# LangChain 系列教程（二）：Model I/O

## 介绍

LangChain 提供的 LangChain Expression Language(LCEL) 让开发可以很方便地将多个组件连接成 AI 工作流。如下是一个简单的工作流：

chain = prompt | chatmodel | outputparser  
chain.invoke({"input":"What's your name?"})

其中，通过由|管道操作符连接而成的 LangChain 表达式，我们方便地将三个组件 **prompt chatmodel outparser** 按顺序连接起来，这就形成了一个 AI 工作流。 **invoke()**则是实际运行这个工作流。

而 LangChain 的 Model I/O 模块是与语言模型（LLMs）进行交互的核心组件，它包括**模型输入（Prompts）、模型本身（Models）和模型输出（Output Parsers）**。

在 LangChain 的 Model I/O 模块设计中，包含三个核心部分： **Prompt Template，Model和Output Parser**。

* **Prompt Template**：通过模板化来管理大模型的输入。
* **Model**：使用通用接口调用不同的大语言模型。
* **Output Parser**：用来从模型的推理中提取信息，并按照预先设定好的模版来规范化输出。

## 代码

### 配置环境

!pip install langchain langchain-openai tiktoken  
  
import os  
os.environ["OPENAI\_API\_BASE"] = "https://api.chatanywhere.tech"  
os.environ["OPENAI\_API\_KEY"] = "sk-xxx"

### 示例教程

# 给定一段文本，输出 40 字以内摘要及 3 个关键词（JSON 格式）  
  
# ① 准备——模板、解析器与模型  
from langchain.prompts import PromptTemplate  
from langchain\_core.output\_parsers import JsonOutputParser  
from langchain\_openai import ChatOpenAI  
  
# 构造解析器并取出“输出格式说明”  
parser = JsonOutputParser()  
format\_hint = parser.get\_format\_instructions()  
  
# 定义提示词模板：40 字摘要 + 3 关键词  
prompt = PromptTemplate(  
 template=(  
 "请在不超过40字内总结下文内容，并给出3个关键词，"  
 "返回格式：{format}\n\n文本：{text}"  
 ),  
 input\_variables=["text", "format"],  
)  
  
# 初始化 GPT-3.5（开启流式输出）  
llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0)  
  
# ② 组装链：PromptTemplate → ChatModel → OutputParser  
chain = prompt | llm | parser  
  
# ③ 调用链  
doc = "LangChain 是一个面向 LLM 应用的框架......"  
result = chain.invoke({"text": doc, "format": format\_hint})  
  
print(result) # {'summary': '……', 'keywords': ['LangChain', 'LLM', '框架']}

{'summary': 'LangChain 是用于构建大语言模型应用的框架。', 'keywords': ['LangChain', '大语言模型', '框架']}

### PromptTemplate

PromptTemplate是指生成提示的可重复的方式。它包含一个文本字符串（“模板”），可以接收来自用户的一组参数并生成提示。

可以使用 PromptTemplate 类创建简单的硬编码提示。提示模板可以采用任意数量的输入变量，并且可以格式化以生成提示。

from langchain import PromptTemplate  
  
# 没有输入变量的示例提示  
no\_input\_prompt = PromptTemplate(input\_variables=[], template="给我讲个笑话。")  
print(no\_input\_prompt.format())  
  
# 带有一个输入变量的示例提示  
one\_input\_prompt = PromptTemplate(input\_variables=["adjective"], template="给我讲一个{adjective}笑话。")  
print(one\_input\_prompt.format(adjective="有趣"))  
  
# 具有多个输入变量的示例提示  
multiple\_input\_prompt = PromptTemplate(  
 input\_variables=["adjective", "content"],   
 template="给我讲一个关于{content}的{adjective}笑话。"  
)  
print(multiple\_input\_prompt.format(adjective="有趣", content="鸡"))  
  
# 定义一个包含占位符的提示模板，LangChain 会自动解析并记录所有占位符变量  
template = "给我讲一个关于{content}的{adjective}笑话。"  
auto\_prompt = PromptTemplate.from\_template(template)  
print(auto\_prompt.input\_variables)  
print(auto\_prompt.format(adjective="有趣", content="鸡"))

给我讲个笑话。  
给我讲一个有趣笑话。  
给我讲一个关于鸡的有趣笑话。  
['adjective', 'content']  
给我讲一个关于鸡的有趣笑话。

#### Few-shot PromptTemplate

在大语言模型的提示工程中，**Few-shot examples**是指在同一个提示里先放入少量（通常 1–5 条）示范输入→输出配对，用来向模型展示期望的格式、语气或推理路径。

这种做法属于 in-context learning：模型不需要额外微调，即可从示例中抽取模式并泛化到用户的真实问题。相比于 zero-shot（不给示例）与全量监督微调，few-shot 在提升复杂任务性能与节省标注成本之间取得平衡。​

下面代码演示如何在 LangChain 中使用 Few-shot PromptTemplate 将三条示例嵌入提示，指导 GPT-3.5 对影评做情感判定。

from langchain.prompts import PromptTemplate, FewShotPromptTemplate  
from langchain\_openai import OpenAI  
from langchain\_core.output\_parsers import StrOutputParser  
  
# ① 示例对列表  
examples = [  
 {"review": "这部电影让我热泪盈眶，演员表现惊艳！", "label": "Positive"},  
 {"review": "故事老套，节奏拖沓，看得我想睡觉。", "label": "Negative"},  
 {"review": "视觉效果不错，但剧情一般。", "label": "Neutral"},  
]  
  
# ② 描述“示例长相”的模板  
example\_prompt = PromptTemplate(  
 input\_variables=["review", "label"],  
 template="影评：{review}\n情感：{label}\n",  
)  
  
