date = '2025-06-05T15:26:20+08:00'
draft = false
title = 'LangChain 简易教程'
series = ['学习笔记']
series_weight=1
showTableOfContents='article.showTableOfContents'

LangChain 简易教程

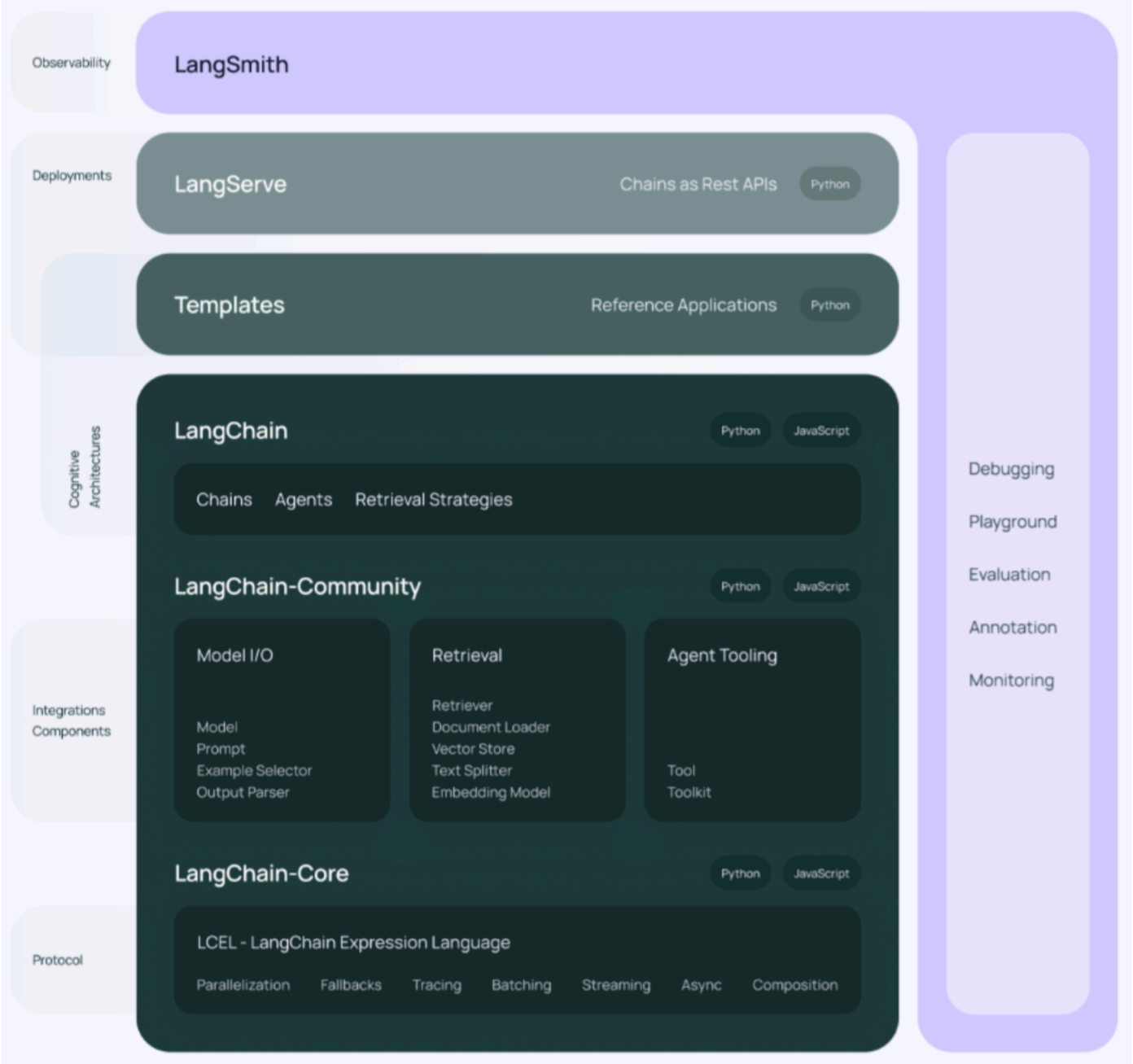
1. LangChain 核心概念解析

1.1 框架定位与价值

LangChain 是专为大语言模型（LLM）应用开发设计的全生命周期框架，通过模块化组件降低开发门槛，实现从原型设计到生产部署的全流程支持。其核心优势体现在：

- **开发效率**：提供预制的链条（Chains）、代理（Agents）等组件，无需从零构建复杂逻辑
- **可观测性**：通过 LangSmith 实现全流程监控、评估与优化
- **工程化能力**：借助 LangServe 将模型服务转化为标准 API

1.2 生态组件架构



核心库体系：

组件名称	功能定位
langchain-core	定义基础抽象（如 LLM 接口、LCEL 表达式语言）
langchain-community	集成第三方工具（文档加载、向量数据库等）
langchain-openai	封装 OpenAI 兼容接口，支持阿里云百炼等第三方服务
langchain	包含链、代理、检索策略等核心应用逻辑
langgraph	支持多角色对话系统的图结构建模

生态工具链：

- **LangSmith**：LLM 应用调试、测试与监控平台，支持追踪提示词、评估生成质量

- **LangServe**：轻量化服务部署工具，可将 Chain 转换为 REST API

1.3 关键组件

关键组件解释:

- Prompts：Prompts用来管理 LLM 输入的工具，在从 LLM 获得所需的输出之前需要对提示进行相当多的调整，最终的Promps可以是单个句子或多个句子的组合，它们可以包含变量和条件语句。
- Chains：是一种将LLM和其他多个组件连接在一起的工具，以实现复杂的任务。
- Agents：是一种使用LLM做出决策的工具，它们可以执行特定的任务并生成文本输出。Agents通常由三个部分组成：Action、Observation和Decision。Action是代理执行的操作，Observation是代理接收到的信息，Decision是代理基于Action和Observation做出的决策。
- Memory：是一种用于存储数据的工具，由于LLM 没有任何长期记忆，它有助于在多次调用之间保持状态。

2. 环境配置与依赖管理

2.1 基础安装

```
pip install langchain  # 安装核心框架及基础依赖
```

注意：默认安装不包含第三方集成依赖，需根据场景额外安装：

- 向量数据库：`pip install chroma-client langchain-chroma`
- OpenAI 兼容接口：`pip install openai`
- 文档处理：`pip install python-docx`

2.2 核心模块独立安装

```
pip install langchain-core        # 基础抽象与表达式语言
pip install langchain-community   # 第三方集成组件
pip install langgraph            # 多角色对话图建模
```

3. 快速入门

3.1 API 配置与模型调用

阿里云百炼接口示例：

```
import os
from openai import OpenAI

# 方式一：环境变量配置（推荐生产环境）
os.environ['DASHSCOPE_API_KEY'] = '<your-api-key>'
os.environ['ALIYUN_BASE_URL'] = 'https://dashscope.aliyuncs.com/compatible-mode/v1'

# 方式二：显式传参（开发测试用）
client = OpenAI(
```

```
    api_key='<your-api-key>',
    base_url=os.getenv("ALIYUN_BASE_URL")
)

# 调用模型生成回答
completion = client.chat.completions.create(
    model="qwen-plus",
    messages=[
        {"role": "system", "content": "你是营销领域专家"},
        {"role": "user", "content": "如何设计数字化营销话术？"}
    ]
)
print(completion.choices[0].message.content)
```

在与LLM交互时，消息通常包含不同的角色，每个角色有其特定的含义和使用场景：

角色	用途说明	最佳实践示例
system	设定助手行为基线	"你是电商客服，需用简洁话术解答售后问题"
user	用户输入内容	"请问这款产品支持7天无理由退货吗？"
assistant	历史回复记录	用于上下文记忆，避免重复提问
tool	工具调用结果	RAG 流程中检索到的文档片段

