第02章：LangChain使用之Model I/O

讲师：尚硅谷-宋红康

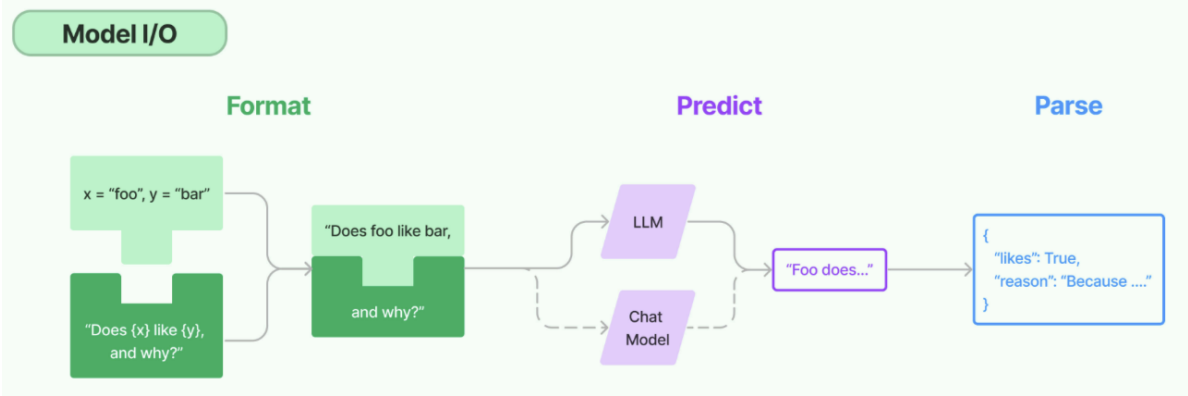
官网：尚硅谷

1、Model I/O介绍

Model I/O 模块是与语言模型（LLMs）进行交互的**核心组件**，在整个框架中有着很重要的地位。

所谓的Model I/O，包括输入提示(Format)、调用模型(Predict)、输出解析(Parse)。分别对应着**Prompt Template**， **Model** 和 **Output Parser**。

> 简单来说，就是输入、模型处理、输出这三个步骤。



针对每个环节，LangChain都提供了模板和工具，可以快捷的调用各种语言模型的接口。

2、Model I/O之调用模型1

LangChain作为一个“工具”，不提供任何 LLMs，而是依赖于第三方集成各种大模型。比如，将OpenAI、Anthropic、Hugging Face 、LlaMA、阿里Qwen、ChatGLM等平台的模型无缝接入到你的应用。

2.1 模型的不同分类方式

> 简单来说，就是用谁家的API以什么方式调用哪种类型的大模型

角度1：按照模型功能的不同：

- 非对话模型（LLMs、Text Model）
- 对话模型（Chat Models）（**推荐**）
- 嵌入模型（Embedding Models）（**暂不考虑**）

角度2：模型调用时，几个重要参数的书写位置的不同：

- 硬编码：写在代码文件中

- 使用环境变量
- 使用配置文件（推荐）

角度3：具体调用的API

- OpenAI提供的API
- 其它大模型自家提供的API
- LangChain的统一方式调用API（推荐）

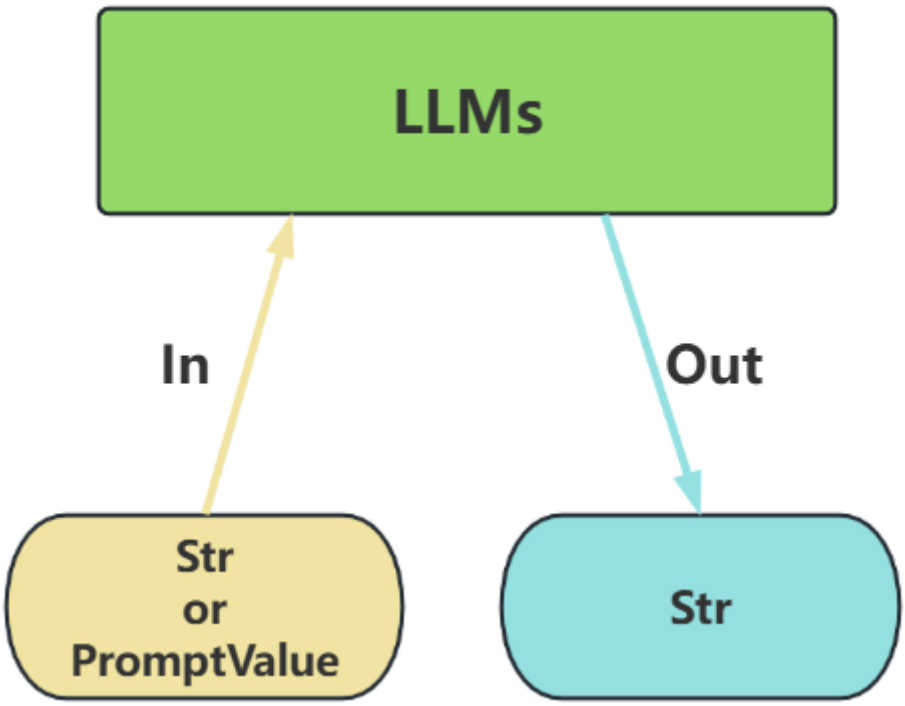
> 背景小知识：
>
> OpenAI的GPT系列模型影响了大模型技术发展的开发范式和标准。所以无论是Qwen、ChatGLM等模型，它们的使用方法和函数调用逻辑基本**遵循OpenAI定义的规范**，没有太大差异。这就使得大部分的开源项目能够通过一个较为通用的接口来接入和使用不同的模型。

2.2 角度1出发：按照功能不同举例

类型1：LLMs(非对话模型)

LLMs，也叫Text Model、非对话模型，是许多语言模型应用程序的支柱。主要特点如下：

- **输入**：接受**文本字符串**或**PromptValue**对象
- **输出**：总是返回**文本字符串**



- **适用场景**：仅需单次文本生成任务（如摘要生成、翻译、代码生成、单次问答）或对接不支持消息结构的旧模型（如部分本地部署模型）（**言外之意，优先推荐ChatModel**）
- **不支持多轮对话上下文。**每次调用独立处理输入，无法自动关联历史对话（需手动拼接历史文本）。
- **局限性**：无法处理角色分工或复杂对话逻辑。

举例：

```python
import os
import dotenv
from langchain_openai import OpenAI
dotenv.load_dotenv()
os.environ["OPENAI_API_KEY"] = os.getenv("OPENAI_API_KEY1")
os.environ["OPENAI_BASE_URL"] = os.getenv("OPENAI_BASE_URL")


############核心代码#############
llm = OpenAI()
str = llm.invoke("写一首关于春天的诗")  # 直接输入字符串
print(str)
```

```
春风拂过大地
万物复苏欣欣向荣
花开满山谷
鸟儿啁啾歌声飘
春雨滋润田野
农夫欢呼收获丰
春日暖阳温柔
人们心情也随之舒
春天来了
带来新的开始
带走冬日的寒冷
带来生机与希望
春天啊，美丽的季节
让我们一起欢庆
迎接春天的到来
让快乐永远驻留心中。
```

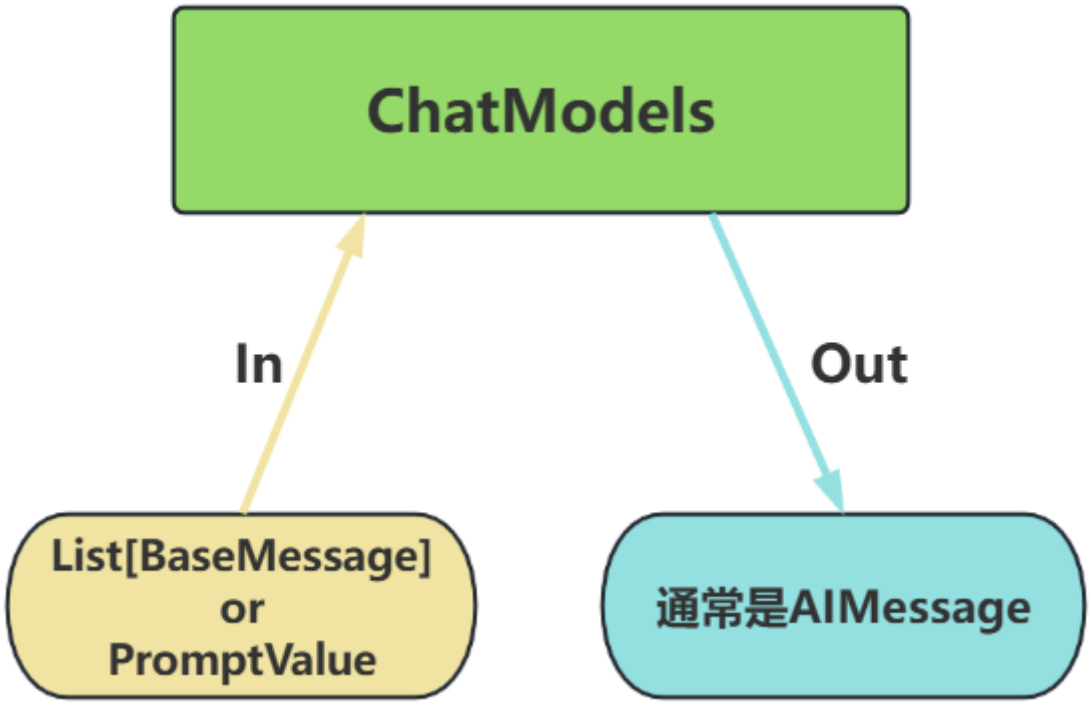
类型2：Chat Models(对话模型)

ChatModels，也叫聊天模型、对话模型，底层使用LLMs。

大语言模型调用，以 ChatModel 为主！

主要特点如下：

- **输入**：接收消息列表 `List[BaseMessage]` 或 `PromptValue`，每条消息需指定角色（如 SystemMessage、HumanMessage、AIMessage）
- **输出**：总是返回带角色的 `消息对象`（`BaseMessage` 子类），通常是 `AIMessage`



- **原生支持多轮对话。**通过消息列表维护上下文（例如：`[SystemMessage, HumanMessage, AIMessage, ...]`），模型可基于完整对话历史生成回复。
- **适用场景**：对话系统（如客服机器人、长期交互的AI助手）

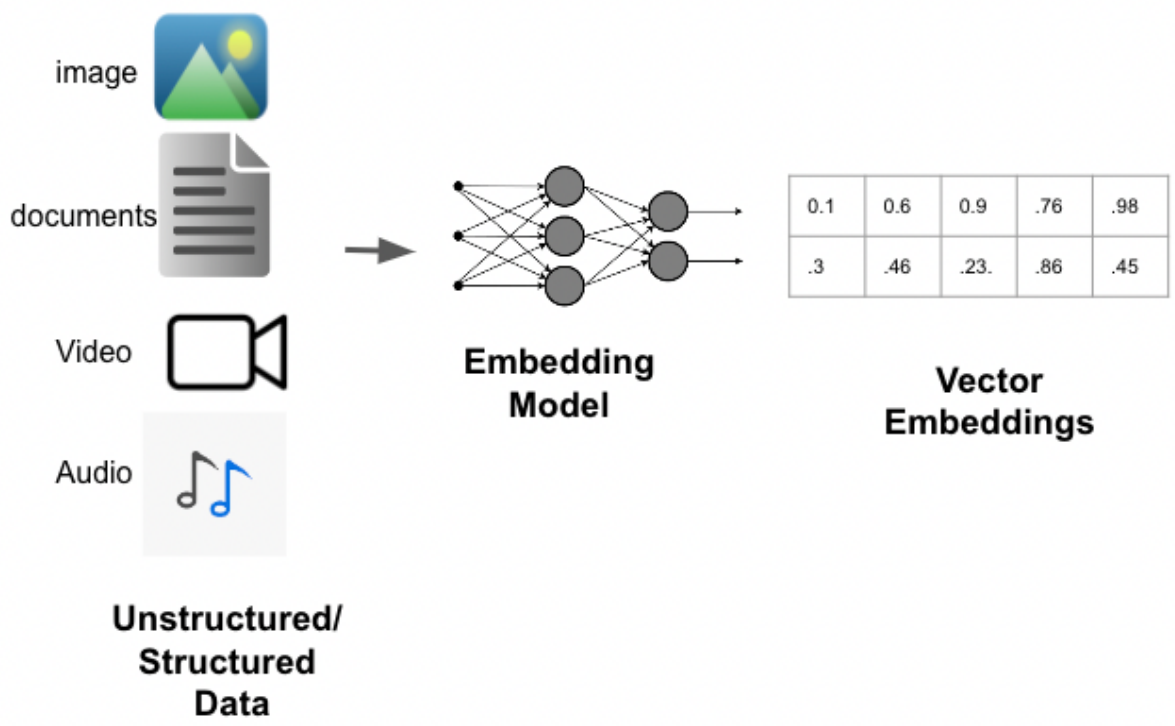
举例：

```python
from langchain_openai import ChatOpenAI
from langchain_core.messages import SystemMessage, HumanMessage
import os
import dotenv
dotenv.load_dotenv()
os.environ["OPENAI_API_KEY"] = os.getenv("OPENAI_API_KEY1")
os.environ["OPENAI_BASE_URL"] = os.getenv("OPENAI_BASE_URL")


########核心代码############
chat_model = ChatOpenAI(model="gpt-4o-mini")

messages = [
    SystemMessage(content="我是人工智能助手，我叫小智"),
    HumanMessage(content="你好，我是小明，很高兴认识你")
]
response = chat_model.invoke(messages)  # 输入消息列表

print(type(response))  # <class 'langchain_core.messages.ai.AIMessage'>
print(response.content)
```

```
<class 'langchain_core.messages.ai.AIMessage'>
你好，小明！很高兴认识你，有什么我可以帮你的吗？
```

类型3：Embedding Model(嵌入模型)

Embedding Model：也叫文本嵌入模型，这些模型将 **文本** 作为输入并返回 **浮点数列表**，也就是Embedding。（后面章节《07-LangChain使用之Retrieval》重点讲）




```python
import os
import dotenv
from langchain_openai import OpenAIEmbeddings
dotenv.load_dotenv()
os.environ['OPENAI_API_KEY'] = os.getenv("OPENAI_API_KEY1")
os.environ['OPENAI_BASE_URL'] = os.getenv("OPENAI_BASE_URL")


##############
embeddings_model = OpenAIEmbeddings(
    model="text-embedding-ada-002"
)

res1 = embeddings_model.embed_query('我是文档中的数据')
print(res1)
# 打印结果：[-0.004306625574827194, 0.003083756659179926, -0.013916781172156334, ...., ]
```

2.3 角度2出发：参数位置不同举例

这里以 **LangChain** 的API为准，使用对话模型，进行测试。

2.3.1模型调用的主要方法及参数

相关方法及属性：

- **OpenAI(...) / ChatOpenAI(...)**：创建一个模型对象（非对话类/对话类）
- **model.invoke(xxx)**：执行调用，将用户输入发送给模型
- **.content**：提取模型返回的实际文本内容

模型调用函数使用时需初始化模型，并设置必要的参数。

1）必须设置的参数：

- `base_url`：大模型 API 服务的根地址
- `api_key`：用于身份验证的密钥，由大模型服务商（如 OpenAI、百度千帆）提供
- `model/model_name`：指定要调用的具体大模型名称（如 `gpt-4-turbo`、`ERNIE-3.5-8K` 等）

2）其它参数：

- `temperature`：温度，控制生成文本的“随机性”，取值范围为0～1。
  - `值越低` → 输出越确定、保守（适合事实回答）
  - `值越高` → 输出越多样、有创意（适合创意写作）

  通常，根据需要设置如下：
  - 精确模式（0.5或更低）：生成的文本更加安全可靠，但可能缺乏创意和多样性。
  - 平衡模式（通常是0.8）：生成的文本通常既有一定的多样性，又能保持较好的连贯性和准确性。
  - 创意模式（通常是1）：生成的文本更有创意，但也更容易出现语法错误或不合逻辑的内容。
- `max_tokens`：限制生成文本的最大长度，防止输出过长。

  Token是什么?

  `基本单位`：大模型处理文本的最小单位是token（相当于自然语言中的词或字），输出时逐个token依次生成。

  `收费依据`：大语言模型(LLM)通常也是以token的数量作为其计量(或收费)的依据。
  - 1个Token≈1-1.8个汉字，1个Token≈3-4个英文字母
  - Token与字符转化的可视化工具：
    - OpenAI提供：https://platform.openai.com/tokenizer
    - 百度智能云提供：https://console.bce.baidu.com/support/#/tokenizer

  max_tokens设置建议：
  - 客服短回复：128-256。比如：生成一句客服回复（如“订单已发货，预计明天送达”）
  - 常规对话、多轮对话：512-1024
  - 长内容生成：1024-4096。比如：生成一篇产品说明书（包含功能、使用方法等结构）

2.3.2 模型调用推荐平台：closeai

这里推荐使用的平台：

考虑到OpenAI等模型在国内访问及充值的不便，大家可以使用CloseAI网站注册和充值，`具体费用自理`。

https://www.closeai-asia.com

2.3.3 方式1：硬编码

直接将 API Key 和模型参数写入代码，**仅适用于临时测试**，存在密钥泄露风险，在`生产环境不推荐`。

```python
from langchain_openai import ChatOpenAI

# 硬编码 API Key 和模型参数
llm = ChatOpenAI(
    api_key="sk-xxxxxxxxx",  # 明文暴露密钥
    base_url="https://api.openai-proxy.org/v1",
    model="gpt-3.5-turbo",
)

# 调用示例
response = llm.invoke("解释神经网络原理")
print(response.content)
```

神经网络是一种模拟人脑神经元之间信息传递的数学模型。神经网络由多层神经元组成，每个神经元接收来自上一层神经元的输入信号，并通过激活函数将输入信号进行加权求和并输出一个结果。

在神经网络中，数据通过输入层传递到隐藏层，最终到输出层。在每一层中，神经元通过学习算法不断调整连接权重，以使神经网络能够准确地对输入数据进行分类或预测。

神经网络的训练过程是通过反向传播算法进行的。该算法通过计算神经网络输出结果与实际结果之间的误差，然后根据误差调整神经元之间的连接权重，以提高神经网络的准确性。

总的来说，神经网络通过模拟人脑神经元之间的信息传递过程，利用数学模型来实现对复杂问题的学习和预测。神经网络的原理是基于神经元之间的连接权重调整和误差反向传播的机制。

