# 2.1.1 通用大型语言模型（LLMs）

大型语言模型（LLMs）是 LangChain 的核心组件。LangChain 不提供自己的 LLMs，而是提供了与许多不同的 LLMs 进行交互的标准接口。

我们开始逐个解释这些元素，通过编写代码来解析它们。

有许多 LLM 提供商（OpenAI、Cohere、HuggingFace 等）—— langchain.llms 类旨在为所有提供商提供标准接口。

使用 OpenAI LLM 包装器，这些方法对于所有 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

然后设置好密钥：

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

### 使用 LLM 的最简单方法: 字符串 in -> 字符串 out

# Run basic query with OpenAI wrapper  
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。

接下来我们转入第二种专用聊天模型。

# 2.1.2 专用聊天模型

当前最大的应用场景便是 “聊天 Chat” ，就像 OpenAI 的热门应用 ChatGPT 一样。为了紧跟用户需求，LangChain 推出了专门专用聊天模型 Chat Models，以便我们能与各种聊天模型进行无缝交互。

在上一节我们提到专用聊天模型的不同之处在于：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。

这一节，我们通过代码，看看输入聊天消息列表的专用聊天模型, 究竟与通用 LLMs 有什么区别？

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

然后设置好密钥：

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

### 使用 Chat Models 的最简单方法

为了通过 LangChain 与聊天模型交互，我们将导入一个由三个部分组成的模式：一个 AI 消息 AIMessage，一个人类消息 HumanMessage 和一个系统消息 SystemMessage。然后，我们将导入 ChatOpenAI。

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

SystemMessage 是您在使用模型时用于配置系统的消息，而 HumanMessage 是用户消息。

我们现在将系统消息和人类消息组合成一个聊天信息列表，然后输入到 Chat Models 聊天模型。

这里我使用的是模型名称是：gpt-3.5-turbo。如果你有 gpt4，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')

运行后获得 AIMessage 的结果是：

当然！以下是一些包含"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！如果你有其他要求或者想要更多的建议，请随时告诉我。

本节中，我们着重研究了如何运用专用聊天模型。在 LangChain 的持续迭代过程中，我们目睹了模型和 LangChain 都在不断进化。

# 2.1.3 文本嵌入模型

想要造航母，仅仅一个大语言模型是不够的。

假设我们想要创造一个《红楼梦》聊天机器人应用，并且询问关于这本书的知识。一个大语言模型就够了吗？

还有我们上传专业的论文，长达几十页的临床医学实验报告，或者使用搜索或外部数据的东西，跟真实的数据连接起来，大语言模型能处理吗？

虽然最新的模型一直在突破极限，但是仍然不能满足我们想要大模型阅读大型文档、书籍的需求。

面对这些挑战，我们可能需要总结或寻找其他方法来应对。因为大型语言模型可以传递的标记数量是有限的，大多数模型最多可以传递 1024 到 16K 个标记，尽管一些新的模型可以处理更多的标记。这就意味着，我们需要思考如何有效地利用这些有限的标记，以达到我们的应用需求。

此外，理解模型的类型和其各自的特点也同样重要。模型在高层次上可以分为两种不同类型：语言模型和文本嵌入模型。嵌入模型将文本转换为向量标记，让我们可以将文本标记为向量特征，显示在向量空间中。了解这两种模型的特性和差异，能帮助我们更好地选择和使用模型，进一步优化我们的应用。

## 嵌入模型将文本转换为向量标记的例子

在自然语言处理（NLP）中，嵌入模型是将文本转换为向量的重要工具，这样我们可以在向量空间中表示文本，以便进行后续的机器学习或深度学习任务。举个例子，我们可以用 Word2Vec，一个常见的嵌入模型，来演示这个过程。

假设我们有以下的语料库：

1. I love learning.  
2. I like reading books.  
3. Books are great for learning.

首先，我们需要建立一个词汇库，也就是列出所有在语料库中出现过的独立单词：

"I", "love", "learning", "like", "reading", "books", "are", "great", "for"

接下来，我们用 Word2Vec 来训练这个语料库。Word2Vec 是一个预测模型：它试图从上下文预测目标单词，或者从目标单词预测上下文。训练过程中，Word2Vec 会学习到一个单词和它的上下文单词之间的关系，并将这些信息编码到向量中。

训练完成后，每个单词都会被赋予一个向量。例如，“books”的向量可能是：

[0.1, -0.2, 0.3, 0.5, -0.1]