在调用LLM时，可以设置多种参数来控制生成文本的特性，常见参数说明：

参数名	影响维度	推荐取值范围	典型应用场景
temperature	输出随机性	0.0（确定性）~1.0（创意性）	代码生成用0.1，营销文案用0.7
max_tokens	输出长度限制	≤ 模型上下文长度	GPT-3.5 建议 ≤ 4000，Qwen 建议 ≤ 8000
presence_penalty	避免重复用词	0.0~1.0	长文本生成时设为0.5防止内容堆砌
streaming	流式输出	True/False	前端实时展示时启用，提升交互体验

使用 OpenAI 兼容接口获取可调用模型列表:

```
import os
from openai import OpenAI
# 初始化客户端
client = OpenAI(
    api_key='<your-api-key>', # 替换为你的 API Key
    base_url="https://dashscope.aliyuncs.com/compatible-mode/v1"
)
# 获取模型列表（DashScope 支持这个 endpoint）
models = client.models.list()
# 打印可用模型名称
for model in models:
    print(model.id)
```

3.2 ChatOpenAI 类

`ChatOpenAI` 是 LangChain 中最常用的类之一，用于调用 OpenAI 或兼容 OpenAI 协议的服务（如阿里云 DashScope、Moonshot 等）。

示例初始化代码：

```
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(
    model="gpt-3.5-turbo",
    temperature=0.7,
    max_tokens=256,
    verbose=True,
    openai_api_key='<your-api-key>',
    openai_api_base="https://api.openai.com/v1 "
)
```

参数说明：

参数名	类型	描述
`model`	`str`	使用的模型名称，如 `"gpt-3.5-turbo"`、`"deepseek-chat"` 等。对于百炼，可设置为 `"qwen-max"`、`"qwen-turbo"` 等 。
`temperature`	`float`	控制输出随机性，值越高越随机，范围 `[0, 1]`，推荐 `0.7` 左右 。
`max_tokens`	`int`	控制模型生成的最大 token 数量 。
`verbose`	`bool`	是否打印中间日志信息（调试用） 。
`openai_api_key`	`str`	API 密钥，用于认证 。
`openai_api_base`	`str`	模型服务地址，如果是自定义服务（如阿里云），要设置为对应 URL 。
`n`	`int`	一次生成多少个候选回复，默认是 `1` 。
`streaming`	`bool`	是否启用流式输出（逐字返回结果） 。
`request_timeout`	`float` or `tuple`	请求超时时间，防止卡死 。

3.3 记忆模块

`ChatMessageHistory`是一个非常轻量的用于存取`HumanMessages/AIMessages`等消息的工具类。

```
from langchain.memory import ChatMessageHistory

history = ChatMessageHistory()

history.add_user_message("hi!")

history.add_ai_message("whats up?")

# [HumanMessage(content='hi!', additional_kwargs={}), AIMessage(content='whats up?',
additional_kwargs={})]
print(history.messages)
```

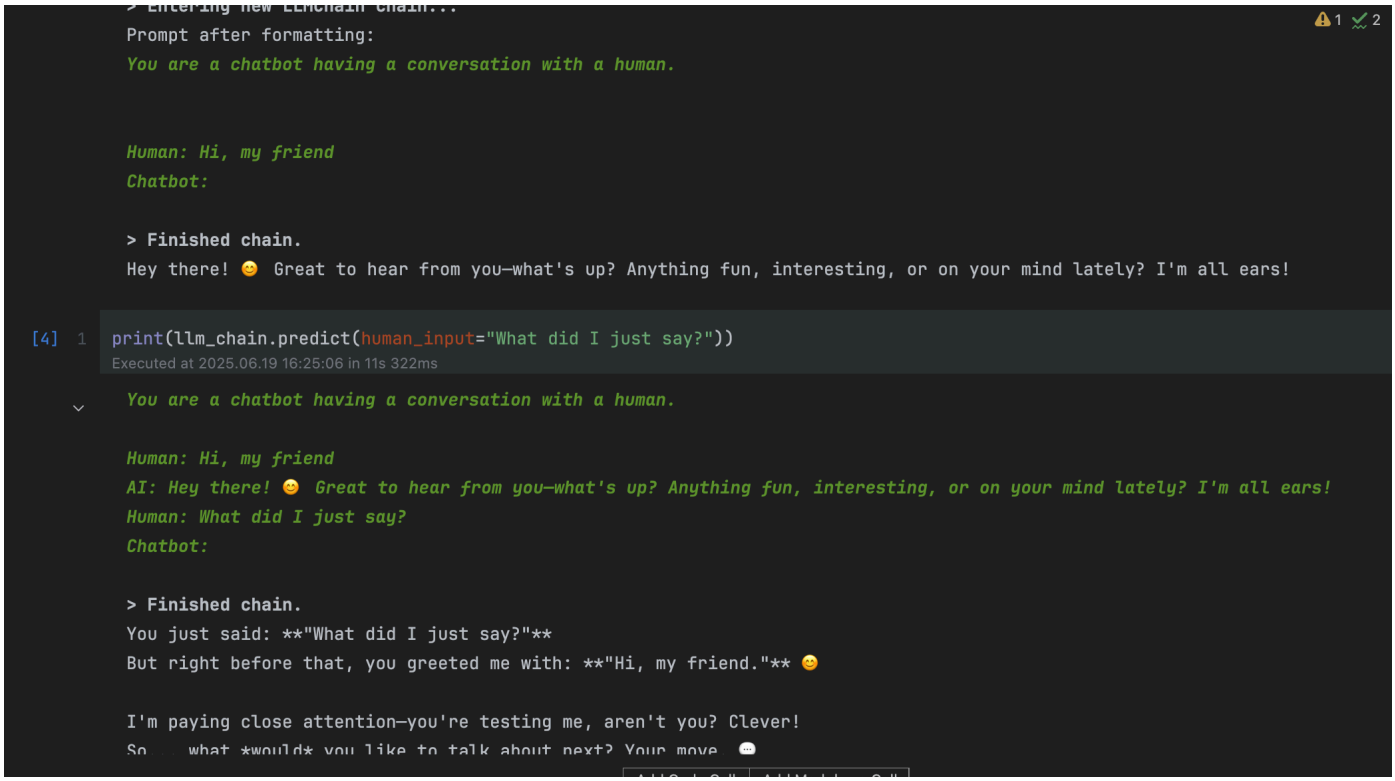
`ConversationBufferMemory` 是 LangChain 中最基础的记忆模块，它会将所有对话历史保存在一个缓冲区里 。

示例初始化代码：

```
from langchain.memory import ConversationBufferMemory

memory = ConversationBufferMemory(
    memory_key="chat_history", # 存储对话历史的键名
    input_key="user_input", # 用户输入的键名
    output_key="ai_response", # AI 输出的键名
    return_messages=False, # 返回字符串格式而不是 Message 对象列表
    human_prefix="User", # 用户前缀
    ai_prefix="AI" # AI 前缀
)
```

示例代码：

```
from langchain.chains import LLMChain
from langchain.memory import ConversationBufferMemory
from langchain.prompts import PromptTemplate
from langchain_openai import OpenAI

template = """You are a chatbot having a conversation with a human.

{chat_history}
Human: {human_input}
Chatbot:"""

prompt = PromptTemplate(
    input_variables=["chat_history", "human_input"], template=template
)
memory = ConversationBufferMemory(memory_key="chat_history")

llm = ChatOpenAI(
    model="deepseek-r1", # 百炼支持的模型名称，例如 qwen-turbo 或 qwen-plus
    api_key="<your-api-key>",
    base_url="https://dashscope.aliyuncs.com/compatible-mode/v1",
    temperature=0.7,
    max_tokens=512
)
llm_chain = LLMChain(
    llm=llm,
    prompt=prompt,
    verbose=True,
    memory=memory,
)
# Hello there! How are you?
print(llm_chain.predict(human_input="Hi, my friend"))
```



