3 模型 I/O

LangChain 打通了大语言模型应用开发的“最后一公里”。

2023年后，大语言模型（LLM）的模型平台如同春雨后的竹笋，一根接一根地冒出来。其中，知名度较高的几个包括OpenAI的GPT系列、Anthropic 的Claude 系列、谷歌的PaLM系列，以及最近Meta公司发布的LLaMA系列。这些模型都由各自的模型平台发布，并配备了API接口供开发者使用。

对于开发者来说，要想充分利用这些模型的能力，首先需要了解并掌握每个模型平台的API调用接口。有了这些知识，开发者就可以发起调用，向模型输入数据，并获取模型的输出结果。

问题是，对于初学者来说，面对众多的大语言模型平台和各自不同的API调用协议，确实可能会感到困惑甚至望而却步。毕竟，每个模型平台都有其特定的调用方式和规范，初学者需要投入大量的时间和精力去学习和理解。例如，OpenAI就发布了十几种不同的大语言模型型号。比如最新的GPT-4模型，它就需要使用"Chat"类型的API进行调用。这意味着，每当我们想要使用一个新的模型，就需要重新学习和理解这个模型特定的API调用方式，这对开发者来说无疑增加了额外的工作负担。这就像是每当我们遇到一个新的语言环境，就需要重新学习一种新的语言一样，既费时又费力。

对于那些想要利用大语言模型构建应用的开发者来说，同样如此。以应用程序为例，一个复杂的应用可能包含各种不同的功能需求，这就意味着可能需要调用不同类型的模型来满足这些需求。比如说，在处理文本分类的任务时，我们可能只需要一个参数较少、模型规模较小的模型就能够胜任。但在处理聊天场景时，我们就需要一个能够理解用户输入、并能让对话具有“说人话”的感觉，比如GPT-4。这样一来，我们就需要掌握和管理更多的模型调用方式，这无疑增加了开发的复杂度。

为了解决这些问题，LangChain推出了模型 I/O ，这是一种与大语言模型交互的基础组件。模型 I/O的设计目标是使开发者无需深入理解各个模型平台的API调用协议，就可以方便地与各种大语言模型进行交互。本质上来说，模型 I/O 组件是对各个模型平台API 的封装，这个组件下封装了50多个模型接口。

这就好比LangChain提供了这种“通用包装器”— 模型 I/O，无论你要和哪种模型进行交互，都可以通过这个“包装器”来实现。开发者可以很方便地与最新、最强大的模型（如2023年7月的GPT-4）进行交互，也可以与本地私有化部署的语言模型，甚至在HuggingFace上找到的开源模型进行交互。只需要几行代码，就可以实现与这些模型的对话，而无需关心模型平台的底层API调用方式。

那如何使用 LangChain 的基础组件模型 I/O来访问各个平台的大型语言模型？

模型 I/O 组件提供了三个核心功能，如图 3-1 所示。

[模型](file:///D:\docs\modules\model_io\models\)包装器：通过接口调用语言模型，即模型Model 的包装（图3-1中Predict部分）。

[提示](file:///D:\docs\modules\model_io\prompts\)管理：将模型输入模板化、动态选择和管理，即模型输入Model I（图3-1中Format部分）。

[输出解析器](file:///D:\docs\modules\model_io\output_parsers\)：从模型输出中提取信息 ，即模型输出Model O（图3-1中Parse部分）。

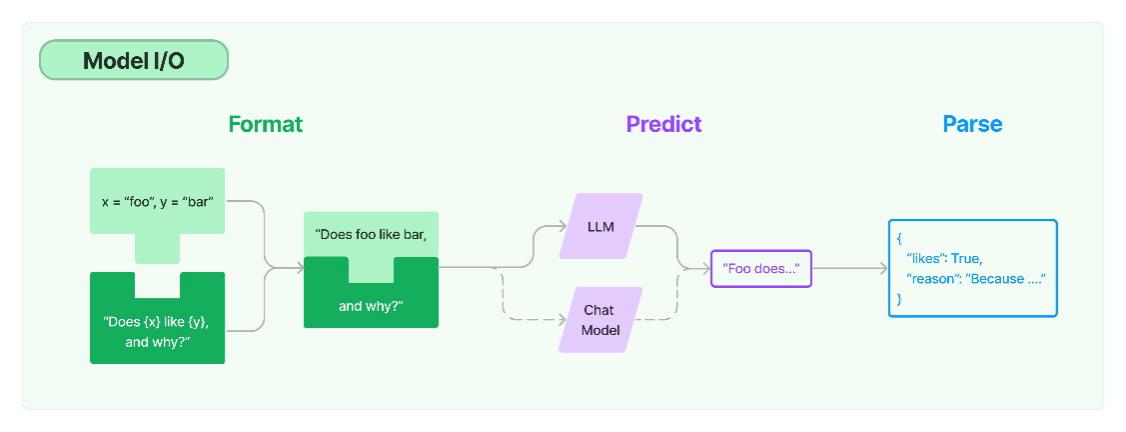


图 3-1

3.1 模型包装器

截至2023年7月，LangChain支持的大语言模型已经超过了50种，这其中包括了来自OpenAI、Facebook、Google等顶尖科技公司的开发大语言模型，以及各类优秀的开源大语言模型。对这些大语言模型，LangChain都提供了模型包装器进行交互。

随着大语言模型的发展，LangChain的模型包装器组件也在不断升级，以适应各大模型平台的API变化。到2023年，OpenAI发布了GPT-3.5-turbo模型，并且在他们的平台上增加了一个新的API类型，即"CHAT"类型API。这种API类型更适合处理聊天场景和复杂的应用场景。截至2023年7月，最新的GPT-4模型和Anthropic的Claude 2模型都采用了"CHAT"类型API。这种API类型正在成为模型平台API的发展趋势。如果不使用这种API，我们将无法利用最强大的GPT-4模型，也无法生成达到"人类质量"的自然语言对话文本。因此，选择适合自己应用需求的API类型，以及配套的LangChain模型包装器组件，是我们在使用大语言模型进行开发时，必须考虑的重要因素。

LangChain的模型包装器组件是基于各个模型平台的API协议进行开发的，它主要提供了两种类型的包装器。一种是通用的LLM类型 —— 文本补全“Completion”，另一种是专门针对"CHAT"类型API的ChatModel类型。

LLM类型的包装器适用于大部分的大语言模型，它们可以处理各种常见的自然语言处理任务，如文本分类、命名实体识别、情感分析等。在大部分情况下，LLM类型的包装器就能满足我们的需求。

然而，当我们要处理的是聊天场景或者更复杂的应用场景时，我们就需要使用ChatModel类型的包装器。ChatModel类型的包装器是专门针对"CHAT"类型API设计的，它能更好地适应聊天场景的特性，例如，它可以处理一系列的消息作为输入，返回一个消息作为输出，这使得它非常适合用于构建聊天机器人或者对话系统。

LangChain的模型包装器组件为我们提供了一种方便的方式来使用各种类型的大语言模型，无论是通用的LLM类型，还是专门针对聊天场景的ChatModel类型，都能让我们更高效地利用大语言模型的能力。

### 3.1.1[模型](file:///D:\docs\modules\model_io\models\)包装器分类

为了适应各种 模型平台的不同类型的API接口，LangChain设计了[模型](file:///D:\docs\modules\model_io\models\)包装器，主要支持两种类型的模型交互方式，它们分别是LLM类型、Chat Model类型。

LLM 包装器组件，是一种专门处理与大语言模型的文本补全类型API交互的组件。这种类型的大语言模型主要用于接收一个字符串作为输入，然后返回一个补全的字符串作为输出。比如，你可以输入一个英文句子的一部分，然后让模型生成句子的剩余部分。这种类型的模型非常适合用于自动写作、编写代码、生成创意内容等任务。

例如你想使用OpenAI 的“text-davinci-003”模型，你可以选择使用 OpenAI 包装器，如下所示：

from langchain.llms import OpenAI

openai = OpenAI(model\_name="text-davinci-003")

在这里， openai 是 OpenAI 类的一个实例，它继承了OpenAI类的所有属性和方法，你可以使用这个 openai 对象来调用 OpenAI 的 “text-davinci-003”模型， 导入的OpenAI 类即为 LangChain 的模型包装器，专门用于处理OpenAI 公司的“Completion” 类型API。

2023年以来，LangChain 已经实现了50种不同大语言模型 “Completion” 类型API的包装器，包括 OpenAI，LlamaCpp，Cohere，Anthropic等。也就是说开发者无需关注这50个模型平台的底层 API 是如何被调用的，LangChain 已经包装好了，让开发者可以“即插即用”。

通过 langchain.llms 获取的所有对象都是LLM类型包装器，这些对象我们称为LLM 类包装器。所有的 LLM 类包装器都是 BaseLLM 的子类，它们继承了 BaseLLM 的所有属性和方法，并根据需要添加或覆盖了一些自己的方法。这些包装器封装了各自平台上的大型语言模型的功能，使得开发者可以以面向对象的方式使用这些大型语言模型的功能，而无需直接与各个模型平台的底层 API 进行交互。

需要注意的是，OpenAI 的 “Completion” 类型API在2023年7月进行了最后一次更新，并且该API现在只能用于访问较旧的历史遗留模型，如“2020-2022年”的模型text-davinci-003、text-davinci-002、davinci、curie、babbage和ada等。

“Completion” 类型API与新的“Chat” 类型API不同。 “Completion” 类型API使得开发者可以直接提供一段具有特定上下文的文本，然后让模型在这个上下文的基础上产生相应的输出。尽管这种方式在某些场景下可能会更方便，比如翻译和写文案场景，但在需要模拟对话或者需要更复杂交互的情况下，OpenAI 平台建议使用“Chat” 类型API。

相比之下，如果要使用OpenAI的最新模型，如2023年以后的模型gpt-4和gpt-3.5-turbo，那么你需要通过 Chat 类型API进行访问。这意味着，如果你想充分利用OpenAI最新的技术进展，你需要将你的应用程序或服务从使用 “Completion” 类型API迁移到使用Chat 类型API。

LangChain创建了Chat Model 类型的组件适配了模型平台的Chat 类型API。

Chat Model是一种专用于大语言模型 “Chat” 类型API 交互的包装器组件。设计这类包装器，主要是为了适配GPT4 等先进聊天模型，这类模型主要用于构建对话系统或聊天机器人。它接收一系列的消息作为输入，并返回一个消息作为输出。这种类型的模型非常适合用于构建能与人进行自然语言交流的应用，比如客服机器人、语音助手等。