# ③ Few-shot PromptTemplate：自动把示例 + 用户输入拼接  
few\_shot\_prompt = FewShotPromptTemplate(  
 examples=examples, # 要插入的 few-shot 列表  
 example\_prompt=example\_prompt, # 如何渲染每条示例  
 suffix="影评：{input\_review}\n情感：", # 用户输入部分，真正要问模型的问题，其中 {input\_review} 是占位符  
 input\_variables=["input\_review"], # 运行时还需要哪些字段  
)  
  
# ④ 调用 OpenAI GPT-3.5（非聊天接口）  
llm = OpenAI(model="gpt-3.5-turbo", temperature=0)  
parser = StrOutputParser() # 仅返回文本标签  
chain = few\_shot\_prompt | llm | parser  
  
print(chain.invoke({"input\_review": "动作场面震撼，剧情也扣人心弦。"}))

Positive

prompt 可以通过 json 或者 yaml 进行保存读取

'''  
json文件内容如下：  
{  
 "\_type": "prompt",  
 "input\_variables": ["adjective", "content"],  
 "template": "Tell me a {adjective} joke about {content}."  
}  
'''  
  
from langchain.prompts import load\_prompt  
  
prompt = load\_prompt("simple\_prompt.json")  
print(prompt.format(adjective="funny", content="chickens"))

Tell me a funny joke about chickens.

#### Selector

在 LangChain 的 few-shot 提示中，Selector 用于动态选取最合适的示例，而不是每次都把固定示例全部塞进提示里。

这样可以 (1) 缩短上下文、节省成本；(2) 提高示例与当前输入的相似度，从而获得更准确的回答。

# pip install "langchain-core>=0.2" langchain-openai faiss-cpu tiktoken  
  
from langchain\_core.example\_selectors import SemanticSimilarityExampleSelector # 新路径✅  
from langchain\_core.prompts import FewShotPromptTemplate, PromptTemplate  
from langchain\_openai import OpenAIEmbeddings, OpenAI  
from langchain.vectorstores.faiss import FAISS  
from langchain\_core.output\_parsers import StrOutputParser  
  
# ---------- 1) 准备 few-shot 示例 ----------  
examples = [  
 {"review": "这部电影让我热泪盈眶，演员表现惊艳！", "label": "Positive"},  
 {"review": "故事老套，节奏拖沓，看得我想睡觉。", "label": "Negative"},  
 {"review": "视觉效果不错，但剧情一般。", "label": "Neutral"},  
 {"review": "配乐出彩，但人物塑造单薄。", "label": "Neutral"},  
 {"review": "情节紧凑，反转惊喜，全程无尿点!", "label": "Positive"},  
]  
  
# ---------- 2) 定义“每条示例如何呈现” ----------  
example\_prompt = PromptTemplate(  
 template="影评：{review}\n情感：{label}\n",  
 input\_variables=["review", "label"],  
)  
  
# ---------- 3) 创建 Semantic Selector（选 k=3 条最相似示例） ----------  
selector = SemanticSimilarityExampleSelector.from\_examples(  
 examples = examples, # 示例库  
 embeddings = OpenAIEmbeddings(),# 向量模型  
 vectorstore\_cls = FAISS, # 内存向量库  
 k = 3, # 每次选 3 条  
)  
  
# ---------- 4) Few-shot PromptTemplate 与 Selector 结合 ----------  
few\_shot\_prompt = FewShotPromptTemplate(  
 example\_prompt = example\_prompt, # 示例渲染方式  
 example\_selector = selector, # 动态选择器  
 suffix = "影评：{input\_review}\n情感：", # 用户输入部分  
 input\_variables = ["input\_review"],  
)  
  
# ---------- 5) 组装链并调用 ----------  
llm = OpenAI(model="gpt-3.5-turbo-instruct", temperature=0)  
parser = StrOutputParser()  
chain = few\_shot\_prompt | llm | parser  
  
query = "节奏明快，剪辑流畅，就是结尾稍显仓促。"  
  
# selector.select\_examples() 会：  
# 1) 将 query 嵌入为向量  
# 2) 与向量库中的示例做余弦相似度比对  
# 3) 取相似度最高的 k 条示例（k 在创建 selector 时设定）  
# 注意：输入字典的 key 必须对应 selector 创建时使用的占位符，  
# 例如前文 FewShotPromptTemplate 的占位符是 {input\_review}，  
# 此处就应该写 {"input\_review": query}  
selected\_examples = selector.select\_examples({"input\_review": query})  
for example in selected\_examples:  
 for k, v in example.items():  
 print(f"{k}: {v}")  
  
print(chain.invoke({"input\_review": query})) # → Positive / Neutral / Negative

review: 故事老套，节奏拖沓，看得我想睡觉。  
label: Negative  
review: 情节紧凑，反转惊喜，全程无尿点!  
label: Positive  
review: 视觉效果不错，但剧情一般。  
label: Neutral  
Neutral

### Model

Langchain作为一个“工具”它并没有提供自己的LLM，而是提供了一个接口，用于与许多不同类型的LLM进行交互，比如耳熟能详的openai、huggingface或者是cohere等，都可以通过langchain快速调用。

from langchain.llms import OpenAI  
  
llm = OpenAI() # 创建 LLM 对象  
print(llm('你是谁')) # 单条调用  
print(llm.generate(["给我背诵一首古诗", "给我讲个100字小故事"]\*10)) # 批量调用