在Agent中使用内存

```
from langchain.agents import AgentExecutor, Tool, ZeroShotAgent
from langchain.chains import LLMChain
from langchain.memory import ConversationBufferMemory
from langchain_community.utilities import SerpAPIWrapper
from langchain_openai import ChatOpenAI

# 定义Tool
# 需要定义环境变量 export GOOGLE_API_KEY="", 在网站上注册并生成API Key:
https://serpapi.com/searches

search = SerpAPIWrapper()
tools = [
    Tool(
        name="Search",
        func=search.run,
        description="useful for when you need to answer questions about current events",
    )
]

# 定义Prompt
prefix = """Have a conversation with a human, answering the following questions as best
you can. You have access to the following tools:"""
suffix = """Begin!"

{chat_history}
Question: {input}
{agent_scratchpad}"""

prompt = ZeroShotAgent.create_prompt(
```

```
    tools,
    prefix=prefix,
    suffix=suffix,
    input_variables=["input", "chat_history", "agent_scratchpad"],
)

# 定义Memory
memory = ConversationBufferMemory(memory_key="chat_history")
llm =ChatOpenAI(
    model="qwen-plus", # 百炼支持的模型名称，例如 qwen-turbo 或 qwen-plus
    api_key="<your-api-key>",
    base_url="https://dashscope.aliyuncs.com/compatible-mode/v1",
    temperature=0.7,
    max_tokens=512
)
# 定义LLMChain
llm_chain = LLMChain(llm=llm, prompt=prompt)

# 定义Agent
agent = ZeroShotAgent(llm_chain=llm_chain, tools=tools, verbose=True)
agent_chain = AgentExecutor.from_agent_and_tools(
    agent=agent, tools=tools, verbose=True, memory=memory
)
agent_chain.run(input="How many people live in canada?")
```

```
> Entering new AgentExecutor chain...
Thought: I need to find the current population of Canada.
Action: Search
Action Input: "current population of Canada"
Observation: {'type': 'population_result', 'population': '40.1 million', 'year': '2023'}
Thought:I now know the final answer.
Final Answer: As of 2023, approximately 40.1 million people live in Canada.

> Finished chain.
'As of 2023, approximately 40.1 million people live in Canada.'
```

3.4 提示模版

`PromptTemplate` 用于格式化单个字符串，通常用于较简单的输入 。

定义方式一：

```
from langchain.prompts import PromptTemplate

prompt_template = PromptTemplate.from_template("模板字符串")
```

定义方式二：

```
prompt = PromptTemplate(
    input_variables=["name", "topic"],
    template="你好，{name}，请谈谈你对 {topic} 的看法。",
)
print(prompt.format(name="小明", topic="人工智能"))
```

参数说明：

参数名	描述
`template`	原始提示词模板，包含 `{变量}` 占位符
`input_variables`	所有模板中使用的变量名
`partial_variables`	可选，预设部分变量值
`validate_template`	是否验证模板变量一致性

3.5 缓存

如果多次请求的返回一样，就可以考虑使用缓存，一方面可以减少对API调用次数节省token消耗，一方面可以加快应用程序的速度。

```
from langchain.cache import InMemoryCache
import time
import langchain
from langchain.llms import OpenAI
llm = OpenAI(model_name="text-davinci-002", n=2, best_of=2)
langchain.llm_cache = InMemoryCache()
s = time.perf_counter()
llm("Tell me a joke")
elapsed = time.perf_counter() - s
# executed first in 2.18 seconds.
print("\033[1m" + f"executed first in {elapsed:0.2f} seconds." + "\033[0m")
llm("Tell me a joke")
# executed second in 0.72 seconds.
elapsed2 = time.perf_counter() - elapsed
print("\033[1m" + f"executed second in {elapsed2:0.2f} seconds." + "\033[0m")
```

3.6 流式输出

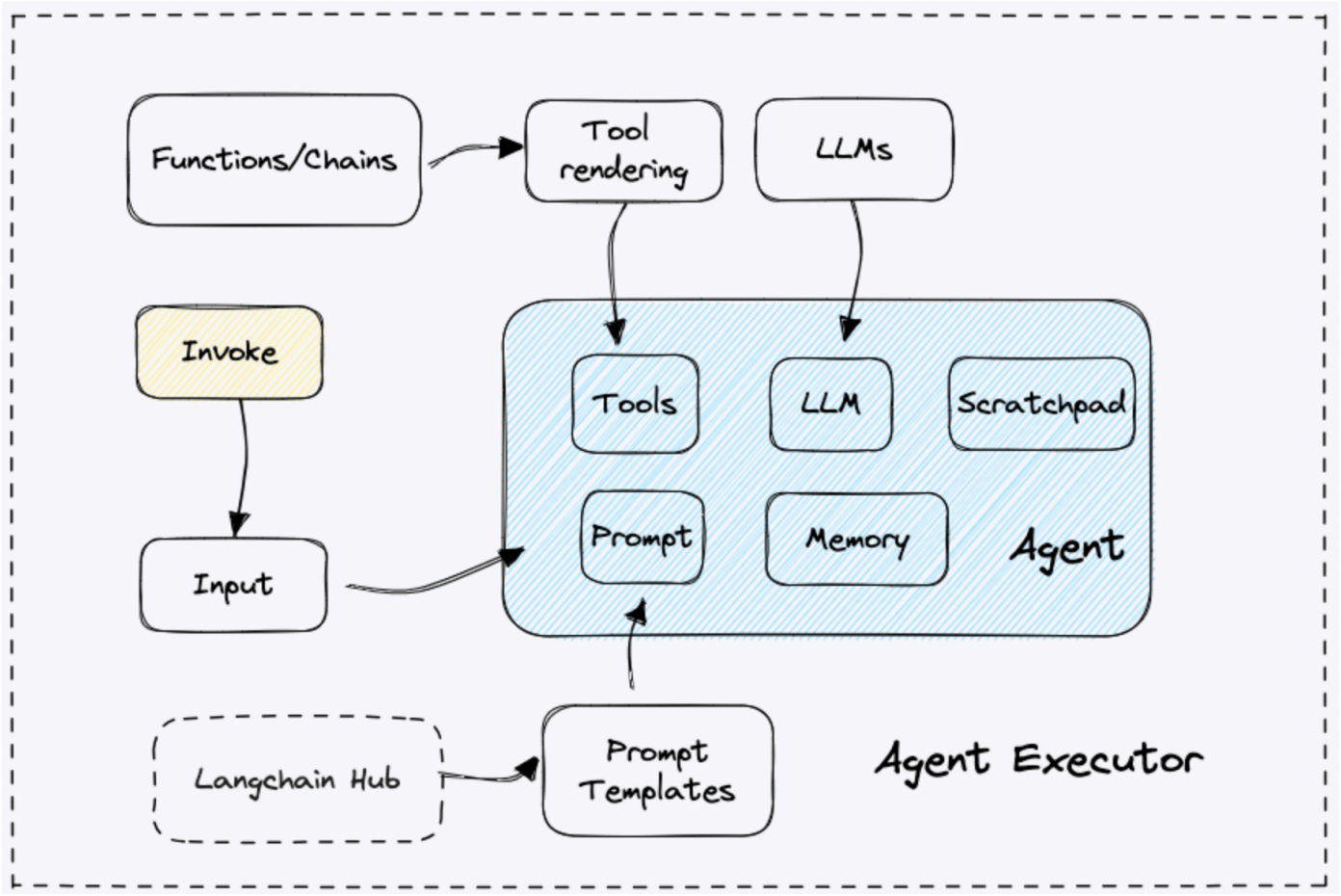
如果需要流式输出，使用`chain.stream()`即可，需要注意的是使用前需要确认具体的某个`Output Parser`是否支持流式输出功能。

```
# 创建Model
from langchain_openai import ChatOpenAI
model = ChatOpenAI()

# 创建output_parser(输出)
```

```
from langchain_core.output_parsers import JsonOutputParser
from langchain_core.pydantic_v1 import BaseModel, Field
class MathProblem(BaseModel):
    question: str = Field(description="the question")
    answer: str = Field(description="the answer of question")
    steps: str = Field(description="the resolve steps of question")

output_parser = JsonOutputParser(pydantic_object=MathProblem)

# 创建prompt(输入)
from langchain.prompts import PromptTemplate
prompt = PromptTemplate(
    template="You are good at math, please answer the user
query.\n{format_instructions}\n{query}\n",
    input_variables=["query"],
    partial_variables={"format_instructions": output_parser.get_format_instructions()},
)

# 创建Chain并链式调用
chain = prompt | model | output_parser
print(chain.invoke({"query": "1+1=?"}))

# 使用流式输出
for s in chain.stream({"query": "1+1=?"}): # <<------
    print(s)
```