2023年7月，LangChain 已经实现了6个针对不同模型平台的Chat Model类型的包装器，包括：

- ChatOpenAI ：用于包装OpenAI Chat 大型语言模型（如 gpt-4和gpt-3.5-turbo）

- AzureChatOpenAI ：用于包装 Azure 平台上的 OpenAI 模型

- PromptLayerChatOpenAI ：用于包装 PromptLayer 平台上的 OpenAI 模型

- ChatAnthropic ：用于包装 Anthropic 平台的大语言模型

- ChatGooglePalm ：用于包装 Google Palm 平台的大语言模型

- ChatVertexAI：用于包装 Vertex AI 平台的大语言模型

不同于LLM 类包装器是 BaseLLM的子类，Chat Model类包装器都是 BaseChatModel 的子类，继承了 BaseChatModel 的所有属性和方法，并根据需要添加或覆盖了一些自己的方法。

例如，如果你想要使用最先进的 GPT-4 模型，你可以选择使用 ChatOpenAI 包装器，如下所示：

from langchain.chat\_models import ChatOpenAI

llm = ChatOpenAI(temperature=0, model\_name="gpt-4")

所有通过 langchain.chat\_models 获取的所有对象都是Chat Model类型，这些对象我们称为Chat Model 类包装器。在这里， llm 是 ChatOpenAI 类的一个实例，你可以使用这个 llm 对象来调用 OpenAI Chat 的 GPT-4 模型的功能。

LLM 类和Chat Model类包装器，都是LangChain对各个大语言模型底层API 的封装，开发者无需关注各个模型平台的底层 API的实现方式，开发者只需关注像模型输入什么，以及模型输出什么。

LangChain 的官网文档中，凡是涉及到模型输入、输出相关的链 （Chain）和代理（Agent）的示例代码，都会提供两份。一份是使用LLM类型，一份是使用Chat Model类型。如图3-1中Predict部分，同样在区分为了LLM 和Chat Model。这是因为两者之间存在着细微但是很重要的区别。

LLM与Chat Model的区别

学习大语言模型应用开发的捷径是关注模型的输入和输出。

从输入和输出的角度来看，LLM 类型和 Chat Model 类型的包装器组件主要有以下区别：

1. 输入：

对于LLM 类型，其输入通常是一个单一的字符串提示（prompt）。例如，你可以输入"Translate the following English text to French: '{text}'"，然后模型会生成对应的法文翻译。主要用于文本完成任务，例如给定一个提示（例如："今天的天气如何？"），模型会生成一个相应的完成（例如："今天的天气很好。"）。

Chat Model 类型，其输入则是一系列的聊天消息。通常这些消息都带有发言人的标签（通常是系统、AI和人类）。每条消息都有一个role（角色）和 content（内容）。例如，你可以输入[{"role": "user", "content": 'Translate the following English text to French: "{text}"'}]，模型会返回对应的法文翻译。

2. 输出：

对于LLM 类型，其输出是一个字符串，这个字符串是模型对输入提示的补全。

Chat Model 类型，其输出是一个聊天消息，这个消息是模型对输入消息的响应。

虽然LLM 类型和Chat Model 类型在处理输入和输出的方式上有所不同，但是为了使它们可以互换使用，它们都实现了基础语言模型接口。这个接口公开了两个常见的方法：predict（接受一个字符串并返回一个字符串）和 predict messages （接受消息并返回一个消息）。这样，无论你是使用特定的模型，还是创建一个应该与不同类型的模型一起工作的应用，都可以通过这个共享的接口来进行操作。

之所以要区分这两种类型，主要是因为它们处理输入和输出的方式不同，且各自适用于不同的使用场景。通过这种方式，可以更好地利用不同类型的语言模型，提高模型的适用性和灵活性。

模型I/O的学习路径

在Langchain的发展迭代过程中，每个模块调用模型I/O功能，都精细地提供了LLM 和Chat Model两种代码方式。因为OpenAI平台的底层API 发生了迭代，Langchain为了不增加开发者修改代码量，更好地适配新的大语言模型发展要求，也做了类型的划分。这种划分已经形成了技术的趋势，同时也为我们学习LangChain提供了线索。当我们看到一个类的名字内包含 “Chat” ，比如ChatAgent ，那么我们要给模型输入的是消息类型的信息，而我们也可以预期 ChatAgent 输出的是消息列表类型。

相比之下，如果你使用的是 LangChain 的 llms 模块导出的对象，这些对象是 LLM 类包装器，主要用于处理自由形式的文本。输入的是一段或多段自由形式的文本，输出的则是模型生成的新文本。这些输出文本可能是对输入文本的回答、延续或其他形式的响应。

最后，如果你使用的是 LangChain 的 chat\_models 模块导出的对象，这些对象是专门设计用来处理对话消息的。输入的是一个对话消息列表，每条消息都由角色（如 user）和内容组成。这样的输入给了大语言模型一定的上下文环境，可以提高输出的质量。输出的也是一个消息类型，这些消息是连续对话内容的响应。

### 3.1.2 LLM 包装器

LLM 包装器是 LangChain 的核心组件之一。LangChain 不提供自己的大语言模型，而是提供了与许多不同的模型平台进行交互的标准接口。

下面我们通过示例代码，演示如何使用LLM 包装器。

示例代码我们使用的LLM 包装器是OpenAI包装器，它封装了OpenAI 平台的接口，导入方式和实例化方法对于所有 LLM 类型都是通用的。以下是示例代码：

安装和设置密钥

首先，我们需要安装 OpenAI Python 包：

pip install openai langchain

然后导入OpenAI包装器并设置好密钥：

from langchain.llms import OpenAI  
OpenAI.openai\_api\_key = "YOUR\_OPENAI\_API\_TOKEN"

使用 LLM 包装器的最简单方法： 输入一个字符串，输出一个字符串

# 运行一个最基本的LLM 包装器，由模型平台 OpenAI 提供的文本生成能力。 b  
llm = OpenAI()  
llm("Tell me a joke")

运行后的结果：

'Why did the chicken cross the road?\n\nTo get to the other side.'

说明： 这里的运行结果是随机的，而不是唯一固定的回答。

### 3.1.3 Chat Model 包装器

当前最大的应用场景便是 “聊天 Chat” ，就像模型平台OpenAI的热门应用ChatGPT一样。为了紧跟用户需求，LangChain推出了专门用于聊天场景的Chat Model 包装器，以便我们能与各种模型平台的 “ Chat” 类型 API 进行交互。

在3.1.1节我们提到Chat Model 包装器的不同之处在于，提供以 “聊天消息” 作为输入和输出的接口。它们的输入不是单个字符串，而是聊天消息的列表。

在本节，我们将通过示例代码，演示输入聊天消息列表的Chat Model 包装器, 与上一节的 LLM包装器在使用上有什么区别？

安装和设置密钥

首先，我们需要安装 OpenAI Python 包。

pip install openai langchain

然后设置好密钥：

import os  
  
os.environ['OPENAI\_API\_KEY'] = ''

使用 Chat Model 包装器的方法： 输入一个消息列表，输出一个字符串

为了使用 Chat Model 包装器，我们将导入三个数据模式 （schema）：一个由 AI 生成的消息的数据模式（AIMessage），一个是人类用户输入的消息的数据模式（HumanMessage）和一个是系统消息的数据模式 （SystemMessage），通常用于设置聊天环境或提供上下文信息。它们定义了应用程序与大语言模型进行聊天时，不同类型消息的数据格式。然后导入 Chat Model 包装器—— ChatOpenAI，这个包装器封装了 OpenAI 平台 Chat 类型的API, 我们无需关注OpenAI 平台的接口如何调用，我们只需关注我们向这个 ChatOpenAI 包装器输入的内容。

from langchain.schema import (  
 AIMessage,  
 HumanMessage,  
 SystemMessage  
)  
from langchain.chat\_models import ChatOpenAI

我们向ChatOpenAI 包装器输入的内容要是一个消息列表。消息列表的数据格式需要符合 AIMessage, HumanMessage 和 SystemMessage 这三种类型的数据模式。这种设计的目的是为了提供一种标准和一致的方式来表示和序列化这些消息。序列化是将数据结构转换为可以存储或传输的格式的过程。在ChatOpenAI 包装器中，序列化是指将这些消息对象转换为可以通过 API 发送的数据格式。这样，接收消息的一方（OpenAI 平台的服务器）就能知道如何正确地解析和处理每个消息。

此时，SystemMessage是指在使用模型时用于配置系统的消息，而HumanMessage是指用户消息。下面将SystemMessage和HumanMessage组合成一个聊天消息列表，输入聊天模型。这里笔者使用的模型名称是“gpt-3.5-turbo”。如果你有“gpt4”，那么也可以使用“gpt4”。

chat = ChatOpenAI(model\_name="gpt-3.5-turbo",temperature=0.3)  
messages = [  
 SystemMessage(content="你是个取名大师，你擅长为创业公司取名字。"),  
 HumanMessage(content="帮我给新公司取个名字，要包含AI的")  
]  
response=chat(messages)  
  
print(response.content,end='\n')

我们创建一个消息列表 messages，这个列表包含了一系列 SystemMessage 和 HumanMessage 对象。每个消息对象都有一个 content 属性，用于存储实际的消息内容。例如，在这个示例代码中，系统消息的内容是“你是个取名大师，你擅长为创业公司取名字。” 人类消息的内容是“帮我给新公司取个名字，要包含AI的”。

当你调用 chat(messages) 时，ChatOpenAI 对象会接收这个消息列表，然后按照AIMessage, HumanMessage 和 SystemMessage 这三种类型的数据模式，将其序列化并发送到 OpenAI 平台的服务器。服务器会处理这些消息，生成 AI 的回应，然后将这个回应发送回 ChatOpenAI 包装器。这个包装器接收回应，回应是一个 AIMessage 对象，你可以通过 response.content 获取它的内容。

包含“ AI ” 的创业公司名称的建议：  
1. AIgenius  
2. AItech  
3. AIvision  
4. AIpros  
5. AIlink  
6. AIsense  
7. AIsolutions  
8. AIwave  
9. AInova  
10. AIboost  
  
