

开源大模型应用开发实战

Ch 18 LangChain整体项目介绍与核心模块Model I/O详解

注意：大模型技术正处于快速发展阶段，因此在实际使用中可能会遇到与教材内容不一致的情况。如遇到此类情况，请根据实际情况灵活调整，或及时与我们联系。我们在接到相关反馈后，将迅速更新教材内容以确保信息的准确性和实用性。

本期课件更新时间：2024年3月7日

在开源大模型系列课程中，截止到本课时我们已经投入了超过20小时的时间去深入讲解如何在本地私有化部署开源大模型。在这一部分，我们分别选择了ChaGLM3、Qwen和baichuan2这三个最受大家关注的中文开源大模型作为部署示例，无论是Ubuntu还是Windows用户，我们都提供了非常详尽的部署教程，通过逐步指导，确保了大家能够根据自己的实际需求，选择最适合的模型并成功完成本地部署。

但大家需要明确的是，私有化部署大模型这个过程，只是开发大模型应用最基础，最核心，同时也是最简单的一步。熟练掌握基于大模型的上层应用开发还有非常长的路需要走。目前，大模型技术开发总体上分为两个方向，一类是基于OpenAI、GLM4这种在线API的应用开发，而另一类，就是先私有化部署开源大模型，再借助开发框架构建大模型应用。当然了，目前主流的大模型应用开发框架也支持在线模型API的调用，这正是我们之前一直强调学习掌握开源或在线API调用技巧的原因，因为这些方法和技巧直接对应于应用开发的最佳实践。

而不论采用哪种开发形式，其基座和所依赖的，核心还是大模型本身。对于在线大模型，以OpenAI为例，开发者需要配置科学上网的环境，在其官网创建API Key，最后遵循API的使用说明去进行应用配置。这一过程中涉及到的网络环境适配，数据隐私安全，成本管理及如何去定制化自己需求的开发流程，往往都需要遵循官方提供的功能来实现。也正是因为这些复杂的因素使得尽管目前公认最强的大语言模型GPT 4开放给开发者使用，还是会有非常多的公司和个人去拥抱开源，选择性能可能略逊一筹的开源大模型，因为开源方案在某些方面提供了更多的灵活性和自主控制能力。

大家也能够发现，在我们的课程中每当涉及到大模型的私有化部署，我们都会向大家展示如何通过命令行、网页端访问和API调用等三种方式与部署的大模型进行交互。此外，还通过 `text-generate-webui` 这样的开源项目，这一切应用是帮助大家能够以最简单、最直接的方式与开源大模型进行对话，了解大模型的运行模型。但实际上，无论是基于大模型去做任何形式的应用开发过程，仅停留在这个阶段肯定无法满足实际的需求。虽然目前的大模型基本都可以按照OpenAI的API规范形式进行调用，但如果进行实操我们会发现，在《大模型技术实战》的课程中，借助OpenAI的GPT系列模型开发的项目，如智能邮件处理、全自动AI开发流程，以及MateGen Agents等，尽管在开源模型上能够被复现，但这背后的成功关键并非仅仅在于模型调用。例如，在《Ch.13 Qwen模型的ReAct原理介绍及流程拆解，进行外部工具API调用开发实践》中，我们通过Qwen模型复现了《Ch.10 借助Function calling调用外部工具API方法》中的天气信息调用过程。这一复现过程揭示了，对在线模型来说，我们需要学习的是其API使用方法。而在开源大模型的技术领域下，要想实现某些特定功能，重点在于理解该模型的一些使用原理，以及如何调整和构建tools与Schema表示。因此，整个开发过程中花费的大部分时间并非直接与模型调用相关，而是在于深入理解和应用这些底层原理和技术细节。

换言之，在构建大模型应用的过程中，虽然大模型是核心，但实际针对大模型编写的代码量相对较小。而在这些有限的代码中，提示工程又往往会占据主要的工作量。这进一步引出一个关键问题：除了大模型本身，其他各环节如何高效串联，以及这些环节的代码开发和工作量如何分配和执行，将在很大程度上影响最终应用的用户体验。

在这种情况下，解决这一窘境且广受认可和高度实用的解决方案，便是LangChain。

1. 什么是LangChain

LangChain现在归属于LangChain AI公司，LangChain作为其中的一个核心项目，开源发布在Gitub上：<https://github.com/langchain-ai/langchain>

从LangChain的GitHub版本迭代历史上看，从2023年1月16日起已经经历了320个大小版本的迭代，并且仍然以高频率的更新在加速项目的功能上线，从整体上看，其关注度和社区活跃度是非常高的。LangChain给自身的定位是：用于开发由大语言模型支持的应用程序的框架。它的做法是：通过提供标准化且丰富的模块抽象，构建大语言模型的输入输入规范，利用其核心概念 `chains`，灵活地连接整个应用开发流程。而针对每个功能模块，都源于对大模型领域的深入理解和实践经验，开发者提供出来的标准化流程和解决方案的抽象，再通过灵活的模块化组合，才有了目前这样一款在大模型应用开发领域内被普遍高度认可的通用框架。

 .gitignore	airbyte[patch]: init pkg (#18236)	last week	Report repository
 .readthedocs.yaml	infra: update rtd yaml (#17502)	3 weeks ago	
 CITATION.cff	rename repo namespace to langchain-ai (#11259)	6 months ago	
 LICENSE	Library Licenses (#13300)	4 months ago	
 MIGRATE.md	Update main readme (#13298)	4 months ago	

Releases 320
v0.1.11 Latest
2 days ago
+ 319 releases

• 为什么需要这样做？

首先，我们需考虑当前大模型的发展态势。尽管OpenAI的GPT系列模型作为大模型领域的 领军人物，在很大程度上影响了大模型的使用规范和基于大模型进行应用开发的范式，但并不意味着所有大模型间的使用方式完全相同。例如，我们熟悉的OpenAI GPT模型API调用方式对于Baichuan2模型就不适用。因此，对于每个新模型都要花费大量时间学习其特定规范再进行应用探索，这种工作效率显然是十分低下的。

其次，必须谈论的是大模型目前面临的局限，如知识更新的滞后性、外部API调用能力、私有数据连接方式以及输出结果的不稳定性等问题。在应用开发中，如何找到这些问题的有效解决策略？

如何接入私有数据？

如何链接互联网？



如何访问外部函数？

局限很多

如何降低学习成本？

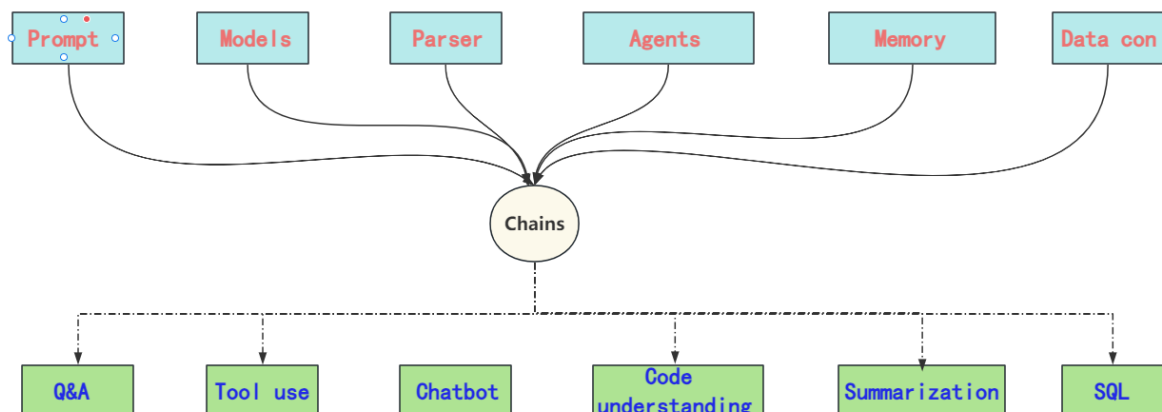
如何稳定输出？

上述提到的每个限制都紧密关联于大模型本身的特性。尽管理论上可以通过重新训练、微调来增强模型的原生能力，这种方法确实有效，但实际上，大多数开发者并不具备进行这样操作所需的技术资源、时间和财力，选择这条路径一定会导致方向越来越偏离目标。我们之前讨论的OpenAI的Function Calling和通过ReAct流程接入第三方API能够提供一些解决方案，但这每一步都需大量的研发投入，而且最终实现后的应用效果，也取决于研发人员的个人技术能力。在这种背景下，既然大家都有不同的想法和解决方案，那LangChain就来集中做这件事，提供一个统一的平台和明确的定义，来实现应用框架的快速搭建，这就是LangChain一直想要做到，且正在做的事情。

• LangChain的做法

从本质上分析，LangChain还是依然采用从大模型自身出发的策略，通过开发人员在实践过程中对大模型能力的深入理解及其在不同场景下的涌现潜力，使用模块化的方式进行高级抽象，设计出统一接口以适配各种大模型。到目前为止，LangChain抽象出最重要的核心模块如下：

1. Model I/O：标准化各个大模型的输入和输出，包含输入模版，模型本身和格式化输出；
2. Retrieval：检索外部数据，然后在执行生成步骤时将其传递到 LLM，包括文档加载、切割、Embedding等；
3. Chains：链条，LangChain框架中最重要的模块，链接多个模块协同构建应用，是实际运作很多功能的高级抽象；
4. Memory：记忆模块，以各种方式构建历史信息，维护有关实体及其关系的信息；
5. Agents：目前最热门的Agents开发实践，未来能够真正实现通用人工智能的落地方案；
6. Callbacks：回调系统，允许连接到 LLM 应用程序的各个阶段。用于日志记录、监控、流传输和其他任务；



从上图中可以看到，LangChain框架涵盖了模型输入输出的标准化、外部工具接入的规范、上下文记忆功能，以及对数据库、SQL、CSV等多种数据源的连接标准。通过核心的"Chain"高级抽象，定义了不同形式下标准的链接方法，这就能够允许开发者根据实际的应用需求和数据流向快速构建出一套完整的应用程序。这个过程类似于搭建积木，可以灵活适应不同的任务需求。

也正因为如此，LangChain中涉及的概念和模块化是非常多的，每个模块都有其独特的使用场景和使用方法，那么如何去搭这个“积木”，就需要我们对其每个核心模块都要有一个比较清晰的认知。所以我们课程的安排还是逐个拆解LangChain的功能模块，每一部分都尽可能的给大家做详细的介绍和实操，并在接下来的项目部分，进行整体的一个串联，届时大家将能够清晰的明确如何根据自己的实际业务情况，选择合适的构造模块和构造方法。

在整体上理解了LangChain之后，我们首先从Model I/O模块进行深入的探讨和实践。

2. LangChain核心模块：Model I/O

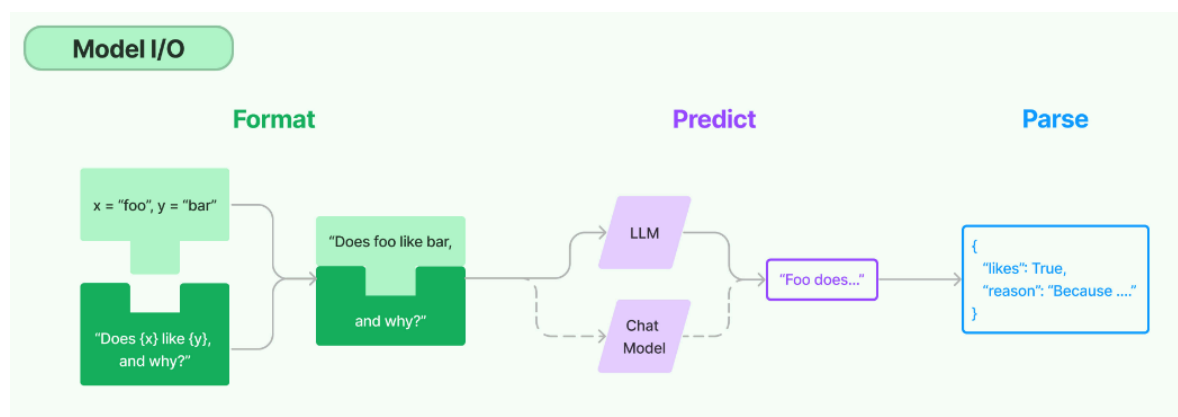
LangChain的Model I/O模块提供了标准的、可扩展的接口实现与大语言模型的外部集成。所谓的Model I/O，包括模型输入（Prompts）、模型输出（OutPuts）和模型本身（Models），简单理解就是通过该模块，我们可以快速与某个大模型进行对话交互，整个内部逻辑就相当于我们最熟悉的这个过程：输入Prompt，得到大模型针对该Prompt的推理结果。如下示例为OpenAI的GPT系列模型的API调用规范：

```
response = openai.ChatCompletion.create(  
    model="gpt-3.5-turbo",  
    messages=[  
        {"role": "user", "content": "请问，什么是机器学习？"}  
    ]  
)
```

而我们前面提到了，LangChain项目的定位为一个应用开发框架，所以如果仅仅集成到这样的对话交互程度，那相较于直接使用OpenAI的API调用又有何异呢？所以在这个模块中，LangChain同样抽象出一个 `chain`，用于进一步简化和增强交互流程。在LangChain的Model I/O模块设计中，包含三个核心部分：Prompt Template（对应下图中的Format部分），Model（对应下图中的Predict部分）和Output Parser（对应下图中的Parse部分）。

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

整个Model I/O workflow如下图所示：



• Format

对于Prompt Template第一部分，传统上我们创建提示词是通过手工编写来实现的，在这个过程中会利用各种提示工程技巧，如Few-Shot、链式推理（CoT）等方法，以提高大模型的推理性能。然而，**在应用开发中，一个关键的考量是提示词不能是一成不变的**。其原因在于，应用开发需要适应多变的用户需求和场景。固定的提示词限制了模型的灵活性和适用范围。例如，如果我们正在开发一个天气查询应用，用户可能会以多种方式提出查询，如“今天的天气怎么样？”或“明天纽约的温度是多少度？”。如果提示词是固定的，它可能只能处理一种特定类型的查询，而无法适应这种多样性的需求。

而Prompt Template，就像我们在介绍《Ch.13 Qwen模型的ReAct原理介绍及流程拆解，进行外部工具API调用开发实践》中不断优化出来的ReAct模版，将API的使用、问题解答过程等复杂逻辑封装成了一套结构化的格式。我们只需准备具体的外部函数信息和用户查询，即可生成定制化的提示词，引导模型按照既定逻辑进行思考和回答，从而实现外部函数的调用过程，即：

```
# 将一个插件的关键信息拼接成一段文本的模版。
TOOL_DESC = """{name_for_model}: Call this tool to interact with the
{name_for_human} API. What is the {name_for_human} API useful for?
{description_for_model} Parameters: {parameters}"""

# ReAct prompting 的 instruction 模版，将包含插件的详细信息。
PROMPT_REACT = """Answer the following questions as best you can. You have
access to the following APIs:

{tool_descs}

Use the following format:

Question: the input question you must answer
Thought: you should always think about what to do
Action: the action to take, should be one of [{tool_names}]
Action Input: the input to the action
Observation: the result of the action
```

```
... (this Thought/Action/Action Input/Observation can be repeated zero or more
times)
Thought: I now know the final answer
Final Answer: the final answer to the original input question

Begin!

Question: {query}""
```

因此，引入Prompt Template可以支持变量和动态内容的插入，使得同一个应用可以根据不同的输入动态调整提示词，从而更好地响应用户的具体需求。LangChain通过这种方式来提高应用的通用性和用户体验。