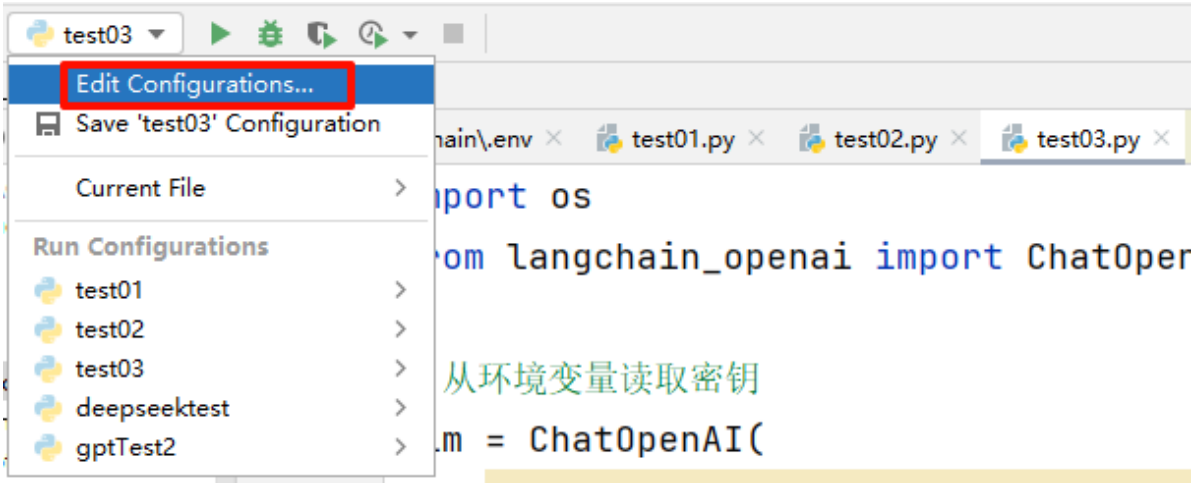
2.3.4 方式2：配置环境变量

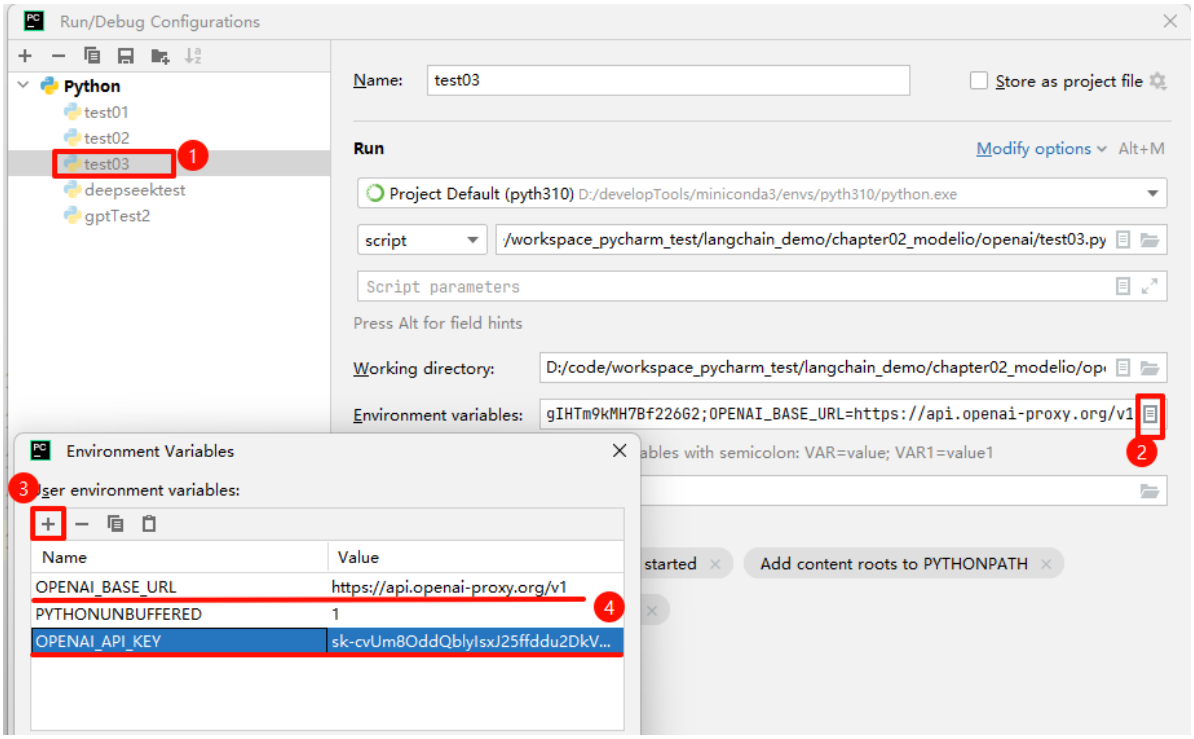
通过系统环境变量存储密钥，避免代码明文暴露。

方式1：终端设置环境变量（临时生效）：

```bash
export OPENAI_API_KEY="sk-xxxxxxxxxxxxxxxxxxxx"  # Linux/Mac
set OPENAI_API_KEY="sk-xxxxxxxxxxxxxxxxxxxx"    # Windows
```

方式2：从PyCharm设置





举例：

```python
import os
from langchain_openai import ChatOpenAI

# 从环境变量读取密钥
llm = ChatOpenAI(
    api_key=os.environ["OPENAI_API_KEY"], # 动态获取
    base_url=os.environ["OPENAI_BASE_URL"],
    model="gpt-4o-mini",
)

response = llm.invoke("LangChain 是什么？")
print(response.content)
```

输出：略

优点：密钥与代码分离，适合单机开发

缺点：切换终端或文件后，变量失效，需重新设置。

2.3.5 方式3：使用.env配置文件

使用 `python-dotenv` 加载本地配置文件，支持多环境管理（开发/生产）。

1）安装依赖

```
pip install python-dotenv

#或者

conda install python-dotenv
```

2）创建 `.env` 文件（项目根目录）：

```
OPENAI_API_KEY="sk-xxxxxxxxx" # 需填写自己的API KEY
OPENAI_BASE_URL="https://api.openai-proxy.org/v1"
```

3）举例

方式1

```
from dotenv import load_dotenv
from langchain_openai import ChatOpenAI
import os

load_dotenv() # 自动加载 .env 文件

# print(os.getenv("OPENAI_API_KEY"))
# print(os.getenv("OPENAI_BASE_URL"))

llm = ChatOpenAI(
    api_key=os.getenv("OPENAI_API_KEY"), # 安全读取
    base_url=os.getenv("OPENAI_BASE_URL"),
    model="gpt-4o-mini",
    temperature=0.7,
)

response = llm.invoke("RAG 技术的核心流程")
print(response.content)
```

```
输出：略
```

方式2：给os内部的环境变量赋值

```
from langchain_openai import ChatOpenAI
import dotenv
dotenv.load_dotenv()

import os

os.environ["OPENAI_API_KEY"] = os.getenv("OPENAI_API_KEY1")
os.environ["OPENAI_API_BASE"] = os.getenv("OPENAI_BASE_URL")

text = "猫王是猫吗？"

chat_model = ChatOpenAI(
    model="gpt-4o-mini",
    temperature=0.7,
    max_tokens=300,

)
response = chat_model.invoke(text)
print(type(response))
print(response.content)
```

<class 'langchain_core.messages.ai.AIMessage'>
“猫王”是对美国著名歌手埃尔维斯·普雷斯利（Elvis Presley）的昵称，他被称为“猫王”是因为他的音乐风格和独特的魅力，而不是因为他是一只猫。埃尔维斯是摇滚乐的传奇人物，对音乐和文化产生了深远的影响。

核心优势：

- 配置文件可加入 .gitignore 避免泄露
- 结合 LangChain 可扩展其它模型（如 DeepSeek、阿里云）
- 生产环境推荐

小结：

方式	安全性	持久性	适用场景
硬编码	⚠ 低	❌ 临时	本地快速测试
环境变量	☑ 中	⚠ 会话级	短期开发调试
`.env` 配置文件	☑☑ 高	☑ 永久	生产环境、团队协作

以上3种方式，适合于所有的LLM的获取。

2.4 角度3出发：各平台API的调用举例(了解)

2.4.1 OpenAI 官方API

考虑到OpenAI在国内访问及充值的不便，我们仍然使用CloseAI网站（https://www.closeai-asia.com）注册和充值，**具体费用自理**。

调用非对话模型：

```
from openai import OpenAI

# 从环境变量读取API密钥（推荐安全存储）
client = OpenAI(api_key="sk-zD4CB2Qe7G2Dp6veCfPRSxeDx9fQPxCUIfOFAk20ETV5B2VA", # 填写自己的api-key
                base_url="https://api.openai-proxy.org/v1") #通过代码示例获取

# 调用Completion接口
response = client.completions.create(
    model="gpt-3.5-turbo-instruct", # 非对话模型
    prompt="请将以下英文翻译成中文：\n'Artificial intelligence will reshape the future.'",
    max_tokens=100, # 生成文本最大长度
    temperature=0.7, # 控制随机性
)
# 提取结果
print(response.choices[0].text.strip())
```

调用对话模型：

写法1：

```python
from openai import OpenAI

client = OpenAI(api_key="sk-zD4CB2Qe7G2Dp6veCfPRSxeDx9fQPxCUIfOFAk20ETV5B2VA", #填写自己的api-key
                base_url="https://api.openai-proxy.org/v1")

completion = client.chat.completions.create(
    model="gpt-3.5-turbo", # 对话模型
    messages=[
        {"role": "system", "content": "你是一个乐于助人的智能AI小助手"},
        {"role": "user", "content": "你好，请你介绍一下你自己"}
    ],
    max_tokens=150,
    temperature=0.5
)

print(completion.choices[0].message)
```

```
ChatCompletionMessage(content='你好，我是一个智能AI助手，可以回答各种问题、提供信息和建议。无论是日常生活中的疑问，还是学习工', refusal=None, role='assistant', annotations=None, audio=None, function_call=None, tool_calls=None)
```

写法2：

```python
from openai import OpenAI
import os
import dotenv
dotenv.load_dotenv()
os.environ["OPENAI_API_KEY"] = os.getenv("OPENAI_API_KEY1")
os.environ["OPENAI_BASE_URL"] = os.getenv("OPENAI_BASE_URL")


client = OpenAI()
response = client.chat.completions.create(
    model="gpt-3.5-turbo",
    messages=[
        {"role": "system", "content": "我是一位乐于助人的AI智能小助手"},
        {"role": "user", "content": "你好，请你介绍一下你自己。"}
    ]
)

print(response.choices[0])
```

2.4.2 百度千帆平台

开发参考文档：

https://cloud.baidu.com/doc/qianfan-docs/s/Mm8r1mejk

获取API Key和App ID：

创建API Key：https://console.bce.baidu.com/qianfan/ais/console/apiKey

创建App ID：https://console.bce.baidu.com/qianfan/ais/console/applicationConsole/application/v2

代码举例：

```
from openai import OpenAI

client = OpenAI(
    api_key="bce-v3/ALTAK-KZke********/f1d6ee**********",  # 千帆bearer token
    base_url="https://qianfan.baidubce.com/v2",  # 千帆域名
    default_headers={"appid": "app-MuYR79q6"}  # 用户在千帆上的appid，非必传
)

completion = client.chat.completions.create(
    model="ernie-4.0-turbo-8k", # 预置服务请查看模型列表，定制服务请填入API地址
    messages=[{'role': 'system', 'content': 'You are a helpful assistant.'},
            {'role': 'user', 'content': 'Hello!'}]
)

print(completion.choices[0].message)
```

2.4.3 阿里云百炼平台

注册与key的获取：

提前开通百炼平台账号并申请API KEY：https://bailian.console.aliyun.com/#/home

对应的配置文件：

```
DASHSCOPE_API_KEY= "sk-f1a87324#####e6a819a482"  #使用自己的api key
DASHSCOPE_BASE_URL= "https://dashscope.aliyuncs.com/compatible-mode/v1"
```

模型的调用：

参考具体模型的代码示例。这里以DeepSeek为例：

举例1：通过OpenAI SDK

```
pip install openai
```

```
import os
from openai import OpenAI

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="sk-xxx",
    api_key=os.getenv("DASHSCOPE_API_KEY"), # 如何获取API Key:https://help.aliyun.com/zh/model-studio/developer-reference/get-api-key
    base_url=os.getenv("DASHSCOPE_BASE_URL")
)

completion = client.chat.completions.create(
    model="deepseek-r1", # 此处以 deepseek-r1 为例，可按需更换模型名称。
    messages=[
        {'role': 'user', 'content': '9.9和9.11谁大'}
    ]
)

# 通过reasoning_content字段打印思考过程
print("思考过程：")
print(completion.choices[0].message.reasoning_content)

# 通过content字段打印最终答案
print("最终答案：")
print(completion.choices[0].message.content)
```

举例2：通过DashScope SDK

```
pip install dashscope
```

```
import os
import dashscope

messages = [
    {'role': 'user', 'content': '你是谁？'}
]

response = dashscope.Generation.call(
    # 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="sk-xxx",
    api_key=os.getenv('DASHSCOPE_API_KEY'),
    model="deepseek-r1", # 此处以 deepseek-r1 为例，可按需更换模型名称。
    messages=messages,
    # result_format参数不可以设置为"text"。
    result_format='message'
)

print("="*20 + "思考过程" + "="*20)
print(response.output.choices[0].message.reasoning_content)
print("="*20 + "最终答案" + "="*20)
print(response.output.choices[0].message.content)
```

2.4.4 智谱的GLM

注册智谱模型并获取API Key：

https://www.bigmodel.cn/usercenter/proj-mgmt/apikeys


```
#记录自己的api key，声明在.env文件中
ZHIPUAI_API_KEY= "63a0f275b3a9################rA4Y8daGaLydxQ"
```

查看《开发文档》

或者选择如下《参考文档》皆可：

https://www.bigmodel.cn/dev/api/normal-model/glm-4

代码举例

举例1：使用OpenAI SDK

```python
from openai import OpenAI

client = OpenAI(
    api_key=os.getenv("ZHIPUAI_API_KEY"),
    base_url=os.getenv("ZHIPUAI_URL")
)

completion = client.chat.completions.create(
    model="glm-4-air-250414",
    messages=[
        {"role": "system", "content": "你是一个聪明且富有创造力的小说作家"},
        {"role": "user", "content": "请你作为童话故事大王，写一篇短篇童话故事，故事的主题是要永远保持一颗善良的心，要能够激发儿童的学习兴趣和想象力，同时也能够帮助儿童更好地理解和接受故事中所蕴含的道理和价值观。"}
    ],
    top_p=0.7,
    temperature=0.9
)

print(completion.choices[0].message)
```

举例2：使用Langchain SDK

```python
import os
from langchain_openai import ChatOpenAI
from langchain.prompts import (
    ChatPromptTemplate,
    MessagesPlaceholder,
    SystemMessagePromptTemplate,
    HumanMessagePromptTemplate,
)
from langchain.chains import LLMChain
from langchain.memory import ConversationBufferMemory

llm = ChatOpenAI(
    temperature=0.95,
    model="glm-4-air-250414",
    openai_api_key= os.getenv("ZHIPUAI_API_KEY"),
    openai_api_base=os.getenv("ZHIPUAI_URL"),
)
prompt = ChatPromptTemplate(
    messages=[
        SystemMessagePromptTemplate.from_template(
            "You are a nice chatbot having a conversation with a human."
        ),
        MessagesPlaceholder(variable_name="chat_history"),
        HumanMessagePromptTemplate.from_template("{question}")
    ]
)

memory = ConversationBufferMemory(memory_key="chat_history", return_messages=True)
```

```
conversation = LLMChain(
    llm=llm,
    prompt=prompt,
    verbose=True,
    memory=memory
)
conversation.invoke({"question": "给我讲个冷笑话"})
```

举例3：参考langchain的文档

https://imooc-langchain.shortvar.com/docs/integrations/chat/zhipuai/

安装包：

```
pip install  langchain-community

pip install pyjwt
```

代码示例：

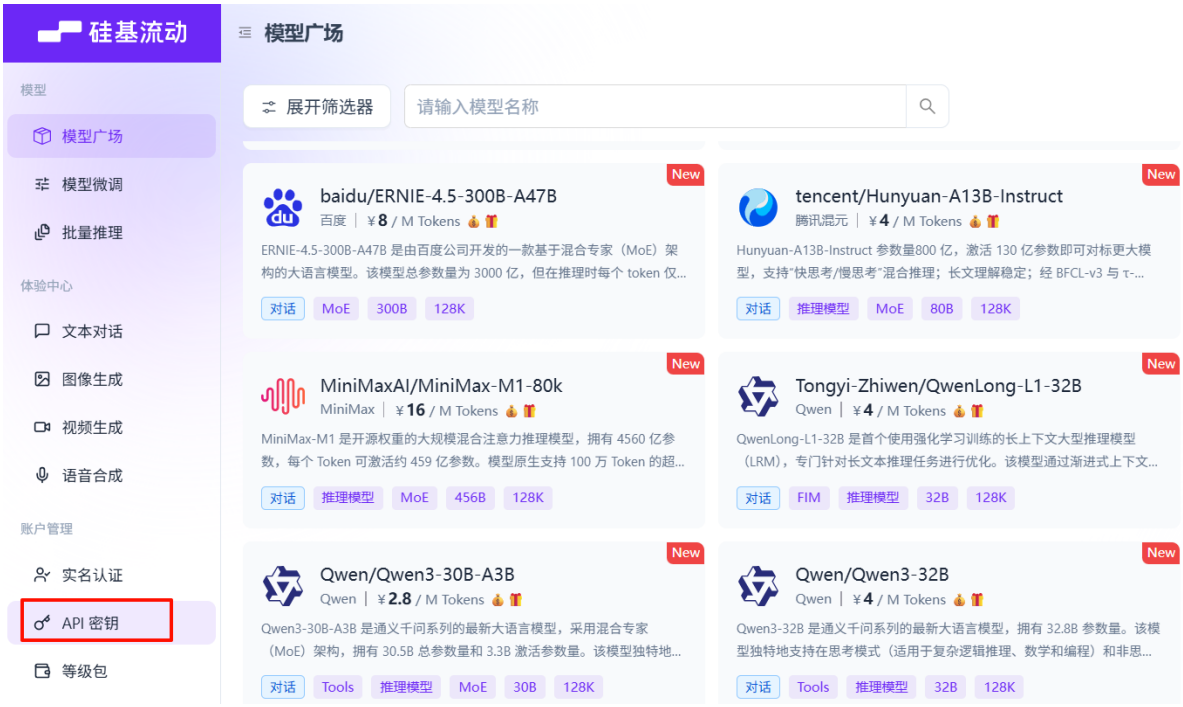
```
import dotenv
from langchain_community.chat_models import ChatZhipuAI
from langchain_core.messages import AIMessage, SystemMessage, HumanMessage

#智谱大模型：参考langchain的大模型

dotenv.load_dotenv()

import os
os.environ["ZHIPUAI_API_KEY"] = os.getenv("ZHIPUAI_API_KEY")

chat = ChatZhipuAI(
    model="glm-4",
    temperature=0.5,
)

messages = [
    AIMessage(content="哈罗~"),
    SystemMessage(content="你是一个诗人"),
    HumanMessage(content="写一首关于AI的七言绝句"),
]

response = chat.invoke(messages)
print(response.content)  # 显示 AI 生成的诗
```

```
智能助手显神通，
万物互联慧眼中。
编码世界藏诗意，
共融未来路无穷。
```

2.4.5 硅基流动平台

官网：https://www.siliconflow.cn/

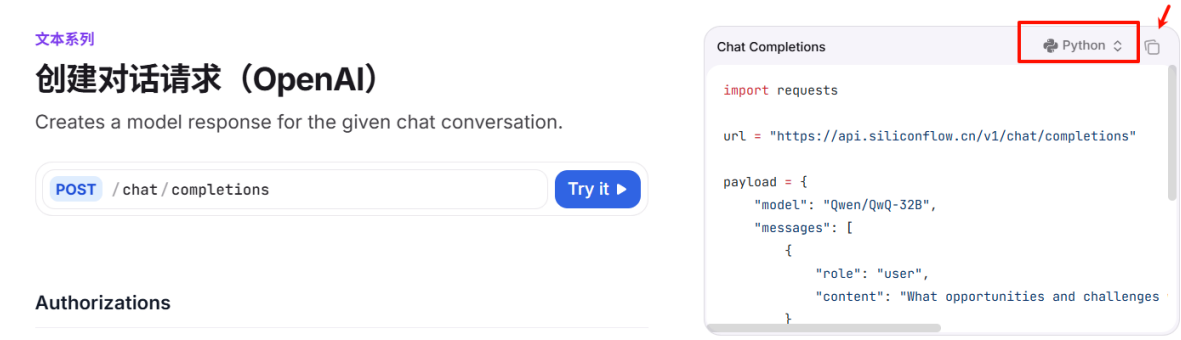
申请API Key：



参考文档：https://docs.siliconflow.cn/cn/userguide/quickstart

```python
from openai import OpenAI

client = OpenAI(api_key=os.getenv("SILICON_API_KEY"),
                base_url="https://api.siliconflow.cn/v1")
response = client.chat.completions.create(
    model='Pro/deepseek-ai/DeepSeek-R1',
    # model="Qwen/Qwen2.5-72B-Instruct",
    messages=[
        {'role': 'user',
         'content': "推理模型会给市场带来哪些新的机会"}
    ],
    stream=True
)

for chunk in response:
    if not chunk.choices:
        continue
    if chunk.choices[0].delta.content:
        print(chunk.choices[0].delta.content, end="", flush=True)
    if chunk.choices[0].delta.reasoning_content:
        print(chunk.choices[0].delta.reasoning_content, end="", flush=True)
```

或者：




```python
import requests

url = "https://api.siliconflow.cn/v1/chat/completions"

payload = {
    "model": "deepseek-ai/DeepSeek-R1", #填写你选择的大模型
    "messages": [
        {
            "role": "user",
            "content": "1 +2 * 3 = ? "
        }
    ]
}
headers = {
    "Authorization": "Bearer sk-auciaxqpz.....zepozralhwleyrdoyjani", #填写你的api-key
    "Content-Type": "application/json"
}

response = requests.post(url, json=payload, headers=headers)

print(response.json())
```

2.5 如何选择合适的大模型

2.5.1 有没有最好的大模型

凡是问「哪个大模型最好？」的，都是不懂的。

不妨反问：「无论做什么，有都表现最好的员工吗？」

划重点：**没有最好的大模型，只有最适合的大模型**

基础模型选型，合规和安全是首要考量因素！

为什么不要依赖榜单?