希望这些名称能够给你一些启发！

正如你所提到的，Chat Model包装器在使用过程中，相比于LLM包装器，确实更显复杂一些。这主要是因为，Chat Model包装器需要我们先导入三种数据模式，并且需要我们组合一个消息列表messages，最后从响应对象中解析出我们需要的结果。而LLM包装器则简单许多，我们只需要输入一个字符串，然后就能直接得到一个字符串结果。

那么，为什么Chat Model包装器会设计得如此复杂呢？这其实是因为各个模型平台的"chat"类型API接收的数据格式并不统一。为了能够正确地与这些API进行交互，我们必须定义各种消息的数据格式，以满足各个模型平台的"chat"类型API所需的数据格式，以便获取到我们期望的聊天消息结果。

否则，如果我们不使用统一的数据模式，每次向不同的模型平台提交输入时，都需要对输入进行单独的格式处理，这无疑会增加我们的开发工作量。而且，如果我们不进行类型检查，那么一旦出现类型错误，可能会在代码运行时才被发现，这可能会导致程序崩溃，或者产生不符合预期的结果。

尽管Chat Model包装器在使用过程中的复杂性超过了LLM包装器，但这种复杂性是有价值的。通过前置数据处理为标准化格式的工作，我们可以简化和统一数据处理流程，减少出错的可能性。这样，开发者在使用大语言模型时，可以将更多的注意力放在业务逻辑的开发上，而不是被各种复杂的数据处理和错误处理所困扰。

这种设计理念，可以让开发者无需为不同模型平台的API调用方式而烦恼，可以更快速地集成和使用这些强大的大语言模型。

这个演示代码也体现了Chat Model包装器和LLM包装器的最大差异，在于它们所接受的输入数据格式的不同。

Chat Model包装器的设计目标是处理复杂的对话场景，它需要处理的输入是一个聊天消息列表。这个列表中的每一条消息都包含了消息的角色（AI或用户）和消息内容，它们合在一起构成了一个完整的对话上下文。这种输入方式非常适合处理那些需要引入历史对话内容，以便生成更加连贯、有上下文的聊天响应的任务。

LLM包装器的设计目标是处理那些只需要单一输入就可以完成的任务，如文本翻译、文本分类等。因此，它只需要一个字符串作为输入，不需要复杂的消息列表和对话上下文。这种简洁的输入方式，使得LLM包装器在处理一些简单任务时，使用起来更加简单。

## 3.2 提示词

一切都始于“提示词”（Prompt）。事实上，LangChain的一切都是围绕“提示词”构建的。下面我们介绍一下“什么是‘提示词’”，以及在LangChain中的“提示词模板”的概念。

在大语言模型的世界里，“提示词”是一个关键的概念。提示词是引导大语言模型做出人类预期响应的文本。它由应用程序发送到模型平台，这个文本描述了我们希望解决的问题，或者表达了我们期望得到的结果。例如，最简单的提示词可能就是 “你好! ”。当我们输入“你好! ” 这个提示词时，我们期望的是模型能够以一个友好的方式回应，也可能是想确认AI是否在线。

如果大语言模型是个魔法世界，提示词的角色就像哈利波特手中的魔杖，它是我们引导模型产生预期响应的关键。每一个提示词，就像一条魔法咒语，我们挥舞手中的魔杖，将提示词“咒语”输入给模型，希望模型像魔法世界一样，呈现出我们期待的结果。我们对于提示词的探索就如同哈利波特拿到人生的第一支魔杖一样，充满了热情和好奇。人人都想拥有高级的魔杖，实施更强大的魔法，以至于2023年出现了一个全新的大语言模型开发领域，人们称之为“提示词工程”。

诞生“提示词工程”的原因在于实际应用场景的复杂性和多样性，简单的提示词不能满足实际场景的需求，而高级的“提示词”输入给模型，可以提高模型输出结果的质量。例如，当我们希望构建更复杂的应用，如生成专业报告、编写长篇小说或者提供专业咨询等，我们需要“提示词工程”构建出来的提示词，它可能是包含示例，引入外部知识库数据，清晰表达需求，且数据结构化的文本，而不仅仅是一句话。

“提示词工程”，通过对提示词的精心设计和优化，我们可以引导大语言模型生成更符合我们需求的输出。“提示词工程”是一门技术，需要我们对大语言模型的理解，对目标任务的理解，还需要我们的创新和实验。

在LangChain框架中，提示词不是简单的字符串，而是一个更复杂的结构，一个“提示词工程”。这个结构由一个或多个提示词模板（Prompt Template）组成，每个模板可以接受一些输入变量，并根据这些变量生成对应的提示词。这样，我们就可以根据具体的需求和情境，动态地创建出各种各样的提示词。

例如，我们可能有一个提示词模板用于生成写邮件的提示词，这个模板可能需要接受如 "收件人"、"主题"、"邮件内容" 等输入变量。然后，根据这些变量，模板可以生成出如 "写一封给 {收件人} 的邮件，主题是 {主题}，内容是 {邮件内容}" 这样的提示词。

这种工程化的提示词构建方式，使得我们可以构造出适应各种应用程序的结果，而不仅仅是简单的聊天界面。这些复杂的提示词可以用于生成文章、写邮件、回答问题、执行任务等各种场景，大大提高了大语言模型的实用性和可用性。

在LangChain框架中，这种工程化的方式来构建提示词，是通过各种“PromptTemplate”类来生成提示词。在LangChain中，每一个“PromptTemplate” 类的实例都定义了一种特定类型的提示词的格式和生成规则。

“PromptTemplate” 类实际上提供了一个模板，可以在这个模板的基础上，方便地生成复杂的提示词。这就是提示词模板的核心思想和工作方式。

### 3.2.1 提示词模板

提示词可以被视为我们向大语言模型提出的请求，它们决定了我们希望模型进行何种反应。提示词的质量，直接影响了模型的回答质量和我们能否成功完成更复杂的任务。

在LangChain框架中，提示词是由“提示词模板”（PromptTemplate）这个包装器对象生成的。也就是说你在LangChain中，你要构造提示词，你必须学会使用这个包装器。

你可以将提示词模板视为一个幕后工作者，他在幕后默默地工作，用户只看到他们自己输入的简单关键词，却得到了模型出色的响应。这是因为开发者将用户的输入嵌入到预先设计好的提示词模板中，这个模板就是一个包装器，用户的输入经过包装器的包装后，最终输出一个高效的提示词。这个包装器还能组合不同的提示词，提供各种格式化、检验参数的工具，帮助开发者构造复杂的提示词。

提示词模板背后的魔力是，它让用户的简单输入转化为了高效的提示词，从而使得大语言模型能够提供高质量的回答和服务。

提示词模板的定义

在LangChain框架中，提示词模板就是可批量生成提示词的包装器，这个包装器由 PromptTemplate 类生成的。它可以接收开发者对任务的描述文本，也可以接受用户输入的一系列参数。比如说，我们要给公司的产品取一个好听的名字，用户输入的就是产品的品类名字，例如“袜子”或“毛巾”。然而，我们并不需要为每一个品类都编写一个提示词，而是使用这个模板，根据用户输入的不同品类词，生成对应的提示词：“请给公司的 {品类名}，取一个简单容易传播的产品名字” 。

提示词模板的职责就是根据模型的API接口类型，包装并生成适合的提示词。为了满足不同类型的模型平台API需求，PromptTemplate 类的设计具有高度的灵活性。对于需要字符串输入的LLM类型的包装器，我们会使用 to\_string 方法将提示词转化为一个字符串。而对于需要消息列表输入的Chat Model类型的API，我们会使用 to\_messages 方法将提示词转化为一个消息列表。

这就是说，提示词模板可以将用户的简单输入，根据需求转化为复杂的、结构化的提示词。这个过程中，开发者可以向模板中添加实例来影响模型的生成效果，描述任务的特性，甚至包含聊天记录，让模型记住用户的特定信息，如用户的名字。所有这些都是为了让模型能够产生更符合用户需求的回答。

总的来说，提示词模板就是一种能够产生动态提示词的包装器，开发者组织数据输入给包装器，经过包装后，输出的是适配各个模型平台的提示词。

提示词模板的输入

提示词模板是一个输入数据和输出提示词的包装器，那开发者可以向它输入什么样的数据呢？

开发者向提示词模板输入的数据可以来自很多来源，我们可以将其分类为内部数据和外部数据。

内部数据是指那些已经被LangChain框架封装好的数据以及开发者写的示例和需求描述文本。比如，LangChain的许多Agent和Chain实例对象都内置了自己的提示词。这些提示词都被预先定义在源码 prompt.py 文件中，例如我们可以导入预制的API\_RESPONSE\_PROMPT ，它是引导模型根据API 响应回答用户问题的提示词，导入方式是 from langchain.chains.api.prompt import API\_RESPONSE\_PROMPT。

API\_RESPONSE\_PROMPT 在源码中的定义是这样：

API\_RESPONSE\_PROMPT\_TEMPLATE = (

API\_URL\_PROMPT\_TEMPLATE

+ """ {api\_url}

Here is the response from the API:

{api\_response}

Summarize this response to answer the original question.