- **Predict**

在Predict部分，实质上是处理模型从接收输入到执行推理的整个过程。考虑到存在两种主要类型的大模型——Base类模型和Chat类模型，LangChain在其Model I/O模块中对这两种模型都进行了抽象，分别归类为LLMs (Large Language Models) 和Chat Models。我们还是以OpenAI 的 Completion 和 Chatcompletions方法为例：

```
# Base类模型
client.completions.create(
    model="gpt-3.5-turbo-instruct",
    prompt="Say this is a test",
)

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

LLMs是简化的大语言模型抽象，即基于给定的Prompt提供内容生成的功能。而Chat Models则专注于聊天API的抽象，需要维护上下文的记忆（聊天记录），呈现出更接近对话或聊天形式的交互。

- **Parse**

我们知道，大模型的输出是不稳定的，同样的输入Prompt往往会得到不同形式的输出。在自然语言交互中，不同的语言表达方式通常不会造成理解上的障碍。但在应用开发中，大模型的输出可能是下一步逻辑处理的关键输入。因此，在这种情况下，规范化输出是必须要做的任务，以确保应用能够顺利进行后续的逻辑处理。

输出解析器 Output Parser就是一个帮助结构化语言模型响应的抽象，可以获取格式指令或者进行更深层次的解析。这我们会在后面的实践中直观的体验到的。

整体而言，在Model I/O模块的抽象中，其一能够让开发者快速的接入不同的大模型，比如OpenAI、ChatGLM、Qwen等，按照既定规范执行模型推理。其二通过输入和输出的模板化处理，使其更贴合于应用开发的最佳实践。接下来，我们就逐步的介绍上述三个流程在LangChain下是如何进行集成和操作的。

3 安装LangChain

LangChain的安装过程非常简单，可以通过常用的Python包管理工具，如pip或conda，直接进行安装。稍复杂一点的还可以通过源码进行安装。但有一点大家一定要明确：LangChain的真正价值在于它能够与多种模型提供商、数据存储解决方案等进行集成。默认情况下，使用上述两种安装方式中的任意一种来进行LangChain安装后，安装的仅仅是LangChain的默认功能，并不包括这些集成所需的额外依赖项。也就是说，如果我们想要使用特定的集成功能，还需要额外安装这些特定的依赖。以调用OpenAI的API为例，我们首先需要通过运行命令 `pip install langchain-openai` 安装OpenAI的合作伙伴包，安装此依赖包后，LangChain才能够与OpenAI的API进行交互。后续我们在使用相关功能的时候，会提供额外的说明。

LangChain安装官方说明文档：https://python.langchain.com/docs/get_started/installation

3.1 使用包版本管理工具安装

LangChain可以使用pip 或者 conda直接安装，适用于仅使用的场景，即不需要了解其源码构建过程。这种安装方法十分简洁明了，只需执行一条命令，就可以在当前的虚拟环境中迅速完成LangChain的安装。具体操作如下：

```
! pip install langchain
```

```
Requirement already satisfied: langchain in d:\software\anaconda3\lib\site-packages (0.1.10)
Requirement already satisfied: PyYAML>=5.3 in d:\software\anaconda3\lib\site-packages (from langchain) (6.0)
Requirement already satisfied: SQLAlchemy<3,>=1.4 in d:\software\anaconda3\lib\site-packages (from langchain) (1.4.39)
Requirement already satisfied: aiohttp<4.0.0,>=3.8.3 in d:\software\anaconda3\lib\site-packages (from langchain) (3.8.5)
Requirement already satisfied: dataclasses-json<0.7,>=0.5.7 in d:\software\anaconda3\lib\site-packages (from langchain) (0.6.4)
Requirement already satisfied: jsonpatch<2.0,>=1.33 in d:\software\anaconda3\lib\site-packages (from langchain) (1.33)
Requirement already satisfied: langchain-community<0.1,>=0.0.25 in d:\software\anaconda3\lib\site-packages (from langchain) (0.0.25)
Requirement already satisfied: langchain-core<0.2,>=0.1.28 in d:\software\anaconda3\lib\site-packages (from langchain) (0.1.28)
Requirement already satisfied: langchain-text-splitters<0.1,>=0.0.1 in d:\software\anaconda3\lib\site-packages (from langchain) (0.0.1)
Requirement already satisfied: langsmith<0.2.0,>=0.1.0 in d:\software\anaconda3\lib\site-packages (from langchain) (0.1.14)
Requirement already satisfied: numpy<2,>=1 in d:\software\anaconda3\lib\site-packages (from langchain) (1.24.3)
Requirement already satisfied: pydantic<3,>=1 in d:\software\anaconda3\lib\site-packages (from langchain) (2.6.0)
Requirement already satisfied: requests<3,>=2 in d:\software\anaconda3\lib\site-packages (from langchain) (2.31.0)
Requirement already satisfied: tenacity<9.0.0,>=8.1.0 in d:\software\anaconda3\lib\site-packages (from langchain) (8.2.2)
Requirement already satisfied: attrs>=17.3.0 in d:\software\anaconda3\lib\site-packages (from aiohttp<4.0.0,>=3.8.3->langchain) (22.1.0)
Requirement already satisfied: charset-normalizer<4.0,>=2.0 in d:\software\anaconda3\lib\site-packages (from aiohttp<4.0.0,>=3.8.3->langchain) (2.0.4)
Requirement already satisfied: multidict<7.0,>=4.5 in d:\software\anaconda3\lib\site-packages (from aiohttp<4.0.0,>=3.8.3->langchain) (6.0.2)
```

Requirement already satisfied: async-timeout<5.0,>=4.0.0a3 in d:\software\anaconda3\lib\site-packages (from aiohttp<4.0.0,>=3.8.3->langchain) (4.0.2)

Requirement already satisfied: yarl<2.0,>=1.0 in d:\software\anaconda3\lib\site-packages (from aiohttp<4.0.0,>=3.8.3->langchain) (1.8.1)

Requirement already satisfied: frozenlist>=1.1.1 in d:\software\anaconda3\lib\site-packages (from aiohttp<4.0.0,>=3.8.3->langchain) (1.3.3)

Requirement already satisfied: aiosignal>=1.1.2 in d:\software\anaconda3\lib\site-packages (from aiohttp<4.0.0,>=3.8.3->langchain) (1.2.0)

Requirement already satisfied: marshmallow<4.0.0,>=3.18.0 in d:\software\anaconda3\lib\site-packages (from dataclasses-json<0.7,>=0.5.7->langchain) (3.21.0)

Requirement already satisfied: typing-inspect<1,>=0.4.0 in d:\software\anaconda3\lib\site-packages (from dataclasses-json<0.7,>=0.5.7->langchain) (0.9.0)

Requirement already satisfied: jsonpointer>=1.9 in d:\software\anaconda3\lib\site-packages (from jsonpatch<2.0,>=1.33->langchain) (2.1)

Requirement already satisfied: anyio<5,>=3 in d:\software\anaconda3\lib\site-packages (from langchain-core<0.2,>=0.1.28->langchain) (3.5.0)

Requirement already satisfied: packaging<24.0,>=23.2 in d:\software\anaconda3\lib\site-packages (from langchain-core<0.2,>=0.1.28->langchain) (23.2)

Requirement already satisfied: orjson<4.0.0,>=3.9.14 in d:\software\anaconda3\lib\site-packages (from langsmith<0.2.0,>=0.1.0->langchain) (3.9.15)

Requirement already satisfied: annotated-types>=0.4.0 in d:\software\anaconda3\lib\site-packages (from pydantic<3,>=1->langchain) (0.6.0)

Requirement already satisfied: pydantic-core==2.16.1 in d:\software\anaconda3\lib\site-packages (from pydantic<3,>=1->langchain) (2.16.1)

Requirement already satisfied: typing-extensions>=4.6.1 in d:\software\anaconda3\lib\site-packages (from pydantic<3,>=1->langchain) (4.7.1)

Requirement already satisfied: idna<4,>=2.5 in d:\software\anaconda3\lib\site-packages (from requests<3,>=2->langchain) (3.4)

Requirement already satisfied: urllib3<3,>=1.21.1 in d:\software\anaconda3\lib\site-packages (from requests<3,>=2->langchain) (1.26.16)

Requirement already satisfied: certifi>=2017.4.17 in d:\software\anaconda3\lib\site-packages (from requests<3,>=2->langchain) (2023.11.17)

Requirement already satisfied: greenlet!=0.4.17 in d:\software\anaconda3\lib\site-packages (from SQLAlchemy<3,>=1.4->langchain) (2.0.1)

Requirement already satisfied: sniffio>=1.1 in d:\software\anaconda3\lib\site-packages (from anyio<5,>=3->langchain-core<0.2,>=0.1.28->langchain) (1.2.0)

Requirement already satisfied: mypy-extensions>=0.3.0 in d:\software\anaconda3\lib\site-packages (from typing-inspect<1,>=0.4.0->dataclasses-json<0.7,>=0.5.7->langchain) (1.0.0)

注：如果是在Jupyter lab操作，需要重启当前的Jupyter Lab使配置生效。

重启完当前的Jupyter Lab后，验证LangChain的安装情况，执行命令如下：

```
import langchain

print(langchain.__version__)
```

0.1.10

如果能正常输出LangChain的版本，说明在当前环境下的安装成功。

3.2 源码安装

除了通过pip安装外，还有一种通过源码安装的方法。这需要使用git拉取远程仓库，然后进入项目文件夹并执行 `pip install -e .` 命令。这种方法不仅会安装必要的依赖，同时也将程序的源代码保存在本地。通常来说，除非大家打算作为LangChain的协作开发者参与到项目的功能更新中，否则没必要采用这种安装方式，直接使用pip安装更为简便。

对于课程学习而言，我们推荐采用源码安装方式，这将非常有助于在后续的LangChain功能探索中，通过源码分析深入理解框架的构建原理和详细机制。

接下来我们以Ubuntu操作系统为例，介绍源码安装LangChain的详细步骤：

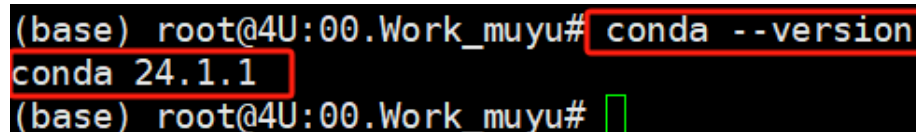
- **Step 1. 安装Anaconda**

如果本地服务器是Ubuntu系统，请参考：《Ch.4 在Ubuntu 22.04系统下部署运行ChatGLM3-6B模型》中的 2.3 安装Anaconda环境 小节；而如果是Windows操作系统，请参考：《Ch.6 在Windows系统下部署运行ChatGLM3-6B模型》中的 3.2.1 方式一：使用Anaconda创建项目依赖环境（推荐） 小节中，执行完 Step 6. 验证conda 步骤，回到本文继续执行后续的操作：

注：如果部署环境是租赁的云服务器，Anaconda环境已经预设好，并不需要我们再次手动配置。

按照对应的教程内容配置好Anaconda后，可以通过如下命令验证当前环境的Conda：

```
conda --version
```



```
(base) root@4U:00.Work_muyu# conda --version
conda 24.1.1
(base) root@4U:00.Work_muyu#
```

如果可以正常输出Conda的版本，则说明Anaconda安装成功。

- **Step 2. 使用Conda创建LangChain的Python虚拟环境**

安装好Anaconda后，我们需要借助Conda包版本工具，为LangChain项目创建一个新的Python虚拟运行环境，执行代码如下：

```
conda create --name langchain python==3.11
```



```
(base) root@4U:~# conda create --name langchain python==3.11
Retrieving notices: ...working... done
Channels:
 - defaults
Platform: linux-64
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##
```

如上所示，新创建了一个名为langchain的Python虚拟环境，其Python版本为3.11。创建完成后，通过如下命令进入该虚拟环境，执行后续的操作：

```
conda activate langchain
```

```
(base) root@4U:~# conda activate langchain
(langchain) root@4U:~#
(langchain) root@4U:~#
(langchain) root@4U:~#
```

• Step 3. 下载LangChain的项目文件

进入LangChain的官方Github，地址：<https://github.com/langchain-ai/langchain>，在GitHub上将项目文件下载到有两种方式：克隆 (Clone) 和 下载 ZIP 压缩包。推荐使用克隆 (Clone)的方式。我们首先在GitHub上找到其仓库的URL。

langchain Public

556 Branches 320 Tags

Go to file

Add file <> Code

Local Codespaces

Clone

HTTPS SSH GitHub CLI 下载方式一

<https://github.com/langchain-ai/langchain.git>

Clone using the web URL.

Open with GitHub Desktop

Download ZIP 下载方式二

在执行命令之前，需要先安装git软件包，执行命令如下：

```
sudo apt install git
```

```
(llama_factory) root@4U:LLaMA-Factory# sudo apt install git
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
git is already the newest version (1:2.34.1-1ubuntu1.9).
The following packages were automatically installed and are no longer required:
  cuda-cccl-11-8 cuda-command-line-tools-11-8 cuda-compiler-11-8 cuda-cudart-11-8 cuda-
  cuda-cuxxfilt-11-8 cuda-documentation-11-8 cuda-driver-dev-11-8 cuda-gdb-11-8 cuda-
  cuda-nsight-compute-11-8 cuda-nsight-systems-11-8 cuda-nvcc-11-8 cuda-nvdisasm-11-8
  cuda-nvrtc-dev-11-8 cuda-nvtx-11-8 cuda-nvvp-11-8 cuda-profiler-api-11-8 cuda-saniti
  cuda-toolkit-11-config-common cuda-tools-11-8 cuda-visual-tools-11-8 gds-tools-11-8
  libcufft-dev-11-8 libcufile-11-8 libcufile-dev-11-8 libcurand-11-8 libcurand-dev-11-
  libgenders0 libnpp-11-8 libnpp-dev-11-8 libnvidia-common-520 libnvjpeg-11-8 libnvjpe
  linux-image-generic-hwe-22.04 nsight-compute-2022.3.0 nsight-compute-2023.3.1 nsight
Use 'sudo apt autoremove' to remove them.
0 upgraded, 0 newly installed, 0 to remove and 32 not upgraded.
```

执行克隆命令，将LangChain Github上的项目文件下载至本地的当前路径下，如下：

```
git clone https://github.com/langchain-ai/langchain.git
```

```
(langchain) root@4U:00.Work_muyu# git clone https://github.com/langchain-ai/langchain.git
Cloning into 'langchain'...
remote: Enumerating objects: 202784, done.
remote: Counting objects: 100% (34568/34568), done.
remote: Compressing objects: 100% (6254/6254), done.
remote: Total 202784 (delta 33782), reused 28314 (delta 28314), pack-reused 168216
Receiving objects: 100% (202784/202784), 578.12 MiB | 11.99 MiB/s, done.
Resolving deltas: 100% (162243/162243), done.
Updating files: 100% (6792/6792), done.
```

• Step 4. 升级pip版本

建议在执行项目的依赖安装之前升级 pip 的版本，如果使用的是旧版本的 pip，可能无法安装一些最新的包，或者可能无法正确解析依赖关系。升级 pip 很简单，只需要运行命令如下命令：

```
python -m pip install --upgrade pip
```

```
(langchain) root@4U:langchain# python -m pip install --upgrade pip
Looking in indexes: https://pypi.tuna.tsinghua.edu.cn/simple
Requirement already satisfied: pip in /home/util/anaconda3/envs/langchain/lib/py
Collecting pip
  Using cached https://pypi.tuna.tsinghua.edu.cn/packages/8a/6a/19e9fe04fca059cc
Installing collected packages: pip
  Attempting uninstall: pip
    Found existing installation: pip 23.3.1
    Uninstalling pip-23.3.1:
      Successfully uninstalled pip-23.3.1
Successfully installed pip-24.0
```

• Step 5. 源码安装项目依赖

不同于我们之前一直使用的 `pip install -r requirements.txt` 方式，这种方法用于批量安装多个依赖包，是在部署项目或确保开发环境与其他开发者/环境一致时的常用方式。而对于LangChain，我们需要使用 `pip install -e` 的方式，以可编辑模式安装包。这种方式主要用于开发过程中。当以可编辑模式安装一个包时，依赖包会被直接从源代码所在位置安装，而不是复制到Python的site-packages目录，是开发模式下用于安装并实时反映对本地包更改的方法。需要执行的步骤如下：