");  
 return -1;  
 }  
 }  
   
 if (strlen(str) > 1 && !strcmp(str, "无语"))  
 {  
 if (query("weiwang") >= 100000)  
 {  
 message\_vision(HIY "$N向$n瞥了一眼，道：你这个问题问得实在是……\n" NOR, this\_object(), me);  
 message\_vision(HIY "$N接着叹了一口气，道：没事，没事，我也不想多说什么。\n" NOR, this\_object(), me);  
 return -1;  
 }  
 else  
 {  
 message\_vision(HIY "$N向$n瞥了一眼，道：你问我这个干什么？\n" NOR, this\_object(), me);  
 return -1;  
 }  
 }  
   
 if (strlen(str) > 1 && !strcmp(str, "什么"))  
 {  
 message\_vision(HIY "$N向$n瞥了一眼，道：你想知道什么？\n" NOR, this  
generations=[[Generation(text='\n李白的《静夜思》：\n\n床前明月光，疑是地上霜。\n举头望明月，低头思故乡。\n', generation\_info={'finish\_reason': 'stop', 'logprobs': None})], [Generation(text='\n\n从前，在一个小镇上，有一位老人，他每天都会坐在村口的长椅上，观察着路过的人们。他总是非常和蔼地向每个人打招呼，给他们祝福。即使有些人没有回应，他也不会生气，依然保持着微笑。村里的人都很喜欢和他聊天，因为他总能给他们带来快乐和温暖。\n\n有一天，一位小女孩路过，她看到老人的手里拿着一把小刀在雕刻着一块木头。她很好奇，便走过去问老人在做什么。老人笑着回答说，他在做一只小熊玩具，准备送给自己的孙子。\n\n小女孩感动地看着', generation\_info={'finish\_reason': 'length', 'logprobs': None})], [Generation(text='\n\n《静夜思》-李白\n床前明月光，疑是地上霜。\n举头望明月，低头思故乡。\n', generation\_info={'finish\_reason': 'stop', 'logprobs': None})], [Generation(text='\n\n从前有一个小男孩，他非常喜欢吃糖果。每天放学回家，他都会先去买一袋糖果，然后一边吃一边玩耍。他的妈妈总是担心他吃太多糖会伤害身体，可是他总是不听。直到有一天，他发现自己的牙齿开始疼痛，妈妈带他去看牙医，医生告诉他，因为吃太多糖，他的牙齿都蛀了。小男孩非常后悔，他决定从此以后不再吃糖了。经过一段时间的护理，他的牙齿慢慢好转，变得更加健康。从此以后，他也学会了控制自己的食', generation\_info={'finish\_reason': 'length', 'logprobs': None})], [Generation(text='\n\n《静夜思》\n床前明月光，疑是地上霜。\n举头望明月，低头思故乡。\n', generation\_info={'finish\_reason': 'stop', 'logprobs': None})], [Generation(text='\n\n从前有一个小男孩，他非常喜欢跑步，每天都会跑去公园里练习。有一天，他突然发现一只小鸟的腿受伤了，无法飞行。小男孩心里很难过，于是他决定每天帮助小鸟找食物，直到它的腿好了为止。经过一段时间的照顾，小鸟的腿终于完全康复了，它也能够飞回天空了。小男孩很高兴，但是小鸟却不想离开他，它每天都会来公园和小男孩一起跑步。从此以后，小男孩和小鸟成为了最好的朋友，一起享受着跑步和自由的快乐。小男孩也明白了', generation\_info={'finish\_reason': 'length', 'logprobs': None})], [Generation(text='\n\n《春晓》- 孟浩然\n\n春眠不觉晓，\n处处闻啼鸟。\n夜来风雨声，\n花落知多少。\n\n天街小雨润如酥，\n草色遥看近却无。\n最是一年春好处，\n绝胜烟柳满皇都。', generation\_info={'finish\_reason': 'stop', 'logprobs': None})], [Generation(text='\n从前有一位老人，每天都会在公园里散步。他喜欢观察周围的一切，尤其是动物们。有一天，他发现一只小鸟摔断了翅膀，无法飞行。老人立刻心生怜悯，将小鸟带回家，给它包扎伤口，并细心地喂食。\n\n经过几天的照顾，小鸟的伤口渐渐愈合，它也恢复了飞行能力。但是，当老人打开窗户，让小鸟自由飞出去时，它却留在了老人的家中，不肯离开。\n\n老人感到十分困惑，于是他决定带着小鸟去公园，希望能让它重新回到自然中。看着小', generation\_info={'finish\_reason': 'length', 'logprobs': None})], [Generation(text='\n《登鹳雀楼》\n作者：王之涣\n\n白日依山尽，黄河入海流。\n欲穷千里目，更上一层楼。\n', generation\_info={'finish\_reason': 'stop', 'logprobs': None})], [Generation(text='\n\n从前有一只小猫，它是一只非常调皮的小猫。每天它都会跑来跑去，捣蛋闹事，让主人非常头疼。主人是一位老人，她对小猫非常宽容，总是宠爱它。有一天，小猫偷偷溜出家门，来到了一座花园。花园里有一只美丽的小鸟，小猫非常想抓到它。它蹲在花丛中，悄悄地等待着。可是小鸟却飞到了树上，小猫追了上去，却不小心掉进了一口井里。它害怕地喵喵叫，但是没有人听见。就在这时，主人来到了花园，她听到了', generation\_info={'finish\_reason': 'length', 'logprobs': None})], [Generation(text='\n\n《静夜思》\n床前明月光，疑是地上霜。\n举头望明月，低头思故乡。\n', generation\_info={'finish\_reason': 'stop', 'logprobs': None})], [Generation(text='\n\n小明是一位快乐的小男孩，他住在一个美丽的小村庄里。每天，他都会和小伙伴们一起在村子里玩耍，一起探索周围的大自然。他们常常会在田野里捉迷藏，或者在河边捉鱼虾。小明最喜欢的是和爷爷一起去山里采草药。爷爷总是会给他讲许多关于植物的知识，小明也因此对自然产生了浓厚的兴趣。\n\n有一天，小明和小伙伴们在河边玩耍时，发现了一只受伤的小鹿。小明立刻带着小伙伴们一起去找爷爷帮忙。爷爷看到小鹿受伤的', generation\_info={'finish\_reason': 'length', 'logprobs': None})], [Generation(text='\n\n《登鹳雀楼》\n\n白日依山尽，黄河入海流。