- 榜单已被应试教育污染，还算值得相信的榜单：LMSYS Chatbot Arena Leaderboard
- 榜单体现的是整体能力，放到一件具体事情上，排名低的可能反倒更好
- 榜单体现不出成本差异

本课程主要以OpenAI为例展开后续的课程。因为：

1、OpenAI 最流行，即便国内也是如此

2、OpenAI 最先进。别的模型有的能力，OpenAI一定都有。OpenAI有的，别的模型不一定有。

3、其它模型都在追赶和模仿OpenAI　gpt-4o-mini

> 学会OpenAI，其它模型触类旁通。反之，不一定

2.5.2 小结：获取大模型的标准方式

后续的各种模型测试，都基于如下的模型展开：

非对话模型：

```
import os
import dotenv
from langchain_openai import OpenAI

dotenv.load_dotenv()

os.environ['OPENAI_API_KEY'] = os.getenv("OPENAI_API_KEY")
os.environ['OPENAI_BASE_URL'] = os.getenv("OPENAI_BASE_URL")

llm = OpenAI(  #非对话模型
    #max_tokens=512,
    #temperature=0.7,
)
```

对话模型：

```
import os
import dotenv
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()

os.environ['OPENAI_API_KEY'] = os.getenv("OPENAI_API_KEY")
os.environ['OPENAI_BASE_URL'] = os.getenv("OPENAI_BASE_URL")

chat_model = ChatOpenAI(  #对话模型
    model="gpt-4o-mini",
    #max_tokens=512,
```

```
    #temperature=0.7,
)
```

对应的配置文件：

```
OPENAI_API_KEY="sk-xxxxxx"  #从CloseAI平台，注册自己的账号，并获取API KEY
OPENAI_BASE_URL="https://api.openai-proxy.org/v1"
```

3、Model I/O之调用模型2

3.1 关于对话模型的Message(消息)

聊天模型，出了将字符串作为输入外，还可以使用 **聊天消息** 作为输入，并返回 **聊天消息** 作为输出。

LangChain有一些内置的消息类型：

- **SystemMessage**：设定AI行为规则或背景信息。比如设定AI的初始状态、行为模式或对话的总体目标。比如“作为一个代码专家”，或者“返回json格式”。通常作为输入消息序列中的第一个传递。
- **HumanMessage**：表示来自用户输入。比如“实现 一个快速排序方法”
- **AIMessage**：存储AI回复的内容。这可以是文本，也可以是调用工具的请求
- **ChatMessage**：可以自定义角色的通用消息类型
- **FunctionMessage/ToolMessage**：函数调用/工具消息，用于函数调用结果的消息类型

> 注意：
>
> FunctionMessage和ToolMessage分别是在函数调用和工具调用场景下才会使用的特殊消息类型，HumanMessage、AIMessage和SystemMessage才是最常用的消息类型。

举例1：

```
from langchain_core.messages import HumanMessage, SystemMessage

messages = [SystemMessage(content="你是一位乐于助人的智能小助手"),
            HumanMessage(content="你好，请你介绍一下你自己"),]

print(messages)
```

```
[SystemMessage(content='你是一位乐于助人的智能小助手', additional_kwargs={}, response_metadata={}), HumanMessage(content='你好，请你介绍一下你自己', additional_kwargs={}, response_metadata={})]
```

举例2：

```python
from langchain_core.messages import HumanMessage, AIMessage, SystemMessage

messages = [
    SystemMessage(content=["你是一个数学家,只会回答数学问题", "每次你都能给出详细的方案"]),
    HumanMessage(content="1 + 2 * 3 = ?"),
    AIMessage(content="1 + 2 * 3 的结果是7"),
]

print(messages)
```

```
[SystemMessage(content=['你是一个数学家,只会回答数学问题', '每次你都能给出详细的方案'], additional_kwargs={}, response_metadata={}), HumanMessage(content='1 + 2 * 3 = ?', additional_kwargs={}, response_metadata={}), AIMessage(content='1 + 2 * 3 的结果是7', additional_kwargs={}, response_metadata={})]
```

举例3：

```python
#1.导入相关包
from langchain_core.messages import SystemMessage, HumanMessage, AIMessage

# 2.直接创建不同类型消息
systemMessage = SystemMessage(
  content="你是一个AI开发工程师",
  additional_kwargs={"tool": "invoke_tool()"}
)
humanMessage = HumanMessage(
  content="你能开发哪些AI应用?"
)
aiMessage = AIMessage(
  content="我能开发很多AI应用, 比如聊天机器人, 图像识别, 自然语言处理等"
)
# 3.打印消息列表
messages = [systemMessage,humanMessage,aiMessage]
print(messages)
```

```
[SystemMessage(content='你是一个AI开发工程师', additional_kwargs={'你的名字': '小谷AI'}, response_metadata={}), HumanMessage(content='你能开发哪些AI应用?', additional_kwargs={}, response_metadata={}), AIMessage(content='我能开发很多AI应用, 比如聊天机器人, 图像识别, 自然语言处理等', additional_kwargs={}, response_metadata={})]
```

举例4：

```python
from langchain_core.messages import (
    AIMessage,
    HumanMessage,
    SystemMessage,
    ChatMessage
)

# 创建不同类型的消息
```

```
system_message = SystemMessage(content="你是一个专业的数据科学家")
human_message = HumanMessage(content="解释一下随机森林算法")
ai_message = AIMessage(content="随机森林是一种集成学习方法...")
custom_message = ChatMessage(role="analyst", content="补充一点关于超参数调优的信息")

print(system_message.content)
print(human_message.content)
print(ai_message.content)
print(custom_message.content)
```

```
你是一个专业的数据科学家
解释一下随机森林算法
随机森林是一种集成学习方法...
补充一点关于超参数调优的信息
```

举例5：结合大模型使用

```
import os
from langchain_core.messages import SystemMessage,HumanMessage

dotenv.load_dotenv()

os.environ['OPENAI_API_KEY'] = os.getenv("OPENAI_API_KEY")
os.environ['OPENAI_BASE_URL'] = os.getenv("OPENAI_BASE_URL")

chat_model = ChatOpenAI(
    model="gpt-4o-mini",
)

# 组成消息列表
messages = [
    SystemMessage(content="你是一个擅长人工智能相关学科的专家"),
    HumanMessage(content="请解释一下什么是机器学习？")
]

response = chat_model.invoke(messages)
print(response.content)
print(type(response))  #<class 'langchain_core.messages.ai.AIMessage'>
```

```
机器学习是人工智能的一个分支，它旨在通过经验自动改进系统的性能。换句话说，机器学习使计算机能够从数据中学习和识别模式，从而进行预测或决策，而不需要明确编程。

机器学习的基本过程通常包括以下几个步骤：

1. **数据收集**：收集相关数据，这些数据可以是结构化的（如数据库中的表格）或非结构化的（如文本、图像等）。

2. **数据预处理**：对收集到的数据进行清洗和处理，以去除噪声和不相关的信息，填补缺失值，并对数据进行标准化或归一化。

```

```
3. **特征选择与提取**：选择对模型有预测能力的特征，或者创造新的特征，以便更好地表示问题。

4. **模型选择**：选择合适的机器学习算法与模型，这些算法可以分为监督学习、无监督学习和强化学习等类型。

5. **训练模型**：使用训练数据来调整模型的参数，以便其能够根据输入数据做出准确的预测或分类。

6. **评估模型**：使用测试数据来评估模型的性能，常用的评估指标包括准确率、精确率、召回率和F1分数等。

7. **优化与迭代**：根据评估结果对模型进行优化，可能需要返回前面的步骤进行调整，直到满意为止。

8. **部署与监控**：将训练好的模型投入实际应用中，并持续监控其表现，以便在遇到新数据时进行调优。

机器学习已经广泛应用于各个领域，包括图像识别、自然语言处理、推荐系统、金融预测、医疗诊断等，其目标是通过数据驱动的方式来提高决策的效率和准确性。
<class 'langchain_core.messages.ai.AIMessage'>
```

3.2 关于多轮对话与上下文记忆

前提：获取大模型

```python
import os
import dotenv
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()

os.environ['OPENAI_API_KEY'] = os.getenv("OPENAI_API_KEY1")
os.environ['OPENAI_BASE_URL'] = os.getenv("OPENAI_BASE_URL")

chat_model = ChatOpenAI(
    model="gpt-4o-mini"
)
```

测试1：

```python
from langchain_core.messages import SystemMessage, HumanMessage

sys_message = SystemMessage(
    content="我是一个人工智能的助手，我的名字叫小智",
)
human_message = HumanMessage(content="猫王是一只猫吗？")

messages = [sys_message, human_message]

#调用大模型，传入messages
response = chat_model.invoke(messages)
```

```
print(response.content)


response1 = chat_model.invoke("你叫什么名字？")
print(response1.content)
```

测试2：

```
from langchain_core.messages import SystemMessage, HumanMessage

sys_message = SystemMessage(
    content="我是一个人工智能的助手，我的名字叫小智",
)
human_message = HumanMessage(content="猫王是一只猫吗？")
human_message1 = HumanMessage(content="你叫什么名字？")

messages = [sys_message, human_message,human_message1]

#调用大模型，传入messages
response = chat_model.invoke(messages)
print(response.content)

```

测试3：

```
from langchain_core.messages import SystemMessage, HumanMessage

sys_message = SystemMessage(
    content="我是一个人工智能的助手，我的名字叫小智",
)
human_message = HumanMessage(content="猫王是一只猫吗？")

sys_message1 = SystemMessage(
    content="我可以做很多事情，有需要就找我吧",
)

human_message1 = HumanMessage(content="你叫什么名字？")

messages = [sys_message, human_message,sys_message1,human_message1]

#调用大模型，传入messages
response = chat_model.invoke(messages)
print(response.content)

```

测试4：

```
from langchain_core.messages import SystemMessage, HumanMessage

# 第1组
sys_message = SystemMessage(
    content="我是一个人工智能的助手，我的名字叫小智",
```

```
)
human_message = HumanMessage(content="猫王是一只猫吗？")

messages = [sys_message, human_message]

# 第2组
sys_message1 = SystemMessage(
    content="我可以做很多事情，有需要就找我吧",
)

human_message1 = HumanMessage(content="你叫什么名字？")

messages1 = [sys_message1,human_message1]

#调用大模型，传入messages
response = chat_model.invoke(messages)
print(response.content)

response = chat_model.invoke(messages1)
print(response.content)
```

测试5：

```
from langchain_core.messages import SystemMessage, HumanMessage, AIMessage

messages = [
    SystemMessage(content="我是一个人工智能助手，我的名字叫小智"),
    HumanMessage(content="人工智能英文怎么说？"),
    AIMessage(content="AI"),
    HumanMessage(content="你叫什么名字"),
]

messages1 = [
    SystemMessage(content="我是一个人工智能助手，我的名字叫小智"),
    HumanMessage(content="很高兴认识你"),
    AIMessage(content="我也很高兴认识你"),
    HumanMessage(content="你叫什么名字"),
]

messages2 = [
    SystemMessage(content="我是一个人工智能助手，我的名字叫小智"),
    HumanMessage(content="人工智能英文怎么说？"),
    AIMessage(content="AI"),
    HumanMessage(content="你叫什么名字"),
]

chat_model.invoke(messages2)
```

3.3 关于模型调用的方法

为了尽可能简化自定义链的创建，我们实现了一个"Runnable"协议。许多LangChain组件实现了Runnable 协议，包括聊天模型、提示词模板、输出解析器、检索器、代理(智能体)等。

Runnable 定义的公共的调用方法如下：

- `invoke`：处理单条输入，等待LLM完全推理完成后再返回调用结果
- `stream`：流式响应，逐字输出LLM的响应结果
- `batch`：处理批量输入

这些也有相应的异步方法，应该与 asyncio 的`await`语法一起使用以实现并发：

- `astream`：异步流式响应
- `ainvoke`：异步处理单条输入
- `abatch`：异步处理批量输入
- `astream_log`：异步流式返回中间步骤，以及最终响应
- `astream_events`：（测试版）异步流式返回链中发生的事件（在 langchain-core 0.1.14 中引入）

3.3.1 流式输出与非流式输出

在Langchain中，语言模型的输出分为了两种主要的模式：**流式输出**与**非流式输出**。

下面是两个场景：

- 非流式输出：这是Langchain与LLM交互时的`默认行为`，是最简单、最稳定的语言模型调用方式。当用户发出请求后，系统在后台等待模型`生成完整响应`，然后`一次性将全部结果返回`。
  - 举例：用户提问，请编写一首诗，系统在静默数秒后`突然弹出`了完整的诗歌。（体验较单调）
  - 在大多数问答、摘要、信息抽取类任务中，非流式输出提供了结构清晰、逻辑完整的结果，适合快速集成和部署。
- 流式输出：一种`更具交互感`的模型输出方式，用户不再需要等待完整答案，而是能看到模型`逐个token`地实时返回内容。
  - 举例：用户提问，请编写一首诗，当问题刚刚发送，系统就开始`一字一句`（逐个token）进行回复，感觉是一边思考一边输出。
  - 更像是“实时对话”，更为贴近人类交互的习惯，更有吸引力。
  - 适合构建强调“实时反馈”的应用，如聊天机器人、写作助手等。
  - Langchain 中通过设置`stream=True`并配合 **回调机制** 来启用流式输出。

非流式输出：

举例1：

```
import os
import dotenv
from langchain_core.messages import HumanMessage
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()

os.environ['OPENAI_API_KEY'] = os.getenv("OPENAI_API_KEY1")
os.environ['OPENAI_BASE_URL'] = os.getenv("OPENAI_BASE_URL")

```

```python
#初始化大模型
chat_model = ChatOpenAI(model="gpt-4o-mini")

# 创建消息
messages = [HumanMessage(content="你好，请介绍一下自己")]

# 非流式调用LLM获取响应
response = chat_model.invoke(messages)

# 打印响应内容
print(response)
```

输出结果如下，是直接全部输出的。

```
content='你好！我是一个人工智能助手，专门设计来提供信息和解答问题。我可以帮助你解答各种问题，比如学习、科技、文化、生活等方面的内容。如果你有任何具体的问题或者需要了解的主题，欢迎随时问我！' additional_kwargs={'refusal': None} response_metadata={'token_usage': {'completion_tokens': 57, 'prompt_tokens': 12, 'total_tokens': 69, 'completion_tokens_details': {'accepted_prediction_tokens': 0, 'audio_tokens': 0, 'reasoning_tokens': 0, 'rejected_prediction_tokens': 0}, 'prompt_tokens_details': {'audio_tokens': 0, 'cached_tokens': 0}}, 'model_name': 'gpt-4o-mini-2024-07-18', 'system_fingerprint': 'fp_efad92c60b', 'id': 'chatcmpl-BmdJTYMLA9iiUFDIAJLJREFdJN5Us', 'service_tier': None, 'finish_reason': 'stop', 'logprobs': None} id='run--2b25b74a-12b0-4162-80fc-7d348b3ed3fb-0' usage_metadata={'input_tokens': 12, 'output_tokens': 57, 'total_tokens': 69, 'input_token_details': {'audio': 0, 'cache_read': 0}, 'output_token_details': {'audio': 0, 'reasoning': 0}}
```

举例2：

```python
import os
import dotenv
from langchain_core.messages import HumanMessage, SystemMessage
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()

os.environ['OPENAI_API_KEY'] = os.getenv("OPENAI_API_KEY1")
os.environ['OPENAI_BASE_URL'] = os.getenv("OPENAI_BASE_URL")

# 初始化大模型
chat_model = ChatOpenAI(model="gpt-4o-mini")

# 支持多个消息作为输入
messages = [
    SystemMessage(content="你是一位乐于助人的助手。你叫于老师"),
    HumanMessage(content="你是谁？")
]
response = chat_model.invoke(messages)
print(response.content)
```

```
我叫于老师，是一位乐于助人的助手。在这里我可以帮助你解答问题、提供信息或是进行交流。有需要帮助的地方吗?
```

举例3：

```python
import os
import dotenv
from langchain_core.messages import HumanMessage, SystemMessage
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()
1
os.environ['OPENAI_API_KEY'] = os.getenv("OPENAI_API_KEY1")
os.environ['OPENAI_BASE_URL'] = os.getenv("OPENAI_BASE_URL")

# 初始化大模型
chat_model = ChatOpenAI(model="gpt-4o-mini")

# 支持多个消息作为输入
messages = [
    SystemMessage(content="你是一位乐于助人的助手。你叫于老师"),
    HumanMessage(content="你是谁？")
]
response = chat_model(messages)   #特别的写法
print(response.content)
```

第19行，底层调用 `BaseChatModel.__call__`，内部调用的还是invoke()。后续还会有这种写法出现，了解即可。

流式输出

一种更具交互感的模型输出方式，用户不再需要等待完整答案，而是能看到模型**逐个 token** 地实时返回内容。适合构建强调“实时反馈”的应用，如聊天机器人、写作助手等。

Langchain 中通过设置 `streaming=True` 并配合 **回调机制** 来启用流式输出。

举例：

```python
import os
import dotenv
from langchain_core.messages import HumanMessage
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()

os.environ['OPENAI_API_KEY'] = os.getenv("OPENAI_API_KEY")
os.environ['OPENAI_BASE_URL'] = os.getenv("OPENAI_BASE_URL")

# 初始化大模型
chat_model = ChatOpenAI(model="gpt-4o-mini",
                        streaming=True  # 启用流式输出
                        )

# 创建消息
```

```python
messages = [HumanMessage(content="你好，请介绍一下自己")]

# 流式调用LLM获取响应
print("开始流式输出：")
for chunk in chat_model.stream(messages):
    # 逐个打印内容块
    print(chunk.content, end="", flush=True) # 刷新缓冲区（无换行符，缓冲区未刷新，内容可能不会立即显示）

print("\n流式输出结束")
```

输出结果如下（一段段文字逐个输出）

```
开始流式输出：
你好！我是一个人工智能助手，旨在帮助用户回答问题、提供信息和解决问题。我可以处理各种主题，包括科技、历史、文化、语言学习等。无论你有什么问题，尽管问我，我会尽力提供准确和有用的回答！
流式输出结束
```

3.3.2 批量调用

举例：

```python
import os
import dotenv
from langchain_core.messages import HumanMessage, SystemMessage
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()

os.environ['OPENAI_API_KEY'] = os.getenv("OPENAI_API_KEY")
os.environ['OPENAI_BASE_URL'] = os.getenv("OPENAI_BASE_URL")

# 初始化大模型
chat_model = ChatOpenAI(model="gpt-4o-mini")

messages1 = [SystemMessage(content="你是一位乐于助人的智能小助手"),
             HumanMessage(content="请帮我介绍一下什么是机器学习"), ]

messages2 = [SystemMessage(content="你是一位乐于助人的智能小助手"),
             HumanMessage(content="请帮我介绍一下什么是AIGC"), ]

messages3 = [SystemMessage(content="你是一位乐于助人的智能小助手"),
             HumanMessage(content="请帮我介绍一下什么是大模型技术"), ]

messages = [messages1, messages2, messages3]

# 调用batch
response = chat_model.batch(messages)

print(response)