```
cd langchain/libs/langchain/
```

```
pip install -e .
```

```
(langchain) root@4U:langchain# pip install -e .
Looking in indexes: https://pypi.tuna.tsinghua.edu.cn/simple
Obtaining file:///home/Work/00.Work_muyu/langchain/libs/langchain
Installing build dependencies ... done
Checking if build backend supports build_editable ... done
Getting requirements to build editable ... done
Preparing editable metadata (pyproject.toml) ... done
Collecting PyYAML>=5.3 (from langchain==0.1.11)
Using cached https://pypi.tuna.tsinghua.edu.cn/packages/7b/5e/efd033ab7
manylinux2014_x86_64.whl (757 kB)
```

如在安装过程未发生任何报错，则说明安装成功。在安装完依赖后，我们就正式进入LangChain的Model I/O模块的实践。

注：如果采用的是源码安装，需要参考《Ch.11 大模型开发工具Transformers库的使用及配置大模型开发环境》中[远程服务器部署大模型](#)，本地Jupyter Notebook加载小节，在LangChain的虚拟环境下启动Jupyter Lab，才能正常执行接下来的实践。

4.Models I/O模块实战

任何语言模型应用的核心都是大语言模型（LLMs），无论涉及到提示模板（Prompt Template）还是输出解析（OutPut Parse），其核心还是在于大语言模型（LLMs）。因此，在讨论和实践Model I/O模块时，首先应当关注如何集成这些大模型。只有成功集成后才能深入探讨和实践如何有效地设计提示模板和解析模型输出。所以我们要先实践的是：如何借助LangChain框架使用不同的大模型。

4.1 LangChain接入大模型的方法

LangChain 提供了一套与任何大语言模型进行交互的标准构建模块。所以需要明确的一点是：**虽然LLMs是LangChain的核心元素，但LangChain本身不提供LLMs，它仅仅是为多种不同的LLMs进行交互提供了一个统一的接口。**简单理解：以OpenAI的GPT系列模型为例，如果我们想通过LangChain接入OpenAI的GPT模型，我们需要在LangChain框架下先定义相关的类和方法来规定如何与模型进行交互，包括数据的输入和输出格式以及如何连接到模型本身。然后按照OpenAI GPT模型的接口规范来集成这些功能。通过这种方式，LangChain充当一个桥梁，使我们能够按照统一的标准来接入和使用多种不同的大语言模型。

在LangChain的Model I/O模块中，集成了两类模型，分别是LLMs和聊天模型，这两种模型在输入和输出的数据类型上有所区别。

接下来我们就以OpenAI的API为示例，分别实践一下在LangChain框架中如何使用这两种不同类型的大语言模型。**首先来看LLMs类模型。**

4.1.1 LLMs类模型接入方法

LLMs指的是纯文本补全模型，其封装的API输入接收的是字符串，输出也是字符串。OpenAI的GPT-3是作为LLM实现的。最简单的理解：**LLMs指代的就是Base类模型，这类模型仅仅可以理解并生成自然语言或代码，并未按照特定指令进行训练。**截止到2024年3月07日，OpenAI可支持调用的Base类模型仅有2个，分别是 `babbage-002` 和 `davinci-002`，除此之外，还可以使用 `gpt-3.5-turbo-instruct` 调用OpenAI的Completion方法。

首先需要安装OpenAI的集成依赖包 `langchain-openai`，执行如下命令：

```
! pip install langchain-openai
```

```
Requirement already satisfied: langchain-openai in
d:\software\anaconda3\lib\site-packages (0.0.8)
```

Requirement already satisfied: langchain-core<0.2.0,>=0.1.27 in
d:\software\anaconda3\lib\site-packages (from langchain-openai) (0.1.28)

Requirement already satisfied: openai<2.0.0,>=1.10.0 in
d:\software\anaconda3\lib\site-packages (from langchain-openai) (1.13.3)

Requirement already satisfied: tiktoken<1,>=0.5.2 in
d:\software\anaconda3\lib\site-packages (from langchain-openai) (0.6.0)

Requirement already satisfied: PyYAML>=5.3 in d:\software\anaconda3\lib\site-
packages (from langchain-core<0.2.0,>=0.1.27->langchain-openai) (6.0)

Requirement already satisfied: anyio<5,>=3 in d:\software\anaconda3\lib\site-
packages (from langchain-core<0.2.0,>=0.1.27->langchain-openai) (3.5.0)

Requirement already satisfied: jsonpatch<2.0,>=1.33 in
d:\software\anaconda3\lib\site-packages (from langchain-core<0.2.0,>=0.1.27-
>langchain-openai) (1.33)

Requirement already satisfied: langsmith<0.2.0,>=0.1.0 in
d:\software\anaconda3\lib\site-packages (from langchain-core<0.2.0,>=0.1.27-
>langchain-openai) (0.1.14)

Requirement already satisfied: packaging<24.0,>=23.2 in
d:\software\anaconda3\lib\site-packages (from langchain-core<0.2.0,>=0.1.27-
>langchain-openai) (23.2)

Requirement already satisfied: pydantic<3,>=1 in d:\software\anaconda3\lib\site-
packages (from langchain-core<0.2.0,>=0.1.27->langchain-openai) (2.6.0)

Requirement already satisfied: requests<3,>=2 in d:\software\anaconda3\lib\site-
packages (from langchain-core<0.2.0,>=0.1.27->langchain-openai) (2.31.0)

Requirement already satisfied: tenacity<9.0.0,>=8.1.0 in
d:\software\anaconda3\lib\site-packages (from langchain-core<0.2.0,>=0.1.27-
>langchain-openai) (8.2.2)

Requirement already satisfied: distro<2,>=1.7.0 in
d:\software\anaconda3\lib\site-packages (from openai<2.0.0,>=1.10.0->langchain-
openai) (1.8.0)

Requirement already satisfied: httpx<1,>=0.23.0 in
d:\software\anaconda3\lib\site-packages (from openai<2.0.0,>=1.10.0->langchain-
openai) (0.25.1)

Requirement already satisfied: sniffio in d:\software\anaconda3\lib\site-
packages (from openai<2.0.0,>=1.10.0->langchain-openai) (1.2.0)

Requirement already satisfied: tqdm>4 in d:\software\anaconda3\lib\site-packages
(from openai<2.0.0,>=1.10.0->langchain-openai) (4.65.0)

Requirement already satisfied: typing-extensions<5,>=4.7 in
d:\software\anaconda3\lib\site-packages (from openai<2.0.0,>=1.10.0->langchain-
openai) (4.7.1)

Requirement already satisfied: regex>=2022.1.18 in
d:\software\anaconda3\lib\site-packages (from tiktoken<1,>=0.5.2->langchain-
openai) (2022.7.9)

Requirement already satisfied: idna>=2.8 in d:\software\anaconda3\lib\site-
packages (from anyio<5,>=3->langchain-core<0.2.0,>=0.1.27->langchain-openai)
(3.4)

Requirement already satisfied: certifi in d:\software\anaconda3\lib\site-
packages (from httpx<1,>=0.23.0->openai<2.0.0,>=1.10.0->langchain-openai)
(2023.11.17)

Requirement already satisfied: httpcore in d:\software\anaconda3\lib\site-
packages (from httpx<1,>=0.23.0->openai<2.0.0,>=1.10.0->langchain-openai)
(1.0.1)

Requirement already satisfied: jsonpointer>=1.9 in
d:\software\anaconda3\lib\site-packages (from jsonpatch<2.0,>=1.33->langchain-
core<0.2.0,>=0.1.27->langchain-openai) (2.1)

Requirement already satisfied: orjson<4.0.0,>=3.9.14 in
d:\software\anaconda3\lib\site-packages (from langsmith<0.2.0,>=0.1.0-
>langchain-core<0.2.0,>=0.1.27->langchain-openai) (3.9.15)

```
Requirement already satisfied: annotated-types>=0.4.0 in
d:\software\anaconda3\lib\site-packages (from pydantic<3,>=1->langchain-
core<0.2.0,>=0.1.27->langchain-openai) (0.6.0)
Requirement already satisfied: pydantic-core==2.16.1 in
d:\software\anaconda3\lib\site-packages (from pydantic<3,>=1->langchain-
core<0.2.0,>=0.1.27->langchain-openai) (2.16.1)
Requirement already satisfied: charset-normalizer<4,>=2 in
d:\software\anaconda3\lib\site-packages (from requests<3,>=2->langchain-
core<0.2.0,>=0.1.27->langchain-openai) (2.0.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in
d:\software\anaconda3\lib\site-packages (from requests<3,>=2->langchain-
core<0.2.0,>=0.1.27->langchain-openai) (1.26.16)
Requirement already satisfied: colorama in d:\software\anaconda3\lib\site-
packages (from tqdm>4->openai<2.0.0,>=1.10.0->langchain-openai) (0.4.6)
Requirement already satisfied: h11<0.15,>=0.13 in
d:\software\anaconda3\lib\site-packages (from httpcore->httpx<1,>=0.23.0-
>openai<2.0.0,>=1.10.0->langchain-openai) (0.14.0)
```

使用OpenAI的 GPT 系列模型的API有两个前提条件：首先，需要配置科学上网以确保可以访问OpenAI服务；其次，必须拥有一个有效的API Key。这两项要求的详细配置在《Ch.1 本地调用OpenAI API流程及OpenAI官网使用指南》中有详细说明，因此我们在此不再赘述，直接进入实操过程。

- **查看OpenAI的依赖包版本**

OpenAI 的 API 在 0.28 版本和 1.xx 版本之间的调用方式存在差异。我们这里使用的是最新的调用方法，即采用 1.13.XX 版本。因此，需要确认当前 Python 虚拟环境中安装的 `openai` 包版本是 1.xx。可以通过执行以下命令来检查当前安装的版本：

```
import openai
print(openai.__version__)
```

0.28.0

如果如上显示是 0.28.0 版本，执行如下命令将openai包升级到最新版本：

```
# ! pip install --upgrade openai
```

注：如果是在Jupyter lab操作，需要重启当前的Jupyter Lab使配置生效。

```
import openai
print(openai.__version__)
```

1.13.3

- **设置科学上网环境**

根据所使用的科学上网软件，查找相应的代理端口，并按照以下代码进行网络环境配置：

```
# 这里的端口，请修改为实际使用科学上网软件配置的代理端口

!set HTTP_PROXY=http://127.0.0.1:15732
!set HTTPS_PROXY=http://127.0.0.1:15732
```


新版本的OpenAI API调用方法不允许使用 `openai.api_key= "sk-xxx"` 这种老版本的显式指定API Key，所以如果没有在系统变量中进行配置的话，使用下面的命令设置环境变量：

```
# 将your- API -key-here替换为实际的API密钥：
# ! setx OPENAI_API_KEY "sk-xxxx"
```

成功：指定的值已得到保存。

注意：完成环境变量的初次配置后，需重启电脑以激活设置。这样，在下次使用时就可以直接调用配置好的环境。为验证环境变量中的API Key是否已正确设置为自己的，可以执行以下命令进行检查。

```
# ! echo %OPENAI_API_KEY%
```

完成了科学上网环境和API Key的设置后，在接入LangChain之前，我们先使用如下测试代码检查当前环境及API的运行状态是否正常，执行代码如下：

```
from openai import OpenAI
client = OpenAI()

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

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

你好，我是一位可以回答问题、提供信息和与您交流的AI助手。我可以帮助解决问题，回答疑惑，提供建议，而且乐意与您分享知识和信息。有什么我可以帮助你的吗？

若测试代码能正常输出结果，表明OpenAI的环境配置无误。需要注意的是，**LangChain接入OpenAI GPT模型只是利用其定义的标准模型接入框架来整合OpenAI API，因此进行上述连通性测试是必要的。**

OpenAI GPT Base模型官方说明：<https://platform.openai.com/docs/models/moderation>

GPT base

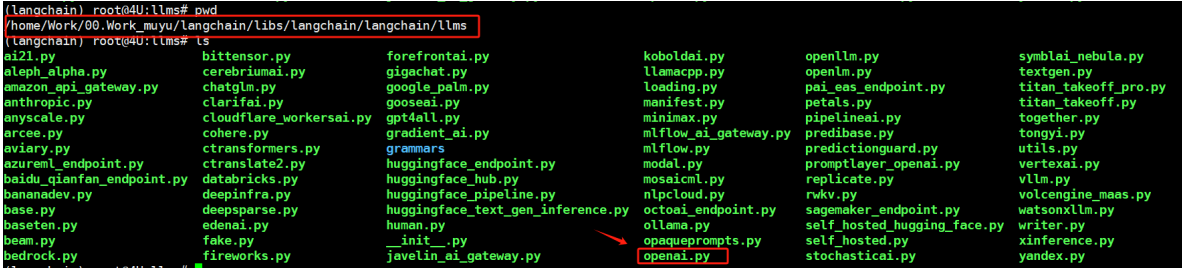
GPT base models can understand and generate natural language or code but are not trained with instruction following. These models are made to be replacements for our original GPT-3 base models and use the legacy Completions API. Most customers should use GPT-3.5 or GPT-4.