\n欲穷千里目，更上一层楼。\n', generation\_info={'finish\_reason': 'stop', 'logprobs': None})], [Generation(text='\n\n从前，有一位聪明的小女孩，她非常喜欢探索和发现新事物。一天，她发现了一只小鸟，它的羽毛非常漂亮，但是却受了伤，不能飞行。小女孩心里很难过，便决定帮助它。她用自己的小手指做了一个小绷带，把小鸟的伤口包扎好，还给它喂食。经过几天的照顾，小鸟的伤口愈合了，它也恢复了健康，可以自由地飞翔了。小女孩看着小鸟飞走，心里感到非常满足和快乐。从此以后，小女孩更加热爱大自然，也学会了关爱生命，用', generation\_info={'finish\_reason': 'length', 'logprobs': None})], [Generation(text='\n\n《静夜思》 - 李白\n床前明月光，疑是地上霜。\n举头望明月，低头思故乡。\n', generation\_info={'finish\_reason': 'stop', 'logprobs': None})], [Generation(text='\n\n从前有一只小鸟，它叫小红。小红是一个非常勇敢的小鸟，它有着美丽的羽毛和灵巧的翅膀。一天，小红飞到了一个陌生的地方，它想要探索一下这个地方。但是，小红却遇到了一只凶恶的老鹰，它正盯着小红准备要吃掉它。小红非常害怕，但是它并没有放弃，它想要用自己的勇气和智慧来应对老鹰。\n\n小红想到了一个好主意，它开始唱歌，用美妙的歌声来打动老鹰。老鹰被小红的歌声吸引住了，停止了攻击。小红趁', generation\_info={'finish\_reason': 'length', 'logprobs': None})], [Generation(text='吧\n\n《登鹳雀楼》- 王之涣\n\n白日依山尽，黄河入海流。\n欲穷千里目，更上一层楼。\n', generation\_info={'finish\_reason': 'stop', 'logprobs': None})], [Generation(text='\n\n从前有一个小姑娘，她非常喜欢画画，每天都会拿起画笔和颜料，画出各种美丽的图案。她的父母非常支持她，每次看到她的作品都会表扬她。小姑娘也很善良，经常会把自己的作品送给邻居和朋友。\n\n一天，小姑娘听说村子里有一位老奶奶生病了，家里没有人能照顾她。她决定用自己的画作换取一些粥水，帮助老奶奶照顾她。老奶奶非常感动，她把小姑娘的画作挂在墙上，每天都会看着它，心里暖暖的。\n\n小姑娘也经常去看望老', generation\_info={'finish\_reason': 'length', 'logprobs': None})], [Generation(text='\n\n关山月\n\n明月出天山，苍茫云海间。\n长风几万里，吹度玉门关。\n汉下白登道，胡窥青海湾。\n由来征战地，不见有人还。\n戍客望边色，思归多苦颜。\n高楼当此夜，叹息未应闲。\n天阶夜色寒，银汉鸦啼眠。\n但愿人长久，千里共婵娟。', generation\_info={'finish\_reason': 'stop', 'logprobs': None})], [Generation(text='\n\n有一只小松鼠，它非常喜欢收集树上的松果。每天早上，它都会跳上树枝，开始收集松果。它把松果放在一个小洞里，作为自己的财富。\n\n有一天，小松鼠遇到了一只鸟儿，它的翅膀受伤无法飞行。小松鼠看到它落在地上，立刻跑过去帮助它。它用爪子抓住一根树枝，让鸟儿站在上面，然后带它飞到树上。鸟儿非常感激小松鼠的帮助，它决定用自己的歌声来回报小松鼠。\n\n从此，小松鼠和', generation\_info={'finish\_reason': 'length', 'logprobs': None})]] llm\_output={'token\_usage': {'prompt\_tokens': 230, 'completion\_tokens': 3187, 'total\_tokens': 3417}, 'model\_name': 'gpt-3.5-turbo-instruct'} run=[RunInfo(run\_id=UUID('96b87007-d652-4760-ba2f-9c3051238a24')), RunInfo(run\_id=UUID('57995714-f640-41cf-9815-5a156fb683dd')), RunInfo(run\_id=UUID('1de577e8-156a-49f6-892a-7da44386f149')), RunInfo(run\_id=UUID('8d37b464-19a0-4522-b058-1ad4ff89eac3')), RunInfo(run\_id=UUID('c0185803-86fc-4140-a097-5ccf65a3ca78')), RunInfo(run\_id=UUID('6efb722a-935f-4ec5-8bf7-e2505429e19a')), RunInfo(run\_id=UUID('9debff07-bf16-46da-9a76-b057f2b2f98c')), RunInfo(run\_id=UUID('56207be2-7dde-44d7-98ab-fbcebf66b96d')), RunInfo(run\_id=UUID('3ad9e5b6-815b-4228-975f-31c6482450f3')), RunInfo(run\_id=UUID('ec052a8a-da81-46ae-822e-21e0930cb782')), RunInfo(run\_id=UUID('b62455f7-58e7-452d-b9de-5e1c7bd68389')), RunInfo(run\_id=UUID('48b0afca-9ee5-4408-8b80-cf0f805b87d9')), RunInfo(run\_id=UUID('bf36e293-71e8-46bf-86c8-3a7e60f15d7d')), RunInfo(run\_id=UUID('c523593e-c1dd-419e-8668-c428eb5615f2')), RunInfo(run\_id=UUID('f4518e12-d716-4f94-8b2d-9d20fea9b194')), RunInfo(run\_id=UUID('ba9d9cab-6a73-40e7-8b10-8e23e8d5cc1e')), RunInfo(run\_id=UUID('3711938d-5df1-4ebb-a8db-91f7c013a4d3')), RunInfo(run\_id=UUID('a3d1c05f-6fc1-4df8-9dd3-da9ebf4f18d0')), RunInfo(run\_id=UUID('d9ad44a4-a519-4916-be2a-d17dd8155f14')), RunInfo(run\_id=UUID('361c9490-67b4-40d1-aba2-cb7880975429'))] type='LLMResult'