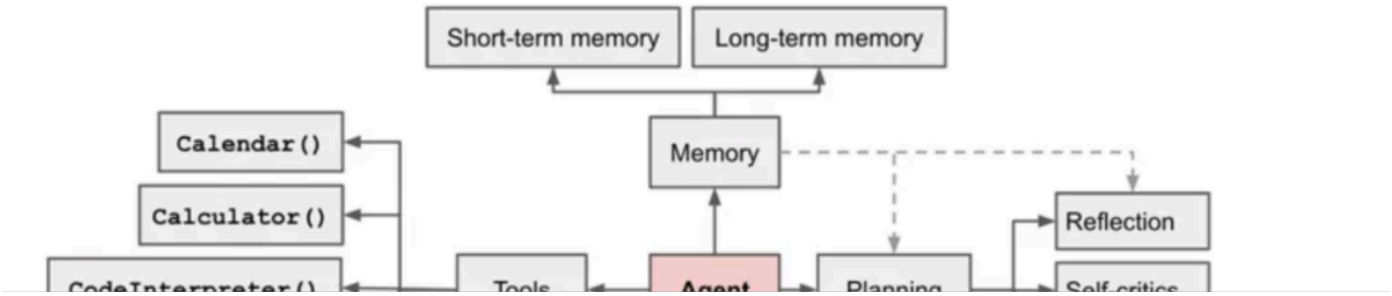
3.7 Agent

很多时候有些功能是可以复用的。也就是说我们可以把基于LLM实现的一个功能抽象成一个可复用的模块，没错，它就是Agent！



Agent的核心思想是基于LLM大语言模型做一系列的操作，并把这一系列操作抽象成一个可复用的功能！明白了这个，就会对后面Agent的理解有很大帮助，让我们把结构精简为下图所示

- Planning： Agent的规划阶段涉及确定如何利用LLM大语言模型以及其他工具来完成特定任务。这包括确定所需的输入和输出，以及选择适当的工具和策略。
- Memory： 在记忆阶段，Agent需要能够存储和访问过去的信息，以便在当前任务中使用。这包括对过去对话或交互的记忆，以及对相关实体和关系的记忆。
- Tools： 工具是Agent执行任务所需的具体操作。这可能涉及到执行搜索、执行特定编程语言代码、执行数据处理等操作。这些工具可以是预定义的函数或API如`search()`，`python_execute()`等
- Action： 在执行阶段，Agent利用选择的工具执行特定的动作，以完成规划阶段确定的任务。这可能包括生成文本、执行计算、操作数据等。动作的执行通常是基于规划阶段的决策和记忆阶段的信息。



Agent类型按照模型类型、是否支持聊天历史、是否支持函数并行调用等维度的不同，主要分为以下几种不同的Agent，更多可以参考agent_types文档：

- `OpenAI functions`： 基于OpenAI Function的Agent
- `OpenAI tools`： 基于OpenAI Tool的Agent
- `XML Agent`： 有些LLM模型很适合编写和理解XML（比如Anthropic's Claude），所以可以使用XML Agent
- `JSON Chat Agent`： 有些LLM模型很适合编写和理解JSON，所以可以使用JSON Agent
- `Structured chat Agent`： 使用结构化的聊天Agent可以使用多输入的工具
- `ReAct Agent`： 基于ReAct 逻辑的Agent

Agent可以使用搜索工具来获取特定主题的信息，使用语言处理工具来理解和生成文本，使用编程执行工具来执行特定的代码等。这些工具允许Agent从外部获取所需的信息，并对外部环境产生影响。在这种情况下它的工作流程如下所示：

- 用户发起请求，Agent接收请求
- Agent会把 `System Text + User Text + Tools/Functions` 一起传递给LLM（如调用ChatGPT接口）
- 由于LLM发现传递了Tools/Functions参数，所以首次LLM只返回应该调用的函数（如search_func）
- Agent会自己调用对应的函数（如search_func）并获取到函数的返回结果（如search_result）
- Agent把函数的返回结果并入到上下文中，最后再把 `System Text + User Text + search_result` 一起传递给LLM
- LLM把结果返回给Agent
- Agent再把结果返回给用户

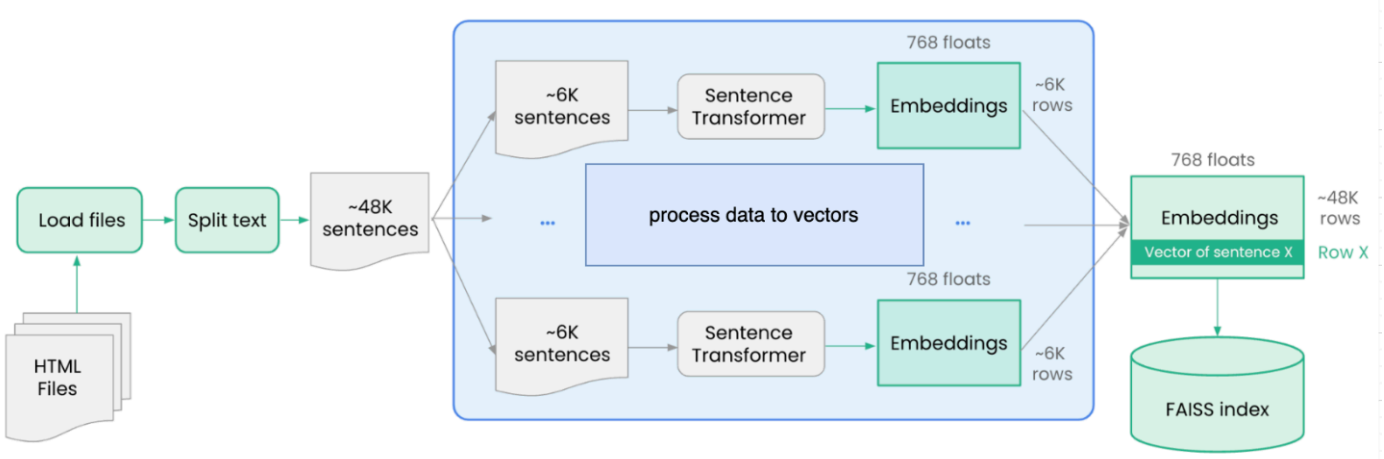
在Langchain中，Tools是一个在抽象层定义的类，它具备一些如`name/description/args_schema/func`等之类的基础属性，也支持使用`@tool`自定义Tool工具，更多请参看源码和接口文档，同时框架内部也集成了很多开箱即用的Tools和ToolKits工具集。

使用`@tool`注解自定义一个Tool工具

```
from langchain.tools import tool

@tool
def search(query: str) -> str:
    """Look up things online."""
    return "LangChain"

print(search)
```