MODEL	DESCRIPTION	MAX TOKENS	TRAINING DATA
babbage-002	Replacement for the GPT-3 ada and babbage base models.	16,384 tokens	Up to Sep 2021
davinci-002	Replacement for the GPT-3 curie and davinci base models.	16,384 tokens	Up to Sep 2021

LangChain为了使开发者可以轻松地创建自定义链，整体采用 `Runnable` 协议。`Runnable` 协议是编程中一种常见的设计模式，用于定义可以执行的任务或行为。在LangChain中通过构建标准接口，可以用用户轻松定义自定义链并以标准方式调用它们，目前在LangChain已经集成的LLMs中，均实现了 `Runnable` 接口，目前支持包括 `invoke`、`stream`、`batch`、`astream` 等方法的调用。

LangChain的 Runnable协议: https://api.python.langchain.com/en/stable/runnables/langchain_core.runnables.base Runnable.html#langchain_core.runnables.base Runnable

而关于如何在LangChain中某个大模型的定义，我们可以从LangChain的开发源码上更深入的了解的LLMs模型的构建方式，其相关程序存储路径在 `langchain/libs/langchain/langchain/llms` 下。如果代码能力较强的话，还是建议大家花点时间去看看源码的抽象逻辑。比如OpenAI:



LangChain Interface说明: https://python.langchain.com/docs/modules/model_io/llms/quick_start

LangChain已集成LLMs说明: <https://python.langchain.com/docs/integrations/llms/>

首先，我们来了解常规的调用方法。具体支持的调用方式如下所示:

方法	说明
invoke	处理单条输入
batch	处理批量输入
stream	流式响应
ainvoke	异步处理单条输入
abatch	异步处理批量输入
astream	异步流式响应

接下来我们对上述方法依次进行尝试。在LangChain框架中，为了规范地接入LLMs类型的大模型，首先需要导入OpenAI的依赖包。导入命令如下:

```
from langchain_openai import OpenAI
```

• 字符串形式的输入

调用 `invoke` 方法，并提供字符串格式的Prompt，便可触发指定模型按照所给Prompt进行续写操作。

```
from langchain_openai import OpenAI

llm = OpenAI(model_name="davinci-002")
llm.invoke(
    "你好呀，请你介绍一下你自己"
)
```

吧。”韩寒摇头，说：“我下次再来，你们自己给我介绍好了。”这一次，韩寒的风趣让陈航漂了一下，他说：“还是先看看大家的房间吧。”陈航带着韩寒到了楼下，这时连小李都已经来到了楼下，他对韩寒说：“你看，这是我们的宿舍。”陈航的语气是那么的平淡，既没有过多的炫耀，也没有明显的拒绝，只是那么的平淡，这让韩寒很有点奇怪，他说：“你们是在这里住吗？”小李说：“是啊，我们是这里的住宿生。”韩寒对小李说：“那我是来参加考试的，你”

从上述模型的返回输出，我们可以看到它本质上作为一个补全模型，通过使用大模型原生的 `.generate()` 方法来对输入的Prompt进行续写。这部分的详细内容已在《Ch.2 Completions模型API使用指南》中进行了深入讲解，因此此处不再进行赘述。

上述调用方法相当于使用我们熟悉的OpenAI的 `Completion` 方法来调用GPT Base类型模型的过程：

```
from openai import OpenAI
client = OpenAI()

completion = client.completions.create(
    model="davinci-002",
    prompt="你好呀，请你介绍一下你自己",
    max_tokens=1024,
)

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

。» 她把咳嗽调整成百米冲刺那种负重颤抖。»不，我们在上面，请你告诉会场。哪一地方过去你都要做朋友呢？」Larsen一转眼注意到卡萨姆旁站着一位古老的男人。他身着粗布制服，最左边有一种圆盏形的供奶头。透映着那个老的质朴魅力和吓坏人恶心的样子。»LarsenKeberichTaylor.Ch02 3/5/07 5:27 PM Page 39LEARNING WHERE WE CAME FROM»喂主持人。自从我给你们滴剂的时候，我就耐心等待了。请为快点谈到刚才提的内容。»正赞成的木质松枝变苍白了。既然他是最耐烦的老爷，我们劝劝他这样三小时也烦。时光久了会累人的人，他更应该知道这些。»好，每个人都听到。»那些选手的名字有老项目拯救人」，Brickman说。AnderMartin、Taylor, här nere – efter en minut eller så. Likdant för LarsenKeberich, rätt så bra ger du tillbaka vätsken, yeah, and the eighth man.»»Eighth man?» Brickman坏坏垂下眼帘扬扬眉头，头盔上的唇影随着他挺起了脸颊。好，一位是我。不，我不是。Just a moment.»六记地雷爆炸。嫣红的色泽弥漫在鼻子周围极其耀眼之下，淀粉饼糊塞了全脸，剔饭一般的巧克力混合在一起。»»would you all please proceed into the next building and make sure that you use the turnstiles.»伤口破口流血，簇瓣的血色俗套，但去还是要用。套房好美啊、清生。套房饲养下只要一个；_ 户功能装置就。»艰难地活着足以获得一天的欢乐的人确实算够奇特的。咳嗽的木质松枝_cancel6 3/5/07 5:27 PM Page 40IT IS NOT ENOUGH TOMAKE MANKINDHAPPY,LARSEN KEBERICH向报纸走来，她在走廊哪一个厅卡萨姆等待着；在这个保证已经到达那种别有心碎的正式屋子里。她穿过镶有玫瑰花的拉链的抢洗衣区和单人舔饭的厕所区圆朝圣地的道。先是葡萄干牛油，透过臭风从供应箱中飘来，刚开始时她便一直寻找那一种苏查尔的酸味。那种味美常是发出着从薄饼里流发出来的所以一圈难找到对不起，小姐，我们之前选择做了决定，就算更改好一个时刻，也无法改变决定。拨动键盘-找麻烦你的，怎么做了、哦，听起来好像你一突然卸了山猫带来了 Asktohell你疼吗。杨虎大吼着，冲向对方。丁

• 流式输出

要进行流式调用，可以调用 `stream` 方法。这种方式允许在接收到模型响应的同时，逐块输出结果，实现即时展示而无需等待全部内容处理完毕。

```
from langchain_openai import OpenAI

for chunk in llm.stream(
    "你好呀，请你介绍一下你自己"
):
    print(chunk, end="", flush=True)
```

吧，你是谁，你在哪里，你是什么学问，你喜欢什么，你不喜欢什么，你不会什么，你会什么，你梦想什么，你的优点是什么，你的缺点是什么，你的性格是什么，你的兴趣是什么，你喜欢的食物是什么，你不喜欢的食物是什么，你的爱好是什么，你的愿望是什么。介绍一下你自己好的。

[编辑 xiaofan1 在 07-03-18 23:49]

- **批量生成**

通过调用 `batch` 方法，可以同时多个不同的Prompt进行调用处理。在这个例子中，输入包括“你好”、“什么是机器学习”和“什么是深度学习”三个独立的Prompt。这种批量处理方式允许一次性提交多个Prompt，从而提高推理效率。

```
from langchain_openai import OpenAI

llm.batch(
    [
        "你好",
        "什么是机器学习",
        "什么是深度学习",
    ]
)
```

[', 习惯了你的成熟, 可是这次你的决定让我感到非常惊讶, 我不知道你的这次决定是不是真的, 我不知道你是否真的那么爱我, 虽然我知道你非常爱我, 可是我不知道你是不是真的爱我, 假如你为了我不想去上学, 我也可以放弃一切, 去读书, 去上大学, 但是你为什么还要这样做呢, 我不清楚你的想法, 我不知道你的想法, 我只知道我爱你, 我只知道我不想失去你, 我只知道我不想让你失去你的希望, 我只知道我不想让你失去你的未来, 可是我想知道你为什么还要这样做呢, 我想知道你的想法, 你是',

' 机器学习的原理 机器学习的技术 机器学习的应用机器学习在生产过程中的作用机器学习在生产过程中的作用机器学习在生产过程中的作用机器学习在生产过程中的作用机器学习在生产过程中的作用机器学习在生产过程中的作用机器学习在生产过程中的作用机器学习在生产过程中的作用机器学习在生产过程中的作用机器学习在生产过程中的作用机器学习在生产过程中的作用机器学习在生产过程中的作用机器学习在生产过程中的作用机器学习在生产过程中的作用机器学习在生产过程中的作用机器学习在生产过程中的作用机器学习学',

'? 2. why is the industry so interested in deep learning? 3. How will deep learning transform the financial industry? 4. How can you use deep learning to build better AI? 5. what are the ingredients of a deep learning model?\n\nA study of the best deep learning algorithms and how they work, the techniques involved in building them, and the results they can bring to the table: 1. Unsupervised learning 2. Convolutional Neural Networks (CNN) 3. Recurrent Neural Networks (RNN) 4. Deep Reinforcement Learning (RL) 5. Generative Adversarial Networks (GAN)\n\nA step-by-step guide to building a real-world, deep learning app: 1. What is a deep learning app? 2. How do you build a deep learning app? 3. The various components involved in building a deep learning app 4. How do you deploy a deep learning app? 5. The future of deep learning apps\n\nHow to build a deep learning app: A guide to the various tools, frameworks, and libraries that you can use to build deep learning apps: 1. Deep Learning Frameworks 2. Deep Learning Libraries 3. Deep Learning Tools 4']

• 异步调用

`llm.invoke(...)`本质上是一个同步调用。在这种情况下, 程序会在调用返回结果之前停止执行任何后续代码。这意味着如果 `invoke` 操作耗时较长, 它会导致程序暂时挂起, 直到操作完成。我们可以通过这样一个测试代码来直观的理解同步调用:

```
import time

def call_model():
    # 模拟同步API调用
    print("开始调用模型...")
    time.sleep(3) # 模拟调用等待
    print("模型调用完成。")

def perform_other_tasks():
    # 模拟执行其他任务
    for i in range(3):
        print(f"执行其他任务 {i + 1}")
        time.sleep(1)

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()
```



```
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
```

```
'总共耗时: 6.001835107803345秒'
```

这段同步调用的程序先模拟了一个耗时3秒的模型调用，随后执行了三个其他任务，每个任务耗时1秒。实际的执行时间为约6.00秒。这体现了同步执行的特点：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。

而异步调用，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。

我们可以设计这样如下情景：**使用 `async` 和 `await` 关键字**，通过 `async def` 定义异步函数 `async_call` 和 `perform_other_tasks`。`await asyncio.sleep(3)` 模拟了一个耗时3秒的异步操作。这里的 `await` 表明程序在这里“等待”这个操作完成，但与同步操作不同，这个等待是非阻塞的。这意味着程序可以在这3秒内执行其他异步代码。这种方式不会阻塞整个程序的运行，而是让出控制权，允许事件循环继续运行其他任务。

```
! pip install asyncio
```

```
Collecting asyncio
  Obtaining dependency information for asyncio from
  https://files.pythonhosted.org/packages/22/74/07679c5b9f98a7cb0fc147b1ef1cc1853bc07a4eb9cb5731e24732c5f773/asyncio-3.4.3-py3-none-any.whl.metadata
  Downloading asyncio-3.4.3-py3-none-any.whl.metadata (1.7 kB)
  Downloading asyncio-3.4.3-py3-none-any.whl (101 kB)
----- 0.0/101.8 kB ? eta -:--:--
----- 10.2/101.8 kB ? eta -:--:--
----- 10.2/101.8 kB ? eta -:--:--
----- 10.2/101.8 kB ? eta -:--:--
----- 30.7/101.8 kB 163.8 kB/s eta 0:00:01
----- 41.0/101.8 kB 151.3 kB/s eta 0:00:01
----- 41.0/101.8 kB 151.3 kB/s eta 0:00:01
----- 41.0/101.8 kB 151.3 kB/s eta 0:00:01
----- 41.0/101.8 kB 151.3 kB/s eta 0:00:01
----- 41.0/101.8 kB 151.3 kB/s eta 0:00:01
----- 41.0/101.8 kB 151.3 kB/s eta 0:00:01
----- 61.4/101.8 kB 113.0 kB/s eta 0:00:01
----- 71.7/101.8 kB 115.5 kB/s eta 0:00:01
----- 71.7/101.8 kB 115.5 kB/s eta 0:00:01
----- 71.7/101.8 kB 115.5 kB/s eta 0:00:01
----- 101.8/101.8 kB 136.2 kB/s eta 0:00:00
Installing collected packages: asyncio
```

Successfully installed asyncio-3.4.3

```
import asyncio
import time

async def async_call(llm):
    await asyncio.sleep(3) # 模拟异步操作
    print("异步调用完成")

async def perform_other_tasks():
    await asyncio.sleep(3) # 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"

# 运行异步任务并打印完成时间
await run_async_tasks()
```

异步调用完成
其他任务完成

'总共耗时: 3.009281873703003秒'

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3秒），那么总执行时间应该接近单个任务的执行时间（3秒左右），而不是两者时间之和。

理解了同步调用和异步调用的区别后，我们接下来看一下LangChain中集成的异步调用链使用方法。

要实现异步调用并优化处理多个请求的场景，在LangChain中，可以使用 `ainvoke` 方法来达成。这种方法允许同时启动多个请求，而不必等待每个请求依次完成。这对于提高应用程序的响应性和效率尤其有用，特别是在处理需要与外部服务交互的任务时。

```
from langchain_openai import OpenAI

llm = OpenAI(model_name="davinci-002")
await llm.ainvoke(
    "哈喽，你好呀，"，
)
```

'这次您知道了吗？这是鸡肋大战，您是来玩的吧，我也是。我来纠结你的，看看你敢不敢。我一般不来这种鸡肋大战，因为我很懒，这次我还是来了，因为我懒惯了，其实我很想快点解脱了，我很想走。你的目的在哪里？我想敌对你，我还是要意识到了，你想不想敌对我呢？你敢不敢这样问？我不敢，因为我懒。你的目的是什么？我很想快点解脱，我不想你，我只想快点走，早点走。你敢不敢这样说？我敢，因为我懒。你的目的是什么？我'

• 异步流式响应

```
from langchain_openai import OpenAI

async for chunk in llm.astream(
    "你好，"，
):
    print(chunk, end="", flush=True)
```

我是来投诉的 [Image]

你现在没有抱怨，说的确实很有道理，如果他们为了你不让你上，你能不上吗，我想大多数人应该会上，所以他们是不会因为你不上而阻止你上的，但是如果你上了，你的优惠和你的一切都是不可能了，你上了，他们是不能为你划上你的学费的，所以我觉得就算你上了也没有用，他们不会为你划学费，因为你上了从而给了他们非常大的麻烦，你上了，他们是不能有一点好处的，他们还要为你上的学费等等付出，所以他们会说你上不了，就是为了不让你上，不让你上，他们不会为你划学费，所以我觉得你如果上了，你就没有任何

• 异步批量处理

```
from langchain_openai import OpenAI
await llm.abatch(
    [
        "你好"，
        "什么是机器学习"，
        "什么是深度学习"，
    ]
)
```

['，前来亲自拜访的，因为我们这里很少有人来拜访，你就打算在这里待上几个礼拜了吧？’“我是回来找你，我是到桐谷这里来的。”李晓婷冷冷的回道。这一次很明显的不是她想让他留下来的，而是让他走了。众人看着李晓婷，这里的她就这么冷了，竟然连郑晓东都不看一眼，郑晓东是她的男朋友，他们是一对，她怎么可能不看一眼？“那你为什么不和我说，你的事情怎么样了？”郑晓东看向李晓婷的脸，她的表情很冷，她怎么不看一眼自己’，

'? \n\n2:王景仪 2021-01-07 23:12:57\n\nnannounced\n\n3:李桂林 2021-01-01 23:12:57\n\nMy Sunshine is a TV drama adapted from Gu Man's novel of the same title. It stars Wallace Huo and Tang Yan, premiering in January, 2015. The drama depicts a man and a woman's sweet love story.\n\n4:王威 2020-12-27 23:12:57\n\nThe new data released yesterday represent the first snapshot of global trade for 2015. And the figures suggest that 2015 was not a good year for world trade, as it had been hoped.\n\n5:苏晓波 2021-01-01 23:12:57\n\nMark Rutte, the Dutch prime minister whose People's Party for Freedom and Democracy (VVD) is slightly ahead of the PVV, has seen his poll numbers rise in recent days because of his hardline stance with Ankara. On Tuesday he called Mr Erdogan's comments "a disgusting distortion of history", adding: "we will not lower ourselves to this level. It is totally unacceptable."\n\n',

'? 深度学习是一种模仿大脑思维的机器学习方法，深度学习被广泛用于图像识别、语音识别、自然语言处理等领域。深度学习过程中，模型经常需要大量的训练数据，因此大量的数据获取和数据刷新是有必要。·
1) Find a free server to host the application in. I have been trying to create a simple neural network by using the TensorFlow Object Detection API to run an object detection model. Cloud TPU is a Google Cloud service that accelerates machine learning workloads. Major cloud providers offer free tier of services to help you get started. Google Cloud Storage is a RESTful web service for storing and accessing data on Google Cloud Platform. In this video, you will learn about Cloud TPU, a high-performance, scalable, and cost-effective machine learning service. This IP address 157. 0, a neural network framework. Contribute. TensorFlow Lite for mobile and embedded devices For Production TensorFlow Extended for end-to-end ML components. The Google Cloud Platform']

除此之外，在Jupyter Notebook或Python交互式解释器中进行操作的话，可以使用?来查看关于llm中内置的方法和帮助文档，执行如下代码：

llm?