Summary:"""

)

API\_RESPONSE\_PROMPT = PromptTemplate(

input\_variables=["api\_docs", "question", "api\_url", "api\_response"],

template=API\_RESPONSE\_PROMPT\_TEMPLATE,

)

我们可以直接导入 API\_RESPONSE\_PROMPT 这个内置模板后，format 外部的输入数据，将提示词提交给模型平台的 API。

from langchain.chains.api.prompt import API\_RESPONSE\_PROMPT

prompt= API\_RESPONSE\_PROMPT.format(api\_docs=“”,question=“”, api\_url=“”, api\_response=“”)

构建复杂的提示词，像是盖房子，LangChain的提示词模板做了“盖房子”的基建工程，比如检查数据格式，规划提示词的结构，格式化提示词等。这些基建工程通常用于描述模型的任务，或者用于指示模型的行为。

外部数据则是开发者可以自由添加的数据，这些数据可以来自各种来源。最主要的数据是用户的输入，用户和模型的聊天记录，以及开发者为模型增加的外部知识库数据、程序运行的上下文管理信息。例如，开发者可以收集并使用聊天历史记录，这些历史记录可以帮助模型理解之前的对话上下文，从而生成更加连贯和有用的回答。开发者也可以使用用户的输入，这些输入被填充到模板中，可以帮助模型理解用户的需求，从而生成更加符合用户期望的回答。此外，开发者还可以编写自己的示例文本，或者导入外部的文档片段，这些示例文本和文档片段可以帮助模型理解任务的需求，从而生成更加高质量的回答。

总的来说，无论是内部数据还是外部数据，都是利用提示词模板这个包装器，让包装器输出更好的提示词。

结合内部数据和外部数据，我们可以构建出更加复杂和强大的提示词，从而开发出更加强大的大语言模型应用。构建提示词的过程，就是一项“提示词工程”。

提示词模板的输出

在 LangChain 中，我们使用提示词模板构造器来构造适应各种模型平台 API 类型的提示词。具体来说，我们可以根据模型平台的 API 类型，生成出两种类型的提示词。例如，OpenAI 平台的 API 就有两种类型，一种是 LLM 类，一种是 Chat 类。这两种 API 类型对应了两种不同类型的交互方式。LLM 类的 API 接收的是一个字符串作为输入，而 Chat 类的 API 接收的是一个消息列表。因此，如果你要使用的是 OpenAI 的 GPT-4 模型，那么你就需要准备一个消息类型的提示词。

对于 LLM 类的 API，我们有 BasePromptTemplate 类和它的子类 PromptTemplate，这些类实例化为一个包装器对象后，可以生成一个字符串的提示词。在这种情况下，包装器可以构造出一个字符串的提示词。这个字符串可能包含了一些特定的任务描述，用户的问题，或者其他的上下文信息，它们都被整合在一起，构成了一个完整的、用于引导模型生成预期输出的提示词。字符串的提示词如下：

' You are a helpful assistant that translates English to French '

而对于 Chat 类的 API，我们有 BaseChatPromptTemplate 类和它的子类 ChatPromptTemplate，实例化为一个包装器对象后，可以生成一个消息列表的提示词。因为 chat 类型的 API，我们通常需要一个消息列表作为输入。在这种情况下，包装器构造出一个包含多个消息对象的提示词。每个消息对象都代表了一条消息，它可能是一个用户的问题，一个 AI 的回答，或者一个系统的指令。这些消息被组织在一起，形成了一个清晰的对话流程，用于引导模型进行复杂的对话任务。消息列表的提示词如下：

[SystemMessage(content='You are a helpful assistant that translates English to French.', additional\_kwargs={}),

HumanMessage(content='I love programming.', additional\_kwargs={})]

无论你使用哪种类型的 API，只要选择对应的提示词模板类，就可以轻松地生成出符合 API 要求的提示词。这极大地简化了构建提示词的过程，使得开发者可以将更多的精力放在解决业务逻辑上，而无需手动处理复杂的数据转换和格式化工作。

模板的三个组成部分

提示词模板主要由三个部分组成：语言模型的说明，一组少数示例，以及一个问题或任务。Langchain将这三个部分组合起来，并为输出结果进行格式化的处理，以生成一个完整的提示。

提示词模板有许多实际应用场景。比如，它在做摘要或分类时非常实用。以新闻摘要为例，我们可以构建一个模板，模型的任务就是生成新闻的摘要。用户的输入可能是复制黏贴了一篇完整的新闻文章，给语言模型的说明可能是“生成这篇新闻的摘要”，示例则可以是一些已经生成过的新闻摘要。然后，Langchain将这三个部分组合起来，生成一个完整的提示，如：“请为这篇新闻生成一个摘要。举例” 根据这个提示，模型就能生成一个符合要求的新闻摘要。这样，我们就可以利用提示词模板，轻松地为任意一篇新闻生成摘要。

值得庆幸的是Langchain提示词模板构造的提示词，在任何大语言模型中是可用的，但是输出的回答质量取决于大语言模型自身的能力。

构造提示词的步骤

构造提示词是通过特定的格式和规则，将用户的输入数据包装为能够引导模型生成期望输出的提示词。LangChain 提供了一套内置提示词模板，这些模板可以用来生成各种任务的提示词。在一些基础和通用的场景中，使用内置模板可能就足够了，但在一些特定和复杂的场景中，可能需要创建自定义模板。这里我们介绍最常见的使用 LangChain 提供的内置模板来构造提示词。

PromptTemplate是LangChain提示词组件中最核心的一个类，构造提示词的步骤本质上是实例化这个类的过程。这个类被实例化为对象，也就是我们所说的提示词包装器，在LangChain各个链组件中被调用。在实例化PromptTemplate时，两个关键参数是template和input\_variables。我们只需要准备好这两个参数，就可以实例化一个基础的PromptTemplate ，结果就是一个PromptTemplate对象, 即一个提示词模板的包装器。

from langchain import PromptTemplate  
template = """  
You are an expert data scientist with an expertise in building deep learning models.   
Explain the concept of {concept} in a couple of lines  
"""

#实例化PromptTemplate：

prompt = PromptTemplate(template=template, input\_variables=["concept"])

在LangChain中，这个包装器会像快递箱一样，被传递给链组件进行调用。链组件的调用方式非常直观，例如上面代码的实例化后的对象 prompt，将这个包装器传递给LLMChain：chain = LLMChain(llm, prompt=prompt)。最后，我们可以使用 chain({"concept ": "NLP"}) 来运行链。这种使用方式是LangChain中最常见的模式。

值得注意的是，PromptTemplate包装器接受内部数据（即在实例化时定义的template和input\_variables）和外部数据（即在运行链时传递的数据）。在我们使用链组件调用时，外部数据（即用户的输入）是通过链组件传递的，而不是直接传递给提示词模板包装器。

如果我们不需要链组件进行调用，PromptTemplate包装器还提供了一些其他方法。例如，format方法可以将PromptTemplate包装器的内部和外部数据合并，形成一个完整的提示词，方便我们查看完整的提示词内容。

from langchain import PromptTemplate  
template = """  
You are an expert data scientist with an expertise in building deep learning models.   
Explain the concept of {concept} in a couple of lines  
"""

#实例化模板的第一种方式：

prompt = PromptTemplate(template=template, input\_variables=["concept"])

# 实例化模板的第二种方式：  
# prompt = PromptTemplate.from\_template(template)

# 将用户的输入通过format 方法嵌入到提示词模板，并且做格式化处理

final\_prompt = prompt.format(concept="NLP")

我们打印 final\_prompt 提示词的结果是：

'\nYou are an expert data scientist with an expertise in building deep learning models. \nExplain the concept of nlp in a couple of lines\n'

创建提示词模板主要涉及以下两个要求：

它需要有一个 input\_variables 属性，这个属性指定了提示词模板期望的输入变量。这些输入变量是在生成提示词时需要的数据，比如在我们的示例中，输入变量就是 {concept}。

如果不想显示指定输入变量，还可以使用 from\_template 方法，代码中的第二种方式。这个方法接收与预期的 input\_variables 对应的关键字参数，并返回格式化的提示词。在我们的示例中，from\_template 方法实例化模板，format 方法接收 concept="NLP" 作为输入，并返回格式化完成的提示词。

PromptTemplate类实例化的结果是包装器，包装器可以是自己被链组件调用，也可以调用方法（连结外部的用户输入和内部定义的关键参数），最终的结果是实现内部数据和外部数据的整合，形成一个完整的提示词。

同样地，Langchain 为Chat Model 类型的API 定义了ChatPromptTemplate 类，用于构造消息列表的提示词。构造的步骤基本与上面的LLM 类型一致，但是仍然有区别。

构造一个Chat Model类型的提示词

Chat Model类型的提示词模板的不同之处，构造的提示词是消息列表, 支持输出 Message 对象。Langchain 提供了聊天内置模板 （ChatPromptTemplate）和角色消息内置模板。角色消息提示词模板包含AIMessagePromptTemplate, SystemMessagePromptTemplate 和 HumanMessagePromptTemplate 三种角色消息提示词模板。

无论看起来多么复杂，构造都遵循构造提示词的步骤，实例化内置的模板类为包装器对象，包装器来格式化外部的用户输入，包装器调用类方法输出提示词。

我们将上一个示例代码改造为Chat Model类型的提示词模板对象，下面是代码示例：

先导入聊天内置模板 （ChatPromptTemplate）和角色消息内置模板：

from langchain.prompts import (  
 ChatPromptTemplate,  
 PromptTemplate,  
 SystemMessagePromptTemplate,  
 AIMessagePromptTemplate,  
 HumanMessagePromptTemplate,  
)

改造思路是生成人类和系统的消息提示类型的提示词对象，实例化SystemMessagePromptTemplate 类和HumanMessagePromptTemplate 类为包装器，再实例化 ChatPromptTemplate类，将前面两个对象作为参数传递给ChatPromptTemplate类实例化后的包装器，调用其from\_messages 方法，生成为消息列表的提示词包装器：

先使用 from\_template 方法实例化消息类型的内置模板， SystemMessagePromptTemplate 类和 HumanMessagePromptTemplate 类。传入定义的template模板字符串，实例化人类和系统两种模板对象。

from langchain import PromptTemplate  
  
template = """  
You are an expert data scientist with an expertise in building deep learning models.   
"""  
  
system\_message\_prompt = SystemMessagePromptTemplate.from\_template(template)  
human\_template="Explain the concept of {concept} in a couple of lines"  
human\_message\_prompt = HumanMessagePromptTemplate.from\_template(human\_template)

然后将上面两个模板对象作为参数传入ChatPromptTemplate 类的实例化方法 from\_messages，转化为 ChatPromptTemplate 的提示词包装器：

chat\_prompt=ChatPromptTemplate.from\_messages([system\_message\_prompt,human\_message\_prompt])

打印这个结果：

ChatPromptTemplate(input\_variables=['concept'], output\_parser=None, partial\_variables={}, messages=[SystemMessagePromptTemplate(prompt=PromptTemplate(input\_variables=[], output\_parser=None, partial\_variables={}, template='\nYou are an expert data scientist with an expertise in building deep learning models. \n', template\_format='f-string', validate\_template=True), additional\_kwargs={}), HumanMessagePromptTemplate(prompt=PromptTemplate(input\_variables=['concept'], output\_parser=None, partial\_variables={}, template='Explain the concept of {concept} in a couple of lines', template\_format='f-string', validate\_template=True), additional\_kwargs={})])

最后使用包装器对象的format 方法，将用户的输入传递进包装器，组合为完整的提示词对象：

chat\_prompt.format\_prompt(concept="NLP")

调用format\_prompt 方法，获得的是 ChatPromptValue 对象：

ChatPromptValue(messages=[SystemMessage(content='\nYou are an expert data scientist with an expertise in building deep learning models. \n', additional\_kwargs={}), HumanMessage(content='Explain the concept of NLP in a couple of lines', additional\_kwargs={}, example=False)])