import os, time, asyncio, openai  
from langchain.llms import OpenAI  
  
# ---------- 同步调用 ----------  
def generate\_sync(n: int = 10):  
 llm = OpenAI(temperature=0.9)  
 t0 = time.perf\_counter()  
 for \_ in range(n):  
 resp = llm.generate(["Hello, how are you?"])  
 print(resp.generations[0][0].text.strip())  
 dt = time.perf\_counter() - t0  
 print(f"\n同步模式耗时：{dt:.2f} 秒")  
  
# ---------- 异步调用 ----------  
async def async\_generate\_once(llm):  
 resp = await llm.agenerate(["Hello, how are you?"])  
 print(resp.generations[0][0].text.strip())  
  
async def generate\_async(n: int = 10):  
 llm = OpenAI(temperature=0.9)  
 t0 = time.perf\_counter()  
 tasks = [async\_generate\_once(llm) for \_ in range(n)]  
 await asyncio.gather(\*tasks)  
 dt = time.perf\_counter() - t0  
 print(f"\n异步并发耗时：{dt:.2f} 秒")  
  
# ---------- 运行对比 ----------  
if \_\_name\_\_ == "\_\_main\_\_":  
 print("=== 同步调用 ===")  
 generate\_sync()  
  
 print("\n=== 异步调用 ===")  
 await generate\_async()

=== 同步调用 ===  
I am an AI and do not have emotions, but thank you for asking. How can I assist you?  
I'm an AI and I don't experience emotions, but thank you for asking. Is there something I can assist you with today?  
I'm an AI language model, so I don't have feelings or emotions. But I am functioning well, thank you for asking. How about you?  
I am an AI, I do not have physical or emotional capabilities, but thank you for asking. Is there something you would like to talk about?  
I'm an AI and have no emotions, but thank you for asking! How may I assist you today?  
I am an AI, so I do not have emotions. But thank you for asking. How may I assist you?  
I am an AI and do not have emotions, but thank you for asking. How can I assist you?  
I am an AI and do not have emotions, but thank you for asking. How can I assist you today?  
I am an AI and cannot feel emotions, but thank you for asking. How can I assist you?  
I am an AI and don't have the ability to feel emotions. But thank you for asking! How can I assist you today?  
  
同步模式耗时：16.55 秒  
  
=== 异步调用 ===  
I am an AI so I do not have emotions, but thank you for asking. How can I assist you today?  
I am an AI and do not have the ability to feel emotions. How can I assist you?  
I am an AI, I do not have emotions like humans do, but thank you for asking. How can I assist you?  
I am a language model created by OpenAI and I do not have feelings. But thank you for asking, how can I assist you?  
I'm doing well, thank you. How about you?  
I'm an AI so I don't have feelings like humans, but thank you for asking! How can I assist you?  
I'm doing well, thank you for asking. How about you?  
I am an AI and do not have emotions, but thank you for asking. How can I assist you today?  
I am an AI language model. I do not have emotions but thank you for asking. Is there anything I can assist you with?  
I am an AI language model created by OpenAI. I do not have emotions, but thank you for asking. How can I assist you today?  
  
异步并发耗时：4.21 秒

#### 自定义LLM

在开发过程中如果遇到需要调用不同的LLM时，可以通过自定义LLM实现效率的提高。

自定义LLM时，必须要实现的是\_call方法，通过这个方法接受一个字符串、一些可选的索引字，并最终返回一个字符串。

除了该方法之外，还可以选择性生成一些方法用于以字典的模式返回该自定义LLM类的各属性。

from langchain.llms.base import LLM  
from langchain.callbacks.manager import CallbackManagerForLLMRun  
from typing import Optional, List, Any, Mapping  
  
class CustomLLM(LLM):  
 """  
 一个最小示范 LLM：只会把输入 prompt 截断到前 n 个字符返回。  
 继承自 LangChain 的 LLM 抽象类，因而可与链、代理等组件无缝配合。  
 """  
  
 n: int # 控制返回长度的参数，例如 n=10 就只保留前 10 个字符  
  
 # ---- 必需接口 1：声明模型类型 ----  
 @property  
 def \_llm\_type(self) -> str:  
 return "custom"  
  
 # ---- 必需接口 2：真正执行推理的函数 ----  
 def \_call(  
 self,  
 prompt: str, # 输入文本  
 stop: Optional[List[str]] = None, # 不支持 stop，因此若传入将报错  
 run\_manager: Optional[CallbackManagerForLLMRun] = None,  
 \*\*kwargs: Any,  
 ) -> str:  
 if stop is not None:  
 raise ValueError("stop kwargs are not permitted.")  
 return prompt[: self.n] # 简单地截断并返回  
  
 # ---- 必需接口 3：用于缓存与复现的“模型参数” ----  
 @property  
 def \_identifying\_params(self) -> Mapping[str, Any]:  
 # 让 LangChain 知道：参数只有 n  
 return {"n": self.n}  
   
  
toy\_llm = CustomLLM(n=8)  
print(toy\_llm("LangChain Makes LLM Apps Easy!"))