```
[1;31mSignature:•[0m
•[0mllm•[0m•[1;33m(•[0m•[1;33m
•[0m      •[0mprompt•[0m•[1;33m:•[0m•[1;34m'str'•[0m•[1;33m,•[0m•[1;33m
•[0m      •[0mstop•[0m•[1;33m:•[0m•[1;34m'Optional[List[str]]'•[0m•[1;33m=•[0m
•[1;32mNone•[0m•[1;33m,•[0m•[1;33m
•[0m      •[0mcallbacks•[0m•[1;33m:•[0m•[1;34m'Callbacks'•[0m•[1;33m=•[0m
•[1;32mNone•[0m•[1;33m,•[0m•[1;33m
•[0m      •[1;33m*•[0m•[1;33m,•[0m•[1;33m
•[0m      •[0mtags•[0m•[1;33m:•[0m•[1;34m'Optional[List[str]]'•[0m•[1;33m=•[0m
•[1;32mNone•[0m•[1;33m,•[0m•[1;33m
•[0m      •[0mmetadata•[0m•[1;33m:•[0m•[1;34m'Optional[Dict[str, Any]]'•[0m
•[1;33m=•[0m•[1;32mNone•[0m•[1;33m,•[0m•[1;33m
•[0m      •[1;33m**•[0m•[0mkwargs•[0m•[1;33m:•[0m
•[1;34m'Any'•[0m•[1;33m,•[0m•[1;33m
•[0m•[1;33m)•[0m•[1;33m->•[0m•[1;34m'str'•[0m•[1;33m•[0m•[1;33m•[0m•[0m
•[1;31mType:•[0m      OpenAI
•[1;31mString form:•[0m
•[1mOpenAI•[0m
```

```
Params: {'model_name': 'davinci-002', 'temperature': 0.7, 'top_p': 1,
'frequency_penalty': 0, 'presence_penalty': 0, 'n': 1, 'logit_bias': {},
'max_tokens': 256}
```

```
•[1;31mFile:•[0m          d:\software\anaconda3\lib\site-
packages\langchain_openai\llms\base.py
•[1;31mDocstring:•[0m
OpenAI large language models.
```

To use, you should have the ``openai`` python package installed, and the environment variable ``OPENAI_API_KEY`` set with your API key.

Any parameters that are valid to be passed to the `openai.create` call can be passed in, even if not explicitly saved on this class.

Example:

```
.. code-block:: python
```

```
    from langchain_community.llms import OpenAI
    openai = OpenAI(model_name="gpt-3.5-turbo-instruct")
```

```
•[1;31mInit docstring:•[0m
```

Create a new model by parsing and validating input data from keyword arguments.

Raises `ValidationError` if the input data cannot be parsed to form a valid model.

```
•[1;31mCall docstring:•[0m
```

```
[*Deprecated*] Check Cache and run the LLM on the given prompt and input.
```

Notes

```
.. deprecated:: 0.1.7
    use invoke instead.
```

这里能看到非常多的关键信息，比如：

- 默认参数: Params: {'model_name': 'davinci-002', 'temperature': 0.7, 'top_p': 1, 'frequency_penalty': 0, 'presence_penalty': 0, 'n': 1, 'logit_bias': {}, 'max_tokens': 256}

当不输入 `model_name` 时，默认会调用 `davinci-002` 模型：

```
from langchain_openai import OpenAI
```

```
llm = OpenAI()
```

```
await llm.ainvoke(
    "哈喽，你好呀，",
)
```

```
'我是小影。很高兴认识你，有什么可以帮到你的吗？'
```

同时，也可以调整温度 `temperature`：


```
from langchain_openai import OpenAI

llm = OpenAI()
await llm.ainvoke(
    "哈喽，你好呀，",
    temperature=0
)
```

'我是一个人工智能助手，很高兴认识你！有什么可以帮到你的吗？'

- model: 必选参数，具体调用的Completions模型名称，可以调用的模型包括text-davinci-003、text-davinci-002、text-curie-001、text-babbage-001、text-ada-001等，不同模型参数规模不同；这里需要注意，大模型领域不同于机器学习领域，后者哪怕是简单模型在某些场景下可能也会拥有比复杂模型更好的表现。在大模型领域，（就OpenAI提供的A、B、C、D四大模型来看）参数规模越大、越新版本的模型效果更好（当然费用也更高），因此课程中主要以text-davinci-003使用为例进行讲解；
- prompt: 必选参数，提示词；
- suffix: 可选参数，默认为空，具体指模型返回结果的后缀；
- max_tokens: 可选参数，默认为16，代表返回结果的token数量；
- temperature: 可选参数，取值范围为0-2，默认值为1。参数代表采样温度，数值越小，则模型会倾向于选择概率较高的词汇，生成的文本会更加保守；而当temperature值较高时，模型会更多地选择概率较低的词汇，生成的文本会更加多样；
- top_p: 可选参数，取值范围为0-1，默认值为1，和temperature作用类似，用于控制输出文本的随机性，数值越趋近于1，输出文本随机性越强，越趋近于0文本随机性越弱；通常来说若要调节文本随机性，top_p和temperature两个参数选择一个进行调整即可；这里更推荐使用temperature参数进行文本随机性调整；
- n: 可选参数，默认值为1，表示一个提示返回几个Completion；
- stream: 可选参数，默认值为False，表示回复响应的方式，当为False时，模型会等待返回结果全部生成后一次性返回全部结果，而为True时，则会逐个字进行返回；
- logprobs: 可选参数，默认为null，该参数用于指定模型返回前N个概率最高的token及其对数概率。例如，如果logprobs设为10，那么对于生成的每个token，API会返回模型预测的前10个token及其对数概率；
- echo: 可选参数，默认为False，该参数用于控制模型是否应该简单地复述用户的输入。如果设为True，模型的响应会尽可能地复述用户的输入；
- stop: 可选参数，默认为null，该参数接受一个或多个字符串，用于指定生成文本的停止信号。当模型生成的文本遇到这些字符串中的任何一个时，会立即停止生成。这可以用来控制模型的输出长度或格式；
- presence_penalty: 可选参数，默认为0，取值范围为[-2, 2]，该参数用于调整模型生成新内容（例如新的概念或主题）的倾向性。较高的值会使模型更倾向于生成新内容，而较低的值则会使模型更倾向于坚持已有的内容，当返回结果篇幅较大并且存在前后主题重复时，可以提高该参数的取值；
- frequency_penalty: 可选参数，默认为0，取值范围为[-2, 2]，该参数用于调整模型重复自身的倾向性。较高的值会使模型更倾向于避免重复，而较低的值则会使模型更可能重复自身；当返回结果篇幅较大并且存在前后语言重复时，可以提高该参数的取值；
- best_of: 该参数用于控制模型的生成过程。它会让模型进行多次尝试（例如，生成5个不同的响应），然后选择这些响应中得分最高的一个；
- logit_bias: 该参数接受一个字典，用于调整特定token的概率。字典的键是token的ID，值是应用于该token的对数概率的偏置；在GPT中我们可以使用tokenizer tool查看文本Token的标记。一般

不建议修改；

- user：可选参数，使用用户的身份标记，可以通过人为设置标记，来注明当前使用者身份。需要注意的是，Completion.create函数中的user和后续介绍的对话类模型的user参数含义并不相同，需要注意区分；

更详细的参数说明和使用细节可在《Ch.2 Completions模型API使用指南》中找到。此处不在展开重复说明。

目前在LangChain中已经集成的LLMs有82个，均已经实现了对异步、流式处理和批处理的基本支持，查看地址如下：<https://python.langchain.com/docs/integrations/llms/>

Model	Invoke	Async invoke	Stream	Async stream	Batch	Async batch
AI21	✓	✗	✗	✗	✗	✗
AlephAlpha	✓	✗	✗	✗	✗	✗
AmazonAPIGateway	✓	✗	✗	✗	✗	✗
Anthropic	✓	✓	✓	✓	✗	✗
Anyscale	✓	✓	✓	✓	✓	✓
Aphrodite	✓	✗	✗	✗	✓	✗
Arcee	✓	✗	✗	✗	✗	✗
Aviary	✓	✗	✗	✗	✗	✗
AzureMLOnlineEndpoint	✓	✗	✗	✗	✓	✗
AzureOpenAI	✓	✓	✓	✓	✓	✓

4.1.2 Chat Models接入方法

所谓聊天模型，简单理解就是我们经常使用的 xxx-chat 类模型，比如Baichuan2-7b-chat，Qwen-14b-chat，这类模型是基于Base模型经过针对特定数据格式的微调得到，以更贴近人类对话的方式进行输入和输出。所以这类模型不会使用“文本输入、文本输出”的形式，而是采用聊天消息作为输入并返回聊天消息作为输出（非纯文本）。LangChain框架与OpenAI、Cohere、Hugging Face等多家模型提供商已经实现集成，提供了一个标准化的接口，以便与这些不同的模型进行交互。

我们先回顾下OpenAI规范下使用Chat模型进行一次交互需遵循的代码执行方式：

```
from openai import OpenAI
client = OpenAI()

completion = client.chat.completions.create(
    model="gpt-3.5-turbo-0125",
    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"}
    ]
)

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

您好，我是一个智能AI助手，可以帮助您回答问题、提供信息、解决问题等。如果您有任何疑问或需要帮助的话，都可以随时向我提问。我会尽力帮助您解决问题。

这里不得不再次提到，LangChain作为一个应用开发框架，需要集成各种不同的大模型，如上述OpenAI的GPT系列模型调用示例，通过Message数据输入规范，定义不同的role，即system、user和assistant来区分对话过程，但对于其他大模型，并不意味着一定会遵守这种输入输出及角色的定义，所以LangChain的做法是，因为Chat Model基于消息而不是原始文本，LangChain目前就抽象出来的消息类型有 AIMessage、HumanMessage、SystemMessage、FunctionMessage 和 ChatMessage，但大多时候我们只需要处理 HumanMessage、AIMessage 和 SystemMessage，即：

- SystemMessage：用于启动 AI 行为，作为输入消息序列中的第一个传入。
- HumanMessage：表示来自与聊天模型交互的人的消息。
- AIMessage：表示来自聊天模型的消息。这可以是文本，也可以是调用工具的请求。

同样，Chat Models实现了 Runnable 接口，也支持 invoke、ainvoke、stream、astream、batch、abatch 等调用。即：

方法	说明
invoke	处理单条输入
batch	处理批量输入
stream	流式响应
ainvoke	异步处理单条输入
abatch	异步处理批量输入
astream	异步流式响应

所以对于LangChain中调用OpenAI的GPT模型，首先需要导入对应的模块，代码如下：

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

• 消息形式输入调用

定义消息对象：

```
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"),]
```

```
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')]
```

```
from langchain_openai import ChatOpenAI  
  
chat = ChatOpenAI(model_name="gpt-3.5-turbo-0125")
```

执行推理：

```
chat.invoke(messages)
```

```
AIMessage(content='你好，我是一位可以回答问题、提供帮助的智能小助手。无论是关于知识、技能还是日常生活中的问题，我都会尽力为您提供准确和有用的答案。有什么可以帮助您的吗？')
```

如果只想获取到纯净的输出，可以通过.content来格式化。

```
reponse.content
```

```
'你好！我是一位智能助手，可以回答各种问题，提供信息和帮助。无论是关于知识、技能、娱乐、生活等方面，我都会尽力为你提供帮助。有什么问题需要我帮忙的吗？'
```

- 流式调用

```
for chunk in chat.stream(messages):  
    print(chunk.content, end="", flush=True)
```

```
你好！我是一位智能助手，专门为用户提供信息咨询、问题解答和帮助服务。无论你有什么疑问或需求，我都会尽力帮助你，让你更方便地获取所需的信息和支持。有什么我可以帮助你的吗？
```

- 批量调用

先定义三个不同的消息对象：

```
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深度学习"),]
```

```
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"),]
```

将上述三个消息对象放在一个列表中，使用.batch方法执行批量调用。

```
reponse = chat.batch([messages1,  
                      messages2,  
                      messages3,])
```

```
reponse
```

```
[AIMessage(content='机器学习是人工智能的一个子领域，其目的是使计算机系统通过学习数据和经验来改善自身的性能，而无需明确地进行编程。简而言之，机器学习是一种让计算机系统从数据中学习和提取规律、模式以及知识的技术。通过利用统计学和算法来使计算机自动地学习和改进，从而实现更准确和高效的预测、分类、识别等任务。机器学习应用广泛，包括推荐系统、自然语言处理、图像识别、医疗诊断、金融预测等领域。')，
```

```
AIMessage(content='深度学习是一种机器学习的子领域，它模仿人类大脑的结构和功能，通过一系列神经网络层来学习和理解复杂的数据。深度学习模型可以自动发现数据中的模式和特征，从而实现诸如图像识别、语音识别、自然语言处理等任务。深度学习的核心思想是通过训练神经网络来优化模型参数，使其能够对新数据进行准确预测和分类。深度学习在各个领域都取得了显著的成就，被广泛应用于人工智能、医疗、金融、交通等各个领域。')，
```

```
AIMessage(content='大模型技术是指利用大规模、高性能的计算资源和先进的机器学习算法来训练和部署复杂的深度学习模型。这些模型通常具有数十亿到数万亿个参数，需要大量的数据和计算资源来训练和优化。\\n\\n大模型技术各个领域都有广泛的应用，如自然语言处理、计算机视觉、语音识别等。通过使用大模型技术，研究人员和工程师可以构建更加准确和复杂的模型，从而提升机器学习系统的性能和效果。\\n\\n然而，大模型技术也面临着一些挑战，如计算资源的需求高、训练时间长、模型部署复杂等。因此，研究人员和工程师在使用大模型技术时需要权衡各种因素，以确保系统的性能和稳定性。')]
```

格式化输出：

```
# 使用列表生成式打印每一个消息的内容
contents = [msg.content for msg in reponse]

# 打印出内容
for content in contents:
    print(content, "\\n---\\n")
```

机器学习是人工智能的一个子领域，其目的是使计算机系统通过学习数据和经验来改善自身的性能，而无需明确地进行编程。简而言之，机器学习是一种让计算机系统从数据中学习和提取规律、模式以及知识的技术。通过利用统计学和算法来使计算机自动地学习和改进，从而实现更准确和高效的预测、分类、识别等任务。机器学习应用广泛，包括推荐系统、自然语言处理、图像识别、医疗诊断、金融预测等领域。

深度学习是一种机器学习的子领域，它模仿人类大脑的结构和功能，通过一系列神经网络层来学习和理解复杂的数据。深度学习模型可以自动发现数据中的模式和特征，从而实现诸如图像识别、语音识别、自然语言处理等任务。深度学习的核心思想是通过训练神经网络来优化模型参数，使其能够对新数据进行准确预测和分类。深度学习在各个领域都取得了显著的成就，被广泛应用于人工智能、医疗、金融、交通等各个领域。

大模型技术是指利用大规模、高性能的计算资源和先进的机器学习算法来训练和部署复杂的深度学习模型。这些模型通常具有数十亿到数万亿个参数，需要大量的数据和计算资源来训练和优化。

大模型技术在各个领域都有广泛的应用，如自然语言处理、计算机视觉、语音识别等。通过使用大模型技术，研究人员和工程师可以构建更加准确和复杂的模型，从而提升机器学习系统的性能和效果。

然而，大模型技术也面临着一些挑战，如计算资源的需求高、训练时间长、模型部署复杂等。因此，研究人员和工程师在使用大模型技术时需要权衡各种因素，以确保系统的性能和稳定性。

- 异步调用


```
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"),]
```

```
from langchain_openai import ChatOpenAI

chat = ChatOpenAI(model_name="gpt-3.5-turbo-0125")
```

```
reponse = await chat.ainvoke(messages1)

reponse
```

AIMessage(content='机器学习是一种人工智能的分支领域，旨在让计算机系统通过数据和经验自动学习并改进性能，而无需显式地进行编程。通过机器学习，计算机系统可以识别模式、预测结果、做出决策，并不断优化自身的表现。机器学习的应用非常广泛，包括语音识别、图像识别、自然语言处理、推荐系统等领域。常见的机器学习方法包括监督学习、无监督学习、强化学习等。')

```
reponse.content
```