```

[AIMessage(content='机器学习是人工智能（AI）的一种分支，它使计算机能够通过经验自动改进其性能，而不需要明确的编程指令。简而言之，机器学习关注的是让计算机从数据中学习，并在此基础上做出决策或预测。\n\n机器学习的基本流程通常包括以下几个步骤：\n\n1. **数据收集**：收集和准备数据，这是机器学习的基础。数据可以是结构化的（如表格）或非结构化的（如文本、图像）。\n\n2. **数据预处理**：对数据进行清洗、整理和转换，以适合后续的分析和建模。这可能包括去除噪声、填补缺失值和标准化数据等。\n\n3. **选择模型**：根据问题的性质选择合适的机器学习算法或模型。例如，线性回归、决策树、支持向量机、神经网络等。\n\n4. **训练模型**：使用准备好的数据来训练模型。模型通过调整内部参数来最小化预测与实际结果之间的差距。\n\n5. **评估模型**：使用测试数据集来评估模型的性能，通常通过准确率、召回率、F1值等指标。\n\n6. **模型优化**：根据评估结果调整模型参数或选择不同的算法，以提高模型的表现。\n\n7. **部署应用**：将训练好的模型应用于实际问题，进行实时或批量预测。\n\n机器学习通常被分为三大类：\n\n1. **监督学习**：模型在拥有标记数据的情况下学习，目标是预测新的输入数据的结果。常用例子包括分类和回归任务。\n\n2. **无监督学习**：模型在没有标记数据的情况下学习，目标是发现数据的结构和模式。常用例子包括聚类和降维。\n\n3. **强化学习**：模型通过与环境交互来学习，目标是通过试错过程来优化决策，最大化累积奖励。\n\n机器学习的应用非常广泛，包括图像识别、语言处理、推荐系统、金融预测、医疗诊断等领域。随着数据的增加和计算能力的提升，机器学习正在快速发展，越来越多地被应用于实际问题中。', additional_kwargs={'refusal': None}, response_metadata={'token_usage': {'completion_tokens': 490, 'prompt_tokens': 30, 'total_tokens': 520, 'completion_tokens_details': {'accepted_prediction_tokens': 0, 'audio_tokens': 0, 'reasoning_tokens': 0, 'rejected_prediction_tokens': 0}, 'prompt_tokens_details': {'audio_tokens': 0, 'cached_tokens': 0}}, 'model_name': 'gpt-4o-mini-2024-07-18', 'system_fingerprint': 'fp_57db37749c', 'id': 'chatcmpl-Bms2AQa5fLnKKMWybQvs0oLm3mc7C', 'service_tier': None, 'finish_reason': 'stop', 'logprobs': None}, id='run--6ab9c603-e5bc-4cc7-bba4-706bdc6f28d9-0', usage_metadata={'input_tokens': 30, 'output_tokens': 490, 'total_tokens': 520, 'input_token_details': {'audio': 0, 'cache_read': 0}, 'output_token_details': {'audio': 0, 'reasoning': 0}}), AIMessage(content='AIGC（人工智能生成内容，Artificial Intelligence Generated Content）是指利用人工智能技术自动生成各种内容的过程。这些内容可以包括文本、图像、音频、视频等。随着深度学习、自然语言处理和计算机视觉等技术的发展，AIGC在各个领域展现出巨大的潜力和应用前景。\n\n例如：\n\n1. **文本生成**：使用语言模型，如OpenAI的GPT系列，生成文章、故事、诗歌等。它可以用于内容创作、新闻报道的自动撰写等。\n\n2. **图像生成**：基于生成对抗网络（GAN）等技术，创建新的图像或艺术作品。例如，DALL-E和Midjourney等工具能够根据用户输入的文本描述生成相应的图片。\n\n3. **音频生成**：通过AI算法生成音乐、语音或其他音频内容，能够用于音乐创作、虚拟助理中的语音输出等。\n\n4. **视频生成**：AI也可以生成或编辑视频内容，应用于影视制作、广告创意等领域。\n\nAIGC的优势在于其高效性和创新能力，它能大幅度提高内容生产的速度，降低成本，同时也可以帮助创作者激发灵感。不过，AIGC也面临一些挑战，如版权问题、生成内容的质量和真实性等。随着技术的不断进步，AIGC的应用领域和影响力仍在不断扩大。', additional_kwargs={'refusal': None}, response_metadata={'token_usage': {'completion_tokens': 318, 'prompt_tokens': 31, 'total_tokens': 349, 'completion_tokens_details': {'accepted_prediction_tokens': 0, 'audio_tokens': 0, 'reasoning_tokens': 0, 'rejected_prediction_tokens': 0}, 'prompt_tokens_details': {'audio_tokens': 0, 'cached_tokens': 0}}, 'model_name': 'gpt-4o-mini-2024-07-18', 'system_fingerprint': 'fp_efad92c60b', 'id': 'chatcmpl-Bms2A0v8agsIGevCNImWTIq7ICZgq', 'service_tier': None, 'finish_reason': 'stop', 'logprobs': None}, id='run--7aa515e5-37b2-49e7-91be-b29a231537d1-0', usage_metadata={'input_tokens': 31, 'output_tokens': 318, 'total_tokens': 349, 'input_token_details': {'audio': 0, 'cache_read': 0}, 'output_token_details': {'audio': 0, 'reasoning': 0}}), AIMessage(content='大模型技术是指通过大规模的数据集和强大的计算能力训练出的复杂机器学习模型，特别是深度学习模型。这些模型通常具有数亿甚至数万亿个参数，能够在多个领域中执行各种任务，如自然语言处理、计算机视觉、语音识别等。\n\n以下是大模型技术的一些关键特点和应用：\n\n1. **规模与复杂性**：大模型通常由多层神经网络构成，具备高维度和显著的表达能力。这使得它们可以捕捉到数据中的复杂模式。

\n\n2. **预训练与微调**：大模型通常采用预训练和微调的方法。模型先在大规模通用数据集上进行预训练，然后再在特定任务的数据集上进行微调，以提高在特定任务上的性能。\n\n3. **迁移学习**：大模型可以将从一个任务中学到的知识迁移到其他相关任务上，从而减少各个任务之间的训练时间和数据需求。\n\n4. **多模态学习**：一些大模型技术支持多模态输入，即可以同时处理文本、图像、声音等多种数据类型，增强了模型的通用性和应用范围。\n\n5. **应用范围广泛**：大模型被广泛应用于聊天机器人、自动翻译、图像生成、医学影像分析、内容推荐等多个领域。\n\n6. **技术挑战**：尽管大模型技术具有很高的性能，但在训练和推理过程中，也带来了计算资源需求大、能耗高、模型解释性差等挑战。\n\n总的来说，大模型技术代表了机器学习和人工智能领域的一种趋势和发展方向，其强大的能力使其在许多实际应用中表现出色。', additional_kwargs={'refusal': None}, response_metadata={'token_usage': {'completion_tokens': 383, 'prompt_tokens': 31, 'total_tokens': 414, 'completion_tokens_details': {'accepted_prediction_tokens': 0, 'audio_tokens': 0, 'reasoning_tokens': 0, 'rejected_prediction_tokens': 0}, 'prompt_tokens_details': {'audio_tokens': 0, 'cached_tokens': 0}}, 'model_name': 'gpt-4o-mini-2024-07-18', 'system_fingerprint': 'fp_efad92c60b', 'id': 'chatcmpl-Bms2Atgkau6s0fmfThAacknwoZNal', 'service_tier': None, 'finish_reason': 'stop', 'logprobs': None}, id='run--93b50346-c340-4821-9847-562f80fb8cbc-0', usage_metadata={'input_tokens': 31, 'output_tokens': 383, 'total_tokens': 414, 'input_token_details': {'audio': 0, 'cache_read': 0}, 'output_token_details': {'audio': 0, 'reasoning': 0}})]

3.3.3 同步调用与异步调用(了解)

同步调用

举例：

```python
import time

def call_model():
    # 模拟同步API调用
    print("开始调用模型...")
    time.sleep(5)  # 模拟调用等待,单位：秒
    print("模型调用完成。")

def perform_other_tasks():
    # 模拟执行其他任务
    for i in range(5):
        print(f"执行其他任务 {i + 1}")
        time.sleep(1)  # 单位：秒

def main():
    start_time = time.time()
    call_model()
    perform_other_tasks()
    end_time = time.time()
    total_time = end_time - start_time
    return f"总共耗时：{total_time}秒"

# 运行同步任务并打印完成时间
main_time = main()
print(main_time)
```

```
开始调用模型...
模型调用完成。
执行其他任务 1
执行其他任务 2
执行其他任务 3
执行其他任务 4
执行其他任务 5
总共耗时：10.061029434204102秒
```

之前的`llm.invoke(...)`本质上是一个同步调用。每个操作依次执行，直到当前操作完成后才开始下一个操作，从而导致总的执行时间是各个操作时间的总和。

异步调用

异步调用，允许程序在等待某些操作完成时继续执行其他任务，而不是阻塞等待。这在处理I/O操作（如网络请求、文件读写等）时特别有用，可以显著提高程序的效率和响应性。

举例：

写法1：此写法适合Jupyter Notebook

```python
import asyncio
import time

async def async_call(llm):
    await asyncio.sleep(5)  # 模拟异步操作
    print("异步调用完成")

async def perform_other_tasks():
    await asyncio.sleep(5)  # 模拟异步操作
    print("其他任务完成")

async def run_async_tasks():
    start_time = time.time()
    await asyncio.gather(
        async_call(None),  # 示例调用，使用None模拟LLM对象
        perform_other_tasks()
    )
    end_time = time.time()
    return f"总共耗时：{end_time - start_time}秒"

# # 正确运行异步任务的方式
# if __name__ == "__main__":
#     # 使用 asyncio.run() 来启动异步程序
#     result = asyncio.run(run_async_tasks())
#     print(result)


# 在 Jupyter 单元格中直接调用
result = await run_async_tasks()
print(result)
```

```
异步调用完成
其他任务完成
总共耗时：5.001038551330566秒
```

写法2：(此写法不适合Jupyter Notebook)

```python
import asyncio
import time

async def async_call(llm):
    await asyncio.sleep(5)  # 模拟异步操作
    print("异步调用完成")

async def perform_other_tasks():
    await asyncio.sleep(5)  # 模拟异步操作
    print("其他任务完成")

async def run_async_tasks():
    start_time = time.time()
    await asyncio.gather(
        async_call(None),  # 示例调用，替换None为模拟的LLM对象
        perform_other_tasks()
    )
    end_time = time.time()
    return f"总共耗时：{end_time - start_time}秒"

# 正确运行异步任务的方式
if __name__ == "__main__":
    # 使用 asyncio.run() 来启动异步程序
    result = asyncio.run(run_async_tasks())
    print(result)

```

使用`asyncio.gather()`并行执行时，理想情况下，因为两个任务几乎同时开始，它们的执行时间将重叠。如果两个任务的执行时间相同（这里都是5秒），那么总执行时间应该接近单个任务的执行时间，而不是两者时间之和。

异步调用之ainvoke

举例1：验证ainvoke是否是异步

```python
# 方式1
import inspect

print("ainvoke 是协程函数:", inspect.iscoroutinefunction(chat_model.ainvoke))
print("invoke 是协程函数:", inspect.iscoroutinefunction(chat_model.invoke))
```

```
ainvoke 是协程函数: True
invoke 是协程函数: False
```

举例2：（不能在Jupyter Notebook中测试）

```python
import asyncio
import os
import time

import dotenv
from langchain_core.messages import HumanMessage, SystemMessage
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()

os.environ['OPENAI_API_KEY'] = os.getenv("OPENAI_API_KEY1")
os.environ['OPENAI_BASE_URL'] = os.getenv("OPENAI_BASE_URL")

# 初始化大模型
chat_model = ChatOpenAI(model="gpt-4o-mini")

# 同步调用（对比组）
def sync_test():
    messages1 = [SystemMessage(content="你是一位乐于助人的智能小助手"),
                 HumanMessage(content="请帮我介绍一下什么是机器学习"), ]
    start_time = time.time()
    response = chat_model.invoke(messages1)  # 同步调用
    duration = time.time() - start_time
    print(f"同步调用耗时：{duration:.2f}秒")
    return response, duration


# 异步调用（实验组）
async def async_test():
    messages1 = [SystemMessage(content="你是一位乐于助人的智能小助手"),
                 HumanMessage(content="请帮我介绍一下什么是机器学习"), ]
    start_time = time.time()
    response = await chat_model.ainvoke(messages1)  # 异步调用
    duration = time.time() - start_time
    print(f"异步调用耗时：{duration:.2f}秒")
    return response, duration


# 运行测试
if __name__ == "__main__":
    # 运行同步测试
    sync_response, sync_duration = sync_test()
    print(f"同步响应内容: {sync_response.content[:100]}...\n")

    # 运行异步测试
    async_response, async_duration = asyncio.run(async_test())
    print(f"异步响应内容: {async_response.content[:100]}...\n")

    # 并发测试 - 修复版本
    print("\n=== 并发测试 ===")
    start_time = time.time()

```

```python

    async def run_concurrent_tests():
        # 创建3个异步任务
        tasks = [async_test() for _ in range(3)]
        # 并发执行所有任务
        return await asyncio.gather(*tasks)


    # 执行并发测试
    results = asyncio.run(run_concurrent_tests())

    total_time = time.time() - start_time
    print(f"\n3个并发异步调用总耗时: {total_time:.2f}秒")
    print(f"平均每个调用耗时: {total_time / 3:.2f}秒")

```

```
同步调用耗时：5.73秒
同步响应内容: 机器学习是人工智能（AI）的一个子领域，旨在通过让计算机系统从数据中学习和改进其性能，而无需明确的编程。它的核心理念是利用算法和统计模型，分析和识别数据中的模式，从而在新数据出现时做出预测或决策。

...

异步调用耗时：4.68秒
异步响应内容: 机器学习是一种人工智能（AI）技术，它使计算机能够通过经验自我学习和改进，而无需明确编程。换句话说，机器学习使计算机能够从数据中提取模式和规律，并根据这些信息进行预测或决策。

机器学习的基本过程通常...


=== 并发测试 ===
异步调用耗时：3.07秒
异步调用耗时：3.61秒
异步调用耗时：7.43秒

3个并发异步调用总耗时: 7.43秒
平均每个调用耗时: 2.48秒
```

4、Model I/O之Prompt Template

Prompt Template，通过模板管理大模型的输入。

4.1 介绍与分类

Prompt Template 是LangChain中的一个概念，接收用户输入，返回一个传递给LLM的信息（即提示词prompt）。

在应用开发中，固定的提示词限制了模型的灵活性和适用范围。所以，prompt template 是一个**模板化的字符串**，你可以将**变量插入到模板**中，从而创建出不同的提示。调用时：

- 以 **字典** 作为输入，其中每个键代表要填充的提示模板中的变量。
- 输出一个 **PromptValue** 。这个 PromptValue 可以传递给 LLM 或 ChatModel，并且还可以转换为字符串或消息列表。

有几种不同类型的提示模板：

- **PromptTemplate**： LLM提示模板，用于**生成字符串提示。**它使用 Python 的字符串来模板提示。
- **ChatPromptTemplate**： 聊天提示模板，用于**组合各种角色的消息模板**，传入聊天模型。
- **XxxMessagePromptTemplate**： 消息模板词模板，包括：SystemMessagePromptTemplate、HumanMessagePromptTemplate、AIMessagePromptTemplate、ChatMessagePromptTemplate等
- **FewShotPromptTemplate**： 样本提示词模板，通过示例来教模型如何回答
- **PipelinePrompt**： 管道提示词模板，用于把几个提示词组合在一起使用。
- **自定义模板**： 允许基于其它模板类来定制自己的提示词模板。

模版导入

```
from langchain.prompts.prompt import PromptTemplate

from langchain.prompts import ChatPromptTemplate

from langchain.prompts import FewShotPromptTemplate

from langchain.prompts.pipeline import PipelinePromptTemplate

from langchain.prompts import (
    ChatMessagePromptTemplate,
    SystemMessagePromptTemplate,
    AIMessagePromptTemplate,
    HumanMessagePromptTemplate,
)
```

4.2 复习：str.format()

Python的 **str.format()** 方法是一种字符串格式化的手段，允许在 **字符串中插入变量** 。使用这种方法，可以创建包含 **占位符** 的字符串模板，占位符由花括号 **{}** 标识。

- 调用format()方法时，可以传入一个或多个参数，这些参数将被顺序替换进占位符中。
- str.format()提供了灵活的方式来构造字符串，支持多种格式化选项。

在LangChain的默认设置下， **PromptTemplate** 使用 Python 的 **str.format()** 方法进行模板化。这样在模型接收输入前，可以根据需要对数据进行预处理和结构化。

带有位置参数的用法

```
# 使用位置参数
info = "Name: {0}, Age: {1}".format("Jerry", 25)
print(info)
```

```
Name: Jerry, Age: 25
```

带有关键字参数的用法

```
# 使用关键字参数
info = "Name: {name}, Age: {age}".format(name= "Tom", age=25)
print(info)
```

```
Name: Tom, Age: 25
```

使用字典解包的方式

```
# 使用字典解包
person = {"name": "David", "age": 40}
info = "Name: {name}, Age: {age}".format(**person)
print(info)
```

```
Name: David, Age: 40
```

4.3 具体使用：PromptTemplate

4.3.1 使用说明

PromptTemplate类，用于快速构建 **包含变量** 的提示词模板，并通过 **传入不同的参数值** 生成自定义的提示词。

主要参数介绍：

- **template**：定义提示词模板的字符串，其中包含 **文本** 和 **变量占位符（如{name}）**；
- **input_variables**： 列表，指定了模板中使用的变量名称，在调用模板时被替换；
- **partial_variables**：字典，用于定义模板中一些固定的变量名。这些值不需要再每次调用时被替换。

函数介绍：

- **format()**：给input_variables变量赋值，并返回提示词。利用format() 进行格式化时就一定要赋值，否则会报错。当在template中未设置input_variables，则会自动忽略。

4.3.2 两种实例化方式

方式1：使用构造方法

举例1：

```
from langchain.prompts import PromptTemplate

# 定义模板：描述主题的应用
template = PromptTemplate(template= "请简要描述{topic}的应用。",
                          input_variables=["topic"])

print(template)

```

```
# 使用模板生成提示词
prompt_1 = template.format(topic="机器学习")
prompt_2 = template.format(topic="自然语言处理")

print("提示词1:", prompt_1)
print("提示词2:", prompt_2)
```

```
input_variables=['topic'] input_types={} partial_variables={} template='请简要描述{topic}的应用。'
提示词1: 请简要描述机器学习的应用。
提示词2: 请简要描述自然语言处理的应用。
```

可以直观的看到PromptTemplate可以将template中声明的变量topic准确提取出来，使prompt更清晰。

举例2：定义多变量模板

```
from langchain.prompts import PromptTemplate

#定义多变量模板
template = PromptTemplate(
    template="请评价{product}的优缺点，包括{aspect1}和{aspect2}。",
    input_variables=["product", "aspect1", "aspect2"])

#使用模板生成提示词
prompt_1 = template.format(product="智能手机", aspect1="电池续航", aspect2="拍照质量")
prompt_2 = template.format(product="笔记本电脑", aspect1="处理速度", aspect2="便携性")

print("提示词1:",prompt_1)
print("提示词2:",prompt_2)
```

方式2：调用from_template()

举例1：

```
from langchain.prompts import PromptTemplate

prompt_template = PromptTemplate.from_template(
    "请给我一个关于{topic}的{type}解释。"
)

#传入模板中的变量名
prompt = prompt_template.format(type="详细", topic="量子力学")

print(prompt)
```

```
请给我一个关于量子力学的详细解释。
```

举例2：模板支持任意数量的变量，包括不含变量：

```python
#1.导入相关的包
from langchain_core.prompts import PromptTemplate

# 2.定义提示词模版对象
text = """
Tell me a joke
"""

prompt_template = PromptTemplate.from_template(text)
# 3.默认使用f-string进行格式化（返回格式好的字符串）
prompt = prompt_template.format()
print(prompt)
```

```
Tell me a joke
```