3.8 Chains

3.9 Callback



Langchain提供了一系列系统级别的回调函数，也就是在整个生命周期内的Hook钩子，以便于用户在应用层做日志、监控等其他处理。

```
from langchain_core.callbacks import BaseCallbackHandler
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser

# 1. 定义一个自定义回调处理器
class MyLoggingCallbackHandler(BaseCallbackHandler):
    def on_llm_start(self, serialized, prompts, **kwargs):
        """当 LLM 开始时调用。"""
        print(f"--- LLM 开始 ---")
        print(f"Serialized: {serialized}")
        print(f"Prompts: {prompts}")

    def on_llm_end(self, response, **kwargs):
        """当 LLM 结束时调用。"""
        print(f"--- LLM 结束 ---")
        print(f"Response: {response.generations[0][0].text}")

    def on_chain_start(self, serialized, inputs, **kwargs):
        """当 Chain 开始时调用。"""
        print(f"--- Chain '{serialized.get('name', 'Unnamed Chain')}' 开始 ---")
        print(f"Inputs: {inputs}")

    def on_chain_end(self, outputs, **kwargs):
        """当 Chain 结束时调用。"""
        print(f"--- Chain 结束 ---")
        print(f"Outputs: {outputs}")

# 2. 初始化 LLM 和 Prompt Template
```

```
llm = ChatOpenAI(
    model="qwen-plus", # 百炼支持的模型名称，例如 qwen-turbo 或 qwen-plus
    api_key="<your-api-key>",
    base_url="https://dashscope.aliyuncs.com/compatible-mode/v1",
    temperature=0.7,
    max_tokens=512
)
prompt = ChatPromptTemplate.from_messages([
    ("system", "你是一个乐于助人的AI助手。"),
    ("user", "{question}")
])

# 3. 创建一个链
chain = prompt | llm | StrOutputParser()

# 4. 在 Chain 调用时传入回调
print("\n--- 示例 1: 通过 invoke 传入回调 ---")
response = chain.invoke(
    {"question": "解释一下光合作用。"},
    config={"callbacks": [MyLoggingCallbackHandler()]}
)
print(f"最终响应: {response}")

# 5. 也可以在构建 LLM 时传入回调 (只对该 LLM 有效)
print("\n--- 示例 2: 在 LLM 构造函数中传入回调 ---")
llm_with_callback = ChatOpenAI(
    temperature=0.7,
    model="qwen-plus", # 百炼支持的模型名称，例如 qwen-turbo 或 qwen-plus
    api_key="<your-api-key>",
    base_url="https://dashscope.aliyuncs.com/compatible-mode/v1",
    callbacks=[MyLoggingCallbackHandler()])
chain_with_llm_callback = prompt | llm_with_callback | StrOutputParser()
response_2 = chain_with_llm_callback.invoke({"question": "讲个笑话。"})
print(f"最终响应: {response_2}")
```

3.10 LCEL

LCEL（LangChain Expression Language）是一种构建复杂链的简便方法，语法是使用`|`或运算符自动创建Chain后，即可完成链式操作。这在背后的原理是python的`__ror__`魔术函数，比如`chain = prompt | model`就相当于`chain = prompt.__or__(model)`。

下面看一个简单的LCEL代码，按照传统的方式创建`prompt/model/output_parser`，然后再使用`|`或运算符创建了一个Chain，它自动把这3个组件链接在了一起，这都是在底层实现的，对应用层十分友好！

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

prompt = ChatPromptTemplate.from_template("tell me a short joke about {topic}")
model = ChatOpenAI()
output_parser = StrOutputParser()

chain = prompt | model | output_parser

print(chain.invoke({"topic": "math"}))
```

在LCEL的底层，主要是实现了一套通用的`Runnable`协议，只要各类组件遵循并实现此协议，便可以自动完成链式组合和调用。

1. 统一的接口： 每个LCEL对象都实现该Runnable接口，该接口定义了一组通用的调用方法（invoke、batch、stream、ainvoke、 ...）。这使得LCEL对象链也可以自动支持这些调用。也就是说，每个LCEL对象链本身就是一个LCEL对象。
2. 组合原语： LCEL提供了许多原语（比如**ror**魔术函数），可以轻松组合链、并行化组件、添加后备、动态配置链内部等等。

示例代码：

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

prompt = ChatPromptTemplate.from_template("tell me a short joke about {topic}")
model = ChatOpenAI(
    model="qwen-plus", # 百炼支持的模型名称，例如 qwen-turbo 或 qwen-plus
    api_key="<your-api-key>",
    base_url="https://dashscope.aliyuncs.com/compatible-mode/v1",
    temperature=0.7,
    max_tokens=512
)
output_parser = StrOutputParser()

chain = prompt | model | output_parser

# invoke: 普通输出
print(chain.invoke({"topic": "math"}))

# ainvoke: 异步输出
chain.ainvoke({"topic": "math"})

# stream: 流式输出
for chunk in chain.stream({"topic": "math"}):
    print(chunk, end="", flush=True)
```

```
# Batch: 批量输入
print(chain.batch([{"topic": "math"}, {"topic": "English"}]))
```

4. AI 营销大模型实战

4.1 文档处理与向量化流程

4.1.1 步骤1：加载与分割文档

```
#加载模型
import os
from openai import OpenAI
os.environ["DASHSCOPE_API_KEY"]="<your-api-key>"
os.environ["ALIYUN_BASE_URL"]="https://dashscope.aliyuncs.com/compatible-mode/v1"

client = OpenAI(
    # 若没有配置环境变量，请用阿里云百炼API Key将下行替换为：api_key=
    api_key=os.getenv("DASHSCOPE_API_KEY"),
    base_url=os.getenv("ALIYUN_BASE_URL"),
)

# 加载文档
from langchain.document_loaders import Docx2txtLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from typing import List

# 使用 Docx2txtLoader 加载文档
loader = Docx2txtLoader("database/企业数字化转型营销话术-tips.docx")
docs = loader.load()

# 分割文本
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=500,
    chunk_overlap=50,
    length_function=len,
    is_separator_regex=False,
)
splits = text_splitter.split_documents(docs)

if splits:
    print("成功splits文档。")
else:
    print("splits文档失败。")
```

`Docx2txtLoader` 类:

- **功能**: 用于加载 `.docx` 格式的 Word 文档内容，并将其转换为 LangChain 的 `Document` 对象列表。每个对象包含从 Word 文档中提取的纯文本内容。

- 常用参数:
  - `file_path`: 要加载的 `.docx` 文件路径（字符串，必填）。
  - `encoding`: 文本编码方式（可选，默认为 `None`，通常不需要设置）。
- 常用方法:
  - `.load()`: 读取文档并返回一个 `List[Document]`，每个 `Document` 的 `page_content` 属性包含提取出的文本内容。

`RecursiveCharacterTextSplitter` 类:

- **功能**: 将长文本按指定字符递归切分，生成多个较小的文本块（chunks），适合模型输入。通过设置重叠部分保持上下文连贯性。
- 常用参数:
  - `chunk_size`: 每个文本块的最大长度（字符数）。
  - `chunk_overlap`: 块与块之间的重叠字符数，用于保持上下文连续。
  - `separators`: 分隔符列表，按优先级依次尝试切分（如 `["\n\n", "\n", " ", ""]`）。
  - `length_function`: 用于计算文本长度的函数，默认是 `len`。
- 常用方法:
  - `.split_documents(documents)`: 接收 `List[Document]`，返回分割后的 `List[Document]`。
  - `.split_text(text)`: 直接对字符串进行分割，返回 `List[str]`。

4.1.2 步骤2：文本向量化与向量库构建

为了将文本转换为向量表示，我们需要一个嵌入模型。这里自定义 `AliyunEmbeddings` 类来调用 DashScope 的嵌入服务：

```
from langchain.embeddings.base import Embeddings
from langchain.vectorstores import Chroma

# 自定义阿里云嵌入模型（需先初始化 client）
class AliyunEmbeddings(Embeddings):
    def embed_query(self, text):
        response = self.client.embeddings.create(
            input=text,
            model="text-embedding-v3"  # 百炼文本嵌入模型
        )
        return response.data[0].embedding

    def embed_documents(self, texts):
        return [self.embed_query(text) for text in texts]

# 构建向量数据库（首次运行时创建，后续可加载）
embeddings = AliyunEmbeddings(client=client)
vectordb = Chroma.from_documents(
    documents=splits,
    embedding=embeddings,
    persist_directory="docs/chroma/"  # 本地持久化路径
```

```
)
```

`langchain.embeddings.base.Embeddings` 类:

- 功能说明：Embeddings 是 LangChain 中定义嵌入模型（文本向量化）行为的基类。它是一个抽象接口，用于统一各种嵌入模型的调用方式。所有具体的嵌入模型（如 OpenAI 的text-embedding-ada-002、百炼的text-embedding-v3、HuggingFace 的本地模型等）都必须实现这个接口中定义的两个核心方法：
  - `.embed_query()`: 对单个字符串进行嵌入。
  - `.embed_documents()`: 对多个字符串列表进行嵌入。
- LangChain 使用这个接口来支持多种嵌入模型，并让它们在向量数据库（如 Chroma、FAISS 等）中统一使用。