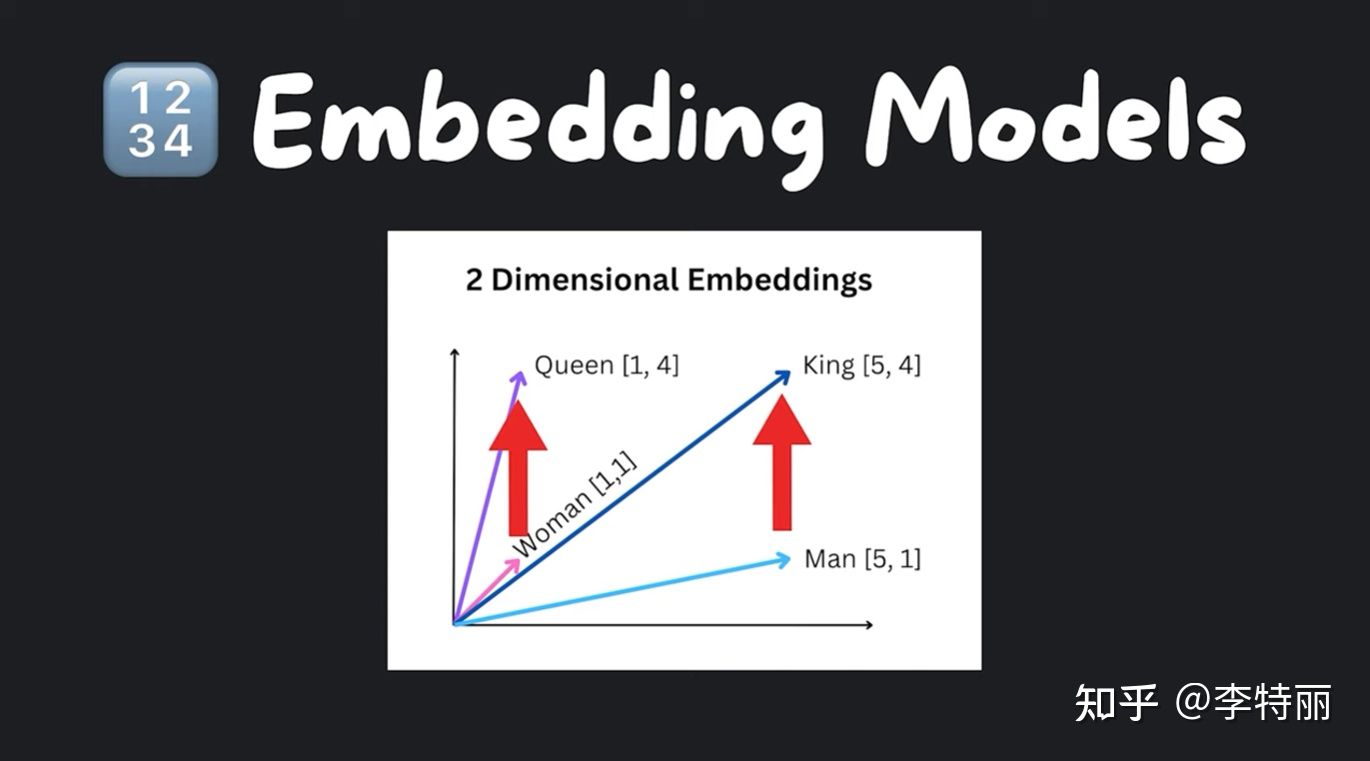
而“learning”的向量可能是：

[-0.1, 0.3, -0.2, 0.1, 0.4]

这些向量现在就代表了对应的单词在向量空间中的位置，且具有一些有趣的属性。比如，语义上相似的单词在向量空间中的位置会比较接近，我们可以通过计算两个向量之间的余弦相似度来衡量这种接近程度。

以上就是嵌入模型将文本转换为向量标记的一个基本例子。

## 嵌入模型的原理



在上面这个图像中，我们可以看到在一个二维空间中，“man”是“king”，“woman”是“queen”，它们代表不同的事物，但我们可以看到一种模式。这个模式就是可以在向量空间中寻找最相似的文本片段，实现语义搜索。

例如，OpenAI 的文本嵌入模型可以精确地嵌入大段文本，具体而言，8100 个标记，根据它们的词对标记比例 0.75，大约可以处理 6143 个单词。它输出 1536 维的向量。

我们可以使用 LangChain 与多个嵌入提供者进行接口交互，例如 OpenAI 和 Cohere 的 API，但我们也可以通过使用 Hugging Faces 的开源嵌入在本地运行，以达到 免费和数据隐私 的目的。

现在，您可以使用仅 4 行代码在自己的计算机上创建自己的嵌入。但是，维度数量可能会有所不同，嵌入的质量可能会较低，这可能会导致检索不太准确。

## 快速入门

### 安装和设置密钥

我们需要安装 Langchain Python 包：

pip install openai langchain

### 使用 Text Embedding Models 的最简单方法

导入 OpenAIEmbeddings：

from langchain.embeddings import OpenAIEmbeddings  
  
embeddings = OpenAIEmbeddings(openai\_api\_key = "YOUR\_OPENAI\_API\_TOKEN")  
result = embeddings.embed\_query("This is an apple.")