4.3.3 两种新的结构形式

形式1：部分提示词模版

在生成prompt前就已经提前初始化部分的提示词，实际进一步导入模版的时候只导入除已初始化的变量即可。

举例1：

方式1：**实例化过程中使用partial_variables变量**

```python
from langchain.prompts import PromptTemplate

#方式2:
template2 = PromptTemplate(
    template="{foo}{bar}",
    input_variables=["foo", "bar"],
    partial_variables={"foo": "hello"}
)

prompt2 = template2.format(bar="world")

print(prompt2)
```

方式2：**使用 PromptTemplate.partial() 方法创建部分提示模板**

```python
from langchain.prompts import PromptTemplate

template1 = PromptTemplate(
    template="{foo}{bar}",
    input_variables=["foo", "bar"]
)

#方式1:
partial_template1 = template1.partial(foo="hello")

prompt1 = partial_template1.format(bar="world")

print(prompt1)
```

举例2:

```python
from langchain_core.prompts import PromptTemplate

# 完整模板
full_template = """你是一个{role}，请用{style}风格回答:
问题：{question}
答案："""

# 预填充角色和风格
partial_template = PromptTemplate.from_template(full_template).partial(
    role="资深厨师",
    style="专业但幽默"
)

# 只需提供剩余变量
print(partial_template.format(question="如何煎牛排？"))
```

```
你是一个资深厨师，请用专业但幽默风格回答:
问题：如何煎牛排？
答案:
```

举例3:

```python
prompt_template = PromptTemplate.from_template(
    template = "请评价{product}的优缺点，包括{aspect1}和{aspect2}。",
    partial_variables={"aspect1":"电池","aspect2":"屏幕"}
)

prompt= prompt_template.format(product="笔记本电脑")
print(prompt)
```

```
请评价笔记本电脑的优缺点，包括电池和屏幕。
```

形式2：组合提示词(了解)

举例：

```
from langchain_core.prompts import PromptTemplate

template = (
    PromptTemplate.from_template("Tell me a joke about {topic}")
    + ", make it funny"
    + "\n\nand in {language}"
)

prompt = template.format(topic="sports", language="spanish")
print(prompt)
```

```
Tell me a joke about sports, make it funny

and in spanish
```

4.3.4 format() 与 invoke()

只要对象是RunnableSerializable接口类型，都可以使用invoke()，替换前面使用format()的调用方式。

format()，返回值为字符串类型；invoke()，返回值为PromptValue类型，接着调用to_string()返回字符串。

举例1：

```
#1.导入相关的包
from langchain_core.prompts import PromptTemplate

# 2.定义提示词模版对象
prompt_template = PromptTemplate.from_template(
    "Tell me a {adjective} joke about {content}."
)
# 3.默认使用f-string进行格式化（返回格式好的字符串）
prompt_template.invoke({"adjective": "funny", "content": "chickens"})
```

```
StringPromptValue(text='Tell me a funny joke about chickens.')
```

举例2：

```python
#1.导入相关的包
from langchain_core.prompts import PromptTemplate

# 2.使用初始化器进行实例化
prompt = PromptTemplate(
    input_variables=["adjective", "content"],
    template="Tell me a {adjective} joke about {content}")

# 3. PromptTemplate底层是RunnableSerializable接口 所以可以直接使用invoke()调用
prompt.invoke({"adjective": "funny", "content": "chickens"})
```

举例3：

```python
from langchain_core.prompts import PromptTemplate

prompt_template = (
        PromptTemplate.from_template("Tell me a joke about {topic}")
        + ", make it funny"
        + " and in {language}"
)

prompt = prompt_template.invoke({"topic": "sports", "language": "spanish"})
print(prompt)
```

4.3.5 结合LLM调用

Prompt 与大模型结合。

问题：这里的大模型，是哪类呢？非对话大模型？对话大模型？

提供大模型：（非对话大模型）

```python
import os
import dotenv
from langchain_openai import OpenAI

dotenv.load_dotenv()

os.environ['OPENAI_API_KEY'] = os.getenv("OPENAI_API_KEY1")
os.environ['OPENAI_BASE_URL'] = os.getenv("OPENAI_BASE_URL")

llm = OpenAI()
```

提供大模型：（对话大模型）

```python
import os
import dotenv
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()

os.environ['OPENAI_API_KEY'] = os.getenv("OPENAI_API_KEY1")
os.environ['OPENAI_BASE_URL'] = os.getenv("OPENAI_BASE_URL")

llm = ChatOpenAI(
    model="gpt-4o-mini"
)
```

调用过程：

```python
prompt_template = PromptTemplate.from_template(
    template = "请评价{product}的优缺点，包括{aspect1}和{aspect2}。"
)

prompt = prompt_template.format(product="电脑",aspect1="性能",aspect2="电池")
# prompt = prompt_template.invoke({"product":"电脑","aspect1":"性能","aspect2":"电池"})

print(type(prompt))

# llm.invoke(prompt) #使用非对话模型调用

llm1.invoke(prompt) #使用对话模型调用
```

<class 'str'>

AIMessage(content='电脑的优缺点可以从多个方面进行分析，包括性能和电池等方面。以下是对电脑的一些优缺点的总结：\n\n### 优点\n\n1. **性能强大**：\n - 现代电脑，尤其是高端型号，配备了强大的处理器（如Intel i7/i9、AMD Ryzen系列）、充足的内存（8GB、16GB或更高）以及快速的固态硬盘（SSD），能够高效地处理复杂任务，如视频编辑、3D建模和游戏等。\n\n2. **多功能性**：\n - 电脑可以用于多种用途，包括办公、学习、编程、设计、娱乐等，满足不同用户的需求。\n\n3. **扩展性**：\n - 许多台式电脑具有良好的扩展性，可以方便地升级硬件（如增加内存、升级显卡或更换硬盘）以提高性能。\n\n4. **大屏幕和高分辨率**：\n - 电脑通常配备比手机和笔记本更大的显示屏，能够提供更好的视觉体验，特别适合进行多任务处理和观看视频。\n\n5. **丰富的软件支持**：\n - 电脑平台（如Windows、macOS、Linux）支持的软件种类繁多，包括一些专业软件和开发工具，适合各类专业用户。\n\n### 缺点\n\n1. **便携性差**：\n - 台式电脑通常体积较大，不适合携带；相比之下，尽管笔记本电脑便携性较好，但通常性能相对较低。\n\n2. **电池续航有限（笔记本电脑）**：\n - 对于笔记本电脑而言，虽然电池技术在不断进步，但高性能笔记本在高负载情况下电池续航时间可能较短，常需要频繁充电。\n\n3. **散热和噪音问题**：\n - 高负载时，电脑可能会产生较高的热量和噪音，尤其是在运行大型应用或游戏时，散热和风扇噪音可能影响使用体验。\n\n4. **更新换代快**：\n - 电脑硬件和软件更新换代非常快，用户需要不时购买新设备或升级硬件，以保持性能。\n\n5. **成本**：\n - 高性能的电脑往往价格昂贵，尤其是游戏电脑和专业工作站，可能不适合预算有限的用户。\n\n总结来说，电脑在性能和多功能性上有显著优势，但在便携性和电池续航等方面存在一定的局限性。用户在选择电脑时，需要根据自己的需求和使用场景进行权衡。', additional_kwargs={'refusal': None}, response_metadata={'token_usage': {'completion_tokens': 589, 'prompt_tokens': 20, 'total_tokens': 609, 'completion_tokens_details': {'accepted_prediction_tokens': 0, 'audio_tokens': 0,

'reasoning_tokens': 0, 'rejected_prediction_tokens': 0}, 'prompt_tokens_details': {'audio_tokens': 0, 'cached_tokens': 0}}, 'model_name': 'gpt-4o-mini-2024-07-18', 'system_fingerprint': 'fp_efad92c60b', 'id': 'chatcmpl-BzjJi5AmgpC71coOr6IgrIRRRlsdB', 'service_tier': None, 'finish_reason': 'stop', 'logprobs': None}, id='run--c1165c77-9944-4d3a-8df7-053000e94f11-0', usage_metadata={'input_tokens': 20, 'output_tokens': 589, 'total_tokens': 609, 'input_token_details': {'audio': 0, 'cache_read': 0}, 'output_token_details': {'audio': 0, 'reasoning': 0}})

4.4 具体使用：ChatPromptTemplate

4.4.1 使用说明

ChatPromptTemplate是创建 **聊天消息列表** 的提示模板。它比普通 PromptTemplate 更适合处理多角色、多轮次的对话场景。

特点：

- 支持 **System** / **Human** / **AI** 等不同角色的消息模板
- 对话历史维护

参数类型：列表参数格式是tuple类型（ **role** :str **content** :str 组合最常用）

元组的格式为：
(role: str | type, content: str | list[dict] | list[object])

- 其中 **role** 是：字符串（如 **"system"** 、 **"human"** 、 **"ai"** ）

4.4.2 两种实例化方式

方式1：使用构造方法

举例：

```python
from langchain_core.prompts import ChatPromptTemplate

#参数类型这里使用的是tuple构成的list
prompt_template = ChatPromptTemplate([
    # 字符串 role + 字符串 content
    ("system", "你是一个AI开发工程师. 你的名字是 {name}."),
    ("human", "你能开发哪些AI应用?"),
    ("ai", "我能开发很多AI应用, 比如聊天机器人, 图像识别, 自然语言处理等."),
    ("human", "{user_input}")
])

#调用format()方法，返回字符串
prompt = prompt_template.invoke(input={"name": "小谷AI", "user_input": "你能帮我做什么?"})
print(type(prompt))
print(prompt)
```

<class 'langchain_core.prompt_values.ChatPromptValue'>
messages=[SystemMessage(content='你是一个AI开发工程师. 你的名字是 小谷AI.', additional_kwargs={}, response_metadata={}), HumanMessage(content='你能开发哪些AI应用?', additional_kwargs={}, response_metadata={}), AIMessage(content='我能开发很多AI应

用, 比如聊天机器人, 图像识别, 自然语言处理等.', additional_kwargs={}, response_metadata={}), HumanMessage(content='你能帮我做什么?', additional_kwargs={}, response_metadata={})]

方式2：调用from_messages()

举例1：

```python
# 导入相关依赖
from langchain_core.prompts import ChatPromptTemplate

# 定义聊天提示词模版
chat_template = ChatPromptTemplate.from_messages(
    [
        ("system", "你是一个有帮助的AI机器人，你的名字是{name}。"),
        ("human", "你好，最近怎么样？"),
        ("ai", "我很好，谢谢！"),
        ("human", "{user_input}"),
    ]
)

# 格式化聊天提示词模版中的变量
messages = chat_template.invoke(input={"name": "小明", "user_input": "你叫什么名字？"})

# 打印格式化后的聊天提示词模版内容
print(messages)
```

```
messages=[SystemMessage(content='你是一个有帮助的AI机器人，你的名字是小明。', additional_kwargs={}, response_metadata={}), HumanMessage(content='你好，最近怎么样？', additional_kwargs={}, response_metadata={}), AIMessage(content='我很好，谢谢！', additional_kwargs={}, response_metadata={}), HumanMessage(content='你叫什么名字？', additional_kwargs={}, response_metadata={})]
```

举例2：了解

```python
# 示例 1: role 为字符串
from langchain_core.prompts import ChatPromptTemplate

prompt = ChatPromptTemplate.from_messages([
    ("system", "你是一个{role}."),
    ("human", "{user_input}"),
])

# 示例 2: role 为消息类 不支持
from langchain_core.messages import SystemMessage, HumanMessage

# prompt = ChatPromptTemplate.from_messages([
#     (SystemMessage, "你是一个{role}."),  # 类对象 role + 字符串 content
#     (HumanMessage, ["你好！", {"type": "text"}]),  # 类对象 role + list[dict] content
# ])
# 修改
prompt = ChatPromptTemplate.from_messages([
```

```
    ("system", ["你好！", {"type": "text"}]),  # 字符串 role + list[dict] content
])
```

4.4.3 模板调用的几种方式

对比：`invoke()`、`format()`、`format_messages()`、`format_prompt()`

方式1：使用 invoke()，前面已经讲过

```
from langchain_core.prompts import ChatPromptTemplate

#参数类型这里使用的是tuple构成的list
prompt_template = ChatPromptTemplate([
    # 字符串 role + 字符串 content
    ("system", "你是一个AI开发工程师. 你的名字是 {name}."),
    ("human", "你能开发哪些AI应用?"),
    ("ai", "我能开发很多AI应用, 比如聊天机器人, 图像识别, 自然语言处理等."),
    ("human", "{user_input}")
])

prompt = prompt_template.invoke({"name": "小谷AI", "user_input": "你能帮我做什么?"})
print(type(prompt))
print(prompt)
print(len(prompt.messages))
```

```
<class 'langchain_core.prompt_values.ChatPromptValue'>
messages=[SystemMessage(content='你是一个AI开发工程师. 你的名字是 小谷AI.', additional_kwargs={}, response_metadata={}), HumanMessage(content='你能开发哪些AI应用?', additional_kwargs={}, response_metadata={}), AIMessage(content='我能开发很多AI应用, 比如聊天机器人, 图像识别, 自然语言处理等.', additional_kwargs={}, response_metadata={}), HumanMessage(content='你能帮我做什么?', additional_kwargs={}, response_metadata={})]
4
```

方式2：使用format()

```
from langchain_core.prompts import ChatPromptTemplate

#参数类型这里使用的是tuple构成的list
prompt_template = ChatPromptTemplate([
    # 字符串 role + 字符串 content
    ("system", "你是一个AI开发工程师. 你的名字是 {name}."),
    ("human", "你能开发哪些AI应用?"),
    ("ai", "我能开发很多AI应用, 比如聊天机器人, 图像识别, 自然语言处理等."),
    ("human", "{user_input}")
])

#方式1：调用format()方法，返回字符串
prompt = prompt_template.format(name="小谷AI", user_input="你能帮我做什么?")
print(type(prompt))
print(prompt)
```

```
<class 'str'>

System: 你是一个AI开发工程师. 你的名字是 小谷AI.
Human: 你能开发哪些AI应用?
AI: 我能开发很多AI应用, 比如聊天机器人, 图像识别, 自然语言处理等.
Human: 你能帮我做什么?
```

方式3：使用format_messages()

```python
from langchain_core.prompts import ChatPromptTemplate

prompt_template = ChatPromptTemplate([
    ("system", "你是一个AI开发工程师. 你的名字是 {name}."),
    ("human", "你能开发哪些AI应用?"),
    ("ai", "我能开发很多AI应用, 比如聊天机器人, 图像识别, 自然语言处理等."),
    ("human", "{user_input}")
])

#调用format_messages()方法，返回消息列表
prompt2 = prompt_template.format_messages(name="小谷AI", user_input="你能帮我做什么?")
print(type(prompt2))
print(prompt2)
```

```
<class 'list'>
[SystemMessage(content='你是一个AI开发工程师. 你的名字是 小谷AI.', additional_kwargs={}, response_metadata={}), HumanMessage(content='你能开发哪些AI应用?', additional_kwargs={}, response_metadata={}), AIMessage(content='我能开发很多AI应用, 比如聊天机器人, 图像识别, 自然语言处理等.', additional_kwargs={}, response_metadata={}), HumanMessage(content='你能帮我做什么?', additional_kwargs={}, response_metadata={})]
```

结论：给占位符赋值，针对于ChatPromptTemplate，推荐使用 **format_messages()** 方法，返回消息列表。

方式4：使用format_prompt()

```python
from langchain_core.prompts import ChatPromptTemplate

#参数类型这里使用的是tuple构成的list
prompt_template = ChatPromptTemplate([
    # 字符串 role + 字符串 content
    ("system", "你是一个AI开发工程师. 你的名字是 {name}."),
    ("human", "你能开发哪些AI应用?"),
    ("ai", "我能开发很多AI应用, 比如聊天机器人, 图像识别, 自然语言处理等."),
    ("human", "{user_input}")
])

prompt = prompt_template.format_prompt(name="小谷AI", user_input="你能帮我做什么?")
print(prompt.to_messages())
print(type(prompt.to_messages()))

```

```
#print(prompt.to_string())
#print(type(prompt.to_string()))
```

```
[SystemMessage(content='你是一个AI开发工程师. 你的名字是 小谷AI.', additional_kwargs={}, response_metadata={}), HumanMessage(content='你能开发哪些AI应用?', additional_kwargs={}, response_metadata={}), AIMessage(content='我能开发很多AI应用, 比如聊天机器人, 图像识别, 自然语言处理等.', additional_kwargs={}, response_metadata={}), HumanMessage(content='你能帮我做什么?', additional_kwargs={}, response_metadata={})]
<class 'list'>
```

4.4.4 更丰富的实例化参数类型

前面讲了ChatPromptTemplate的两种创建方式。我们看到不管使用构造方法，还是使用from_messages()，参数类型都是 **列表类型** 。列表中的元素可以是多种类型，前面我们主要测试了元组类型。

源码：

```
def __init__(self,
             messages: Sequence[BaseMessagePromptTemplate | BaseMessage | BaseChatPromptTemplate | tuple[str | type, str | list[dict] | list[object]] | str | dict[str, Any]],
             *,
             template_format: Literal["f-string", "mustache", "jinja2"] = "f-string",
             **kwargs: Any) -> None
```

源码：

```
@classmethod def from_messages(cls,
                 messages: Sequence[BaseMessagePromptTemplate | BaseMessage | BaseChatPromptTemplate | tuple[str | type, str | list[dict] | list[object]] | str | dict[str, Any]],
                 template_format: Literal["f-string", "mustache", "jinja2"] = "f-string")
  -> ChatPromptTemplate
```

结论：参数是列表类型，列表的元素可以是字符串、字典、字符串构成的元组、消息类型、提示词模板类型、消息提示词模板类型等

类型1：str类型

列表参数格式是str类型（不推荐），**因为默认角色都是human**

```
#1.导入相关依赖
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.messages import SystemMessage,HumanMessage, AIMessage
# 2.定义聊天提示词模版
chat_template = ChatPromptTemplate.from_messages(
  [
    "Hello, {name}!" # 等价于 ("human", "Hello, {name}!")
   ]
)

```

```python
# 3.1格式化聊天提示词模版中的变量(自己提供的)
messages = chat_template.format_messages(name="小谷AI")
# 3.2 使用invoke执行
# messages=chat_template.invoke({"name":"小谷AI"})

# 4.打印格式化后的聊天提示词模版内容
print(messages)
```

类型2：dict类型

列表参数格式是dict类型

```python
# 示例: 字典形式的消息
prompt = ChatPromptTemplate.from_messages([
    {"role": "system", "content": "你是一个{role}."},
    {"role": "human", "content": ["复杂内容", {"type": "text"}]},
])

print(prompt.format_messages(role="教师"))
```

类型3：Message类型

```python
from langchain_core.messages import SystemMessage,HumanMessage


chat_prompt_template = ChatPromptTemplate.from_messages([
    SystemMessage(content="我是一个贴心的智能助手"),
    HumanMessage(content="我的问题是:人工智能英文怎么说？")

])

messages = chat_prompt_template.format_messages()
print(messages)
print(type(messages))
```

```
[SystemMessage(content='我是一个贴心的智能助手', additional_kwargs={}, response_metadata={}), HumanMessage(content='我的问题是:人工智能英文怎么说？', additional_kwargs={}, response_metadata={})]
<class 'list'>
```

类型4：BaseChatPromptTemplate类型

使用 BaseChatPromptTemplate，可以理解为ChatPromptTemplate里嵌套了ChatPromptTemplate。

举例1：不带参数

```python
from langchain_core.prompts import ChatPromptTemplate

# 使用 BaseChatPromptTemplate（嵌套的 ChatPromptTemplate）
nested_prompt_template1 = ChatPromptTemplate.from_messages([("system", "我是一个人工智能助手")])
nested_prompt_template2 = ChatPromptTemplate.from_messages([("human", "很高兴认识你")])

prompt_template = ChatPromptTemplate.from_messages([
    nested_prompt_template1,nested_prompt_template2
])

prompt_template.format_messages()
```

```
[SystemMessage(content='嵌套提示词之System', additional_kwargs={},
response_metadata={}),
HumanMessage(content='嵌套提示词之Human', additional_kwargs={},
response_metadata={})]
```