'机器学习是一种人工智能的分支领域，旨在让计算机系统通过数据和经验自动学习并改进性能，而无需显式地进行编程。通过机器学习，计算机系统可以识别模式、预测结果、做出决策，并不断优化自身的表现。机器学习的应用非常广泛，包括语音识别、图像识别、自然语言处理、推荐系统等领域。常见的机器学习方法包括监督学习、无监督学习、强化学习等。'

• 流式调用

```
async for chunk in chat.astream(messages1):
    print(chunk.content, end="", flush=True)
```

机器学习是一种人工智能的分支，它致力于让计算机系统通过学习数据和模式，从而改进自身的性能。简而言之，机器学习让计算机具有从经验中学习的能力，而不需要明确地编程指令。

在机器学习中，计算机系统利用大量数据进行训练，通过识别数据中的模式和规律来建立模型。这些模型可以用来进行预测、分类、识别等任务。机器学习在各个领域中都有广泛的应用，如自然语言处理、图像识别、医疗诊断、金融预测等。

常见的机器学习算法包括监督学习、无监督学习、强化学习等。监督学习是通过已标记的数据进行训练，无监督学习则是在没有标记的数据中发现模式，强化学习则通过试错的方式学习最优策略。机器学习技术的发展使得计算机系统能够更加智能地处理复杂任务，为人们的生活和工作带来了许多便利。

通过上述描述，我们展示了在LangChain中使用LLMs类模型和Chat Model类模型的不同方法，其核心区别在于输入Prompt的格式。除此之外其他的工作可以直接利用统一的抽象接口，实现与模型交互的快速过程。而针对不同的模型，LangChain也提供个对应的接入方法，其相关说明文档地址：<https://python.langchain.com/docs/integrations/chat/>

比如我们以Baichuan2的在线API模型为例快速接入一下：

Chat models

AI21 Labs
Alibaba Cloud PAI EAS
Anthropic
Anthropic Tools
Anyscale
Azure OpenAI
Azure ML Endpoint
Baichuan Chat
Baidu Qianfan

`batch`, `abatch`, `stream`, `astream`. It is implemented as below:

- *Async* support defaults to calling async functions in your application's background thread.
- *Streaming* support defaults to returning the final result returned by the underlying streaming, which requires native async tokens can work for any of our (
- *Batch* support defaults to calling the executor (in the sync batch case the `max_concurrency` key in `Run`

```
from langchain_community.chat_models import ChatBaichuan
from langchain_core.messages import HumanMessage
```

```
chat = ChatBaichuan(
    # 这里替换成个人的有效 API KEY
    baichuan_api_key="sk-xxx",
    streaming=True,
)
```

```
response = chat([HumanMessage(content="我月薪8块钱，请问在闰年的二月，我月薪多少")])
response
```

```
AIMessage(content='闰年的二月有29天，所以你的月薪是：\n\n$8\times 29$=$232$（元）\n\n所以，你在闰年的二月月薪是232元。')
```

```
response.content
```

```
'闰年的二月有29天，所以你的月薪是：\n\n$8\times 29$=$232$（元）\n\n所以，你在闰年的二月月薪是232元。'
```

如未实践过Baichuan2的在线API，请查看《APICh.16 BaiChuan 2 大模型生态介绍及本地私有化部署方案》中 [1. Baichuan在线大模型生态介绍](#) 小节

可以看出，LangChain能够迅速接入不同的模型API。至于接入本地私有部署的开源大模型的方法，我们将在后续课程中进行详细介绍。

4.2 LangChain中如何使用Prompt Template

提示工程（Prompt Engineering）大家应该比较熟悉，这个概念是指在与大语言模型（LLMs），如GPT-3、Qwen等模型进行交互时，精心设计输入文本（即提示）的过程，以获得更精准、相关或有创造性的输出。正如我们在大模型技术应用课程中介绍的，通过采用Few-Shot、Chain of Thought (CoT)等高级提示技巧，可以显著提高大模型在推理任务上的表现。目前，提示工程已经发展成为一个专业领域，非常多的公司设立了专门的职位，负责为特定任务编写精确且具有创造力的提示。

以使用ChatGPT等网页端对话交互应用中，大部分人常见的做法是将提示（Prompt）做硬编码，例如将一段提示文本固定在System Messages中。而在应用开发领域，开发者往往无法预知用户的具体输入内容，同时又希望大模型能够根据不同的应用任务以一种较为统一的逻辑来处理用户输入。所以，LangChain通过提供Prompt Templates功能，优雅地解决了这个问题。Prompt Templates将用户输入到完整格式化提示的转换逻辑进行封装，使得模型能够更灵活、高效地处理各种输入。

LangChain 提供了创建和使用提示模板的工具，其使用 `PromptTemplate` 方法创建字符串提示模板，因为目前为止，语言模型接收的提示基本都是字符串或聊天消息列表。

4.2.1 使用str.format语言构建模版

在LangChain的默认设置下，`PromptTemplate` 使用 Python 的 `str.format()` 方法进行模板化。

Python的 `str.format()` 方法是一种字符串格式化的手段，允许我们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，包括数字格式化、对齐、填充、宽度设置等。

- 基本用法

```
# 简单示例，直接替换
greeting = "Hello, {}".format("Alice")
print(greeting)
# 输出: Hello, Alice!
```

```
Hello, Alice!
```

- 带有位置参数的用法

```
# 使用位置参数
info = "Name: {0}, Age: {1}".format("Bob", 30)
print(info)
# 输出: Name: Bob, Age: 30
```

```
Name: Bob, Age: 30
```

- 带有关键字参数的用法

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

```
Name: Charlie, 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)
# 输出: Name: David, Age: 40
```

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

在LangChain中,基本采用了Python的原生 `str.format()` 方法对输入数据进行格式化,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,以此来引导大模型进行更准确的推理。

```
from langchain.prompts import PromptTemplate

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

prompt = prompt_template.format(type="详细", topic="量子力学")

prompt
```

```
'请给我一个关于量子力学的详细解释。'
```

如上所示,可以使用 `PromptTemplate` 的 `from_template` 方法创建一个提示模板实例,这个模板包含了两个占位符: `{topic}` 和 `{type}`,这些占位符在实际调用时可以被实际的值替换。

- 调用LLMs模型

```
from langchain_openai import OpenAI
llm = OpenAI(model_name="davinci-002", max_tokens=1000)

result = llm.invoke(prompt)
print(result)
```

```
我觉得我已经对量子力学的基本理解了,但是我还是很想听听有关量子力学的详细解释,比如:量子力学是怎么起作用的。
```

谢谢了。

量子力学是在克里姆林宫试验时的实验结果发现.一开始是用熟悉的物理学法则去解释它.比如说,量子力学是量子力学的结果,而不是量子力学的原因.玻尔兹曼方程就是克里姆林宫试验结论的表示.后来人们发现,等效性,柏格效应等现象.然后才发现,原来我们的光学和电磁学等理论是有问题的.这些现象的实现是在离子束中.所以量子力学是很实际的.

[编辑 stuartm 在 06-07-08 12:11]

```
prompt= prompt_template.format(type="简短", topic="机器学习")
```

```
from langchain_openai import OpenAI
llm = OpenAI(model_name="davinci-002", max_tokens=1000)
```

```
result = llm.invoke(prompt)
print(result)
```

2)目前的时代,机器学习已经发展到了什么程度,也有什么应用场景。 3)目前的研究方向是什么。 4)什么时候会用到这些技术。 5)最后一个问题是,对于我自己来说,是不是还有其他的方向比如说法算法之类的,还有在个人的技术知识里面是不是需要这些东西,不是特别需要。也许这些都是我自己的猜测,如果你有什么好的建议,我也非常感谢。最后,我想问一下,你是在哪里学的,是在哪个大学,我就是在这里问问。谢谢。 2019.6.16

电影《伪装者》中的重要概念

原理: 1. 无线电波的能量不会被吸收,当无线电波被反射后,波的方向就会发生改变(这就是反射) 2. 无线电波的方向不会发生改变,传播途中,波的方向会发生改变(这就是偏转) 3. 无线电波的方向会发生改变,传播途中,波的方向不会发生改变(这就是遮挡) 重要概念: 1. 无线电波的方向不会发生改变,传播途中,波的方向会发生改变(这就是遮挡) 2. 无线电波的方向会发生改变,传播途中,波的方向不会发生改变(这就是遮挡) 3. 无线电波的方向会发生改变,传播途中,波的方向不会发生改变(这就是遮挡) 4. 无线电波的方向会发生改变,传播途中,波的方向不会发生改变(这就是遮挡) 5. 无线电波的方向会发生改变,传播途中,波的方向不会发生改变(这就是遮挡) 6. 无线电波的方向会发生改变,传播途中,波的方向不会发生改变(这就是遮挡) 7. 无线电波的方向会发生改变,传播途中,波的方向不会发生改变(这就是遮挡) 8. 无线电波的方向会发生改变,传播途中,波的方向不会发生改变(这就是遮挡) 原理: 1. 无线电波的能量不会被吸收,当无线电波被反射后,波的方向就会发生改变(这就是反射) 2. 无线电波的方向不会发生改变,传播途中,波的方向会发生改变(这就是偏转) 3. 无线电波的方向会发生改变,传播途中,波的方向不会发生改变(这就是遮挡) 重要概念: 1. 无线电波的方向不会发生改变,传播途中,波的方向会发生改变(这就是遮挡) 2. 无线电波的方向会发生改变,传播途中,波的方向不会发生改变(这就是遮挡) 3. 无线电波的方向会发生改变,传播途中,波的方向不会发生改变(这

除此之外,这种模板生成方式支持任意数量的变量,包括无变量,可以做如下测试:

```
from langchain.prompts import PromptTemplate
```

```
prompt_template = PromptTemplate.from_template("给我讲一个笑话")
prompt_template.format()
```

```
'给我讲一个笑话'
```

```
from langchain_openai import OpenAI
llm = OpenAI(model_name="davinci-002", )

result = llm.invoke(prompt)
print(result)
```

```
如果我是一个机器，会被称为什么？
谢谢
```

• 调用Chat Model

上述示例展示了如何使用字符串模板生成和调用LLMs类模型。对于聊天模型，也需要使用 Python 的 `str.format()` 方法进行模板化，但其构造的输入形式则有所区别，应采用聊天消息列表的形式，每条消息都包含内容和一个名为 `role` 的附加参数。例如，在OpenAI的Chat Completions API中，聊天消息需要包含为assistant、user或system。在这种情况下，应使用 `ChatPromptTemplate` 来构建对话模板。

```
from langchain_core.prompts import ChatPromptTemplate

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}"),
    ]
)

messages = chat_template.format_messages(name="小明", user_input="你叫什么名字？")
```

```
messages
```

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

```
print(chat_template)
```



```
input_variables=['name', 'user_input'] messages=[
    SystemMessagePromptTemplate(prompt=PromptTemplate(input_variables=['name'],
    template='你是一个有帮助的AI机器人，你的名字是{name}。')),
    HumanMessagePromptTemplate(prompt=PromptTemplate(input_variables=[],
    template='你好，最近怎么样? ')),
    AIMessagePromptTemplate(prompt=PromptTemplate(input_variables=[], template='我很好，谢谢! ')),
    HumanMessagePromptTemplate(prompt=PromptTemplate(input_variables=['user_input'], template='{user_input}'))]
```

从输出上看，其构造函数在实例化prompt_template时，主要由两个关键参数进行指定：

- input_variables: 这是一个列表，包含模板中需要动态填充的变量名。这些变量名在模板字符串中以花括号（如{name}）标记。通过指定这些变量，可以在后续过程中动态地替换这些占位符。
- template: 这是定义具体提示文本的模板字符串。它可以包含静态文本和input_variables列表中指定的变量占位符。当调用format方法时，这些占位符会被实际的变量值替换，生成最终的提示文本。

调用Chat Model，这里我们选择GPT 3.5。代码如下：

```
from langchain.chat_models import ChatOpenAI
llm = ChatOpenAI(model_name="gpt-3.5-turbo", max_tokens=1000)

result = llm.invoke(messages)
print(result.content)
```

我叫小明，很高兴为您提供帮助！有什么问题可以帮到您？

```
messages = chat_template.format_messages(name="大熊", user_input="你要去哪里玩? ")
```

```
from langchain.chat_models import ChatOpenAI
llm = ChatOpenAI(model_name="gpt-3.5-turbo", max_tokens=1000)

result = llm.invoke(messages)
print(result.content)
```

作为一个AI机器人，我没有身体，所以无法去任何地方玩耍。但我可以帮助你规划旅行或提供有关旅游目的地信息。有什么我可以帮助你的吗？

通过上述流程，我们演示了如何在LangChain框架内使用 ChatPromptTemplate 来构建与聊天模型交互的对话模板。通过此方法，能够创建一个包含不同参与者角色和定制化消息内容的对话流程，这对于开发复杂的聊天应用或增强AI助手的交互能力至关重要。

4.2.2 构造Few-Shot模版

在LangChain中，很多的功能抽象、链路抽象本质上都是在对大模型的“涌现能力”能够应用落地的一种具体实现方法，而其推理的不稳定，在不修改模型本身参数（微调）的情况下，模型涌现能力极度依赖对模型的提示过程，即对同样一个模型，不同的提示方法将获得质量完全不同的结果。最为简单的提示工程的方法就是通过输入一些类似问题和问题答案，让模型参考学习，并在同一个prompt的末尾提出新的问题，依次提升模型的推理能力。

比如我们在《Ch.3 基于思维链的进阶提示工程》中测试的经典推理问题：

```
import os
import openai
openai.api_key = os.getenv("OPENAI_API_KEY")
```

```
prompt = '艾米需要4分钟才能爬到滑梯顶部，她花了1分钟才滑下来，水滑梯将在15分钟后关闭，请问在关闭之前她能滑多少次？'
prompt
```

'艾米需要4分钟才能爬到滑梯顶部，她花了1分钟才滑下来，水滑梯将在15分钟后关闭，请问在关闭之前她能滑多少次？'

```
from openai import OpenAI
client = OpenAI()

response1 = client.completions.create(
    model="gpt-3.5-turbo-instruct",
    prompt=prompt,
    max_tokens=1000
)

response1.choices[0].text
```

'\n在关闭之前，艾米能滑14次。'

```
from openai import OpenAI
client = OpenAI()

response1 = client.completions.create(
    model="gpt-3.5-turbo-instruct",
    prompt=prompt,
    max_tokens=1000
)

response1.choices[0].text
```

'\n\n15分钟 = 900秒\n\n每次滑下来需要1分钟，所以在15分钟内，可以滑下900秒 / (1分钟+4分钟) = 900/5 = 180次。'