ChatPromptValue 有to\_string 和 to\_messages 方法。

chat\_prompt.format\_prompt(concept="NLP").to\_messages()

我们调用to\_messages 方法获得的结果：

[SystemMessage(content='\nYou are an expert data scientist with an expertise in building deep learning models. \n', additional\_kwargs={}),

HumanMessage(content='Explain the concept of NLP in a couple of lines', additional\_kwargs={}, example=False)]

我们调用to\_string 方法获得的结果：

'System: \nYou are an expert data scientist with an expertise in building deep learning models. \n\nHuman: Explain the concept of NLP in a couple of lines'

值得一提的是，LLM和Chat Model类型的提示词包装器，在实现方式上存在差异，包括他们所使用的内置模板以及实例化的方法。

对于LLM类型，其内置模板是PromptTemplate，而Chat Model类型则使用的是ChatPromptTemplate。PromptTemplate的实例化方法相对简单，只需传递input\_variables和template参数后，直接进行函数式调用或者使用from\_template的类方法进行调用。比如：

PROMPT = PromptTemplate.from\_template (template=template)

相比之下，ChatPromptTemplate 的实例化就复杂得多。它接受的参数是已经实例化的多个对象列表（例如，代码中的system\_message\_prompt和human\_message\_prompt）。如果我们把ChatPromptTemplate实例化的对象视为“大包”，那么传入的包装器就是“小包”，形成了一种“大包装小包”的情况。此外，这个“大包”的实例化类方法也与PromptTemplate不同，它使用的是from\_messages方法，而这个方法只接受消息列表的数组，比如如下代码中的变量 messages ：

messages = [

SystemMessagePromptTemplate.from\_template(system\_template),

HumanMessagePromptTemplate.from\_template("{question}"),

]

CHAT\_PROMPT = ChatPromptTemplate.from\_messages(messages)

尽管存在这些差异，但LLM和Chat Model类型的实例化仍有一定的规律，方便我们记忆和使用。以format为前缀的类方法，主要用于在实例化模板对象后，将外部用户的输入格式化到对象内。如果是实例化LLM类型的内置模板对象，我们使用的是format方法，而对于Chat Model类型，则使用format\_prompt方法。

类似地，以from\_为前缀的类型方法，主要用于实例化内置模板对象。PromptTemplate类只能使用的是from\_template，而ChatPromptTemplate 类则使用from\_messages 方法。

此外，为了实现这两种类型的相互转换，Chat Model类型使用format\_prompt方法实例化对象，生成的对象符合PromptValue数据模式（schema）。所有返回该数据模式的对象都包含以to\_为前缀的方法名，包括to\_string和to\_messages方法，分别用于导出字符串和包含角色的消息列表。

### 3.2.2 少样本提示词模板

少样本提示（FewShotPromptTemplate）是一种机器学习技术，它利用少量的样本（即为提示词）来引导模型进行特定任务的学习和执行。这些提示或示例提供了一种方式，让模型理解我们期望它完成的任务类型和风格。在给定的任务中，这些提示通常包含问题（或任务描述）以及相应的答案或解决方案。

例如，如果我们希望一个语言模型能够以某种特定的风格来回答用户的查询，那么我们可以给模型提供几个已经按照这种风格编写好的问题和答案对。这样，模型就能通过这些示例来理解我们期望的回答风格，并在处理新的用户查询时尽可能地模仿这种风格。

OpenAI的文档中也强调了这种技术的重要性。文档指出，尽管通常情况下，为所有示例提供适用的一般性指令比示例化所有任务的可能性更为高效，但在某些情况下，提供示例可能更为简单，尤其是当你想要模型复制一种难以明确描述的特定响应风格时。在这种情况下，"少样本提示"能够通过少量的示例，帮助模型理解并复制这种特定的响应风格，从而大大提高了模型的使用效率和效果。

FewShotPromptTemplate 是LangChain内置的一种提示词模板的类，其独特之处在于支持动态添加示例和选择示例。这样，示例在提示词中就不再是固定的，而是可以“动态变化”，能够适应不同的需求。这符合OpenAI文档中的建议，LangChain也认为这个内置模板是必须的，它可以为开发者节约大量时间。另外，LangChain还封装了示例选择器，以支持这种模板示例的动态化。

通过观察FewShotPromptTemplate类的源码，我们可以看到它如何实现这种动态化。这个类继承自StringPromptTemplate类，它的实例化方法和PromptTemplate类完全一样。然而，FewShotPromptTemplate类在参数上多了一些东西，例如examples（示例）和example\_selector（示例选择器）。这些参数使得我们可以在实例化对象时添加示例，或者在运行时动态选择示例。

即使这个类添加了一些新的特性，但它的使用方式仍然和PromptTemplate类一样。如果你想要在链组件上使用它，你只需要像使用PromptTemplate类一样来使用它。实际上，FewShotPromptTemplate类只是给实例化对象添加了更多的外部数据，即示例，但并没有改变它的使用方式。

如果你想引导模型得到更好的结果，那么你应该更多地使用FewShotPromptTemplate，因为它在PromptTemplate的基础上添加了示例功能。正如我们前面所说，添加示例的提示词会让模型得到更准确的回答。

FewShotPromptTemplate与PromptTemplate 的区别

我们现在讨论的是PromptTemplate 和 FewShotPromptTemplate 两个类的实例化过程以及它们之间的差异。这两个类都是LangChain的提示词模板，但它们有一些重要的区别。

PromptTemplate 类是一种基本的提示词模板，它接收一个包含变量的模板字符串和一个变量列表。例如，在以下的实例化过程中：

example\_prompt=PromptTemplate(input\_variables=["input","output"],

template="""

词语: {input}\n

反义词: {output}\n

""")

template是一个包含两个变量 {input} 和 {output} 的模板字符串，而 input\_variables 是一个包含这两个变量名称的列表。这个 PromptTemplate 对象可以用来生成提示词，例如通过调用 example\_prompt.format(input="好", output="坏") 可以生成提示词： “\n词语: 好\n\n反义词: 坏\n\n”。

然而， FewShotPromptTemplate 类提供了更高级的功能。它不仅继承了 PromptTemplate 的所有属性和方法，还添加了一些新的参数来支持少样本示例：

few\_shot\_prompt = FewShotPromptTemplate(

examples=examples,

example\_prompt=example\_prompt,

example\_separator="\n",

prefix="我们来玩个反义词游戏，我说词语，你说它的反义词\n",

suffix="词语: {input}\n反义词: ",

input\_variables=["input"],

)

在这个例子中， examples 是示例列表， example\_prompt 是用于格式化这些示例的 PromptTemplate 对象。而 prefix 和 suffix 则构成了用于生成最终提示词的模板，其中 suffix 还接收用户的输入。这种设计使得 FewShotPromptTemplate 可以在给出指导和接收用户输入的同时，还能够展示一系列的示例。

我们可以看到， FewShotPromptTemplate 的 prefix 和 suffix 参数的组合实际上等价于 PromptTemplate 的 template 参数，因此它们的目的和作用是一样的。然而， FewShotPromptTemplate 提供了更多的灵活性，因为它允许我们在提示词中添加示例。这些示例可以是硬编码在模板中的，也可以是动态选择的，这取决于我们是否提供了 ExampleSelector 对象。

总的来说， FewShotPromptTemplate 是 PromptTemplate 的扩展，它在保留了 PromptTemplate 的所有功能的同时，还提供了对少样本示例的支持。这使得我们可以更方便地使用少样本提示技术，而这种技术已被证明能够改善模型的性能。

少样本提示词模板的示例

这里，我们将通过示例探讨如何通过少样本提示词模板，如何加入一些具体的例子进入提示词模板，构造一个包含示例的提示词。

首先，我们需要认识新参数。在LangChain的FewShotPromptTemplate类中，参数example\_selector，example\_prompt，prefix 和 suffix具有以下含义和使用方式：

example\_selector: 这是一个ExampleSelector对象，用于选择要格式化进提示词的示例。如果你想让模型基于一组示例来生成响应，那么你可以提供一个ExampleSelector对象，该对象会根据某种策略（例如随机选择，基于某种标准选择等）从一组示例中选择一部分示例。如果你没有提供ExampleSelector对象，那么你应该直接提供一个示例列表（通过examples参数）。它是必填参数。

example\_prompt: 这是一个PromptTemplate对象，用于格式化单个示例。当你提供了一组示例（无论是直接提供示例列表，还是提供了ExampleSelector对象）后，FewShotPromptTemplate会用example\_prompt来格式化这些示例，生成最终的提示词。它是必填参数。

prefix 和 suffix 参数的组合实际上等价于 PromptTemplate 的 template 参数，因此它们的目的和作用是一样的。suffix 参数是必填的。

现在，让我们通过一个提供一个示例列表（通过examples参数）的例子来理解这个过程，这样很简单也很容易理解，少样本提示词模板的实例化过程。

examples = [

    {"input": "高", "output": "矮"},

    {"input": "胖", "output": "瘦"},

    {"input": "精力充沛", "output": "萎靡不振"},

    {"input": "快乐", "output": "伤心"},

    {"input": "黑", "output": "白"},

]

假设我们现在的任务是让模型进行反义词接龙。在这个任务中，我们会给模型一个词，然后期望模型返回这个词的反义词。因此，我们需要提供一些示例，例如 “高” 对应的反义词是 “矮”，“胖” 的反义词是 “瘦”，以此类推。

example\_prompt=PromptTemplate(input\_variables=["input","output"],

    template="""

词语:  {input}\n

反义词:  {output}\n

""")

我们像构造提示词模板对象一样，构造一个普通的 PromptTemplate 对象，用于格式化单个示例。

example\_prompt.format(\*\*examples[0])

# 打印的结果：'\n词语: 高\n\n反义词: 矮\n\n'

调用 format 方法，填入 input 和 output 参数。 当你写example\_prompt.format(\*\*examples[0])`时，\*\*examples[0] 会将第一个字典的键值对解开，然后作为关键字参数传递给format方法。这等价于example\_prompt.format(input="高", output="矮")。

然后，我们通过实例化 FewShotPromptTemplate类来设置我们的提示词模板。

few\_shot\_prompt = FewShotPromptTemplate(

  examples=examples,

  example\_prompt=example\_prompt,

  example\_separator="\n",

  prefix="我们来玩个反义词游戏，我说词语，你说它的反义词\n",

  suffix="现在轮到你了，词语: {input}\n反义词: ",

  input\_variables=["input"],

)

few\_shot\_prompt.format(input="好")