举例2：带参数

```python
from langchain_core.prompts import ChatPromptTemplate

# 使用 BaseChatPromptTemplate（嵌套的 ChatPromptTemplate）
nested_prompt_template1 = ChatPromptTemplate.from_messages([
    ("system", "我是一个人工智能助手，我的名字叫{name}")
])
nested_prompt_template2 = ChatPromptTemplate.from_messages([
    ("human", "很高兴认识你,我的问题是{question}")
])

prompt_template = ChatPromptTemplate.from_messages([
    nested_prompt_template1,nested_prompt_template2
])

prompt_template.format_messages(name="小智",question="你为什么这么帅？")
```

```
[SystemMessage(content='我是一个人工智能助手，我的名字叫小智', additional_kwargs={},
response_metadata={}),
HumanMessage(content='很高兴认识你,我的问题是你为什么这么帅？', additional_kwargs={},
response_metadata={})]
```

类型5：BaseMessagePromptTemplate类型

LangChain提供不同类型的MessagePromptTemplate。最常用的是`SystemMessagePromptTemplate`、`HumanMessagePromptTemplate`和`AIMessagePromptTemplate`，分别创建系统消息、人工消息和AI消息，它们是ChatMessagePromptTemplate的特定角色子类。

基本概念：

HumanMessagePromptTemplate，专用于生成`用户消息（HumanMessage）`的模板类，是ChatMessagePromptTemplate的特定角色子类。

- `本质`：预定义了 `role="human"` 的 MessagePromptTemplate，且无需无需手动指定角色
- `模板化`：支持使用变量占位符，可以在运行时填充具体值
- `格式化`：能够将模板与输入变量结合生成最终的聊天消息
- `输出类型`：生成 `HumanMessage` 对象（`content` + `role="human"`）
- `设计目的`：简化用户输入消息的模板化构造，避免重复定义角色

SystemMessagePromptTemplate、AIMessagePromptTemplate：类似于上面，不再赘述

ChatMessagePromptTemplate，用于构建聊天消息的模板。它允许你创建可重用的消息模板，这些模板可以动态地插入变量值来生成最终的聊天消息

- `角色指定`：可以为每条消息指定角色（如 "system"、"human"、"ai"） 等，角色灵活。
- `模板化`：支持使用变量占位符，可以在运行时填充具体值
- `格式化`：能够将模板与输入变量结合生成最终的聊天消息

举例1：

```python
# 导入聊天消息类模板
from langchain_core.prompts import ChatPromptTemplate, HumanMessagePromptTemplate, SystemMessagePromptTemplate

# 创建消息模板
system_template = "你是一个专家{role}"
system_message_prompt = SystemMessagePromptTemplate.from_template(system_template)

human_template = "给我解释{concept}，用浅显易懂的语言"
human_message_prompt = HumanMessagePromptTemplate.from_template(human_template)

# 组合成聊天提示模板
chat_prompt = ChatPromptTemplate.from_messages([system_message_prompt, human_message_prompt])

# 格式化提示
formatted_messages = chat_prompt.format_messages(
    role="物理学家",
    concept="相对论"
)
print(formatted_messages)
```

```
[SystemMessage(content='你是一个专家物理学家', additional_kwargs={}, response_metadata={}), HumanMessage(content='给我解释相对论，用浅显易懂的语言', additional_kwargs={}, response_metadata={})]
[SystemMessage(content='你是一个专家物理学家', additional_kwargs={}, response_metadata={}), HumanMessage(content='给我解释相对论，用浅显易懂的语言', additional_kwargs={}, response_metadata={})]
```

举例2：ChatMessagePromptTemplate的理解

```python
# 1.导入先关包
from langchain_core.prompts import ChatMessagePromptTemplate

```

```
# 2.定义模版
prompt = "今天我们授课的内容是{subject}"

# 3.创建自定义角色聊天消息提示词模版
chat_message_prompt = ChatMessagePromptTemplate.from_template(
    role="teacher", template=prompt
)
# 4.格式聊天消息提示词
resp = chat_message_prompt.format(subject="我爱北京天安门")
print(type(resp))
print(resp)
```

```
<class 'langchain_core.messages.chat.ChatMessage'>
content='今天我们授课的内容是我爱北京天安门' additional_kwargs={} response_metadata={}
role='teacher'
```

举例3：综合使用

```
from langchain_core.prompts import (
    ChatPromptTemplate,
    SystemMessagePromptTemplate,
    HumanMessagePromptTemplate,
)
from langchain_core.messages import SystemMessage, HumanMessage

# 示例 1: 使用 BaseMessagePromptTemplate
system_prompt = SystemMessagePromptTemplate.from_template("你是一个{role}.")
human_prompt = HumanMessagePromptTemplate.from_template("{user_input}")

# 示例 2: 使用 BaseMessage（已实例化的消息）
system_msg = SystemMessage(content="你是一个AI工程师。")
human_msg = HumanMessage(content="你好！")

# 示例 3: 使用 BaseChatPromptTemplate（嵌套的 ChatPromptTemplate）
nested_prompt = ChatPromptTemplate.from_messages([("system", "嵌套提示词")])

prompt = ChatPromptTemplate.from_messages([
    system_prompt,  # MessageLike (BaseMessagePromptTemplate)
    human_prompt,   # MessageLike (BaseMessagePromptTemplate)
    system_msg,     # MessageLike (BaseMessage)
    human_msg,      # MessageLike (BaseMessage)
    nested_prompt,  # MessageLike (BaseChatPromptTemplate)
])

prompt.format_messages(role="人工智能专家",user_input="介绍一下大模型的应用场景")
```

类似的：

```
from langchain_core.messages import HumanMessage, AIMessage
from langchain_core.prompts import HumanMessagePromptTemplate,
SystemMessagePromptTemplate
from langchain_core.prompts import ChatPromptTemplate
```

```
4
5  chat_template = ChatPromptTemplate.from_messages(
6      [
7          SystemMessagePromptTemplate.from_template("你是一个AI开发工程师. 你的名字是{name}."),
8          HumanMessage(content=("你能开发哪些AI应用?")),
9          AIMessage(content=("我能开发很多AI应用, 比如聊天机器人, 图像识别, 自然语言处理等.")),
10         HumanMessagePromptTemplate.from_template("{input}")
11     ]
12 )
13 messages = chat_template.format_messages(input="你能帮我做什么?", name="小谷AI")
14 print(messages)
```

```
[SystemMessage(content='你是一个AI开发工程师. 你的名字是 小谷AI.', additional_kwargs={}, response_metadata={}), HumanMessage(content='你能开发哪些AI应用?', additional_kwargs={}, response_metadata={}), AIMessage(content='我能开发很多AI应用, 比如聊天机器人, 图像识别, 自然语言处理等.', additional_kwargs={}, response_metadata={}), HumanMessage(content='你能帮我做什么?', additional_kwargs={}, response_metadata={})]
```

4.4.5 结合LLM

举例1：

```
1  from langchain.prompts.chat import ChatPromptTemplate
2
3  ######1、提供提示词#########
4  chat_prompt = ChatPromptTemplate.from_messages([
5      ("system", "你是一个数学家，你可以计算任何算式"),
6      ("human", "我的问题：{question}"),
7  ])
8
9
10 # 输入提示
11 messages = chat_prompt.format_messages(question="我今年18岁，我的舅舅今年38岁，我的爷爷今年72岁，我和舅舅一共多少岁了？")
12 #print(messages)
13
14 ######2、提供大模型#########
15 import os
16 import dotenv
17 from langchain_openai import ChatOpenAI
18
19 dotenv.load_dotenv()
20
21 os.environ['OPENAI_API_KEY'] = os.getenv("OPENAI_API_KEY1")
22 os.environ['OPENAI_BASE_URL'] = os.getenv("OPENAI_BASE_URL")
23
24 chat_model = ChatOpenAI(model="gpt-4o-mini")
25
26 ######3、结合提示词，调用大模型#########
27
28 # 得到模型的输出
29 output = chat_model.invoke(messages)
```

```
# 打印输出内容
print(output.content)
```

你今年18岁，你的舅舅今年38岁。那么你和舅舅的年龄总和是：

18 + 38 = 56

所以你和舅舅一共56岁。

举例2：

```
from dotenv import load_dotenv
from langchain.prompts.chat import SystemMessagePromptTemplate,
HumanMessagePromptTemplate, AIMessagePromptTemplate
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI

load_dotenv()
llm = ChatOpenAI()

template = ChatPromptTemplate.from_messages(
    [
        SystemMessagePromptTemplate.from_template("你是{product}的客服助手。你的名字叫
{name}"),
        HumanMessagePromptTemplate.from_template("hello 你好吗？"),
        AIMessagePromptTemplate.from_template("我很好 谢谢!"),
        HumanMessagePromptTemplate.from_template("{query}"),
    ]
)

prompt = template.format_messages(
    product="AGI课堂",
    name="Bob",
    query="你是谁"
)

# 提供聊天模型
import os
import dotenv
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()

os.environ['OPENAI_API_KEY'] = os.getenv("OPENAI_API_KEY1")
os.environ['OPENAI_BASE_URL'] = os.getenv("OPENAI_BASE_URL")

chat_model = ChatOpenAI(model="gpt-4o-mini")

# 调用聊天模型
response = chat_model.invoke(prompt)
print(response.content)

```

```
我是Bob，AGI课堂的客服助手。有什么我可以帮助你的吗?
```

4.4.6 插入消息列表：MessagesPlaceholder

当你不确定消息提示模板使用什么角色，或者希望在格式化过程中**插入消息列表**时，该怎么办？ 这就需要使用 MessagesPlaceholder，负责在特定位置添加消息列表。

使用场景：多轮对话系统存储历史消息以及Agent的中间步骤处理此功能非常有用。

举例1：

```python
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_core.messages import HumanMessage

prompt_template = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful assistant"),
    MessagesPlaceholder("msgs")
])
# prompt_template.invoke({"msgs": [HumanMessage(content="hi!")]})

prompt_template.format_messages(msgs=[HumanMessage(content="hi!")])
```

```
[SystemMessage(content='You are a helpful assistant', additional_kwargs={},
response_metadata={}),
HumanMessage(content='hi!', additional_kwargs={}, response_metadata={})]
```

这将生成两条消息，第一条是系统消息，第二条是我们传入的 HumanMessage。 如果我们传入了 5 条消息，那么总共会生成 6 条消息（系统消息加上传入的 5 条消息）。 这对于将一系列消息插入到特定位置非常有用。

举例2：存储对话历史内容

```python
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_core.messages import AIMessage

prompt = ChatPromptTemplate.from_messages(
    [
        ("system", "You are a helpful assistant."),
        MessagesPlaceholder("history"),
        ("human", "{question}")
    ]
)

prompt.format_messages(
    history=[HumanMessage(content="1+2*3 = ?"),AIMessage(content="1+2*3=7")],
    question="我刚才问题是什么？ ")

# prompt.invoke(
#     {
#         "history": [("human", "what's 5 + 2"), ("ai", "5 + 2 is 7")],
#         "question": "now multiply that by 4"
```

```
#    }
#)
```

举例3：

```
#1.导入相关包
from langchain_core.prompts import (ChatPromptTemplate, HumanMessagePromptTemplate,
MessagesPlaceholder)

# 2.定义消息模板
prompt = ChatPromptTemplate.from_messages([
    SystemMessagePromptTemplate.from_template("你是{role}"),
    MessagesPlaceholder(variable_name="intermediate_steps"),
    HumanMessagePromptTemplate.from_template("{query}")
])

# 3.定义消息对象（运行时填充中间步骤的结果）
intermediate = [
    SystemMessage(name="search", content="北京: 晴, 25℃")
]
# 4.格式化聊天消息提示词模版
prompt.format_messages(
    role="天气预报员",
    intermediate_steps=intermediate,
    query="北京天气怎么样？"
)
```

```
[SystemMessage(content='你是天气预报员', additional_kwargs={}, response_metadata={}),
SystemMessage(content='北京: 晴, 25℃', additional_kwargs={}, response_metadata={},
name='search'),
HumanMessage(content='北京天气怎么样？', additional_kwargs={}, response_metadata=
{})]
```

举例4：作为拓展，课下看看即可

```
# 1.导入相关包
from langchain_core.prompts import (ChatPromptTemplate, HumanMessagePromptTemplate,
MessagesPlaceholder)
from langchain_core.messages import AIMessage, HumanMessage

# 2，定义HumanMessage对象
human_message = HumanMessage(content="学习编程的最好方法是什么?")
# 3.定义AIMessage对象
ai_message = AIMessage(
    content="""\
1. 选择一门编程语言：选择一门你想学习的编程语言.

2.从基础开始：熟悉基本的编程概念，如变量、数据类型和控制结构.

3. 练习，练习，再练习：学习编程的最好方法是通过实践经验\
"""
)
```

```
# 4. 定义提示词
human_prompt = "用{word_count}个词总结我们到目前为止的对话"

# 5. 定义提示词模版
human_message_template = HumanMessagePromptTemplate.from_template(human_prompt)

chat_prompt = ChatPromptTemplate.from_messages(
    [
        MessagesPlaceholder(variable_name="conversation"),
        human_message_template
    ]
)
# 6.格式化聊天消息提示词模版
messages1 = chat_prompt.format_messages(
    conversation=[human_message, ai_message], word_count="10"
)
print(messages1)
```

```
[HumanMessage(content='学习编程的最好方法是什么?', additional_kwargs={}, response_metadata={}), AIMessage(content='1. 选择一门编程语言：选择一门你想学习的编程语言.\n\n2.从基础开始：熟悉基本的编程概念，如变量、数据类型和控制结构.\n\n3. 练习，练习，再练习：学习编程的最好方法是通过实践经验', additional_kwargs={}, response_metadata={}), HumanMessage(content='用10个词总结我们到目前为止的对话', additional_kwargs={}, response_metadata={})]
```

4.5 具体使用：少量样本示例的提示词模板

4.5.1 使用说明

在构建prompt时，可以通过构建一个 **少量示例列表** 去进一步格式化prompt，这是一种简单但强大的指导生成的方式，在某些情况下可以 **显著提高模型性能** 。

少量示例提示模板可以由 **一组示例** 或一个负责从定义的集合中选择 **一部分示例** 的示例选择器构建。

- 前者：使用 **FewShotPromptTemplate** 或 **FewShotChatMessagePromptTemplate**
- 后者：使用 **Example selectors(示例选择器)**

每个示例的结构都是一个 **字典** ，其中 **键** 是输入变量， **值** 是输入变量的值。

体会：zeroshot会导致低质量回答

```
from langchain_openai import ChatOpenAI

import os
import dotenv
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()

os.environ['OPENAI_API_KEY'] = os.getenv("OPENAI_API_KEY1")
```

```
os.environ['OPENAI_BASE_URL'] = os.getenv("OPENAI_BASE_URL")

chat_model = ChatOpenAI(model="gpt-4o-mini",
                        temperature=0.4)

res = chat_model.invoke("2 🧬 9是多少?")
print(res.content)
```

2 🧬 9 的计算方式取决于你所用的符号“🧬”的含义。请提供更多信息或者说明这个符号代表什么运算。

4.5.2 FewShotPromptTemplate的使用

举例1：

```
from langchain.prompts import PromptTemplate
from langchain.prompts.few_shot import FewShotPromptTemplate

#1、创建示例集合
examples = [
    {"input": "北京天气怎么样", "output": "北京市"},
    {"input": "南京下雨吗", "output": "南京市"},
    {"input": "武汉热吗", "output": "武汉市"}
]

#2、创建PromptTemplate实例
example_prompt = PromptTemplate.from_template(
    template="Input: {input}\nOutput: {output}"
)

#3、创建FewShotPromptTemplate实例
prompt = FewShotPromptTemplate(
    examples=examples,
    example_prompt=example_prompt,
    suffix="Input: {input}\nOutput:",  # 要放在示例后面的提示模板字符串。
    input_variables=["input"]  # 传入的变量
)

#4、调用
prompt = prompt.invoke({"input": "长沙多少度"})

print("===Prompt===")
print(prompt)


```

```
===Prompt===
Input: 北京天气怎么样
Output: 北京市

Input: 南京下雨吗
Output: 南京市

Input: 武汉热吗
Output: 武汉市

Input: 长沙多少度
Output:

```

结合大模型调用：

```python
import os
import dotenv
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()

os.environ['OPENAI_API_KEY'] = os.getenv("OPENAI_API_KEY1")
os.environ['OPENAI_BASE_URL'] = os.getenv("OPENAI_BASE_URL")

#获取大模型
chat_model = ChatOpenAI(model="gpt-4o-mini")

#调用
print("===Response===")
response = chat_model.invoke(prompt)
print(response.content)
```

```
===Response===
长沙市
```

举例2：

```python
#1、创建提示模板
from langchain.prompts import PromptTemplate

# 创建提示模板，配置一个提示模板，将一个示例格式化为字符串
prompt_template = "你是一个数学专家,算式： {input} 值： {output} 使用： {description} "

# 这是一个提示模板，用于设置每个示例的格式
prompt_sample = PromptTemplate.from_template(prompt_template)

#2、提供示例
examples = [
```

```
    {"input": "2+2", "output": "4", "description": "加法运算"},
    {"input": "5-2", "output": "3", "description": "减法运算"},
]


#3、创建一个FewShotPromptTemplate对象
from langchain.prompts.few_shot import FewShotPromptTemplate


prompt = FewShotPromptTemplate(
    examples=examples,
    example_prompt=prompt_sample,
    suffix="你是一个数学专家,算式: {input} 值: {output}",
    input_variables=["input", "output"]
)
print(prompt.invoke({"input": "2*5", "output": "10"}))

#4、初始化大模型，然后调用
import os
import dotenv
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()

os.environ['OPENAI_API_KEY'] = os.getenv("OPENAI_API_KEY1")
os.environ['OPENAI_BASE_URL'] = os.getenv("OPENAI_BASE_URL")

chat_model = ChatOpenAI(model="gpt-4o-mini")

result = chat_model.invoke(prompt.invoke({"input": "2*5", "output": "10"}))
print(result.content)  # 使用: 乘法运算
```

```
你是一个数学专家,算式： 2*5 值： 10 使用： 乘法运算

如果你还需要其他的数学计算或者有任何问题，请随时告诉我！
```

4.5.3 FewShotChatMessagePromptTemplate的使用

除了FewShotPromptTemplate之外，FewShotChatMessagePromptTemplate是专门为 **聊天对话场景** 设计的少样本（few-shot）提示模板，它继承自 `FewShotPromptTemplate`，但针对聊天消息的格式进行了优化。

特点：

- 自动将示例格式化为聊天消息（`HumanMessage` / `AIMessage` 等）
- 输出结构化聊天消息（`List[BaseMessage]`）
- 保留对话轮次结构

举例1：基本结构

```
from langchain.prompts import (
```

```
    FewShotChatMessagePromptTemplate,
    ChatPromptTemplate
)

# 1.示例消息格式
examples = [
    {"input": "1+1等于几？", "output": "1+1等于2"},
    {"input": "法国的首都是？", "output": "巴黎"}
]

# 2.定义示例的消息格式提示词模版
msg_example_prompt = ChatPromptTemplate.from_messages([
    ("human", "{input}"),
    ("ai", "{output}"),
])

# 3.定义FewShotChatMessagePromptTemplate对象
few_shot_prompt = FewShotChatMessagePromptTemplate(
    example_prompt=msg_example_prompt,
    examples=examples
)
# 4.输出格式化后的消息
print(few_shot_prompt.format())
```

```
Human: 1+1等于几?
AI: 1+1等于2
Human: 法国的首都是?
AI: 巴黎
```