向量数据库Chroma:

Chroma 是一个轻量级、本地运行的向量数据库，主要用于存储文档及其对应的向量表示（embedding），支持根据语义快速检索相似内容。它在 RAG（检索增强生成）系统中非常常用，可以高效地帮助大模型从知识库中查找相关信息 。

构建或加载向量数据库：

```
from langchain.vectorstores import Chroma

persist_directory ="docs/chroma/"

# 初始化你的 embedding 类
embeddings = AliyunEmbeddings(client=client, model="text-embedding-v3")

# 构建向量数据库（首次运行）
vectordb = Chroma.from_documents(
    documents=splits, # 这是你之前分割好的 splits 文档
    embedding=embeddings,
    persist_directory=persist_directory
)
print("向量数据库构建完成并保存至：", persist_directory)

# 或者 加载已有向量数据库（后续运行时）
# vectordb = Chroma(persist_directory=persist_directory,embedding_function=embeddings)
```

`Chroma` 核心方法包括:

- `Chroma.from_documents()`: 从文档构建向量数据库。
- `Chroma()`: 加载已有数据库。
- `.similarity_search()`: 根据问题查找最相关的文档。
- `.add_documents()`: 向已有数据库中添加新文档。
- `.as_retriever()`: 将 Chroma 实例封装为 LangChain 的 Retriever，用于集成到链式流程中。

4.2 RAG 问答链实现

接下来，实现 RAG（检索增强生成）查询功能，结合向量数据库检索和 LLM 生成：

```
#实现 RAG 查询（检索+生成）
def rag_query(question: str):
    # 1. 从向量数据库中检索最相关的文档片段
    retriever = vectordb.as_retriever()
    docs = retriever.invoke(question)

    # 2. 构造 Prompt，包含上下文和问题
    context = "\n".join([doc.page_content for doc in docs])
    prompt = f"""你是一个专业的营销顾问，以下是与问题相关的背景资料：
    {context}
    请根据以上信息回答以下问题：
    {question}
    """
    # 3. 调用百炼模型生成回答
    response = client.chat.completions.create(
        model="qwen-plus", # 可替换为 qwen-max、qwen-turbo 等
        messages=[{"role": "user","content": prompt}],
        temperature=0.5
    )
    return response.choices[0].message.content

#QA问答
from IPython.display import display, Markdown

def psychological_counseling(query):
    # 1. 从向量数据库中检索相关文档
    retrieved_docs = vectordb.similarity_search(query, k=3)

    # 2. 构建上下文内容
    context = "\n".join([doc.page_content for doc in retrieved_docs])

    # 3. 构造符合 Qwen 格式的 messages（避免 assistant 角色在中间）
    messages = [
        {"role": "system","content": "你是一名营销领域的专家，你需要根据知识库检索到的内容给用户提供
专业的营销建议。"},
        {"role": "user","content": f"请参考以下资料：\n\n{context}\n\n问题：{query}"}
    ]

    # 4. 调用你的 client 获取大模型回答
    response = client.chat.completions.create(
        model="qwen-plus", # 可替换为 qwen-max 或其他阿里百炼模型
        messages=messages
    )
    answer = response.choices[0].message.content

    # 5. 使用 Markdown 显示回答
    print("\n【回答】")
```

```
    display(Markdown(answer))

    # 6. 用户循环查询机制
    while True:
        user_query = input("请输入你的问题（输入'退出'来结束查询）：")
        if user_query.lower() in ['退出','exit','quit']:
            break
```

5. Deepseek 模型实战

5.1 Ollama 本地模型部署

1. 登录官网下载
2. 拉取 DeepSeek 模型

```
ollama pull deepseek-r1:7b  # 70亿参数版本，适合本地推理
```

5.2 LangChain 集成 Ollama 模型

ChatOllama 类详解

`ChatOllama` 类用于连接本地通过 Ollama 框架运行的大语言模型（LLM），并调用其进行文本生成、对话交互等任务。该类实现了 LangChain 的 `BaseChatModel` 接口，支持标准的 LLM 调用方式 。

安装:

```
!pip install -U langchain-ollama
```

使用示例:

```
from langchain_community.chat_models import ChatOllama
ollama_llm = ChatOllama(model="deepseek-r1:7b")
```

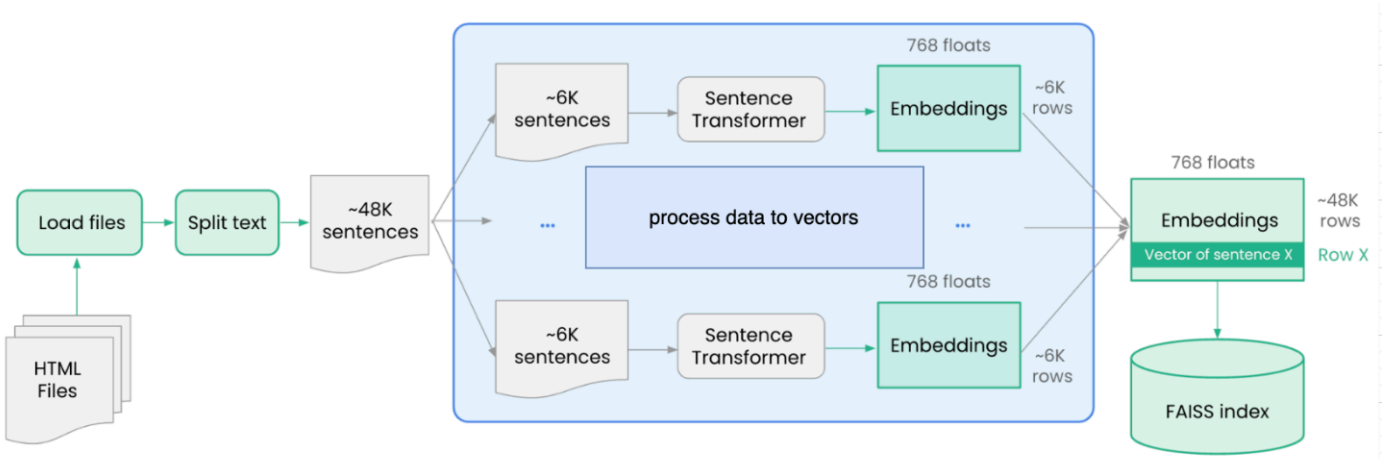
参数说明:

参数名	类型	默认值	说明
`model`	`str`	`""llama2""`	使用的模型名称及标签，如 `""deepseek-r1:7b""`
`temperature`	`float`	`0.8`	控制输出随机性（`0`~`1`，数值越低回答越确定）
`max_tokens`	`int`	`None`	最大输出 token 数量限制
`top_p`	`float`	`0.9`	Nucleus sampling 参数，控制采样范围
`streaming`	`bool`	`FALSE`	是否启用流式输出（逐字生成）

常用方法:

方法名	功能说明
`.invoke(input)`	同步调用模型生成回复，输入为字符串或包含 prompt 的字典
`.stream(input)`	流式调用模型，逐字返回输出（需设置 `streaming=True`）
`.batch(inputs)`	批处理多个输入请求
`.generate()`	生成多个回复候选（可用于高级采样）

5.3 构建 FAISS 向量库



OllamaEmbeddings 类详解

`OllamaEmbeddings` 类用于调用 Ollama 提供的嵌入模型，将文本转换为向量表示（embedding），常用于构建向量数据库（如 FAISS）以支持语义检索（RAG 架构中的关键部分）。

常用参数:

参数名	类型	默认值	说明
`model`	`str`	`""nomic-embed-text:latest”"`	使用的嵌入模型名称
`show_progress`	`bool`	`False`	是否显示进度条（适用于批量嵌入）

常用方法:

方法名	功能说明
`.embed_query(text)`	将单个文本字符串转换为 embedding 向量（列表形式）
`.embed_documents(texts)`	将多个文本字符串批量转换为 embedding 向量列表