我们为模型设置一些标准的示例 （examples），以帮助模型理解任务需求。接下来，我们会实例化一个FewShotPromptTemplate类，然后在这里传入我们的示例。example\_prompt是一个PromptTemplate对象，用于格式化单个示例。我们还会为此设置一个前缀（prefix="我们来玩个反义词游戏，我说词语，你说它的反义词\n"）和一个后缀（suffix="词语: {input}\n反义词: "），这样可以帮助构造出一个结构清晰的提示词文本。

'我们来玩个反义词游戏，我说词语，你说它的反义词

词语: 高

反义词: 矮

词语: 胖

反义词: 瘦

词语: 精力充沛

反义词: 萎靡不振

词语: 快乐

反义词: 伤心

词语: 黑

反义词: 白

词语: 好

反义词: '

可以看到我们仍然是使用 FewShotPromptTemplate 类的函数式调用来实例化对象。然后，我们运行代码，看看模型能否正确地生成我们期望的结果。例如，如果我们输入“冷”，模型就应该返回“热”。这就是我们期望看到的反义词。

from langchain.llms import OpenAI

from langchain.chains import LLMChain

chain = LLMChain(llm=OpenAI(openai\_api\_key="这里填入OpenAI的密钥"),

prompt=few\_shot\_prompt)

chain.run("冷")

在这段代码中，我们首先实例化了一个LLMChain对象。这个对象是LangChain库中的一个核心组件，可以理解为一个执行链，它将各个步骤连接在一起，形成一个完整的运行流程。LLMChain对象在实例化时需要两个关键参数：一个是llm，这里我们使用了OpenAI提供的大型语言模型；另一个是prompt，这里我们传入的是我们刚刚创建的few\_shot\_prompt对象。

然后，我们通过调用LLMChain对象的run方法来运行这个执行链。在这个方法中，我们传入了一个字符串"冷"，这个字符串将作为输入传递给我们的few\_shot\_prompt对象。

' 热'

模型回应了我们“热”。这就是我们期望看到的反义词。

### 3.2.3 示例选择器

在实际应用开发中，我们面临的情况常常更为复杂。例如，我们可能需要将一篇新闻的摘要作为示例加入到提示词中。更具挑战性的是，我们还可能需要在提示词中加入大量的历史聊天信息或者从外部知识库获取的数据。然而，大型语言模型可以处理的字数是有限的容量。如果我们提供的每个示例都是一篇新闻的摘要，那么很可能就会超过模型能够处理的字数。

为了解决这个问题，LangChain在 FewShotPromptTemplate 类上，设计了示例选择器（Example Selector）的参数。示例选择器的作用是在传递给模型的示例中进行选择，以确保示例的数量和长度不会超过模型的处理能力。这样，即使我们有大量的示例，也能够确保模型能够有效地处理我们的提示词，而不会因为示例过多或者过长而导致模型无法处理。而且，尝试适应所有示例可能会很快变得非常昂贵，尤其是在计算资源和时间上。

这就是示例选择器发挥作用的地方，它帮助我们选择最适合的示例来提示模型。示例提示选择器（Example Selector）提供了一套工具，来解决这个问题。这些工具能基于策略选择合适的例子，如根据例子的长度、输入与例子之间的n-gram重叠分数来评估其相似度打分、找到与输入具有最大余弦相似度的例子, 以及多样性等因素来选择例子, 从而保持提示成本的相对稳定。

根据长度选择示例，是很普遍和现实的需求，以下是根据长度选择示例的代码，在上一节少样本提示词模板的实例代码里，我们没有提供示例选择器对象，而是直接提供一个示例列表（通过examples参数）。这一次我们提供 ExampleSelector 参数，使用示例选择器，选择的策略工具根据长度选择示例的LengthBasedExampleSelector 类，其他几种策略工具类，Langchain 都设计好了，开发者可以直接导入使用。

在这段代码中，首先导入了langchain库的LengthBasedExampleSelector类, 其他均重复上一节少样本提示词模板的实例代码。这个类是一个示例选择器，用于根据指定的长度选择示例。

from langchain.prompts.example\_selector import LengthBasedExampleSelector

然后，实例化了一个LengthBasedExampleSelector对象，传入了之前定义的示例（examples）和示例提示模板（example\_prompt），并设置了最大长度（max\_length）为25。这意味着选择器将选择那些长度不超过25的示例。

example\_selector = LengthBasedExampleSelector(

  examples=examples,

  example\_prompt=example\_prompt,

  max\_length=25,

)

接着，创建了一个FewShotPromptTemplate对象，这次传入了新创建的示例选择器（example\_selector）以及其他参数。这样，生成的提示词将根据选择器选择的示例来生成。

example\_selector\_prompt = FewShotPromptTemplate(

  example\_selector=example\_selector,

  example\_prompt=example\_prompt,

  example\_separator="\n",

  prefix="我们来玩个反义词游戏，我说词语，你说它的反义词\n",

  suffix="现在轮到你了，词语: {input}\n反义词: ",

  input\_variables=["input"],

)

example\_selector\_prompt.format(input="好")

当我们调用example\_selector\_prompt.format(input="好")后，程序将根据input值和示例选择器来生成一个提示词。

'我们来玩个反义词游戏，我说词语，你说它的反义词\n\n\n词语: 高\n\n反义词: 矮\n\n\n\n词语: 胖\n\n反义词: 瘦\n\n\n现在轮到你了，词语: 好\n反义词: '

在结果中，我们发现并没有所有的示例都出现在了生成的提示词中，这是因为我们设置的最大长度（max\_length）为25，因此一些过长的示例被选择器过滤掉了。

example\_selector = LengthBasedExampleSelector(

  examples=examples,

  example\_prompt=example\_prompt,

  max\_length=100, # 修改25为100

)

如果我们将最大长度参数改为100（max\_length=100），那么所有的示例都将被选择，因为所有示例的长度都不超过100。

[{'input': '高', 'output': '矮'}, {'input': '胖', 'output': '瘦'}, {'input': '精力充沛', 'output': '萎靡不振'}, {'input': '快乐', 'output': '伤心'}, {'input': '黑', 'output': '白'}]

这段代码展示了如何使用基于长度的示例选择器（LengthBasedExampleSelector）和少样本提示模板（FewShotPromptTemplate）来创建复杂的提示词。这种方法可以有效地管理复杂的示例集，确保生成的提示词不会因为过长而被截断。

实例选择器（Example Selector）是一种用于选择需要在提示词中包含的示例的工具。如果你有大量的示例，实例选择器可以帮助你确定哪些示例应该被包含在提示词中。

在LangChain中，我们提供了多种实例选择器，分别实现了不同的选择策略：

1. 基于长度的选择器（LengthBasedExampleSelector）：这个选择器根据示例的长度来选择示例。这在你担心提示词长度超过模型处理窗口长度时非常有用。对于较长的输入，它会选择较少的示例，而对于较短的输入，它会选择更多的示例。

2. 最大边际相关性选择器（MaxMarginalRelevanceExampleSelector，MMR）：这个选择器根据示例与输入的相似度以及示例之间的多样性来选择示例。它通过找到与输入最相似（即嵌入向量的余弦相似度最大）的示例，然后迭代地添加示例，同时对已选择的示例进行惩罚。

3. 基于n-gram重叠的选择器（NGramOverlapExampleSelector）：这个选择器根据示例与输入的n-gram重叠度来选择和排序示例。n-gram重叠度是一个介于0.0和1.0之间的浮点数。选择器还允许设置一个阈值，重叠度低于或等于阈值的示例将被排除。

4. 基于相似度的选择器（SemanticSimilarityExampleSelector）：这个选择器根据示例与输入的相似度来选择示例。它通过找到与输入最相似（即嵌入向量的余弦相似度最大）的示例来实现。

LangChain设计实例选择器的目的是为了帮助开发者在面对大量示例时，能够有效地选择最适合当前输入的示例，以提升模型的性能和效率。

在这些示例选择器中，实例化参数的确有所不同。它们都需要传入基础的参数，如examples和example\_prompt，但根据选择器的不同，还有一些额外的参数需要设置。

对于LengthBasedExampleSelector，除了examples和example\_prompt外，还需要传递max\_length参数来设置示例的最大长度。

example\_selector = LengthBasedExampleSelector(

examples=examples,

example\_prompt=example\_prompt,

max\_length=25,

)

对于MaxMarginalRelevanceExampleSelector，除了传入示例（examples）外，还需要传入一个用于生成语义相似性测量的嵌入类（OpenAIEmbeddings()），一个用于存储嵌入和执行相似性搜索的VectorStore类（FAISS），以及需要生成的示例数量（k=2）。

example\_selector = MaxMarginalRelevanceExampleSelector.from\_examples(

examples,

OpenAIEmbeddings(),

FAISS,

k=2,

)

对于NGramOverlapExampleSelector，除了examples和example\_prompt外，还有一个threshold参数用于设定选择器的停止阈值。

example\_selector = NGramOverlapExampleSelector(

examples=examples,

example\_prompt=example\_prompt,

threshold=-1.0,

)

对于SemanticSimilarityExampleSelector，除了传入示例（examples）外，还需要传入一个用于生成语义相似性测量的嵌入类（OpenAIEmbeddings()），一个用于存储嵌入和执行相似性搜索的VectorStore类（Chroma 或者其他的VectorStore类都可以），以及需要生成的示例数量（k=1）。

example\_selector = SemanticSimilarityExampleSelector.from\_examples(

examples,

OpenAIEmbeddings(),

Chroma,

k=1

)

每种选择器都有其独特的参数设置，以满足不同的示例选择需求。参数设置并不一样，但是使用的方式跟LengthBasedExampleSelector 类是一样。实例化自身后，通过 example\_selector 参数传递给 FewShotPromptTemplate 类。

我们应该注意的是，每一种示例选择器都可以通过函数的方式实例化，或者使用类方法from\_examples来实例化。比如 MaxMarginalRelevanceExampleSelector 类我们使用类方法from\_examples来实例化，而LengthBasedExampleSelector 则是函数的方式实例化。

### 3.2.4 扩展提示词模板

LangChain提供了极其灵活的提示词模板方法和组合提示的方式，满足各种开发需求。在所有的这些方法中，基础模板和少样本提示词模板是最基础的，其他所有的方法都在此基础上进行扩展。