打印 result ，我们就可以看到 “This is an apple.” 这句话的向量空间表达数组。

[0.012008527293801308,  
 -0.001835253438912332,  
 0.00145026296377182,  
 -0.0030227703973650932,  
 -0.005661344155669212,  
 0.005086636636406183,...]

打印出的数字并不能直观看到这些词在向量空间中的位置，但如果你对向量可视化感兴趣，可以访问 OpenAI 的官方网站，那里有许多关于词嵌入（embedding）可视化的实践场所（playground）供你探索和学习。 # 模型输入输出

LangChain 打通了大语言模型应用开发的“最后一公里”。

任何大语言模型应用的核心元素是 —— 模型的输入 Input 和输出 Output (简称模型输入输出 (Model I/O), 下面都用 I 表示模型的输入 Input，O 表示模型的输出 Output）。

最近一年左右，通过强化学习等方法对模型进行微调（Fine-tuning），以及在提示（Prompt）中使用各种方法，以从这些模型中获得最佳效果。一旦我们能够做到这一点，从这些大语言模型中我们可以得到一些非常好的结果。问题是，它们仍然无法以接口方式访问现有数据和 API, 无法与真实的世界连接起来就很难应用, 尤其是复杂的应用。

如果我们想要使用这些模型构建一个应用程序，我们需要某种方式在大语言模型和传统软件之间进行接口交互。同时，越来越多的大语言模型涌现出来，模型输入输出变得异常重要。LangChain 为我们提供了与任何大语言模型进行交互的基本组件。

LangChain 不仅可以方便地与最新、最强大的模型如 GPT-4 进行交互，还可以与本地私有化部署的语言模型，或是在 HuggingFace 上找到的开源模型进行交互。

只需几行代码，就可以实现与这些模型进行对话。如何使用 Langchain 的基础组件模型输入输出来访问大语言模型？来看看下面这张示意图：

|  |
| --- |
| model\_io 图示 |

model\_io 图示

Model I/O 组件提供了三个核心功能：

* [语言模型 models](/docs/modules/model_io/models/): 通过接口调用语言模型，即 Model。
* [提示 prompts](/docs/modules/model_io/prompts/): 将模型输入模板化、动态选择和管理，即 Model I。
* [输出解析器 output\_parsers](/docs/modules/model_io/output_parsers/): 从模型输出中提取信息 ，即 Model O。

我们将深入了解如何使用 Langchain 的模型输入输出组件来访问大语言模型。我们先认识三大模型类别。

## 语言模型的分类

LangChain 模型输入输出组件的目前有三大模型类型。分别是通用大语言模型 LLMs， 专用聊天模型 Chat Models 和文本嵌入模型 Text Embedding Models。

在 LangChain 中使用的不同类型的完成大语言模型输入和输出。在这一章中，我们将对模型类型进行分类和认识。

### 通用大语言模型

首先介绍的是大语言模型（ LLMs ）。这些模型以文本字符串作为输入，并返回文本字符串作为输出。

### 专用聊天模型

第二种是专用聊天模型。这些模型通常由语言模型支持，但其 API 更加结构化。具体来说，这些模型以聊天消息列表作为输入，并返回聊天消息。

### 文本嵌入模型

文本嵌入模型是以文本作为输入，并返回浮点数列表，也就是向量维度表示的列表。

这三种类型中，通用大语言模型和专用聊天模型很容易被人误解没有必要。实际上，通用大语言模型和专用聊天模型有微妙但重要的区别。

### 通用大语言模型和专用聊天模型的区别

LangChain 中的 通用大语言模型是指纯文本补全模型， 也就是 Text To Text。 它们包装的 API 接受字符串提示作为输入，并输出字符串补全部分。OpenAI 的 GPT-3 就是 LLM 的实现典型。

专用聊天模型通常由 LLMs 支持，但专门用于进行对话。

专用聊天模型的不同之处在于：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。

通常，这些消息带有发言者身份（LangChain 目前支持的消息类型有“AIMessage”，“HumanMessage”，“SystemMessage”）。它们返回一个（“AI”）聊天消息作为输出。GPT-4 和 Anthropic 的 Claude 都是作为专用聊天模型实现的。