LangChai

#### 测试LLM

为了节省我们的成本，当写好一串代码进行测试的时候，通常情况下我们是不希望去真正调用LLM，因为这会消耗token。

Langchain则提供给我们一个“假的”大语言模型，以方便我们进行测试。

# 从langchain.llms.fake模块导入FakeListLLM类，此类可用于模拟或伪造某种行为  
from langchain.llms.fake import FakeListLLM  
from langchain.agents import load\_tools  
from langchain.agents import initialize\_agent  
from langchain.agents import AgentType  
from langchain\_experimental.tools import PythonREPLTool  
  
# 加载"python\_repl"的工具  
tools = [PythonREPLTool()]  
  
# 定义一个响应列表，这些响应是模拟LLM的预期响应  
responses = ["Action: Python REPL\nAction Input: print(2 + 2)", "Final Answer: 4"]  
  
# 使用上面定义的responses初始化一个FakeListLLM对象  
llm = FakeListLLM(responses=responses)  
  
# 调用initialize\_agent函数，使用上面的tools和llm，以及指定的代理类型和verbose参数来初始化一个代理  
agent = initialize\_agent(  
 tools, llm, agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION, verbose=True  
)  
  
# 调用代理的run方法，传递字符串"whats 2 + 2"作为输入，询问代理2加2的结果  
agent.run("whats 2 + 2")

'4'

与模拟llm同理，langchain也提供了一个伪类去模拟人类回复，该功能依赖于wikipedia。

# 从langchain.llms.human模块导入HumanInputLLM类，此类可能允许人类输入或交互来模拟LLM的行为  
from langchain.llms.human import HumanInputLLM  
from langchain.agents import load\_tools  
from langchain.agents import initialize\_agent  
from langchain.agents import AgentType  
  
# 调用load\_tools函数，加载名为"wikipedia"的工具  
tools = load\_tools(["wikipedia"])  
  
# 初始化一个HumanInputLLM对象，其中prompt\_func是一个函数，用于打印提示信息  
llm = HumanInputLLM(  
 prompt\_func=lambda prompt: print(f"\n===PROMPT====\n{prompt}\n=====END OF PROMPT======"))  
  
# 调用initialize\_agent函数，使用上面的tools和llm，以及指定的代理类型和verbose参数来初始化一个代理  
agent = initialize\_agent(tools, llm, agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION, verbose=True)  
  
# 调用代理的run方法，传递字符串"What is 'Bocchi the Rock!'?"作为输入，询问代理关于'Bocchi the Rock!'的信息  
agent.run("What is 'Bocchi the Rock!'?")

#### 缓存LLM

和测试大语言模型具有一样效果的是缓存大语言模型，通过缓存层可以尽可能的减少API的调用次数，从而节省费用。

在Langchain中设置缓存分为两种情况：一是在内存中设置缓存，二是在数据中设置缓存。存储在内存中加载速度较快，但是占用资源并且在关机之后将不再被缓存。

# 第一次调用时：Predict method took 3.4563 seconds to execute.  
# 第二次调用时：Predict method took 0.0007 seconds to execute.  
  
from langchain.cache import InMemoryCache  
import langchain  
from langchain.llms import OpenAI  
import time  
  
langchain.llm\_cache = InMemoryCache()  
  
llm = ChatOpenAI(model\_name="gpt-3.5-turbo")  
  
start\_time = time.time() # 记录开始时间  
print(llm.predict("用中文讲个笑话"))  
end\_time = time.time() # 记录结束时间  
elapsed\_time = end\_time - start\_time # 计算总时间  
print(f"Predict method took {elapsed\_time:.4f} seconds to execute.")  
  
start\_time = time.time() # 记录开始时间  
print(llm.predict("用中文讲个笑话"))  
end\_time = time.time() # 记录结束时间  
elapsed\_time = end\_time - start\_time # 计算总时间  
print(f"Predict method took {elapsed\_time:.4f} seconds to execute.")

有一天，一只小猪走进了一个餐厅，坐下后对服务员说：“我想点菜。”   
服务员奇怪地问：“猪怎么会在这里点菜？”   
小猪自信地回答：“因为我有‘猪’的品味！”   
  
服务员忍不住笑了：“那你想吃什么？”   
小猪眨了眨眼睛，说：“我想要一个‘不倒翁’的沙拉！”   
  
服务员愣了一下：“为什么要点这个？”   
小猪说：“因为它有‘摇摇’的感觉，和我一样可爱！”   
  
感觉这只小猪真是个大幽默家！  
Predict method took 3.4563 seconds to execute.  
有一天，一只小猪走进了一个餐厅，坐下后对服务员说：“我想点菜。”   
服务员奇怪地问：“猪怎么会在这里点菜？”   
小猪自信地回答：“因为我有‘猪’的品味！”   
  
服务员忍不住笑了：“那你想吃什么？”   
小猪眨了眨眼睛，说：“我想要一个‘不倒翁’的沙拉！”   
  
服务员愣了一下：“为什么要点这个？”   
小猪说：“因为它有‘摇摇’的感觉，和我一样可爱！”   
  
感觉这只小猪真是个大幽默家！  
Predict method took 0.0007 seconds to execute.

除了存储在内存中进行缓存，也可以存储在数据库中进行缓存，当开发企业级应用的时候通常都会选择存储在数据库中。

这种方式的加载速度相较于将缓存存储在内存中更慢一些，好处是不占电脑资源，并且存储记录并不会随着关机消失。

# 第一次调用时：Predict method took 2.9873 seconds to execute.  
# 第二次调用时：Predict method took 0.0028 seconds to execute.  
  
from langchain.cache import SQLiteCache  
import langchain  
from langchain.llms import OpenAI  
import time  
  
langchain.llm\_cache = SQLiteCache(database\_path="langchain.db")  
  
llm = ChatOpenAI(model\_name="gpt-3.5-turbo")  
  
start\_time = time.time() # 记录开始时间  
print(llm.predict("用中文讲个笑话"))  
end\_time = time.time() # 记录结束时间  
elapsed\_time = end\_time - start\_time # 计算总时间  
print(f"Predict method took {elapsed\_time:.4f} seconds to execute.")  
  
start\_time = time.time() # 记录开始时间  
print(llm.predict("用中文讲个笑话"))  
end\_time = time.time() # 记录结束时间  
elapsed\_time = end\_time - start\_time # 计算总时间  
print(f"Predict method took {elapsed\_time:.4f} seconds to execute.")