可以看到多次执行，其推理均是错误的。正确的计算过程应该是先计算艾米一次爬上爬下总共需要5分钟，然后滑梯还有15分钟关闭，因此关闭之前能够再滑15/5=3次。

而我们当时在OpenAI的API规范下，是通过这种方式来进行Few-Shot的提示：

```
prompt_Few_shot_CoT4 = 'Q: “罗杰有五个网球，他又买了两盒网球，每盒有3个网球，请问他现在总共有多少个网球？” \n\n      A: “罗杰一开始有五个网球，又购买了两盒网球，每盒3个，共购买了6个网球，因此现在总共由5+6=11个网球。因此答案是11。” \n\n      Q: “食堂总共有23个苹果，如果他们用掉20个苹果，然后又买了6个苹果，请问现在食堂总共有多少个苹果？” \n\n      A: “食堂最初有23个苹果，用掉20个，然后又买了6个，总共有23-20+6=9个苹果，答案是9。” \n\n      Q: “杂耍者可以杂耍16个球。其中一半的球是高尔夫球，其中一半的高尔夫球是蓝色的。请问总共有多少个蓝色高尔夫球？” \n\n      A: “总共有16个球，其中一半是高尔夫球，也就是8个，其中一半是蓝色的，也就是4个，答案是4个。” \n\n      Q: “艾米需要4分钟才能爬到滑梯顶部，她花了1分钟才滑下来，水滑梯将在15分钟后关闭，请问在关闭之前她能滑多少次？” \n\n      A: '\n\nprompt_Few_shot_CoT4
```

```
'Q: “罗杰有五个网球，他又买了两盒网球，每盒有3个网球，请问他现在总共有多少个网球？”\n\n      A: “罗杰一开始有五个网球，又购买了两盒网球，每盒3个，共购买了6个网球，因此现在总共由5+6=11个网球。因此答案是11。”\n\n      Q: “食堂总共有23个苹果，如果他们\n用掉20个苹果，然后又买了6个苹果，请问现在食堂总共有多少个苹果？”\n\n      A: “食堂最初有23个苹果，用掉20个，然后又买了6个，总共有23-20+6=9个苹果，答案是9。”\n\n      Q: “杂耍者可以杂耍16个球。其中一半的球是高尔夫球，其中一半的高尔夫球是蓝色的。请问总共有多少个蓝色高尔夫球？”\n\n      A: “总共有16个球，其中一半是高尔夫球，也就是8个，其中一半是蓝色的，也就是4个，答案是4个。”\n\n      Q: “艾米需要4分钟才能爬到滑梯顶部，她花了1分钟才滑下来，水滑梯将在15分钟后关闭，请问在关闭之前她能滑多少次？”\n\n      A: '
```

在Few-Shot的提示下，再次尝试进行提问：

```
from openai import OpenAI\nclient = OpenAI()\n\nresponse1 = client.completions.create(\n    model="gpt-3.5-turbo-instruct",\n    prompt=prompt_Few_shot_CoT4,\n    max_tokens=1000,\n)\n\nresponse1.choices[0].text
```

```
'"艾米花了1分钟滑下来，滑一次上去需要4分钟，因此在15分钟关闭之前，她能滑15分钟-1分钟=14分钟，14分钟除以4分钟每次，她能滑14÷4=3次，答案是3次。"'
```

从输出结果上看，通过Few-Shot提示可以激发大模型的涌现能力从而回答对复杂推理问题，在应用开发中同样需要这种能力来提升大模型的推理稳定性。所以LangChain抽象了

`FewShotPromptTemplate` 方法来实现。我们可以先测试下在LangChain的接口下直接提问：

- 直接提问

```
prompt1 = '艾米需要4分钟才能爬到滑梯顶部，她花了1分钟才滑下来，水滑梯将在15分钟后关闭，请问在关闭之前她能滑多少次？'  
prompt1
```

```
'艾米需要4分钟才能爬到滑梯顶部，她花了1分钟才滑下来，水滑梯将在15分钟后关闭，请问在关闭之前她能滑多少次？'
```

```
from langchain_openai import OpenAI  
  
llm = OpenAI(model_name="gpt-3.5-turbo-instruct")  
llm.invoke(  
    input=prompt1  
)
```

```
'\n\n在关闭之前，艾米能滑(15-4-1)/4=2次。'
```

```
from langchain_openai import OpenAI  
  
llm = OpenAI(model_name="gpt-3.5-turbo-instruct")  
llm.invoke(  
    input=prompt1  
)
```

```
'\n\n在15分钟内，她能滑15次。'
```

其实也很容易想到，LangChain的工作仅仅是集成并接入OpenAI的GPT模型，因此其本质上与直接调用GPT模型的效果相同。出现错误回答也属于正常情况。接下来，我们根据LangChain对Few-Shot模板的定义规范，进行提示词的构建。

在LangChain中，需要使用 `PromptTemplate` 创建字符串提示模板。模板可以包括说明、少量示例以及适合给定任务的特定上下文和问题。因此，我们要创建一个少量示例的列表。每个示例都是一个字典，其中键是输入变量，值是这些输入变量的值。

- 适合LLMs模型的模版构造形式

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

```

from langchain.prompts.prompt import PromptTemplate

examples = [
    {
        "question": "罗杰有五个网球，他又买了两盒网球，每盒有3个网球，请问他现在总共有多少个网球？",
        "answer": "罗杰一开始有五个网球，又购买了两盒网球，每盒3个，共购买了6个网球，因此现在总共由5+6=11个网球。因此答案是11。"
    },
    {
        "question": "食堂总共有23个苹果，如果他们吃掉20个苹果，然后又买了6个苹果，请问现在食堂总共有多少个苹果？",
        "answer": "食堂最初有23个苹果，吃掉20个，然后又买了6个，总共有23-20+6=9个苹果，答案是9。"
    },
    {
        "question": "杂耍者可以杂耍16个球。其中一半的球是高尔夫球，其中一半的高尔夫球是蓝色的。请问总共有多少个蓝色高尔夫球？",
        "answer": "总共有16个球，其中一半是高尔夫球，也就是8个，其中一半是蓝色的，也就是4个，答案是4个。"
    },
]

```

接下来需要实例化一个 `PromptTemplate` 对象，将少量示例格式化为字符串。

```

example_prompt = PromptTemplate(
    input_variables=["question", "answer"], template="Question:
{question}\n{answer}"
)

print(example_prompt.format(**examples[0]))

```

Question: 罗杰有五个网球，他又买了两盒网球，每盒有3个网球，请问他现在总共有多少个网球？
罗杰一开始有五个网球，又购买了两盒网球，每盒3个，共购买了6个网球，因此现在总共由5+6=11个网球。因此答案是11。

```

example_prompt = PromptTemplate(
    input_variables=["question", "answer"], template="Question:
{question}\n{answer}"
)

print(example_prompt.format(**examples[1]))

```

Question: 食堂总共有23个苹果，如果他们吃掉20个苹果，然后又买了6个苹果，请问现在食堂总共有多少个苹果？
食堂最初有23个苹果，吃掉20个，然后又买了6个，总共有23-20+6=9个苹果，答案是9。

最后，创建一个 `FewShotPromptTemplate` 对象。该对象接受经过上一步格式化后的少数样本示例。

```
# 导入 FewShotPromptTemplate 类
from langchain.prompts.few_shot import FewShotPromptTemplate

# 创建一个 FewShotPromptTemplate 对象
few_shot_prompt = FewShotPromptTemplate(
    examples=examples,          # 最开始定义的 examples 示例
    example_prompt=example_prompt, # 上一步定义的 example_prompt 作为提示模板
    suffix="Question: {input}",   # 后缀模板, 其中 {input} 会被替换为实际输入
    input_variables=["input"]     # 定义输入变量的列表
)

# 使用给定的输入格式化 prompt, 并打印结果
print(few_shot_prompt.format(input="艾米需要4分钟才能爬到滑梯顶部, 她花了1分钟才滑下来, 水滑梯将在15分钟后关闭, 请问在关闭之前她能滑多少次? "))
```

Question: 罗杰有五个网球, 他又买了两盒网球, 每盒有3个网球, 请问他现在总共有多少个网球?
罗杰一开始有五个网球, 又购买了两盒网球, 每盒3个, 共购买了6个网球, 因此现在总共由 $5+6=11$ 个网球。因此答案是11。

Question: 食堂总共有23个苹果, 如果他们吃掉20个苹果, 然后又买了6个苹果, 请问现在食堂总共有多少个苹果?
食堂最初有23个苹果, 吃掉20个, 然后又买了6个, 总共有 $23-20+6=9$ 个苹果, 答案是9。

Question: 杂耍者可以杂耍16个球。其中一半的球是高尔夫球, 其中一半的高尔夫球是蓝色的。请问总共有多少个蓝色高尔夫球?
总共有16个球, 其中一半是高尔夫球, 也就是8个, 其中一半是蓝色的, 也就是4个, 答案是4个。

Question: 艾米需要4分钟才能爬到滑梯顶部, 她花了1分钟才滑下来, 水滑梯将在15分钟后关闭, 请问在关闭之前她能滑多少次?

```
final_input = few_shot_prompt.format(input="艾米需要4分钟才能爬到滑梯顶部, 她花了1分钟才滑下来, 水滑梯将在15分钟后关闭, 请问在关闭之前她能滑多少次? ")
final_input
```

```
'Question: 罗杰有五个网球, 他又买了两盒网球, 每盒有3个网球, 请问他现在总共有多少个网球? \n罗杰一开始有五个网球, 又购买了两盒网球, 每盒3个, 共购买了6个网球, 因此现在总共由5+6=11个网球。因此答案是11。 \n\nQuestion: 食堂总共有23个苹果, 如果他们吃掉20个苹果, 然后又买了6个苹果, 请问现在食堂总共有多少个苹果? \n食堂最初有23个苹果, 吃掉20个, 然后又买了6个, 总共有23-20+6=9个苹果, 答案是9。 \n\nQuestion: 杂耍者可以杂耍16个球。其中一半的球是高尔夫球, 其中一半的高尔夫球是蓝色的。请问总共有多少个蓝色高尔夫球? \n总共有16个球, 其中一半是高尔夫球, 也就是8个, 其中一半是蓝色的, 也就是4个, 答案是4个。 \n\nQuestion: 艾米需要4分钟才能爬到滑梯顶部, 她花了1分钟才滑下来, 水滑梯将在15分钟后关闭, 请问在关闭之前她能滑多少次? '
```

将带有Few-Shot提示的输入, 输入到大模型中执行推理。


```
from langchain_openai import OpenAI

llm = OpenAI(model_name="gpt-3.5-turbo-instruct")
llm.invoke(
    input=final_input
)
```

'\n艾米需要4分钟爬到顶部，1分钟滑下来，一共花费5分钟。在15分钟内，她能滑3次，因为 $15 \div 5 = 3$ 。因此答案是3次。'

```
from langchain_openai import OpenAI

llm = OpenAI(model_name="gpt-3.5-turbo-instruct")
llm.invoke(
    input=final_input
)
```

'\n艾米花了5分钟（4分钟爬上去+1分钟滑下来），因此在15分钟内可以滑3次（ $15 \div 5 = 3$ ）。答案是3次。'

从输出结果来看，显然受到了提示工程的正面影响，通过问题的拆解过程显著提高了回答的准确性。

- **适合Chat Model的提示模版构造形式**

我们在前面解释过LLMs和Chat Model的区别，因其接受的输入形式不同，所以构造的提示模版也是不一样的，对于Chat Model来说，它接收的是聊天形式的对话格式，所以对于其对应的Few-Shot，在LangChain中的基本结构如下：

- examples：要包含在最终提示中的字典示例列表。
- example_prompt：通过其 format_messages 方法将每个示例转换为 1 条或多条消息。一个常见的示例是将每个示例转换为一条人工消息和一条人工智能消息响应，或者一条人工消息后跟一条函数调用消息。

我们还是使用四个数学推理题来进行测试：

```
prompt_Few_shot_CoT4 = 'Q: “罗杰有五个网球，他又买了两盒网球，每盒有3个网球，请问他现在总共有多少个网球？” \
```

```
A: “罗杰一开始有五个网球，又购买了两盒网球，每盒3个，共购买了6个网球，因此现在总共由5+6=11个网球。因此答案是11。” \
```

```
Q: “食堂总共有23个苹果，如果他们吃掉20个苹果，然后又买了6个苹果，请问现在食堂总共有多少个苹果？” \
```

```
A: “食堂最初有23个苹果，吃掉20个，然后又买了6个，总共有23-20+6=9个苹果，答案是9。” \
```

```
Q: “杂耍者可以杂耍16个球。其中一半的球是高尔夫球，其中一半的高尔夫球是蓝色的。请问总共有多少个蓝色高尔夫球？” \
```

```
A: “总共有16个球，其中一半是高尔夫球，也就是8个，其中一半是蓝色的，也就是4个，答案是4个。” \
```

```
Q: “艾米需要4分钟才能爬到滑梯顶部，她花了1分钟才滑下来，水滑梯将在15分钟后关闭，请问在关闭之前她能滑多少次？” \
```

```
A: '
```

```
prompt_Few_shot_CoT4
```

```
'Q: “罗杰有五个网球，他又买了两盒网球，每盒有3个网球，请问他现在总共有多少个网球？”
```

```
A: “罗杰一开始有五个网球，又购买了两盒网球，每盒3个，共购买了6个网球，因此现在总共由5+6=11个网球。因此答案是11。”
```

```
Q: “食堂总共有23个苹果，如果他们吃掉20个苹果，然后又买了6个苹果，请问现在食堂总共有多少个苹果？”
```

```
A: “食堂最初有23个苹果，吃掉20个，然后又买了6个，总共有23-20+6=9个苹果，答案是9。”
```

```
Q: “杂耍者可以杂耍16个球。其中一半的球是高尔夫球，其中一半的高尔夫球是蓝色的。请问总共有多少个蓝色高尔夫球？”
```

```
A: “总共有16个球，其中一半是高尔夫球，也就是8个，其中一半是蓝色的，也就是4个，答案是4个。”
```

```
Q: “艾米需要4分钟才能爬到滑梯顶部，她花了1分钟才滑下来，水滑梯将在15分钟后关闭，请问在关闭之前她能滑多少次？”
```

```
A: '
```

• 直接调用

对于Chat Model，需要将输入转化成对话聊天格式：

```
from langchain_core.messages import HumanMessage, SystemMessage
messages = [SystemMessage(content="你是一个擅长数学推理的专家"),
             HumanMessage(content="艾米需要4分钟才能爬到滑梯顶部，她花了1分钟才滑下来，水滑梯将在15分钟后关闭，请问在关闭之前她能滑多少次?"),]
```

实例化模型进行推理：

```
from langchain_openai import ChatOpenAI

chat = ChatOpenAI(model_name="gpt-3.5-turbo-0125")
```

```
resonse = chat.invoke(messages)
resonse.content
```

```
'首先，艾米花了1分钟滑下来，所以她每次滑下来后需要等待3分钟（4分钟上去 + 1分钟下来 = 5分钟，15分钟总时间 - 5分钟 = 10分钟，10分钟 / 3分钟 = 3次）。\n\n在水滑梯关闭之前，艾米可以滑3次。'
```

&esmp; 实际上，在使用GPT-3.5时，我们可以观察到，即使不采用Few-Shot提示，模型也能以很高的概率正确回答问题，这归功于模型本身已经非常强大的能力。所以我们改为Baichuan2模型来测试：

```
from langchain_community.chat_models import ChatBaichuan
from langchain_core.messages import HumanMessage
```

```
chat = ChatBaichuan(
    # 这里替换成个人的有效 API KEY
    baichuan_api_key="sk-xxx",
    streaming=True,
)
```

```
response = chat([HumanMessage(content="艾米需要4分钟才能爬到滑梯顶部，她花了1分钟才滑下来，水滑梯将在15分钟后关闭，请问在关闭之前她能滑多少次?")])
response.content
```

```
'艾米需要1分钟滑下来，所以她在滑梯关闭前可以滑15次。'
```

```
response = chat([HumanMessage(content="艾米需要4分钟才能爬到滑梯顶部，她花了1分钟才滑下来，水滑梯将在15分钟后关闭，请问在关闭之前她能滑多少次?")])
response.content
```

```
'艾米需要1分钟从滑梯顶部滑下来，所以她可以在滑梯关闭前滑15次。'
```

```
response = chat([HumanMessage(content="艾米需要4分钟才能爬到滑梯顶部，她花了1分钟才滑下来，水滑梯将在15分钟后关闭，请问在关闭之前她能滑多少次?")])
response.content
```

```
'艾米需要4分钟才能爬到滑梯顶部，她花了1分钟才滑下来，所以每次滑行需要4+1=5分钟。
\n\n水滑梯将在15分钟后关闭，所以她能在关闭之前滑行15÷5=3次。'
```