### 通用大语言模型和专用聊天模型的的学习路径

在 Langchain 的发展迭代过程中，每个模块都精细地划分出通用大语言模型（LLM）和专用聊天模型两种类型，紧跟 OpenAI 的前沿技术潮流，以更好地适配新的专用聊天模型。

这一区分已经形成了一种趋势，也为我们学习 Langchain 提供了线索。每当我们看到一个针对通用模型的 Langchain 类或方法，我们便能预期到对应的聊天模型类或方法也必然存在。

然而，Langchain 这样的做法有时候会让人感到困惑，因为它每次都会单独声明通用大语言模型和专用聊天模型的不同代码。但是，如果你从模型的输入输出开始理解，下面的学习过程会变得轻松很多。例如，如果你刚刚学到通用模型，你就预期会有一个聊天模型，你就能更好地做出预测。

## 模型输出的通用接口

无论是通用大模型还是专用聊天模型，Langchain 都暴露了 predict 方法，而且包装进了对话的链或者代理上，使用非常方便。

# 2.2.1 提示模板

实际上, 提示模板是一个 “未完成” 的输入示例。Langchain 提供了我们通过告诉它要替换占位符的值来创建格式化之后的提示。Langchain 内置的提示模板对象 (PromptTemplate), 使我们能够将用户输入插入到提示文本的占位符中，并且可以增加额外的知识, 额外的上下文, 将用户输入进行格式化，最终提供给语言模型高质量的提示, 从而获得更好的输出结果。

### 提示模板的定义

在 Langchain 中，提示模板是一种可复制生成提示的方式。它包含一个文本字符串（即 “模板”），该模板可以接收用户的一系列参数，从而生成一个提示。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，而是使用这个模板，根据用户输入的不同品类，生成对应的提示。

### 模板的三个组成部分

提示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。这三部分中，语言模型的说明和示例是固定的常量，而问题或任务是可以由用户改变的变量。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。

提示模板在 Langchain 中得到了广泛的应用，但也在许多其他的大语言模型提示系统中使用。我们常常能看到类似的情况。Langchain 的提示模板在其他模型中是通用的。

提示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。” 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模板，轻松地为任意一篇新闻生成摘要。

### 一个最简单的示例

这是最基本的示例, 这个提示模板没有包含示例, 但它使我们能够动态更改带有用户输入的提示。

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

当我们传入 concept="NLP" 时, PromptTemplate 方法, 将 “NLP” 注入到模板中

我们得到的结果是:

'\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'

这个代码示例将用户输入变量 input\_variables 和开发者定义的 template 传参给 PromptTemplate 对象, 然后调用 format 方法，格式化为语言模型可用的字符串提示。

### 构造提示模板的步骤

构造模板包含取出模板和格式化两个步骤。from\_template 即取出模板，format 即格式化用户输入。Langchain, 内置了很多模板，比如聊天模型的提示模板，角色的提示模板。格式化可以是普通字符串用作通用语言模型输入，也可以是用作聊天模型输入的 Message 对象。

上面这个代码还可以简化为：

# 导入PromptTemplate和template定义同上  
  
prompt = PromptTemplate.from\_template(template).format(concept="NLP")  
prompt

这段代码可以用自然语言描述为：取出内置模式给 template 使用，传参用户的输入后格式化输出完整提示。无论是哪一种提示模板，都是基于 PromptTemplate 对象，我们基于这个对象构造自己的提示模板。

内置提示模板，是 Langchain 简化开发的方式。即使你不指定 input\_variables , Langchain 会自行推断，只要你告诉了它取出哪个模式。

## 专用聊天模型的提示模板

专用聊天模型的提示模板的不同之处，输入的是消息列表, 支持输出 Message 对象。Langchain 的优势在于提供了聊天提示模板 （ChatPromptTemplate）和角色消息提示模板。角色消息提示模板包含 AIMessagePromptTemplate, SystemMessagePromptTemplate 和 HumanMessagePromptTemplate 三种角色消息提示模板。

无论看起来这些多么复杂，他们都遵循构造模板的步骤，取模式和格式化。聊天模型的提示模板取是消息提示模式，格式化多了消息列表的选择。

我们将上一个示例代码改造为聊天提示模板，下面是代码示例：

先导入消息提示模板：

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，通过 ChatPromptTemplate 整合为消息列表提示： 先使用 from\_template 取出消息模式，传入定义的 template 模板字符串，生成人类和系统两种模板。