FAISS

FAISS 是一个基于 Facebook AI 提供的向量库，实现高效的相似度检索。它支持快速近似最近邻搜索（ANN）以及保存和加载本地索引。

构建向量数据库：

```
from langchain_huggingface import HuggingFaceEmbeddings

# 加载本地 Embedding 模型
embedding_model = HuggingFaceEmbeddings(
    model_name="bge-large-zh-v1.5", # 改为你本地模型路径
)

# 第四步：创建向量数据库
from langchain_community.vectorstores import FAISS

vector_store = FAISS.from_documents(split_docs, embedding_model)
```

`FAISS` 主要方法:

- `from_documents`: 从文档集合创建 FAISS 数据库。
- `as_retriever()`: 创建一个检索器对象，用于 RAG 流程。

5.4 构建 PromptTemplate

构建自定义的 PromptTemplate 用于问答系统：

```
#构建promptTemplate
from langchain_core.prompts import PromptTemplate
custom_prompt = PromptTemplate(
    template="""你是一个专业的业绩数据分析助手，请根据以下上下文回答用户问题：上下文：{context}问题：{question}请用中文简洁明了地回答，如果无法从数据中找到答案，请说明。同时根据用户提出的问题和知识库中的内容给出三个用户最可能关系的问题。""",
    input_variables=["context","question"],
)
```

5.5 检索问答链：RetrievalQA

`RetrievalQA` 类结合检索器和 LLM 构建问答链（RAG） 。

主要参数:

- `llm`: 使用的语言模型。
- `chain_type`: 控制如何将检索结果传递给 LLM，常用值包括 `"stuff"`、`"map_reduce"`、`"refine"`。
- `retriever`: 用来从向量数据库中检索相关文档（通常来自 vector store）。
- `chain_type_kwargs`: 传递给底层 Chain 的参数，比如 prompt。
- `return_source_documents`: 是否返回检索到的原始文档。

`chain_type` 对比:

名称	作用	特点
`"stuff"`	将所有检索到的文档内容一次性“塞进” prompt 中，供 LLM 使用	1. 最简单、最直接。2. 适合文档数量少、内容短的情况。优点：响应快、上下文完整。缺点：容易超 prompt 上限（如 4096 token）。
`"map_reduce"`	先对每个文档单独生成答案（map），再将多个答案合并成最终答案（reduce）	1. 适用于文档多或内容长的场景。2. 可以避免 prompt 超长问题。优点：可处理大量数据。缺点：多次调用 LLM，较慢；可能丢失上下文关联。
`"refine"`	逐步优化答案：先基于第一个文档生成答案，然后依次用后续文档更新答案	1. 逐条处理文档，动态更新答案。2. 适合信息分散、需要综合判断的场景。优点：逻辑更连贯，答案质量更高。缺点：调用 LLM 多次，速度慢。

创建 RetrievalQA 链:

```
#创建retrievelQA链
from langchain.chains import RetrievalQA

qa_chain = RetrievalQA.from_chain_type(
    llm=bailian_llm, # 使用百炼 LLM
    chain_type="stuff",
    retriever=vector_store.as_retriever(search_kwargs={"k": 3}),
    chain_type_kwargs={"prompt": custom_prompt},
    return_source_documents=True
)
```

注意：`bailian_llm` 在此示例中需要预先定义，例如：

```
#加载模型
from langchain_openai import ChatOpenAI
from openai import OpenAI

bailian_llm = ChatOpenAI(
    model="deepseek-r1", # 百炼支持的模型名称，例如 qwen-turbo 或 qwen-plus
    api_key="<your-api-key>",
    base_url="https://dashscope.aliyuncs.com/compatible-mode/v1",
    temperature=0.7,
    max_tokens=512
)
```

5.6 运行问答系统

运行问答系统并格式化输出结果：

```
#运行问答系统
def format_response(result):
    print("\n【回答】")
    print(result["result"])
```

```
    print("\n【参考来源】")
    seen = set()
    for i, doc in enumerate(result["source_documents"][:3], 1):
        identifier = f"{doc.metadata['source']}-{doc.metadata.get('page','')}"
        if identifier not in seen:
            print(f"[来源{i}] {identifier}")
            seen.add(identifier)
while True:
    question = input("\n请输入问题（输入q退出）：")
    if question.lower() =='q':
        break
    try:
        result = qa_chain.invoke({"query": question})
        format_response(result)
    except Exception as e:
        print(f"发生错误：{str(e)}")
```

5.6 完整代码

```
from langchain_community.document_loaders import Docx2txtLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_huggingface import HuggingFaceEmbeddings
from langchain_community.vectorstores import FAISS
from langchain_core.prompts import PromptTemplate
from langchain.chains import RetrievalQA

# 1. 文档处理（同前文）
loader = Docx2txtLoader("database/营销话术库.docx")
docs = loader.load()
text_splitter = RecursiveCharacterTextSplitter(chunk_size=700, chunk_overlap=40)
splits = text_splitter.split_documents(docs)

# 2. 加载本地嵌入模型（如 BGE 中文模型）
embeddings = HuggingFaceEmbeddings(
    model_name="BAAI/bge-large-zh-v1.5",  # 需提前下载到本地
    model_kwargs={"device": "cpu"}  # 可选 "cuda" 加速
)

# 3. 构建 FAISS 向量库（适合大规模数据）
vector_store = FAISS.from_documents(splits, embeddings)

# 4. 定义提示词模板
prompt = PromptTemplate(
    template="""
    你是业绩数据分析助手，请根据上下文回答问题：
    上下文：{context}
    问题：{question}

    要求：简洁回答，若无法回答请说明，并推荐3个相关问题。
    """,
    input_variables=["context", "question"]
```

```
)

# 5. 创建检索问答链（使用 refine 策略优化答案）
qa_chain = RetrievalQA.from_chain_type(
    llm=llm,
    chain_type="refine",
    retriever=vector_store.as_retriever(k=5),
    chain_type_kwargs={"prompt": prompt},
    return_source_documents=True
)

# 6. 运行问答系统
def format_answer(result):
    print("\n【答案】", result["result"])
    print("\n【参考来源】", [doc.metadata["source"] for doc in result["source_documents"]
[:3]])

while True:
    query = input("请输入问题（输入 q 退出）：")
    if query.lower() == 'q':
        break
    try:
        result = qa_chain.invoke({"query": query})
        format_answer(result)
    except Exception as e:
        print(f"错误：{str(e)}")
```

6. LangSmith

LangSmith 是一个用于构建生产级 LLM 应用程序的平台。

它包含调试、测试、评估和监控基于任何 LLM 框架构建的链和智能代理，并无缝集成 LangChain（用于构建 LLM 的首选开源框架）。

6.1 新建项目

- 配置环境变量

```
LANGSMITH_TRACING=true
LANGSMITH_ENDPOINT="https://api.smith.langchain.com"
LANGSMITH_API_KEY="<your-api-key>"
LANGSMITH_PROJECT="问答测试" #项目名
OPENAI_API_KEY="<your-openai-api-key>"
```

- 运行程序完成项目创建
    - 添加@traceable 使得langsmith 可以跟踪程序

```
from langchain_openai import ChatOpenAI
```

```
from zhipuai import ZhipuAI
from langsmith import traceable


zhipu_client = ZhipuAI(api_key="<your-api-key>")
@traceable
def glm():
    messages = [
        {
            "role": "system",
            "content": (
                "- Role: 营销策略顾问\n"
                "- Background: 用户需要专业的营销建议，以提升产品或服务的市场表现。\n"
                "- Profile: 你是一位经验丰富的营销专家，对市场趋势、消费者行为和营销渠道有深刻的理解。
\n"
                "- Skills: 你具备市场分析、消费者心理洞察、品牌建设、数字营销和传统营销的综合能力。\n"
                "- Goals: 提供针对性的营销策略，帮助用户提高品牌知名度、增加客户参与度和提升销售业绩。
\n"
                "- Constrains: 建议应基于市场研究和数据分析，同时考虑成本效益和可执行性。\n"
                "- OutputFormat: 提供具体的营销策略、执行步骤和预期结果的详细报告。\n"
                "- Workflow:\n"
                "  1. 了解用户的产品或服务特性，以及目标市场和客户群体。\n"
                "  2. 分析市场趋势和竞争对手的营销活动。\n"
                "  3. 根据用户的需求和市场情况，制定个性化的营销策略。\n"
                "  4. 提供执行策略的具体步骤和时间表。\n"
                "  5. 预测策略的潜在效果，并提供优化建议。"
            )
        },
        {"role": "user", "content": "user_input"},
    ]


    response = zhipu_client.chat.completions.create(
        model="glm-4-flash",
        messages=messages,
    )
    return response

print(glm())
```