举例2：

使用方式：将原始输入和被选中的示例组一起加入Chat提示词模版中。

```
# 1.导入相关包
from langchain_core.prompts import (FewShotChatMessagePromptTemplate,
ChatPromptTemplate)

# 2.定义示例组
examples = [
    {"input": "2🦜2", "output": "4"},
    {"input": "2🦜3", "output": "8"},
]

# 3.定义示例的消息格式提示词模版
example_prompt = ChatPromptTemplate.from_messages([
    ('human', '{input} 是多少?'),
    ('ai', '{output}')
])

# 4.定义FewShotChatMessagePromptTemplate对象
few_shot_prompt = FewShotChatMessagePromptTemplate(
    examples=examples, # 示例组
    example_prompt=example_prompt, # 示例提示词词模版
```

```
)
# 5.输出完整提示词的消息模版
final_prompt = ChatPromptTemplate.from_messages(
    [
        ('system', '你是一个数学奇才'),
        few_shot_prompt,
        ('human', '{input}'),
    ]
)

#6.提供大模型
import os
import dotenv
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()

os.environ['OPENAI_API_KEY'] = os.getenv("OPENAI_API_KEY1")
os.environ['OPENAI_BASE_URL'] = os.getenv("OPENAI_BASE_URL")

chat_model = ChatOpenAI(model="gpt-4o-mini",
                        temperature=0.4)

chat_model.invoke(final_prompt.invoke(input="2🦜4")).content
```

```
'2🦜4 等于 16。'
```

举例3：与前面类似

```
# 1.导入相关包
from langchain_core.prompts import (FewShotChatMessagePromptTemplate,
ChatPromptTemplate)

# 2.定义示例组
examples = [
    {"input": "2+2", "output": "4"},
    {"input": "2+3", "output": "5"},
]

# 3.定义示例的消息格式提示词模版
example_prompt = ChatPromptTemplate.from_messages([('human', 'What is {input}?'), ('ai',
'{output}')])

# 4.定义FewShotChatMessagePromptTemplate对象
few_shot_prompt = FewShotChatMessagePromptTemplate(
    examples=examples,  # 示例组
    example_prompt=example_prompt,  # 示例提示词词模版
)
# 5.输出完整提示词的消息模版
final_prompt = ChatPromptTemplate.from_messages(
    [
        ('system', 'You are a helpful AI Assistant'),
        few_shot_prompt,
```

```
        ('human', '{input}'),
    ]
)
# 6.格式化完整消息
#final_prompt.format(input="What is 4+4?")
# 或者
final_prompt.format_messages(input="What is 4+4?")
```

```
[SystemMessage(content='You are a helpful AI Assistant', additional_kwargs={}, response_metadata={}),
HumanMessage(content='What is 2+2?', additional_kwargs={}, response_metadata={}),
AIMessage(content='4', additional_kwargs={}, response_metadata={}),
HumanMessage(content='What is 2+3?', additional_kwargs={}, response_metadata={}),
AIMessage(content='5', additional_kwargs={}, response_metadata={}),
HumanMessage(content='What is 4+4?', additional_kwargs={}, response_metadata={})]
```

4.5.4 Example selectors(示例选择器)

前面FewShotPromptTemplate的特点是，无论输入什么问题，都会包含全部示例。在实际开发中，我们可以根据当前输入，使用示例选择器，从大量候选示例中选取最相关的示例子集。

使用的好处：避免盲目传递所有示例，减少 token 消耗的同时，还可以提升输出效果。

示例选择策略：语义相似选择、长度选择、最大边际相关示例选择等

- **语义相似选择**：通过余弦相似度等度量方式评估语义相关性，选择与输入问题最相似的 `k` 个示例。
- **长度选择**：根据输入文本的长度，从候选示例中筛选出长度最匹配的示例。增强模型对文本结构的理解。比语义相似度计算更轻量，适合对响应速度要求高的场景。
- **最大边际相关示例选择**：优先选择与输入问题语义相似的示例；同时，通过惩罚机制避免返回同质化的内容

> - 余弦相似度是通过计算两个向量的夹角余弦值来衡量它们的相似性。它的值范围在-1到1之间：当两个向量方向相同时值为1；夹角为90°时值为0；方向完全相反时为-1。
> - 数学表达式：余弦相似度 = (A·B) / (||A|| * ||B||)。其中A·B是点积，||A||和||B||是向量的模（长度）。

举例1：

```
pip install chromadb
```

```
# 1.导入相关包
from langchain_community.vectorstores import Chroma
from langchain_core.example_selectors import SemanticSimilarityExampleSelector
import os
import dotenv
from langchain_openai import OpenAIEmbeddings

dotenv.load_dotenv()

# 2.定义嵌入模型
os.environ['OPENAI_API_KEY'] = os.getenv("OPENAI_API_KEY1")
```

```python
os.environ['OPENAI_BASE_URL'] = os.getenv("OPENAI_BASE_URL")

embeddings_model = OpenAIEmbeddings(
    model="text-embedding-ada-002"
)

# 3.定义示例组
examples = [
    {
        "question":"谁活得更久，穆罕默德·阿里还是艾伦·图灵?",
        "answer":"""
        接下来还需要问什么问题吗?
        追问：穆罕默德·阿里去世时多大年纪?
        中间答案：穆罕默德·阿里去世时享年74岁。
        """,
    },
    {
        "question":"craigslist的创始人是什么时候出生的？",
        "answer":"""
        接下来还需要问什么问题吗?
        追问：谁是craigslist的创始人?
        中级答案：Craigslist是由克雷格·纽马克创立的。
        """,
    },
    {
        "question":"谁是乔治·华盛顿的外祖父？",
        "answer":"""
        接下来还需要问什么问题吗?
        追问：谁是乔治·华盛顿的母亲?
        中间答案：乔治·华盛顿的母亲是玛丽·鲍尔·华盛顿。
        """,
    },
    {
        "question":"《大白鲨》和《皇家赌场》的导演都来自同一个国家吗？",
        "answer":"""
        接下来还需要问什么问题吗?
        追问：《大白鲨》的导演是谁?
        中级答案：《大白鲨》的导演是史蒂文·斯皮尔伯格。
        """,
    },
]

# 4.定义示例选择器
example_selector = SemanticSimilarityExampleSelector.from_examples(
    # 这是可供选择的示例列表
    examples,
    # 这是用于生成嵌入的嵌入类，用于衡量语义相似性
    embeddings_model,
    # 这是用于存储嵌入并进行相似性搜索的 VectorStore 类
    Chroma,
    # 这是要生成的示例数量
    k=1,
)

# 选择与输入最相似的示例
```

```
question = "玛丽·鲍尔·华盛顿的父亲是谁?"
selected_examples = example_selector.select_examples({"question": question})
print(f"与输入最相似的示例：{selected_examples}")

# for example in selected_examples:
#     print("\n")
#     for k, v in example.items():
#         print(f"{k}: {v}")
```

```
question: 谁是乔治·华盛顿的外祖父?
answer:
接下来还需要问什么问题吗?
追问：谁是乔治·华盛顿的母亲?
中间答案：乔治·华盛顿的母亲是玛丽·鲍尔·华盛顿
```

举例2：结合 FewShotPromptTemplate 使用

这里使用FAISS，需安装：

```
pip install faiss-cpu
#或
conda install faiss-cpu
```

```
# 1.导入相关包
from langchain_community.vectorstores import FAISS
from langchain_core.example_selectors import SemanticSimilarityExampleSelector
from langchain_core.prompts import FewShotPromptTemplate, PromptTemplate
from langchain_openai import OpenAIEmbeddings

# 2.定义示例提示词模版
example_prompt = PromptTemplate.from_template(
    template="Input: {input}\nOutput: {output}",
)

# 3.创建一个示例提示词模版
examples = [
    {"input": "高兴", "output": "悲伤"},
    {"input": "高", "output": "矮"},
    {"input": "长", "output": "短"},
    {"input": "精力充沛", "output": "无精打采"},
    {"input": "阳光", "output": "阴暗"},
    {"input": "粗糙", "output": "光滑"},
    {"input": "干燥", "output": "潮湿"},
    {"input": "富裕", "output": "贫穷"},
]

# 4.定义嵌入模型
embeddings = OpenAIEmbeddings(
    model="text-embedding-ada-002"
)

# 5.创建语义相似性示例选择器
```

```python
example_selector = SemanticSimilarityExampleSelector.from_examples(
    examples,
    embeddings,
    FAISS,
    k=2,
)
#或者
#example_selector = SemanticSimilarityExampleSelector(
#    examples,
#    embeddings,
#    FAISS,
#    k=2
#)

# 6.定义小样本提示词模版
similar_prompt = FewShotPromptTemplate(
    example_selector=example_selector,
    example_prompt=example_prompt,
    prefix="给出每个词组的反义词",
    suffix="Input: {word}\nOutput:",
    input_variables=["word"],
)

response = similar_prompt.invoke({"word": "忧郁"})
print(response.text)
```

```
给出每个词组的反义词

Input: 高兴
Output: 悲伤

Input: 阳光
Output: 阴暗

Input: 忧郁
Output:
```

4.6 具体使用：PipelinePromptTemplate(了解)

用于将多个提示模板**按顺序组合成处理管道**，实现分阶段、模块化的提示构建。它的核心作用类似于软件开发中的 **管道模式**（Pipeline Pattern），通过串联多个提示处理步骤，实现复杂的提示生成逻辑。

特点：

- 将复杂提示拆解为多个处理阶段，每个阶段使用独立的提示模板
- 前一个模板的输出作为下一个模板的输入变量
- 使用场景：解决单一超大提示模板难以维护的问题

说明：PipelinePromptTemplate在langchain 0.3.22版本中被标记为过时，在 langchain-core==1.0之前不会删除它。

https://python.langchain.com/api_reference/core/prompts/langchain_core.prompts.pipeline.PipelinePromptTemplate.html

举例：

```
from langchain_core.prompts.pipeline import PipelinePromptTemplate
from langchain_core.prompts.prompt import PromptTemplate


# 阶段1：问题分析
analysis_template = PromptTemplate.from_template("""
分析这个问题：{question}
关键要素：
""")

# 阶段2：知识检索
retrieval_template = PromptTemplate.from_template("""
基于以下要素搜索资料：
{analysis_result}
搜索关键词：
""")

# 阶段3：生成最终回答
answer_template = PromptTemplate.from_template("""
综合以下信息回答问题：
{retrieval_result}
最终答案：
""")

# 构建管道
pipeline = PipelinePromptTemplate(
    final_prompt=answer_template,
    pipeline_prompts=[
        ("analysis_result", analysis_template),
        ("retrieval_result", retrieval_template)
    ]
)

print(pipeline.format(question="量子计算的优势是什么？"))
```

```
综合以下信息回答问题：

基于以下要素搜索资料：

分析这个问题：量子计算的优势是什么？
关键要素：

搜索关键词：

最终答案：
```

上述代码执行时，提示PipelinePromptTemplate已过时，代码更新如下：

```
from langchain_core.prompts.prompt import PromptTemplate

# 阶段1：问题分析
analysis_template = PromptTemplate.from_template("""
```

```
分析这个问题：{question}
关键要素：
""")

# 阶段2：知识检索
retrieval_template = PromptTemplate.from_template("""
基于以下要素搜索资料：
{analysis_result}
搜索关键词：
""")

# 阶段3：生成最终回答
answer_template = PromptTemplate.from_template("""
综合以下信息回答问题：
{retrieval_result}
最终答案：
""")

# 逐步执行管道提示
pipeline_prompts = [
    ("analysis_result", analysis_template),
    ("retrieval_result", retrieval_template)
]


my_input = {"question": "量子计算的优势是什么？"}

# print(pipeline_prompts)

# [('analysis_result', PromptTemplate(input_variables=['question'], input_types={}, partial_variables={}, template='\n分析这个问题：{question}\n关键要素：\n')), ('retrieval_result', PromptTemplate(input_variables=['analysis_result'], input_types={}, partial_variables={}, template='\n基于以下要素搜索资料：\n{analysis_result}\n搜索关键词：\n'))]

for name, prompt in pipeline_prompts:
    # 调用当前提示模板并获取字符串结果
    result = prompt.invoke(my_input).to_string()
    # 将结果添加到输入字典中供下一步使用
    my_input[name] = result

# 生成最终答案
my_output = answer_template.invoke(my_input).to_string()
print(my_output)
```

4.7 具体使用：自定义提示词模版(了解)

在创建prompt时，我们也可以按照自己的需求去创建自定义的提示模版。

步骤：

- 自定义类继承提示词基类模版BasePromptTemplate
- 重写format、format_prompt、from_template方法

举例：

```python
# 1.导入相关包
from typing import List, Dict, Any
from langchain.prompts import BasePromptTemplate
from langchain.prompts import PromptTemplate
from langchain.schema import PromptValue

#  2.自定义提示词模版
class SimpleCustomPrompt(BasePromptTemplate):
    """简单自定义提示词模板"""
    template: str

    def __init__(self, template: str, **kwargs):
        # 使用PromptTemplate解析输入变量
        prompt = PromptTemplate.from_template(template)

        super().__init__(
            input_variables=prompt.input_variables,
            template=template,
            **kwargs
        )

    def format(self, **kwargs: Any) -> str:
        """格式化提示词"""
        # print("kwargs:", kwargs)
        # print("self.template:", self.template)

        return self.template.format(**kwargs)

    def format_prompt(self, **kwargs: Any) -> PromptValue:
        """实现抽象方法"""
        return PromptValue(text=self.format(**kwargs))

    @classmethod
    def from_template(cls, template: str, **kwargs) -> "SimpleCustomPrompt":
        """从模板创建实例"""
        return cls(template=template, **kwargs)

# 3.使用自定义提示词模版
custom_prompt = SimpleCustomPrompt.from_template(
    template="请回答关于{subject}的问题：{question}"
)

#  4.格式化提示词
formatted = custom_prompt.format(
    subject="人工智能",
    question="什么是LLM？"
)

print(formatted)
```

```
请回答关于人工智能的问题：什么是LLM？
```

4.8 从文档中加载Prompt(了解)

一方面，将想要设定prompt所支持的格式保存为JSON或者YAML格式文件。

另一方面，通过读取指定路径的格式化文件，获取相应的prompt。

目的与使用场景：

- 为了便于共享、存储和加强对prompt的版本控制。
- 当我们的prompt模板数据较大时，我们可以使用外部导入的方式进行管理和维护。

4.8.1 yaml格式提示词

asset下创建yaml文件：prompt.yaml

```yaml
_type:
    "prompt"
input_variables:
    ["name","what"]
template:
    "请给{name}讲一个关于{what}的故事"
```

代码：

```python
from langchain_core.prompts import load_prompt
from dotenv import load_dotenv

load_dotenv()

prompt = load_prompt("asset/prompt.yaml", encoding="utf-8")
# print(prompt)
print(prompt.format(name="年轻人", what="滑稽"))
```

```
请给年轻人讲一个关于滑稽的笑话
```

4.8.2 json格式提示词

asset下创建json文件：prompt.json

```json
{
  "_type": "prompt",
  "input_variables": ["name", "what"],
  "template": "请{name}讲一个{what}的故事。"
}
```

代码：

```python
from langchain_core.prompts import load_prompt
from dotenv import load_dotenv

load_dotenv()

prompt = load_prompt("asset/prompt.json",encoding="utf-8")
print(prompt.format(name="张三",what="搞笑的"))
```

```
请张三讲一个搞笑的的故事。
```

5、Model I/O之Output Parsers

语言模型返回的内容通常都是字符串的格式（文本格式），但在实际AI应用开发过程中，往往希望model可以返回**更直观、更格式化的内容**，以确保应用能够顺利进行后续的逻辑处理。此时，LangChain提供的 `输出解析器` 就派上用场了。

输出解析器（Output Parser）负责获取 LLM 的输出并将其转换为更合适的格式。**这在应用开发中及其重要。**

5.1 输出解析器的分类

LangChain有许多不同类型的输出解析器

- `StrOutputParser`：字符串解析器
- `JsonOutputParser`：JSON解析器，确保输出符合特定JSON对象格式
- `XMLOutputParser`：XML解析器，允许以流行的XML格式从LLM获取结果
- `CommaSeparatedListOutputParser`：CSV解析器，模型的输出以逗号分隔，以列表形式返回输出
- `DatetimeOutputParser`：日期时间解析器，可用于将 LLM 输出解析为日期时间格式

除了上述常用的输出解析器之外，还有：

- `EnumOutputParser`：枚举解析器，将LLM的输出，解析为预定义的枚举值
- `StructuredOutputParser`：将**非结构化文本**转换为预定义格式的**结构化数据**（如字典）
- `OutputFixingParser`：输出修复解析器，用于自动修复格式错误的解析器，比如将返回的不符合预期格式的输出，尝试修正为正确的结构化数据（如 JSON）
- `RetryOutputParser`：重试解析器，当主解析器（如 JSONOutputParser）因格式错误无法解析 LLM 的输出时，通过调用另一个 LLM 自动修正错误，并重新尝试解析

5.2 具体解析器的使用

① 字符串解析器 StrOutputParser

StrOutputParser 简单地将 `任何输入` 转换为 `字符串`。它是一个简单的解析器，从结果中提取content字段

举例：将一个对话模型的输出结果，解析为字符串输出

```python
from langchain_core.messages import HumanMessage, SystemMessage
from langchain_core.output_parsers import StrOutputParser

import os
import dotenv
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()

os.environ['OPENAI_API_KEY'] = os.getenv("OPENAI_API_KEY1")
os.environ['OPENAI_BASE_URL'] = os.getenv("OPENAI_BASE_URL")

chat_model = ChatOpenAI(model="gpt-4o-mini")
```

```python
messages = [
    SystemMessage(content="将以下内容从英语翻译成中文"),
    HumanMessage(content="It's a nice day today"),
]

result = chat_model.invoke(messages)
print(type(result))
print(result)

parser = StrOutputParser()
#使用parser处理model返回的结果
response = parser.invoke(result)
print(type(response))
print(response)
```

```
<class 'langchain_core.messages.ai.AIMessage'>

content='今天是个好天。' additional_kwargs={'refusal': None} response_metadata={'token_usage': {'completion_tokens': 7, 'prompt_tokens': 25, 'total_tokens': 32, 'completion_tokens_details': {'accepted_prediction_tokens': 0, 'audio_tokens': 0, 'reasoning_tokens': 0, 'rejected_prediction_tokens': 0}, 'prompt_tokens_details': {'audio_tokens': 0, 'cached_tokens': 0}}, 'model_name': 'gpt-4o-mini-2024-07-18', 'system_fingerprint': 'fp_efad92c60b', 'id': 'chatcmpl-BpPd126GlvwFI3TpL2EMaInxruhk0', 'service_tier': None, 'finish_reason': 'stop', 'logprobs': None} id='run--690e05f2-39ad-4ff7-98fd-ef3ad00e6133-0' usage_metadata={'input_tokens': 25, 'output_tokens': 7, 'total_tokens': 32, 'input_token_details': {'audio': 0, 'cache_read': 0}, 'output_token_details': {'audio': 0, 'reasoning': 0}}

<class 'str'>

今天是个好天。
```

② JSON解析器 JsonOutputParser

JsonOutputParser，即JSON输出解析器，是一种用于将大模型的`自由文本输出`转换为`结构化JSON数据`的工具。

适合场景：特别适用于需要严格结构化输出的场景，比如 API 调用、数据存储或下游任务处理。

实现方式

- 方式1：用户自己通过提示词指明返回Json格式
- 方式2：借助JsonOutputParser的`get_format_instructions()`，生成格式说明，指导模型输出 JSON 结构

举例1：

```
from langchain_core.output_parsers import JsonOutputParser
from langchain_core.prompts import ChatPromptTemplate

chat_model = ChatOpenAI(model="gpt-4o-mini")

chat_prompt_template = ChatPromptTemplate.from_messages([
    ("system","你是一个靠谱的{role}"),
    ("human","{question}")
])

parser = JsonOutputParser()

# 方式1:
result = chat_model.invoke(chat_prompt_template.format_messages(role="人工智能专家",question="人工智能用英文怎么说？问题用q表示，答案用a表示，返回一个JSON格式"))
print(result)
print(type(result))

parser.invoke(result)

# 方式2:
# chain = chat_prompt_template | chat_model | parser
# chain.invoke({"role":"人工智能专家","question":"人工智能用英文怎么说？问题用q表示，答案用a表示，返回一个JSON格式"})
```

```
content='json\n{\n  "q": "人工智能用英文怎么说？",\n  "a": "Artificial Intelligence"\n}\n```' additional_kwargs={'refusal': None} response_metadata={'token_usage': {'completion_tokens': 28, 'prompt_tokens': 40, 'total_tokens': 68, 'completion_tokens_details': {'accepted_prediction_tokens': 0, 'audio_tokens': 0, 'reasoning_tokens': 0, 'rejected_prediction_tokens': 0}, 'prompt_tokens_details': {'audio_tokens': 0, 'cached_tokens': 0}}, 'model_name': 'gpt-4o-mini-2024-07-18', 'system_fingerprint': 'fp_efad92c60b', 'id': 'chatcmpl-ByBTRPL6LKkHJgsjbRwVawe1L92jJ', 'service_tier': None, 'finish_reason': 'stop', 'logprobs': None} id='run--aadc94df-3608-4b1b-adf6-4a53ef8b640c-0' usage_metadata={'input_tokens': 40, 'output_tokens': 28, 'total_tokens': 68, 'input_token_details': {'audio': 0, 'cache_read': 0}, 'output_token_details': {'audio': 0, 'reasoning': 0}}

<class 'langchain_core.messages.ai.AIMessage'>

{'q': '人工智能用英文怎么说？', 'a': 'Artificial Intelligence'}
```