有一天，小明去上学，老师问他：“小明，如果你有十个苹果，给了别人四个，你还剩几个苹果？”   
小明想了想，回答：“老师，我还剩下六个！”   
老师说：“不对啊，应该是六个。你为什么会那么确定？”   
小明自信地说：“因为我苹果太多了，我从来没试过给人家苹果！”   
  
哈哈，看来小明的“数学”逻辑真是与众不同呀！  
Predict method took 2.9873 seconds to execute.  
有一天，小明去上学，老师问他：“小明，如果你有十个苹果，给了别人四个，你还剩几个苹果？”   
小明想了想，回答：“老师，我还剩下六个！”   
老师说：“不对啊，应该是六个。你为什么会那么确定？”   
小明自信地说：“因为我苹果太多了，我从来没试过给人家苹果！”   
  
哈哈，看来小明的“数学”逻辑真是与众不同呀！  
Predict method took 0.0028 seconds to execute.

#### 跟踪token使用情况

利用get\_openai\_callback可完成对于单条的提问时token的记录，此外对于有多个步骤的链或者agent，langchain也可以追踪到各步骤所耗费的token。

from langchain.agents import load\_tools  
from langchain.agents import initialize\_agent  
from langchain.agents import AgentType  
from langchain.llms import OpenAI  
from langchain.callbacks import get\_openai\_callback  
  
llm = OpenAI(temperature=0)  
tools = load\_tools(["llm-math"], llm=llm)  
agent = initialize\_agent(  
 tools, llm, agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION, verbose=True  
)  
  
with get\_openai\_callback() as cb:  
 result = llm("讲个笑话")  
 print(cb)  
  
with get\_openai\_callback() as cb:  
 response = agent.run("王菲现在的年龄是多少？")  
 print(f"Total Tokens: {cb.total\_tokens}")  
 print(f"Prompt Tokens: {cb.prompt\_tokens}")  
 print(f"Completion Tokens: {cb.completion\_tokens}")  
 print(f"Total Cost (USD): ${cb.total\_cost}")

Tokens Used: 144  
 Prompt Tokens: 5  
 Prompt Tokens Cached: 0  
 Completion Tokens: 139  
 Reasoning Tokens: 0  
Successful Requests: 1  
Total Cost (USD): $0.00028550000000000005   
  
[1m> Entering new AgentExecutor chain...[0m  
[32;1m[1;3m I need to find out how old Wang Fei is now.  
Action: Calculator  
Action Input: 2021 - 1973[0m  
Observation: [36;1m[1;3mAnswer: 48[0m  
Thought:[32;1m[1;3m I now know the final answer  
Final Answer: Wang Fei is 48 years old.[0m  
  
[1m> Finished chain.[0m  
Total Tokens: 724  
Prompt Tokens: 653  
Completion Tokens: 71  
Total Cost (USD): $0.0011215

#### 序列化配置LLM

Langchain也提供一种能力用来保存LLM在训练时使用的各类系数，比如template、 model\_name等。

这类系数通常会被保存在json或者yaml文件中，以json文件为例，配置如下系数，然后利用load\_llm方法即可导入：

'''  
llm.json内容：  
{  
 "model\_name": "gpt-turbo-3.5",  
 "temperature": 0.7,  
 "\_type": "openai"  
}  
'''  
from langchain.llms.loading import load\_llm  
  
llm = load\_llm("llm.json")

### 流式处理LLM的响应：

流式处理意味着**在接收到第一个数据块后就立即开始处理，而不需要等待整个数据包传输完毕**。

这种概念应用在LLM中则可达到**生成响应时就立刻向用户展示此下的响应，或者在生成响应时处理响应**，也就是我们现在看到的**和ai对话时逐字输出的效果**

可以看到实现还是较为方便的只需要直接调用StreamingStdOutCallbackHandler作为callback即可。

from langchain.llms import OpenAI  
from langchain.callbacks.streaming\_stdout import StreamingStdOutCallbackHandler  
  
llm = OpenAI(streaming=True, callbacks=[StreamingStdOutCallbackHandler()], temperature=0)  
resp = llm("Write me a song about sparkling water.")

Verse 1:  
Bubbles dancing in my glass  
Clear and crisp, it's such a blast  
Refreshing taste, it's like a dream  
Sparkling water, you make me beam  
   
Chorus:  
Oh sparkling water, you're my delight  
With every sip, you make me feel so right  
You're like a party in my mouth  
I can't get enough, I'm hooked no doubt  
  
Verse 2:  
No sugar, no calories, just pure bliss  
You're the perfect drink, I must confess  
From lemon to lime, so many flavors to choose  
Sparkling water, you never fail to amuse  
  
Chorus:  
Oh sparkling water, you're my delight  
With every sip, you make me feel so right  
You're like a party in my mouth  
I can't get enough, I'm hooked no doubt  
  
Bridge:  
Some may say you're just plain water  
But to me, you're so much more  
You bring a sparkle to my day  
In every single way  
  
Chorus:  
Oh sparkling water, you're my delight  
With every sip, you make me feel so right  
You're like a party in my mouth  
I can't get enough, I'm hooked no doubt  
  