可以看到，在切换到Baichuan模型进行测试时，三次测试中出现了两次错误回答。因此，我们就使用Baichuan2模型，尝试通过添加Few-Shot来探索是否能增强其推理能力。

首先，安装LangChain的规范，导入生成示例的模块：

```
from langchain.prompts import (
    ChatPromptTemplate,
    FewShotChatMessagePromptTemplate,
)
```

接下来以input-output对的形式构建Few-Shot提示模板。在这里，“input”可以被视作人类发出的Promot，而“output”则是大模型给出的回答。

```
examples = [
    {"input": "罗杰有五个网球，他又买了两盒网球，每盒有3个网球，请问他现在总共有多少个网球？",
     "output": "罗杰一开始有五个网球，又购买了两盒网球，每盒3个，共购买了6个网球，因此现在总共由5+6=11个网球。因此答案是11。"},
    {"input": "食堂总共有23个苹果，如果他们吃掉20个苹果，然后又买了6个苹果，请问现在食堂总共有多少个苹果？",
     "output": "食堂最初有23个苹果，吃掉20个，然后又买了6个，总共有23-20+6=9个苹果，答案是9。"},
    {"input": "杂耍者可以杂耍16个球。其中一半的球是高尔夫球，其中一半的高尔夫球是蓝色的。请问总共有多少个蓝色高尔夫球？",
     "output": "总共有16个球，其中一半是高尔夫球，也就是8个，其中一半是蓝色的，也就是4个，答案是4个。"}
]
```

然后，使用 `from_messages` 方法进行模版构造。

```
# This is a prompt template used to format each individual example.
example_prompt = ChatPromptTemplate.from_messages(
    [
        ("human", "{input}"),
        ("ai", "{output}"),
    ]
)
few_shot_prompt = FewShotChatMessagePromptTemplate(
    example_prompt=example_prompt,
    examples=examples,
)

print(few_shot_prompt.format())
```

Human: 罗杰有五个网球，他又买了两盒网球，每盒有3个网球，请问他现在总共有多少个网球？
AI: 罗杰一开始有五个网球，又购买了两盒网球，每盒3个，共购买了6个网球，因此现在总共由5+6=11个网球。因此答案是11。
Human: 食堂总共有23个苹果，如果他们吃掉20个苹果，然后又买了6个苹果，请问现在食堂总共有多少个苹果？
AI: 食堂最初有23个苹果，吃掉20个，然后又买了6个，总共有23-20+6=9个苹果，答案是9。
Human: 杂耍者可以杂耍16个球。其中一半的球是高尔夫球，其中一半的高尔夫球是蓝色的。请问总共有多少个蓝色高尔夫球？
AI: 总共有16个球，其中一半是高尔夫球，也就是8个，其中一半是蓝色的，也就是4个，答案是4个。

生成最终的输入模版，设置 input 作为动态变量：

```
final_prompt = ChatPromptTemplate.from_messages(
    [
        few_shot_prompt,
        ("human", "{input}"),
    ]
)
```

```
final_prompt
```

```
ChatPromptTemplate(input_variables=['input'], messages=
[FewShotChatMessagePromptTemplate(examples=[{'input': '罗杰有五个网球，他又买了两盒网
球，每盒有3个网球，请问他现在总共有多少个网球？', 'output': '罗杰一开始有五个网球，又购买了两
盒网球，每盒3个，共购买了6个网球，因此现在总共由5+6=11个网球。因此答案是11。'}, {'input':
'食堂总共有23个苹果，如果他们吃掉20个苹果，然后又买了6个苹果，请问现在食堂总共有多少个苹果？',
'output': '食堂最初有23个苹果，吃掉20个，然后又买了6个，总共有23-20+6=9个苹果，答案是
9。'}, {'input': '杂耍者可以杂耍16个球。其中一半的球是高尔夫球，其中一半的高尔夫球是蓝色的。
请问总共有多少个蓝色高尔夫球？', 'output': '总共有16个球，其中一半是高尔夫球，也就是8个，其中
一半是蓝色的，也就是4个，答案是4个。'}]),
example_prompt=ChatPromptTemplate(input_variables=['input', 'output'], messages=
[HumanMessagePromptTemplate(prompt=PromptTemplate(input_variables=['input'],
template='{input}'))),
AIMessagePromptTemplate(prompt=PromptTemplate(input_variables=['output'],
template='{output}'))])),
HumanMessagePromptTemplate(prompt=PromptTemplate(input_variables=['input'],
template='{input}'))))
```

实例化baichuan2模型：

最后，使用LangChain中的 `chain` 的抽象，合并最终的提示和大模型共同执行。

```
chain = final_prompt | ChatBaichuan(
    # 这里替换成个人的有效 API KEY
    baichuan_api_key="sk-e9",
    streaming=True,
)

response = chain.invoke({"input": "艾米需要4分钟才能爬到滑梯顶部，她花了1分钟才滑下来，水
滑梯将在15分钟后关闭，请问在关闭之前她能滑多少次？"})
response.content
```

'艾米需要4分钟才能爬上去，1分钟滑下来，总共需要4+1=5分钟。15分钟可以滑15÷5=3次，所以答案是3次。'

```
chain = final_prompt | ChatBaichuan(  
    # 这里替换成个人的有效 API KEY  
    baichuan_api_key="sk-ed54e9529",  
    streaming=True,  
)
```

```
response = chain.invoke({"input": "艾米需要4分钟才能爬到滑梯顶部，她花了1分钟才滑下来，水滑梯将在15分钟后关闭，请问在关闭之前她能滑多少次？"})  
response.content
```

'艾米爬上去需要4分钟，滑下来只需要1分钟，所以她可以在15分钟内完成4次滑行，因为每次滑行需要4+1=5分钟。'

```
chain = final_prompt | ChatBaichuan(  
    # 这里替换成个人的有效 API KEY  
    baichuan_api_key="sk-ed54695e9529",  
    streaming=True,  
)
```

```
response = chain.invoke({"input": "艾米需要4分钟才能爬到滑梯顶部，她花了1分钟才滑下来，水滑梯将在15分钟后关闭，请问在关闭之前她能滑多少次？"})  
response.content
```

'艾米需要4分钟才能爬到滑梯顶部，然后1分钟滑下来，总共需要4+1=5分钟。因此，在滑梯关闭之前，她可以滑15÷5=3次。'

4.3 LangChain中的Output Parsers

Output Parsers，即输出解析器，这个概念非常好理解，就是负责获取大模型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。在大多数复杂应用场景中，处理逻辑往往环环相扣，执行某项业务逻辑可能需要多次调用大模型，其中上一次的调用结果将被用于指导下一次调用的逻辑。在这种情况下，结构化的信息会比纯文本又有价值，同时这也是输出解析器的价值所在。

LangChain构造的输出解释器必须实现两个主要方法：

- Get format instructions：该方法会返回一个字符串，其中包含有关如何格式化语言模型输出的指令。
- Parse：该方法会接收字符串，并将其解析为某种结构

目前已经支持的解析格式已经包括json、Xml、Csv以及OpenAI的Tools和Functions等多种格式，具体可看：https://python.langchain.com/docs/modules/model_io/output_parsers/

Name	Supports Streaming	Has Format Instructions	Calls LLM	Input Type	Output Type	Description
OpenAITools		(Passes <code>tools</code> to model)		Message (with <code>tool_choice</code>)	JSON object	Uses latest OpenAI function calling args <code>tools</code> and <code>tool_choice</code> to structure the return output. If you are using a model that supports function calling, this is generally the most reliable method.
OpenAIFunctions	✓	(Passes <code>functions</code> to model)		Message (with <code>function_call</code>)	JSON object	Uses legacy OpenAI function calling args <code>functions</code> and <code>function_call</code> to structure the return output.
JSON	✓	✓		str \ Message	JSON object	Returns a JSON object as specified. You can specify a Pydantic model and it will return JSON for that model. Probably the most reliable output parser for getting structured data that does NOT use function calling.
XML	✓	✓		str \ Message	dict	Returns a dictionary of tags. Use when XML output is needed. Use with models that are good at writing XML (like Anthropic's).

简而言之，这个过程本质上是对文本的格式化处理。但与传统的后处理不同，LangChain采取的实
现策略是通过在Prompt上增加输出解析，直接引导大模型输出的格式化方式。

我们以格式化日期来看一下LangChain的构造思路，首先导入相关的模块：

```
from langchain.output_parsers import DatetimeOutputParser
from langchain.prompts import PromptTemplate
from langchain_openai import OpenAI
```

实例化日期解析器：

```
output_parser = DatetimeOutputParser()
```

通过 `.get_format_instructions` 方法，可以查看定义的解析格式：

```
output_parser.get_format_instructions()
```

```
"Write a datetime string that matches the following pattern: '%Y-%m-%dT%H:%M:%S.%fZ'.\n\nExamples: 1181-01-05T02:20:33.675873Z, 1938-12-13T09:46:36.791229Z, 1468-01-17T07:34:51.108688Z\n\nReturn ONLY this string, no other words!"
```

构造输入模版，这里的区别是：在输入的Prompt Template中，加入了OutPut Parse的内容：

```
template = """用户发起的提问：

{question}

{format_instructions}"""
```

生成最终实际输入大模型的Promot内容：

```
prompt = PromptTemplate.from_template(
    template,
    # 预定义的变量，这里我们传入格式化指令
    partial_variables={"format_instructions":
output_parser.get_format_instructions()},
)
```

```
prompt
```

```
PromptTemplate(input_variables=['question'], partial_variables=
{'format_instructions': "write a datetime string that matches the following
pattern: '%Y-%m-%dT%H:%M:%S.%fZ'.\n\nExamples: 1472-10-23T17:39:56.199188Z,
1643-01-14T10:55:11.903091Z, 0631-12-11T23:04:04.345333Z\n\nReturn ONLY this
string, no other words!"), template='用户发起的提
问:\n\n{question}\n\n{format_instructions}')
```

最后，使用LangChain中的 `chain` 的抽象，合并最终的提示、大模型实例及OutPut Parse共同执行。

```
chain = prompt | OpenAI() | output_parser
```

```
output = chain.invoke({"question": "你好，请问你叫什么?"})
```

```
print(output)
```

```
2021-10-07 12:02:13.345333
```

```
chain = prompt | OpenAI() | output_parser
```

```
output = chain.invoke({"question": "你好，请你帮我解释一下什么是机器学习"})
print(output)
```

```
2018-01-23 09:42:32.123456
```

```
chain = prompt | OpenAI() | output_parser
```

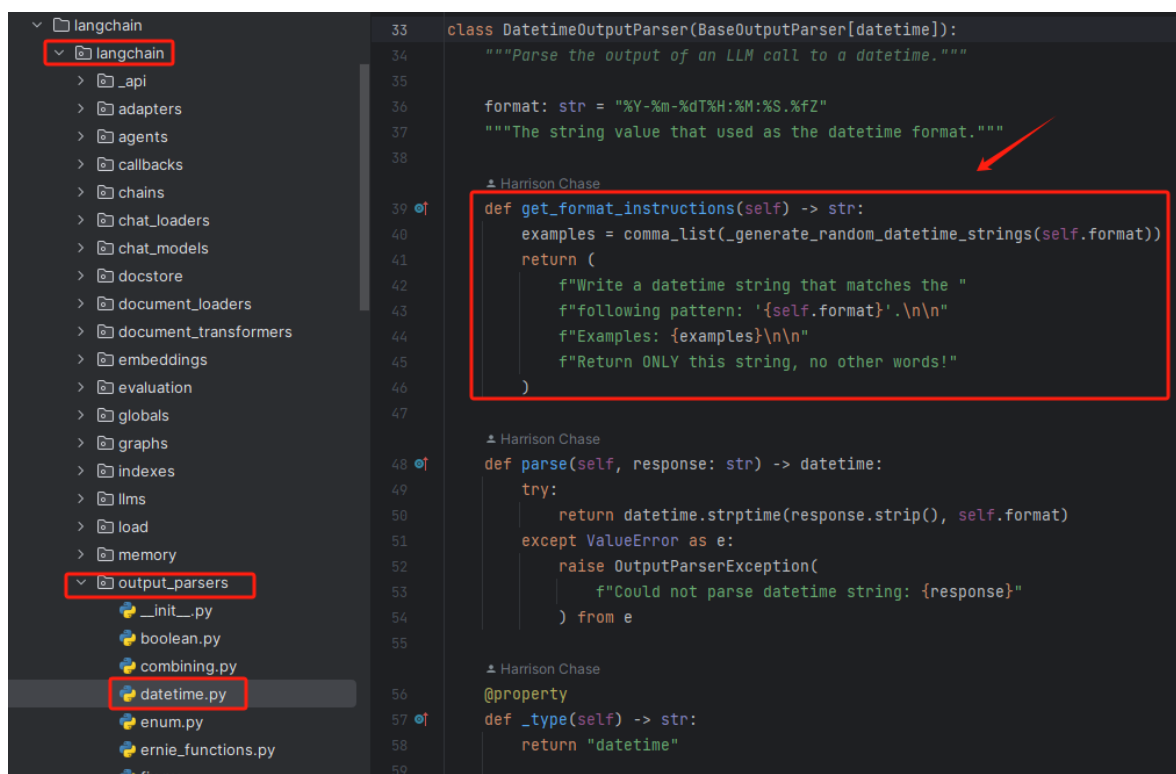
```
output = chain.invoke({"question": "你好，请你帮我解释一下什么是深度学习"})
print(output)
```

```
2019-08-07 18:30:00
```

通过测试可以明显观察到，以此方式定义的输出解析器极大地规范了模型的输出。当然，这也直接受到大模型实际能力和模板定义情况的影响。

LangChain中不同模版的使用方式，除了可以在官方文档上看到，在其源码上看会更加直观，而且可以自定义修改，其所有的OutPut Parse的模版存储路径为：

`langchain/libs/langchain/langchain/output_parsers`，大家可以自行尝试。比如我们上面使用的日期解析的源码对应如下：



```
33 class DatetimeOutputParser(BaseOutputParser[datetime]):
34     """Parse the output of an LLM call to a datetime."""
35
36     format: str = "%Y-%m-%dT%H:%M:%S.%fZ"
37     """The string value that used as the datetime format."""
38
39     def get_format_instructions(self) -> str:
40         examples = comma_list(generate_random_datetime_strings(self.format))
41         return (
42             f"Write a datetime string that matches the "
43             f"following pattern: '{self.format}'.\n\n"
44             f"Examples: {examples}\n\n"
45             f"Return ONLY this string, no other words!"
46         )
47
48     def parse(self, response: str) -> datetime:
49         try:
50             return datetime.strptime(response.strip(), self.format)
51         except ValueError as e:
52             raise OutputParserException(
53                 f"Could not parse datetime string: {response}"
54             ) from e
55
56     @property
57     def _type(self) -> str:
58         return "datetime"
```

到此为止，我们已经非常详尽地介绍了LangChain的核心概念。如果有学习过大模型技术实战课程的同学来说，会发现LangChain的Model I/O模块中讨论的所有概念都是之前接触过的，需要学习的主要是LangChain对大模型进行抽象的方法以及在LangChain框架下的应用开发技巧。

而从LangChain的整体开源情况看，其内容丰富且迭代非常快，通过模块化的方式介绍，可以帮助大家快速构建起对LangChain以及应用开发的思维框架。尽管在Model I/O模块中还有许多更高级的内容，例如如何接入本地部署的开源大模型，关于函数调用、Memory缓存等更深入的主题，这些内容我们将在后续章节中——接触并进行详细的探讨。