进行顺序串联。可以想象的是，对于LCEL来说，最重要的是对齐不同组件的输入/输出。

在整体逻辑层面，除了RunnableSequence，还支持RunnableParallel(并行)，RunnableBranch(分支)等结构，这里不展开叙述了。另外RunnableWithMessageHistory（对话历史）能够将LLM与对话历史绑定，也是比较重要的方法（后文将使用）。

5.2 Langchainic Chatbot

第一步与之前相同，连接LLM：

```
from langchain.chat_models import init_chat_model
import os
from dotenv import load_dotenv
import warnings
warnings.filterwarnings("ignore")

load_dotenv()
llm = init_chat_model(
```

```
    model="deepseek-chat",
    model_provider="openai",
    api_key=os.environ.get('DS_API_KEY'),
    base_url="https://api.deepseek.com",
    temperature = 0.2,
)
```

第二步，创建prompt_template:

```
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_core.messages import SystemMessage

prompt = ChatPromptTemplate.from_messages(
    [
        SystemMessage(content="You are a helpful assistant. Answer all questions to the
best of your ability."),
        MessagesPlaceholder(variable_name="messages"),
    ]
)
```

第三步，将InMemoryChatMessageHistory中的对话内容进行管理，即给出seesion_id，返回对应session的对话历史：

```
from langchain_core.chat_history import BaseChatMessageHistory, InMemoryChatMessageHistory
store = {}
def get_message_history(session_id) -> BaseChatMessageHistory:
    if session_id not in store:
        store[session_id] = InMemoryChatMessageHistory()
    return store[session_id]

config = {"configurable": {"session_id": "a"}} # session_id is required
```

第四步，管理对话历史，这里使用trimmer方法，限制对话历史中的最大token数量。其中先使用tiktoken中的encoding方法计算对话历史的token数量，再通过trim_messages中的参数，如是否包含系统提示词，来限制对话历史。另外，langchain内置的对话历史管理方法还有filter（过滤信息）、merge（合并信息）：

```
from langchain_core.messages import trim_messages
import tiktoken

def count_tokens(messages):
    encoding = tiktoken.encoding_for_model("gpt-3.5-turbo")
    num_tokens = 0
    for message in messages:
        num_tokens += len(encoding.encode(message.content))
    return num_tokens

trimmer = trim_messages(
    max_tokens=40,
    strategy="last",
    token_counter=count_tokens,
    include_system=True,
    allow_partial=False,
    start_on="human",
)
```

第五步，使用LCEL将以上组件，串联成chain，其中trim行为要发生在prompt生成之前。创建好chain之后，可以通过RunnableWithMessageHistory方法，将chain与对话历史绑定：

```python
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables.history import RunnableWithMessageHistory

chain = trimmer | prompt | llm | StrOutputParser()

with_message_history = RunnableWithMessageHistory(chain, get_message_history)

config = {"configurable": {"session_id": "a"}}
```

可以说，经过这样以LCEL为核心的改写之后，已经非常符合langchain的编程、设计风格了。但这样的方式是否满足实际的开发需求、是否易于理解、是否降低了应用开发难度，仍然存在很多争议。但可以确定的是，在Langchain的V1.0版本中，上述这些问题一定会大幅改进。

6 TL;DR

1. 推荐使用miniconda管理环境，poetry/pip管理依赖，cursorAI/vscode/jupyter作为IDE
2. 可以使用同一行代码，接入各种来源的模型
3. 对话历史、提示词，本质上都是对基本数据结构的操作，例如str、dict、list等
4. 可以使用streamlit快速创建llm webui，并通过streamlit cloud进行部署
5. Langchain的核心是LCEL，主要通过runnable类来实现

7 Ref

All Code: https://github.com/LloydS827/ToyChabot

Our Course Repo: https://github.com/LloydS827/WhysAI-LLMFS-4E

Highly Recommended: LangChain Master Class For Beginners 2024 [+20 Examples, LangChain V0.2] - YouTube

Highly Recommended: Langchain official tutorial

Conceptual guide of Langchain

Langchain + Streamlit: streamlit_agent/basic_memory.py

Streamlit: Build an LLM app using LangChain - Streamlit Docs

Memory Module in LangChain

Python Intro Course by Andrew: AI Python for Beginners - DeepLearning.AI