LangChain提供了一套默认的提示词模板，可以生成适用于各种任务的提示。然而，可能会出现默认提示词模板无法满足你的需求的情况。例如，你可能需要创建一个带有特定动态指令的提示词模板。在这种情况下，LangChain 支持你可以创建一个自定义的提示词模板。

为了个性化大语言模型应用，你可能需要将大语言模型应用与特定用户的最新信息进行组合。特征库可以很好地保持这些数据的新鲜度，而LangChain提供了一种方便的方式，可以将这些数据与大语言模型应用进行组合。做法是从提示词模板内部调用特征库，检索值，然后将这些值格式化为提示。

针对聊天模型需求，LangChain提供了不同类型的消息提示词模板。最常用的是AIMessagePromptTemplate，SystemMessagePromptTemplate和HumanMessagePromptTemplate，它们分别创建一个AI消息、系统消息和人类消息。

此外，LangChain还支持”部分”提示词模板，也就是说，传入一部分所需的值，以创建一个只期望剩余子集值的新提示词模板。LangChain以两种方式支持这一点：实例化的时候传递参数（partial\_variables={"foo": "foo"})）或者实例化为对象后调用函数（partial 方法）。

我们可以通过PipelinePrompt来组合多个提示。这在你希望重用部分提示时非常有用。

full\_template = """{introduction}

{example}

{start}"""

full\_prompt = PromptTemplate.from\_template(full\_template)

input\_prompts = [

("introduction", introduction\_prompt),

("example", example\_prompt),

("start", start\_prompt)

]

pipeline\_prompt=PipelinePromptTemplate(final\_prompt=full\_prompt,pipeline\_prompts=input\_prompts)

PipelinePromptTemplate实例化的时候，接收了这里三个实例化的模板对象列表，并且设置了最后的 final\_prompt 模板字符串。将这三个模板对象与模板字符串一起整合为一个完整的提示词对象。

另外，LangChain为了将提示存储为文件，这样可以方便地共享、存储和版本控制提示。提供了支持的文件类型有JSON 和 YAML 的 load\_prompt 方法。

{

"\_type": "few\_shot",

"input\_variables": ["adjective"],

"prefix": "Write antonyms for the following words.",

"example\_prompt": {

"\_type": "prompt",

"input\_variables": ["input", "output"],

"template": "Input: {input}\nOutput: {output}"

},

"examples": "examples.json",

"suffix": "Input: {adjective}\nOutput:"

}

例如你有一个JSON 文件，里面定义了实例化提示词模板类的参数。

prompt = load\_prompt("few\_shot\_prompt.json")

print(prompt.format(adjective="funny"))

使用这个方法，我们很便利地实现了自己的少样本提示词模板。

Write antonyms for the following words.

Input: happy

Output: sad

Input: tall

Output: short

Input: funny

Output:

最后，PromptTemplate会验证模板字符串，检查input\_variables是否与模板中定义的变量匹配。你可以通过将validate\_template设为False来禁用这种行为。这意味着，如果你确信你的模板字符串和输入变量是正确匹配的，你可以选择关闭这个验证功能，以节省一些额外的计算时间。PromptTemplate默认使用Python f-string作为其模板格式。然而，它也可以使用其他格式，如jinja2，这可以通过template\_format参数来指定。这意味着，除了Python的f-string格式，你还可以选择使用像jinja2这样的更强大、更灵活的模板引擎，以适应更复杂的模板格式需求。

## 3.3 输出解析器

在使用 GPT-4 或者类似的大型模型时，一个常见的挑战是如何将模型生成的输出转化为我们可以在代码中直接使用的格式。这里，我们会使用 LangChain 的 输出解析器（OutputParsers） 工具来解决这个问题。

虽然语言模型输出的文本信息可能非常有用，但开发的应用与真实的软件数据世界连接的时候，我们更希望得到的不仅仅是文本，而是更加结构化的数据。为了在应用程序中展示这些信息，我们需要将这些输出转换为某种常见的数据格式。我们可以编写一个函数来提取这个输出，但这并不理想。比如在提示词的模型指导加上“请输出答案为JSON格式”，模型会返回字符串形式的JSON，我们还需要通过函数将其转化为JSON对象。但是在实践中，我们常常会遇到异常问题，例如返回的字符串JSON无法被正确解析。

处理生产环境中的数据时，我们更可能会遇到千奇百怪的输入，导致模型的响应无法解析，增加额外的补丁来进行处理异常。这就使得整个处理流程变得更为复杂。

结构化数据,如数组或JSON对象, 在软件开发中起着至关重要的作用, 它提高了数据处理的效率，便利了数据的存储和检索，支持了数据分析，并且有助于提高数据质量。

还有,大语言模型目前确实存在一些问题，例如机器幻觉，这是指模型在理解或生成文本时产生的错误或误解。另一个问题是为了显得“聪明”，模型有时候会加入不必要的冗长和华丽的语句，这可能会导致模型过度详细，显得“话痨”了。比如你提示的结尾是“你的答案是：”，模型就不会“话痨”了。

在真实的开发环境中，我们不仅希望获取模型的输出结果，而且还希望能够进行后处理，比如解析输出的结构化数据。

这就是为什么在大语言模型的开发中，结构化数据，如数组或JSON对象，显得尤为重要。他们可以帮助我们更好地理解和处理模型的输出结果，比如通过解析输出的JSON对象，我们可以得到模型的预测结果，而不仅仅是一个长长的文本字符串。我们也可以根据需要对这些结果进行进一步的处理，例如提取关键信息，进行数据分析等。这样，我们不仅可以得到模型的“直接回答”，而且可以根据自己的需求进行定制化的**后处理**， 比如传递给下一个任务函数，从而更好地利用大语言模型。

### 3.3.1 输出解析器的功能

这就是输出解析器的用武之地。输出解析器包含两大功能：提示词和解析模型输出的数据。看到这里你也许很奇怪，这个输出解析器跟提示词有什么关系？

确实，从名字上看，输出解析器（OutputParser）似乎与提示词没有太大的关系，因为它听起来像是用于处理和解析模型输出的工具。然而，实际上，OutputParser在构建提示词时起着至关重要的作用。这主要表现在以下两个方面：

1. 格式化模型输出指导：输出解析器可以生成一段指导语句，这个指导语句作为提示词的一部分，可以告诉模型我们希望它以何种格式产出结果。这在处理复杂任务时非常有用，例如，我们需要模型返回一种特定格式的数据，或者我们希望模型按照某种特定的方式回答问题。

2. 解析模型输出：当模型产出结果后，输出解析器还可以将模型的输出从一段普通的文本转化为更为结构化的数据。这对于进一步处理和利用模型的输出非常有帮助。

因此，虽然输出解析器的名字听起来与提示词无关，但实际上，它在构建和使用提示词的过程中发挥着重要的作用。通过输出解析器，我们可以更好地控制模型的行为，并且更方便地处理模型的输出。

在这章的伊始，我们提到提示词通常包括三个部分：对模型的指导，用户的输入，以及用户的问题。输出解析器其实是在构造提示词的一部分：对模型的指导，下面我们称为指导词，指导模型做出反应和行为的文字。再次强调它仅仅是提示词的其中之一，并没有什么特别之处。

举例来说，假设我们要让模型解释“深度学习”这个概念。在这种情况下，我们可能会构建一个如下的提示词：“假设你是人工智能的专家，请你解释深度学习的概念是什么？例如：Q : 请你解释强化学习是什么？A: 强化学习是....” 在这里，“假设你是人工智能的专家”是我们对模型的指导，它告诉模型需要以一个AI专家的身份来回答问题。“深度学习”是用户的输入，“请你解释概念是什么？”则是我们的问题。而那个“QA”示例则是我们提供的少样本示例提示（此处是硬编码示例），它可以帮助模型理解并更好地回答我们的问题。

这就是我们通常如何构建提示词。然而，在实际的应用中，我们可能需要更加复杂和精细的提示词，例如我们可能需要指定模型的输出格式，或者需要考虑到一些特定的需求和限制。为了解决这个问题，我们可以使用OutputParser帮助我们自动化地生成模型的指导词，例如我们可以通过Pydantic的数据模型来生成模型的输出格式指导。这样，我们就可以构建出更加规范和高效的提示词，同时也能使模型的输出更加符合我们的需求。

Langchain 提供了一系列预设的输出解析器，这些输出解析器能够针对不同的数据类型给出合适的模型输出指导，并解析模型的输出。这些输出解析器包括：

1. BooleanOutputParser ：用于解析布尔值类型的输出。

2. CommaSeparatedListOutputParser ：用于解析逗号分隔的列表类型的输出。

3. DatetimeOutputParser ：用于解析日期时间类型的输出。

4. EnumOutputParser ：用于解析枚举类型的输出。

5. ListOutputParser ：用于解析列表类型的输出。

6. PydanticOutputParser ：用于解析符合 Pydantic 模型的输出。

7. StructuredOutputParser ：用于解析具有特定结构的输出。

举例来说，CommaSeparatedListOutputParser 这个解析器就提供了用于指导模型生成逗号分隔的列表类型输出的提示词。同时，这个解析器还包含一个 parse方法，能够将模型的文本输出解析为逗号分隔的列表。这意味着，当你使用 CommaSeparatedListOutputParser 时，你不需要自己编写模型输出指导的提示词，因为 Langchain 已经为你准备好了。

class CommaSeparatedListOutputParser(ListOutputParser):

    """Parse out comma separated lists."""

    def get\_format\_instructions(self) -> str:

        return (

            "Your response should be a list of comma separated values, "

            "eg: `foo, bar, baz`"

        )

    def parse(self, text: str) -> List[str]:

        """Parse the output of an LLM call."""

        return text.strip().split(", ")

CommaSeparatedListOutputParser 这个解析器的源码可以很直观看到提示词的模型指导词部分：

"Your response should be a list of comma separated values, "

            "eg: `foo, bar, baz`"

输出解析器的主要目标是提供预设的提示词来指导模型（如GPT-4），并对模型的输出进行解析，使其更符合我们的需求。一旦我们理解了这一点，我们就可以很容易地使用这些输出解析器。

例如，我们可以使用列表输出解析器 CommaSeparatedListOutputParser。我们首先实例化一个 CommaSeparatedListOutputParser 对象，然后我们可以使用 get\_format\_instructions() 方法获取预设的提示词。这个方法返回的字符串包含了如何格式化语言模型输出的指导。