举例2：使用指定的JSON格式

```
from langchain_core.output_parsers import JsonOutputParser

output_parser = JsonOutputParser()
# 返回一些指令或模板，这些指令告诉系统如何解析或格式化输出数据
format_instructions = output_parser.get_format_instructions()
print(format_instructions)
```

基于此：

```
# 引入依赖包
from langchain_core.output_parsers import JsonOutputParser
from langchain_core.prompts import PromptTemplate

# 初始化语言模型
chat_model = ChatOpenAI(model="gpt-4o-mini")

joke_query = "告诉我一个笑话。"

# 定义Json解析器
parser = JsonOutputParser()

# 定义提示词模版
# 注意，提示词模板中需要部分格式化解析器的格式要求format_instructions
prompt = PromptTemplate(
    template="回答用户的查询.\n{format_instructions}\n{query}\n",
    input_variables=["query"],
    partial_variables={"format_instructions": parser.get_format_instructions()},
)

# 5.使用LCEL语法组合一个简单的链
chain = prompt | chat_model | parser
# 6.执行链
output = chain.invoke({"query": "给我讲一个笑话"})
print(output)
```

```
{'joke': '为什么海洋总是咸的？因为它有太多的"海"湿的事情发生！'}
```

③ XML解析器 XMLOutputParser

XMLOutputParser，将模型的自由文本输出转换为可编程处理的 XML 数据。

如何实现：在 `PromptTemplate` 中指定 XML 格式要求，让模型返回 `<tag>content</tag>` 形式的数据。

注意：XMLOutputParser 不会直接将模型的输出保持为原始XML字符串，而是会解析XML并转换成 `Python字典` （或类似结构化的数据）。目的是为了方便程序后续处理数据，而不是单纯保留XML格式。

举例1：不使用XMLOutputParser，通过大模型的能力，返回xml格式数据

```python
# 初始化语言模型
chat_model = ChatOpenAI(model="gpt-4o-mini")

# 测试模型的xml解析效果
actor_query = "生成汤姆·汉克斯的简短电影记录"
output = chat_model.invoke(f"""{actor_query}请将影片附在<movie></movie>标签中"""
)
print(type(output))  # <class 'langchain_core.messages.ai.AIMessage'>
print(output.content)
```

```
<class 'langchain_core.messages.ai.AIMessage'>
以下是汤姆·汉克斯的一些著名电影记录，使用了`<movie></movie>`标签：

<movie>
<title>大白鲨</title>
<year>1975</year>
<description>汤姆·汉克斯并未参演该片，但它代表了他所崇拜的早期冒险和恐怖电影。
</description>
</movie>

<movie>
<title>费城故事</title>
<year>1993</year>
<description>汉克斯在片中饰演一名因碍于艾滋病而遭受歧视的律师，展现了强大的表演能力。
</description>
</movie>

<movie>
<title>拯救大兵瑞恩</title>
<year>1998</year>
<description>汤姆·汉克斯在这部二战电影中饰演一位军官，领导小队寻找失踪士兵，表现出勇
气与人性。</description>
</movie>

<movie>
<title>阿甘正传</title>
<year>1994</year>
<description>汉克斯饰演智力有所缺陷的阿甘，凭借其纯真和毅力走过了动荡的历史，成为经典
角色。</description>
</movie>

<movie>
<title>云图</title>
<year>2012</year>
<description>这部作品展示了不同历史时期的人物，汉克斯在其中扮演多个角色，体现了人与时
间的关系。</description>
</movie>

<movie>
```

```
<title>大地惊雷</title>
<year>2000</year>
<description>汉克斯在这部影片中担任制片人和主演，讲述了关于希望与重生的感人故事。
</description>
</movie>

<movie>
<title>西线无战事</title>
<year>2022</year>
<description>虽然汉克斯并没有在片中出演，但作为制片人，他对反战主题的传播作出贡献。
</description>
</movie>

这些电影展示了汤姆·汉克斯在不同时期所做出的多样化和深刻的艺术贡献。
```

举例2：体会XMLOutputParser的格式

```python
from langchain_core.output_parsers import XMLOutputParser

output_parser = XMLOutputParser()
# 返回一些指令或模板，这些指令告诉系统如何解析或格式化输出数据
format_instructions = output_parser.get_format_instructions()
print(format_instructions)
```

```
The output should be formatted as a XML file.
1. Output should conform to the tags below.
2. If tags are not given, make them on your own.
3. Remember to always open and close all the tags.

As an example, for the tags ["foo", "bar", "baz"]:
1. String "<foo>
   <bar>
      <baz></baz>
   </bar>
</foo>" is a well-formatted instance of the schema.
2. String "<foo>
   <bar>
   </foo>" is a badly-formatted instance.
3. String "<foo>
   <tag>
   </tag>
</foo>" is a badly-formatted instance.

Here are the output tags:
```
None
```
```

举例3：XMLOutputParser 的使用

```python
# 1.导入相关包
from langchain_core.output_parsers import XMLOutputParser
from langchain_core.prompts import PromptTemplate
from langchain_openai import ChatOpenAI

# 2. 初始化语言模型
chat_model = ChatOpenAI(model= "gpt-4o-mini")

# 3.测试模型的xml解析效果
actor_query = "生成汤姆·汉克斯的简短电影记录,使用中文回复"

# 4.定义XMLOutputParser对象
parser = XMLOutputParser()

# 5.定义提示词模版对象
# prompt = PromptTemplate(
#     template="{query}\n{format_instructions}",
#     input_variables=["query","format_instructions"],
#     partial_variables={"format_instructions": parser.get_format_instructions()},
#)

prompt_template = PromptTemplate.from_template("{query}\n{format_instructions}")

prompt_template1 =
prompt_template.partial(format_instructions=parser.get_format_instructions())

response = chat_model.invoke(prompt_template1.format(query=actor_query))
print(response.content)
```

```xml
<汤姆汉克斯电影记录>
<电影>
<名称>阿甘正传</名称>
<年份>1994</年份>
<角色>福里斯特·甘</角色>
<类型>剧情/喜剧</类型>
</电影>
<电影>
<名称>拯救大兵瑞恩</名称>
<年份>1998</年份>
<角色>米勒上尉</角色>
<类型>战争/剧情</类型>
</电影>
<电影>
<名称>沉默的羔羊</名称>
<年份>2001</年份>
<角色>查克·诺兰</角色>
<类型>冒险/剧情</类型>
</电影>
<电影>
<名称>角斗士</名称>
<年份>2000</年份>
<角色>罗马指挥官</角色>
```

```
    <类型>历史/剧情</类型>
    </电影>
    <电影>
    <名称>大兵瑞恩</名称>
    <年份>1998</年份>
    <角色>米奇·布朗</角色>
    <类型>战争/动作</类型>
    </电影>
    </汤姆汉克斯电影记录>
```

继续：

```
# 方式1
response = chat_model.invoke(prompt_template1.format(query=actor_query))
result = parser.invoke(response)
print(result)
print(type(result))

# 方式2
# chain = prompt_template1 | chat_model | parser
# result = chain.invoke({"query":actor_query})
# print(result)
# print(type(result))
```

```
{'电影记录': [{'演员': [{'名字': '汤姆·汉克斯'}, {'代表作品': [{'电影': [{'标题': '阿甘正传'}, {'年份': '1994'}, {'简介': '讲述了一个智力简单却拥有传奇人生的男子阿甘的故事。'}]}, {'电影': [{'标题': '拯救大兵瑞恩'}, {'年份': '1998'}, {'简介': '一支美国小队在诺曼底登陆后，深入敌后，寻找并拯救被困的士兵瑞恩的故事。'}]}, {'电影': [{'标题': '绿色里小屋'}, {'年份': '1999'}, {'简介': '改编自斯蒂芬·金的小说，讲述了一位狱警与一名死刑犯之间的奇妙关系。'}]}, {'电影': [{'标题': '玩具总动员'}, {'年份': '1995'}, {'简介': '讲述了玩具在主人不在时的冒险故事，是首部全电脑动画电影。'}]}]}]}
<class 'dict'>
```

举例4：与前例类似

```
from langchain_openai import ChatOpenAI
from langchain_core.prompts import PromptTemplate
from langchain_core.output_parsers import XMLOutputParser


model = ChatOpenAI(model="gpt-4o-mini")

actor_query = "生成周星驰的简化电影作品列表，按照最新的时间降序，必要时使用中文"
# 设置解析器 + 将指令注入提示模板。
parser = XMLOutputParser()
prompt = PromptTemplate(
    template="回答用户的查询。\n{format_instructions}\n{query}\n",
    input_variables=["query"],
    partial_variables={"format_instructions": parser.get_format_instructions()},
)

```

```
chain = prompt | model | parser
output = chain.invoke({"query": actor_query})
print(output)
```

```
{'周星驰电影作品列表': [{'电影': [{'标题': '西游伏妖篇'}, {'年份': '2017'}]}, {'电影': [{'标题': '美人鱼'}, {'年份': '2016'}]}, {'电影': [{'标题': '长江7号'}, {'年份': '2008'}]}, {'电影': [{'标题': '大话西游之仙履奇缘'}, {'年份': '1995'}]}, {'电影': [{'标题': '国产凌凌漆'}, {'年份': '1994'}]}, {'电影': [{'标题': '逃学威龙'}, {'年份': '1991'}]}]}
```

④ 列表解析器 CommaSeparatedListOutputParser

列表解析器：利用此解析器可以将模型的文本响应转换为一个用**逗号分隔的列表（List[str]）**。

举例1：

```
from langchain_core.output_parsers import CommaSeparatedListOutputParser

output_parser = CommaSeparatedListOutputParser()

# 返回一些指令或模板，这些指令告诉系统如何解析或格式化输出数据
format_instructions = output_parser.get_format_instructions()
print(format_instructions)

messages = "大象,猩猩,狮子"
result = output_parser.parse(messages)
print(result)
print(type(result))
```

```
Your response should be a list of comma separated values, eg: `foo, bar, baz` or `foo,bar,baz`
['大象', '猩猩', '狮子']
<class 'list'>
```

举例2：

```
from langchain_core.prompts import PromptTemplate
from langchain_openai import ChatOpenAI
from langchain.output_parsers import CommaSeparatedListOutputParser

# 初始化语言模型
chat_model = ChatOpenAI(model="gpt-4o-mini")

# 创建解析器
output_parser = CommaSeparatedListOutputParser()

# 创建LangChain提示模板
chat_prompt = PromptTemplate.from_template(
    "生成5个关于{text}的列表.\n\n{format_instructions}",
```

```
    partial_variables={
     "format_instructions": output_parser.get_format_instructions()
    })

# 提示模板与输出解析器传递输出
# chat_prompt =
chat_prompt.partial(format_instructions=output_parser.get_format_instructions())

# 将提示和模型合并以进行调用
chain = chat_prompt | chat_model | output_parser
res = chain.invoke({"text": "电影"})
print(res)
print(type(res))
```

```
['经典电影', '现代电影', '动作电影', '爱情电影', '科幻电影']
<class 'list'>
```

举例3：

```
from langchain.prompts.chat import HumanMessagePromptTemplate
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
from langchain.output_parsers import CommaSeparatedListOutputParser

# 初始化语言模型
chat_model = ChatOpenAI(model="gpt-4o-mini")

output_parser = CommaSeparatedListOutputParser()

chat_prompt = ChatPromptTemplate.from_messages([
    ("human", "{request}\n{format_instructions}")
    # HumanMessagePromptTemplate.from_template("{request}\n{format_instructions}"),
])

# model_request = chat_prompt.format_messages(
#     request="给我5个心情",
#     format_instructions=output_parser.get_format_instructions()
# )

#方式1:
# result = chat_model.invoke(model_request)
#
# resp = output_parser.parse(result.content)
# print(resp)
# print(type(resp))

# 方式2:
# result = chat_model.invoke(model_request)
# resp = output_parser.invoke(result)
# print(resp)
# print(type(resp))

# 方式3:
```

```
chain = chat_prompt | chat_model | output_parser
resp = chain.invoke({"request": "给我5个心情", "format_instructions": output_parser.get_format_instructions()})
print(resp)
print(type(resp))
```

```
['快乐', '忧伤', '愤怒', '兴奋', '宁静']
<class 'list'>
```

⑤ 日期解析器 DatetimeOutputParser (了解)

利用此解析器可以直接将LLM输出解析为日期时间格式。

- **get_format_instructions()**： 获取日期解析的格式化指令，指令为："Write a datetime string that matches the following pattern: '%Y-%m-%dT%H:%M:%S.%fZ'。
  - 举例：1206-08-16T17:39:06.176399Z

举例1：

```
from langchain.output_parsers import DatetimeOutputParser

output_parser = DatetimeOutputParser()

format_instructions = output_parser.get_format_instructions()
print(format_instructions)
```

```
Write a datetime string that matches the following pattern: '%Y-%m-%dT%H:%M:%S.%fZ'.

Examples: 1563-09-27T04:28:14.640366Z, 1786-06-24T23:46:01.984421Z, 1079-05-27T08:43:24.266403Z

Return ONLY this string, no other words!
```

举例2：

```
from langchain_openai import ChatOpenAI
from langchain.prompts.chat import HumanMessagePromptTemplate
from langchain_core.prompts import ChatPromptTemplate
from langchain.output_parsers import DatetimeOutputParser

chat_model = ChatOpenAI(model="gpt-4o-mini")


chat_prompt = ChatPromptTemplate.from_messages([
    ("system", "{format_instructions}"),
    ("human", "{request}")
])

output_parser = DatetimeOutputParser()
```

```python

# 方式1:
# model_request = chat_prompt.format_messages(
#     request="中华人民共和国是什么时候成立的",
#     format_instructions=output_parser.get_format_instructions()
# )

# response = chat_model.invoke(model_request)
# result = output_parser.invoke(response)
# print(result)
# print(type(result))

# 方式2:
chain = chat_prompt | chat_model | output_parser
resp = chain.invoke({"request": "中华人民共和国是什么时候成立的",
                     "format_instructions":output_parser.get_format_instructions()})
print(resp)
print(type(resp))
```

```
1949-10-01 00:00:00
<class 'datetime.datetime'>
```

6、LangChain调用本地模型

6.1 Ollama的介绍

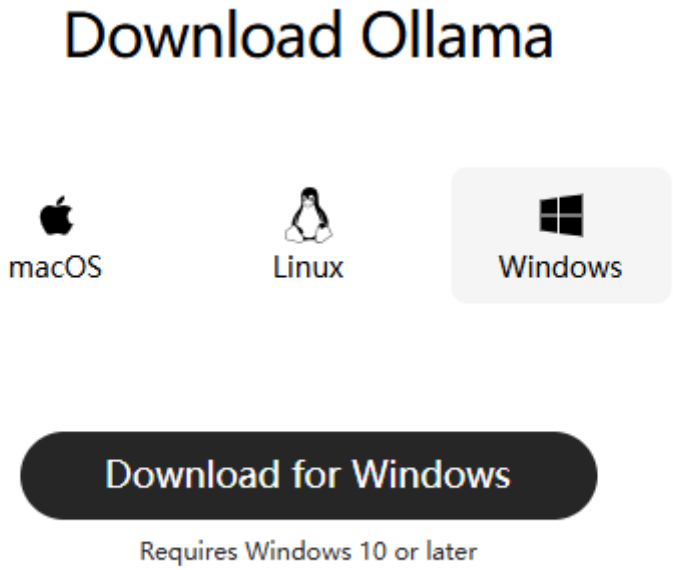
Ollama是在Github上的一个开源项目，其项目定位是：**一个本地运行大模型的集成框架。**目前主要针对主流的LlaMA架构的开源大模型设计，可以实现如 Qwen、Deepseek 等主流大模型的下载、启动和本地运行的自动化部署及推理流程。

目前作为一个非常热门的大模型托管平台，已被包括LangChain、Taskweaver等在内的多个热门项目高度集成。

Ollama官方地址：https://ollama.com

6.2 Ollama的下载-安装

Ollama项目支持跨平台部署，目前已兼容Mac、Linux和Windows操作系统。特别地对于Windows用户提供了非常直观的预览版。



无论使用哪个操作系统，Ollama项目的安装过程都设计得非常简单。

访问 https://ollama.com/download 下载对应系统的安装文件。

- Windows 系统执行 `.exe` 文件安装（大概671M大小）
- Linux 系统执行以下命令安装：

```
curl -fsSL https://ollama.com/install.sh | sh
```

  - 这行命令的目的是从 **https://ollama.com/** 网站读取 **install.sh** 脚本，并立即通过 **sh** 执行该脚本，在安装过程中会包含以下几个主要的操作：
    - 检查当前服务器的基础环境，如系统版本等；
    - 下载Ollama的二进制文件；
    - 配置系统服务，包括创建用户和用户组，添加Ollama的配置信息；
    - 启动Ollama服务；

6.3 模型的下载-安装

访问 https://ollama.com/search 可以查看 Ollama 支持的模型。使用命令行可以下载并运行模型，例如运行 **deepseek-r1:7b** 模型：

```
ollama run deepseek-r1:7b
```

6.4 调用本地私有模型

举例1：

```python
from langchain_community.chat_models import ChatOllama
#from langchain_ollama import ChatOllama

ollama_llm = ChatOllama(model="deepseek-r1:7b")
```

```python
from langchain_core.messages import HumanMessage

messages = [
    HumanMessage(content="你好，请介绍一下你自己")
]

chat_model_response = ollama_llm.invoke(messages)

print(chat_model_response.content)
```

```
您好！我是由中国的深度求索（DeepSeek）公司开发的智能助手DeepSeek-R1。如您有任何任何问题，我会尽我所能为您提供帮助。

您好！我是由中国的深度求索（DeepSeek）公司开发的智能助手DeepSeek-R1。如您有任何任何问题，我会尽我所能为您提供帮助。
```

若 Ollama 不在本地默认端口运行，需指定 `base_url`，即：

```python
ollama_llm = ChatOllama(
    model="deepseek-r1:7b",
    base_url="http://your-ip:port"  # 自定义地址
)
```

```python
print(chat_model_response.content)
```

```
您好！我是由中国的深度求索（DeepSeek）公司开发的智能助手DeepSeek-R1。如您有任何任何问题，我会尽我所能为您提供帮助。
```

举例2：

```python
from langchain.prompts.chat import ChatPromptTemplate
from langchain_community.chat_models import ChatOllama


# 生成对话形式的聊天信息格式
chat_prompt = ChatPromptTemplate.from_messages([
    ("system", "你是一个有用的助手，可以将{input_language}翻译成{output_language}。"),
    ("human", "{text}"),
])

# 格式化变量输入
```

```
messages = chat_prompt.format_messages(input_language="中文", output_language="英语", text="我爱编程")

# 实例化Ollama启动的模型
ollama_llm = ChatOllama(model="deepseek-r1:7b")

# 执行推理
result = ollama_llm.invoke(messages)

print(result.content)
```

好，用户让我把“我爱编程”翻译成英文。首先，“我”翻译成“I”，没问题。“爱”是“love”，常用在表达情感上。“编程”比较合适的是“programming”，这个词很常见，用来指代计算机编程。

所以组合起来就是“I love programming”。听起来挺自然的，符合英语的习惯用法。用户可能是在学习编程或者分享自己的兴趣，所以翻译得简洁明了就可以了。

I love programming.