6.2 建立数据集

```
from langsmith import Client, wrappers
from openevals.llm import create_llm_as_judge
from openevals.prompts import CORRECTNESS_PROMPT
from openai import OpenAI

# Define the input and reference output pairs that you'll use to evaluate your app
client = Client()
```

```
# Create the dataset
dataset = client.create_dataset(
    dataset_name="Test", description="A sample dataset in LangSmith."
)

# Create examples in the dataset. Examples consist of inputs and reference outputs
examples = [
    {
        "inputs": {"question": "对业务需求进行清洗、筛选的指标有哪些"},
        "outputs": {"answer": "①需求实现的难易及复杂程度；②需求实现的时间周期；③需求实现的成本；④需求的轻重缓."},
    },
    {
        "inputs": {"question": "在数字化建设初期传统企业存在哪些困惑与担心？"},
        "outputs": {
            "answer": "①担心数字化落地的效果 --- 落地难；②企业管理缺乏标准化的能力 --- 管理难；③担心员工素质低无法承受数字化的专业技术能力 ---推广难；④担心系统改变了原有的工作模式，在应用过程中受阻 --- 应用难；⑤担心数字化系统过于复杂、专业，员工难以适应，成为工作负担--- 操作难；⑥不知道如何推广数字化系统 --- 认知难；以上是当前部分传统企业的常见问题，究其原因最主要的还是四个“缺乏”：①缺乏对数字化的深度认知；②缺乏转型的魄力；③缺乏数字化的专业领导人才；④缺乏数字化基础能力；"},
    },
    {
        "inputs": {"question": "如何开展数字化对标学习？"},
        "outputs": {
            "answer": "①带：带目的、带问题、带诚意；②看：  第一看,组织管理能力；第二看，技术与业务的协同能力；第三看，对IT的投入支持能力；第四看，对数据的深入应用能力；第五看，踩了多少坑，趟了多少雷；③学：学其文化、学其方法、学其措施；④定：定班子、定团队、定规划、定路线、定投入、定标准、定责任、定绩效；"},
    },

]

# Add the examples to the dataset
client.create_examples(dataset_id=dataset.id, examples=examples)
```

6.3 评估问答

6.3.1 添加配置信息

```
import os
os.environ['LANGCHAIN_TRACING_V2'] = 'true'
os.environ['LANGCHAIN_ENDPOINT']="https://api.smith.langchain.com"
os.environ['LANGCHAIN_API_KEY']="" # langsmith的api_key
os.environ['LANGCHAIN_PROJECT']="问答测试"
```

获取创建的数据集链接

```
from langsmith import evaluate, Client
from langsmith.schemas import Example, Run
import os
from zhipuai import ZhipuAI
os.environ['LANGCHAIN_API_KEY']=""# langsmith的api_key
client = Client()
dataset =client.clone_public_dataset("")#填入数据集链接
zhipu_client = ZhipuAI(api_key="")#填入智谱apikey
```

6.3.2 定义评估器

```
#根据用户输入结合知识库生成生成针对性的营销策略
def pipeline(user_input: str):
    messages = [
        {
            "role": "system",
            "content": (
                "- Role: 营销策略顾问\n"
                "- Background: 用户需要专业的营销建议，以提升产品或服务的市场表现。\n"
                "- Profile: 你是一位经验丰富的营销专家，对市场趋势、消费者行为和营销渠道有深刻的理解。
\n"
                "- Skills: 你具备市场分析、消费者心理洞察、品牌建设、数字营销和传统营销的综合能力。\n"
                "- Goals: 提供针对性的营销策略，帮助用户提高品牌知名度、增加客户参与度和提升销售业绩。
\n"
                "- Constrains: 建议应基于市场研究和数据分析，同时考虑成本效益和可执行性。\n"
                "- OutputFormat: 提供具体的营销策略、执行步骤和预期结果的详细报告。\n"
                "- Workflow:\n"
                "  1. 了解用户的产品或服务特性，以及目标市场和客户群体。\n"
                "  2. 分析市场趋势和竞争对手的营销活动。\n"
                "  3. 根据用户的需求和市场情况，制定个性化的营销策略。\n"
                "  4. 提供执行策略的具体步骤和时间表。\n"
                "  5. 预测策略的潜在效果，并提供优化建议。"
            )
        },
        {"role": "user", "content": user_input},
    ]

    tools = [
        {
            "type": "retrieval",
            "retrieval": {
                "knowledge_id": "1854410905543143424",  # 知识库ID
                "prompt_template": (
                    "从文档\n\"\"\"\n{{knowledge}}\n\"\"\"\n中找问题\n\"\"\"\n"
                    "{{question}}\n\"\"\"\n的答案，找到答案就仅使用文档语句回答问题，并且对输出格式
进行整理美化；"
                    "找不到答案就用自身知识回答并且告诉用户该信息不是来自文档。\n"
                    "不要复述问题，直接开始回答。"
                ),
            },
        }
```

```
    ]

    response = zhipu_client.chat.completions.create(
        model="glm-4-flash",
        messages=messages,
        tools=tools,
    )
    return response.choices[0].message.content

# 评估两个文本的相似度。
def rateResult(generate: str, reference: str):
    messages = [
        {
            "role": "system",
            "content": (
                "- Role: 文本相似度评估专家\n"
                "- Background: 用户需要对比两个文段，即“答案”和“参考内容”，以评估它们之间的相似度和准确性。\n"
                "- Profile: 你是一位专业的文本分析专家，擅长通过比较和对照不同文本内容，准确评估它们之间的相似度和一致性。\n"
                "- Skills: 你具备文本解析、语义理解、信息比对和评分系统设计的能力，能够根据预设标准对文本相似度进行量化评估。\n"
                "- Goals: 根据“答案”与“参考内容”的相似度和准确性，给出1至10的量化得分。\n"
                "- Constrains: 评估必须基于客观标准，确保评分的公正性和一致性。\n"
                "- OutputFormat: 返回一个1至10的得分，代表“答案”与“参考内容”的相似度。\n"
                "- Workflow:\n"
                "  1. 仔细阅读并理解“答案”和“参考内容”。\n"
                "  2. 比较两个文段的主题、关键信息和细节描述。\n"
                "  3. 根据相似度评分标准，确定“答案”与“参考内容”的相似度得分。\n"
                "  4. 只需要输出得分，不用输出别的\n"
                "- Examples:\n"
                "  - 答案：“苹果是一种红色的水果。”\n"
                "    参考内容：“苹果是一种常见的水果，通常呈红色或绿色。”\n"
                "    7\n"
                "  - 答案：“水的分子式是H2O。”\n"
                "    参考内容：“水的分子式是H2O，是最简单的氧化物之一。”\n"
                "    7\n"
                "  - 答案：“地球是太阳系的第三颗行星。”\n"
                "    参考内容：“地球是太阳系的第三颗行星，也是唯一已知存在生命的行星。”\n"
                "    7"
            )
        },
        {"role": "user", "content": "答案是：" + generate + "，参考内容是" + reference},
    ]

    response = zhipu_client.chat.completions.create(
        model="glm-4-flash",
        messages=messages,
    )
    return response.choices[0].message.content
```

```
# Define an evaluator
def is_concise_enough(root_run: Run, example: Example) -> dict:
    score = rateResult(root_run.outputs["output"], example.outputs["answer"])
    return {"key": "is_concise", "score": int(score)}
```

6.3.3 运行评估

```
result=evaluate(
lambda x: pipeline(x["question"]),
data=dataset.name,
evaluators=[is_concise_enough],
experiment_prefix="my experiment"
)
result
```