对于CommaSeparatedListOutputParser，get\_format\_instructions() 方法会返回一条指导模型生成逗号分隔列表的指令。之后，我们可以将这个指令作为变量 format\_instructions 传入 PromptTemplate。这样我们的模型就知道我们希望得到的输出是一个逗号分隔的列表。

输出解析器还提供了 parse方法，它接收一个字符串（假定是模型的响应），并将其解析成某种结构。例如，对于 CommaSeparatedListOutputParser，parse 方法将接收一个字符串，并将其解析为一个逗号分隔的列表，改变了数据的格式。

总的来说，输出解析器为我们提供了一种便捷的方式，使我们能够通过预设的提示词来指导模型的输出，并且能够将模型的输出解析为我们所需要的数据。这使得我们能够更加容易地使用模型来完成实际任务。

### 3.3.2 输出解析器的使用

在Langchain框架中，我们可以利用 PromptTemplate 对象的 partial 方法或在实例化 PromptTemplate 对象时传递 partial\_variables 参数，来增加或选择已经设置好的变量。这样做可以提高代码的灵活性，使得提示词的占位符变量可以根据需要动态地增加或减少。我们使用这种方式来为提示词模板添加指导词部分。

具体操作如下：首先，我们使用 output\_parser.get\_format\_instructions() 获取预设的指导词部分 format\_instructions。然后，我们在实例化 PromptTemplate 类时，将 format\_instructions 作为 partial\_variables 的一部分传入。如此，我们就在提示词模板中追加了提示词模板字符串 format\_instructions 变量。

以下是相关的代码示例：

format\_instructions = output\_parser.get\_format\_instructions()

prompt = PromptTemplate(

template="List five {subject}.\n{format\_instructions}",

input\_variables=["subject"],

partial\_variables={"format\_instructions": format\_instructions}

)

在这段代码中，PromptTemplate 的模板字符串 template 包含两个占位符变量 {subject} 和 {format\_instructions}。我们在实例化 PromptTemplate 对象时，除了传入 input\_variables=["subject"] 参数外，还通过 partial\_variables={"format\_instructions": format\_instructions} 参数预先填充了 {format\_instructions} 变量。这样，我们就成功地为提示词模板添加了输出解析器所提供的指导词部分。

我们现在通过下面的示例代码，完成输出解析器的二大功能 — 添加指导词和解析数据格式，而且展示如何运用到链组件上。

首先，我们引入了 CommaSeparatedListOutputParser，这是一个特定的输出解析器，其主要功能是将模型输出的逗号分隔的文本解析为列表格式。

from langchain.output\_parsers import CommaSeparatedListOutputParser

from langchain.prompts import PromptTemplate

from langchain.llms import OpenAI

output\_parser = CommaSeparatedListOutputParser()

然后，我们使用 output\_parser.get\_format\_instructions() 方法来获取预设的格式化指令。这些指令会指导模型如何将其输出格式化为一个逗号分隔的列表。接下来，我们创建了一个 PromptTemplate 对象。

format\_instructions = output\_parser.get\_format\_instructions()

prompt = PromptTemplate(

template="List five {subject}.\n{format\_instructions}",

input\_variables=["subject"],

partial\_variables={"format\_instructions": format\_instructions}

)

在这个提示模板中，我们定义了一个字符串模板，其中包含两个占位符变量：subject 和 format\_instructions。subject 是我们希望模型产生的列表主题，例如 "ice cream flavors"，而 format\_instructions 是我们之前从输出解析器中获取的预设格式化指令。这里我们引入 OpenAI 模型。

我们可以打印format\_instructions的结果是“Your response should be a list of comma separated values, eg: `foo, bar, baz`”

from langchain.chains import LLMChain

chain = LLMChain(llm=OpenAI(openai\_api\_key="填入OpenAI 的密钥"),

 prompt=prompt )

我们将 subject 的值设为 "ice cream flavors"，然后调用 prompt.format(subject="ice cream flavors") 方法，这将返回一个完整的提示字符串，包含指导模型产生五种冰淇淋口味的指令。

我们导入 LLMChain 链组件，为OpenAI模型类设置密钥，将PromptTemplate 类实例化后的 prompt 对象传入LLMChain 链。

output = chain("ice cream flavors")

运行这个链得到的是一个 JSON 对象，output['text']是模型回答的字符串，然后我们调用输出解析器的 parse() 方法解析这个字符串为一个列表。由于我们的输出解析器是 CommaSeparatedListOutputParser，所以它会将模型输出的逗号分隔的文本解析为列表。

output\_parser.parse(output['text'])

所以，最后得到的结果是一个包含五种冰淇淋口味的列表：

['Vanilla',

'Chocolate',

'Strawberry',

'Mint Chocolate Chip',

'Cookies and Cream']

### 3.3.3 Pydantic JSON解析器

JSON对象，这种格式最大的特点是人和机器都看得懂。

你可以把JSON对象想象成一个大家都认识的“信息盒子”。在这个“信息盒子”里，我们可以存储各种各样的信息，比如你的名字、你的年龄、你最喜欢的食物，甚至是你所有玩具的列表等等。这些信息都被整齐地放在“信息盒子”里，每一种信息都有自己的标签，比如“名字”、“年龄”、“食物”、“玩具”。

在我们开发语言模型应用的时候，我们经常用到这个“信息盒子”。因为它可以帮我们更好地整理和使用机器人的答案。比如，机器人可能会给我们一个包含很多信息的答案，而我们可以用这个“信息盒子”来把这些信息整理得更清晰，更易于理解和使用。

所以，JSON对象就像一个非常有用的“信息盒子”，可以帮助我们更好地使用和理解语言模型的答案。

请记住，大语言模型是有“缺陷”的抽象！你需要使用一个具有足够能力的模型来生成格式良好的JSON。在OpenAI 模型家族中，DaVinci可以做到这一点，但Curie的能力已经大幅下降。Langchain 这种输出解析器可以指定一个任意的JSON结构，并向大语言模型查询，输出符合该架构的JSON。

你可以使用Pydantic来声明你的数据模型。Pydantic的BaseModel就像一个Python数据类，但它具有实际的类型检查和强制转换功能。

下面是最简单的Pydantic JSON解析器代码：

导入语言模型 OpenAI和Prompts模板。

from langchain.prompts import (  
 PromptTemplate,  
 ChatPromptTemplate,  
 HumanMessagePromptTemplate,  
)  
from langchain.llms import OpenAI  
from langchain.chat\_models import ChatOpenAI

获取格式化指令 PydanticOutputParser:

from langchain.output\_parsers import PydanticOutputParser  
from pydantic import BaseModel, Field, validator  
from typing import List

这里我们引入 OpenAI 的模型 text-davinci-003。

model\_name = "text-davinci-003"  
temperature = 0.0  
model = OpenAI(model\_name=model\_name, temperature=temperature)

定义你想要的数据格式：

# Define your desired data structure.  
class Joke(BaseModel):  
 setup: str = Field(description="question to set up a joke")  
 punchline: str = Field(description="answer to resolve the joke")  
  
 # You can add custom validation logic easily with Pydantic.  
 @validator("setup")  
 def question\_ends\_with\_question\_mark(cls, field):  
 if field[-1] != "?":  
 raise ValueError("Badly formed question!")  
 return field  
  
  
# And a query intented to prompt a language model to populate the data structure.  
joke\_query = "Tell me a joke."  
  
# Set up a parser + inject instructions into the prompt template.  
parser = PydanticOutputParser(pydantic\_object=Joke)  
  
prompt = PromptTemplate(  
 template="Answer the user query.\n{format\_instructions}\n{query}\n",  
 input\_variables=["query"],  
 partial\_variables={"format\_instructions": parser.get\_format\_instructions()},  
)  
  
\_input = prompt.format\_prompt(query=joke\_query)  
  
output = model(\_input.to\_string())  
  
parser.parse(output)

将提示传入OpenAI 模型:

\_input = prompt.format(subject="ice cream flavors")  
output = model(\_input)

调用解析器的parse方法,解析数据为列表格式.

output\_parser.parse(output)

最终的结果是:

Joke(setup='Why did the chicken cross the road?', punchline='To get to the other side!')

### 3.3.4 结构化输出解析器

OutputParsers是一组工具，其主要目标是处理和格式化模型的输出。它包含了多个部分，但对于我们实际的开发需求来说，其中最关键的部分是结构化输出解析器（StructuredOutputParser）。这个工具可以将模型原本返回的字符串形式的输出，转化为可以在代码中直接使用的数据结构。

使用结构化输出解析器时，我们首先需要定义我们所期望的输出格式。解析器将根据这个定义来生成模型的提示，从而引导模型产生我们所需的输出。例如，假设我们想要得到的输出是包含“Brand”，“Success Probability”，和“Reasoning”三个部分的 JSON 格式。我们可以将这个要求在解析器中进行定义，随后解析器就会自动为我们生成相应的 prompts。

以下是示例的代码：

# 示例：Structured Output Parser  
response\_pattern = {  
 "Brand": "The brand name is {brand\_name}.",  
 "Success Probability": "The success probability is {success\_probability}.",  
 "Reasoning": "The reasoning is {reasoning}."  
}  
  
output\_parser = StructuredOutputParser(response\_pattern)

在此例中，output\_parser将会把模型的输出按照我们定义的样式进行格式化。当我们运行这个解析器时，我们可以看到它成功地生成了我们需要的格式。最终，模型的输出将被格式化为包含“Brand”，“Success Probability”，和“Reasoning”三个部分的 JSON 格式，我们便可以在代码中直接使用它了。

无论你的应用需要什么样的输出格式，OutputParsers都能够帮助你轻松地得到。只需要定义你希望的输出样式，模型便能为你生成适合的结果，使你能更快地构建应用程序，提供更优质的用户体验。这个输出解析器特别适用于你想返回多个字段的情况。

## 3.4 本章小结

本章，我们深入探讨了大语言模型的输入、输出（模型 I/O）流程, 三个核心功能。我们首先介绍了模型的基础知识，阐述了模型如何理解和处理输入。接着，我们讲解了prompt提示的概念和使用，揭示了其在引导模型生成期望输出中的重要性。最后，我们详细解读了输出解析器的角色，明确了其在转化模型输出为结构化数据中的关键作用。

总的来说，这一章我们走过了从模型输入到输出的全过程，对大语言模型的运作机制有了更深入的理解。这是开发应用的最基础，也是打好基本功的关键。只有深入理解这些基础知识，我们才能在大语言模型的应用开发中乘风破浪，开创新的可能。