Outro:  
So

### Output Parser

Model返回的内容通常都是字符串的模式，但在实际开发过程中，往往希望model可以返回更直观的内容，Langchain提供的输出解析器则将派上用场。

在实现一个输出解析器的过程中，需要实现两种方法：

* 获取格式指令：返回一个字符串的方法，其中包含有关如何格式化语言模型输出的说明。
* Parse：一种接收字符串（假设是来自语言模型的响应）并将其解析为某种结构的方法。

# ===== 列表解析器示例：让 LLM 返回逗号分隔的列表 =====  
# 需求：让模型输出 5 种冰淇淋口味，并直接解析成 Python 列表  
  
# 1) 导入工具 --------------------  
from langchain.output\_parsers import CommaSeparatedListOutputParser # 现成的“逗号列表”解析器  
from langchain.prompts import PromptTemplate # 用于构造提示词  
from langchain.llms import OpenAI # OpenAI 文本模型  
  
# 2) 初始化解析器 -----------------  
output\_parser = CommaSeparatedListOutputParser() # 实例化  
format\_instructions = output\_parser.get\_format\_instructions() # 生成“请用逗号分隔”说明  
  
# 3) 构造 Prompt -----------------  
prompt = PromptTemplate(  
 template="List five {subject}.\n{format\_instructions}", # 指令 + 格式要求  
 input\_variables=["subject"], # 运行时需提供 subject  
 partial\_variables={"format\_instructions": format\_instructions}, # 预填格式指令  
)  
  
# 4) 调用 LLM --------------------  
llm = OpenAI(temperature=0) # 温度 0：输出更确定  
\_input = prompt.format(subject="冰淇淋口味") # 生成最终 prompt  
raw\_out = llm(\_input) # 获得原始字符串输出  
  
# 5) 解析为 Python 列表 -----------  
flavors = output\_parser.parse(raw\_out)  
print(flavors)

['巧克力', '香草', '草莓', '抹茶', '芒果']

# ========= 日期解析器：DatetimeOutputParser =========  
# 作用：让模型输出的日期/时间直接变成 datetime 对象，免去手动解析  
  
from langchain.prompts import PromptTemplate  
from langchain.output\_parsers import DatetimeOutputParser  
from langchain.chains import LLMChain  
from langchain.llms import OpenAI  
  
# 1) 实例化解析器  
date\_parser = DatetimeOutputParser()   
  
# 2) 准备 Prompt，并注入“格式指令”（要求模型输出可解析的日期格式）  
template = """回答用户的问题:  
{question}  
{format\_instructions}""" # {format\_instructions} 会被替换  
  
prompt = PromptTemplate.from\_template(  
 template,  
 partial\_variables={  
 "format\_instructions": date\_parser.get\_format\_instructions()  
 },  
)  
  
# 3) 组装为链  
chain = LLMChain(prompt=prompt, llm=OpenAI())  
  
# 4) 运行并解析  
raw = chain.run("bitcoin是什么时候成立的？用英文格式输出时间") # 模型返回日期字符串  
dt = date\_parser.parse(raw) # 转成 datetime.datetime  
print(dt.date()) # 2009-01-03 之类

2009-01-03

# ========= 枚举解析器：EnumOutputParser =========  
# 作用：强制模型输出枚举成员之一，解析后得到 Enum 对象  
  
from langchain.output\_parsers.enum import EnumOutputParser  
from enum import Enum  
  
class Colors(Enum):  
 RED = "red"  
 GREEN = "green"  
 BLUE = "blue"  
  
enum\_parser = EnumOutputParser(enum=Colors) # 若输出非 red/green/blue 将报错

# ========= PydanticOutputParser 示例 =========  
# 目标：把 LLM 返回的 JSON 字符串解析成 Actor 数据类  
# 同时演示当格式错误时，Pydantic 解析器会抛出异常  
  
from langchain\_core.prompt\_values import StringPromptValue  
from langchain.prompts import PromptTemplate  
from langchain.llms import OpenAI  
from langchain.output\_parsers import PydanticOutputParser, RetryWithErrorOutputParser  
from pydantic import BaseModel, Field  
from typing import List  
  
# 1) 定义数据结构 -----------------------------------  
class Actor(BaseModel):  
 """描述演员及其作品列表的 Pydantic 模型"""  
 name: str = Field(description="name of an actor")  
 film\_names: List[str] = Field(description="films the actor starred in")  
  
# 2) 用 Actor 初始化解析器 ----------------------------  
parser = PydanticOutputParser(pydantic\_object=Actor)  
  
# 3) 模拟一个格式错误的响应（单引号 ➜ JSON 需双引号）----  
misformatted = "{'name': 'Tom Hanks', 'film\_names': ['Forrest Gump']}"  
formatted = "{\"name\": \"Tom Hanks\", \"film\_names\": [\"Forrest Gump\"]}"  
  
# 4) 尝试解析；若格式不合法将抛出 ValidationError -------  
try:  
 actor\_obj = parser.parse(misformatted)  
 print(actor\_obj)  
except Exception as e:  
 print("解析失败，错误信息：", e)  
 # 实际项目中，可在此捕获后用 RetryWithErrorOutputParser 进行自动修复  
  
try:  
 actor\_obj = parser.parse(formatted)  
 print(actor\_obj)  
except Exception as e:  
 print("解析失败，错误信息：", e)  
 # 实际项目中，可在此捕获后用 RetryWithErrorOutputParser 进行自动修复

解析失败，错误信息： Invalid json output: {'name': 'Tom Hanks', 'film\_names': ['Forrest Gump']}  
For troubleshooting, visit: https://python.langchain.com/docs/troubleshooting/errors/OUTPUT\_PARSING\_FAILURE   
name='Tom Hanks' film\_names=['Forrest Gump']