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)

然后转化为聊天提示模板类型：

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

最后指定输出为消息类型的格式, 字符串的输出则是这样：

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

打印输出的是：

'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'

指定格式化为消息对象类型则是这样：

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

打印输出的是：

[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 强化学习 in a couple of lines', additional\_kwargs={}, example=False)]

尽管上述示例并没有展示如何加入示例，但请不要忧虑，接下来我们即将进入下一章节：少样本提示模板。这个章节将全面展示如何利用示例来优化我们的模板。

事实上，所有的提示模板都基于示例选择这一概念。为了从大语言模型中获取更加完整和准确的结果，少量示例学习（Few Shot Learning）就显得尤为重要。下一章少样本提示模板，就是在构造提示时，我们提供一些示例来指导模型的输出。这些示例可以帮助模型更好地理解任务要求，从而生成更加贴合用户预期的答案。这也正是 FewShotPromptTemplate 的价值所在：通过使用少量示例学习，我们能够有效提升模型的输出质量，让模型给出的回答更加准确和具有针对性。

# 2.2.2 少样本提示模板

少样本提示模板 (FewShotPromptTemplate)已经被证明是很有用的。通常用户只要输入关键信息，就可以获得预期满意的回答。这些示例可以硬编码，但如果动态选择，会更强大。Langchain 提供了少样本提示模板接受用户输入，然后返回包含示例列表的提示。

在这篇文章中，我们将详细探讨如何通过调整和增强提示，特别是加入一些具体的例子，让模型通过学习这些例子来提升自身的输出质量。你会惊讶地发现，这种微小的调整会对模型输出的结果产生巨大的影响。

首先，我们需要明白一点，这与我们之前所做的工作并无太大区别，其核心仍然是制作一个优秀的提示模板。现在，让我们通过一个具体的例子来理解这个过程。

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 “happy” 对应的反义词是 “sad”，“tall” 的反义词是 “short”，以此类推。然后，我们通过 LangChain 来设置我们的提示模板。

首先，我们会设置一个前缀，这样可以方便我们逐步构建提示的各个部分，以确保它们能完整地结合在一起。然后，我们会为模型设置一些标准的示例，以帮助模型理解任务需求。接下来，我们会设置一个 Few Shot 提示模板，然后在这里传入我们的示例。我们还会为此设置一个前缀和一个后缀，这样可以帮助模型更好地理解任务。

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 “big”，模型就应该返回 “small”。这就是我们期望看到的反义词。

实际上，我们并没有看到整个提示，因为在大多数情况下，我们并不希望向用户展示完整的提示。因此，这就是使用提示模板制作提示的一种简单方法。

在这个过程中，标准提示模板和 Few Shot 提示模板都发挥了重要的作用。要注意的是，Few Shot 提示模板的关键在于示例，因此你可以在这里使用一系列示例。这个示例只需要展示输入是什么，以及你期望的输出是什么。

我们通过下面代码，创建一个包含几个示例的少样本提示模板, 来解释 FewShotPromptTemplate 的作用。

pip install openai langchain

导入 PromptTemplate, FewShotPromptTemplate 对象

from langchain.prompts.few\_shot import FewShotPromptTemplate  
from langchain.prompts.prompt import PromptTemplate

创建一个包含几个示例的列表

examples = [  
 {"word": "happy", "antonym": "sad"},  
 {"word": "tall", "antonym": "short"},  
]

实例化 PromptTemplate 对象

example\_prompt\_format = """Word: {word}  
Antonym: {antonym}  
"""  
example\_prompt = PromptTemplate(input\_variables=["word","antonym"], template=example\_prompt\_format)  
  
print(example\_prompt.format(\*\*examples[0]))

实例化 FewShotPromptTemplate 对象

我们创建 FewShotPromptTemplate 对象，传入示例、示例格式化器、前缀、命令和后缀，这些都在指导 LLM 的输出。

此外，我们还可以提供输入变量 examples, example\_prompt 和分隔符 example\_separator = “”，用于将示例与前缀 prefix 和后缀 suffix 分开。

few\_shot\_prompt = FewShotPromptTemplate(  
 examples=examples,  
 example\_prompt=example\_prompt, # 上一步实例化PromptTemplate对象  
 prefix="Give the antonym of every input\n",  
 suffix="Word: {input}\nAntonym: ",  
 input\_variables=["input"],  
 example\_separator="\n",  
)

我们可以生成一个提示，它看起来像这样。

print(few\_shot\_prompt.format(input="big"))  
  
Give the antonym of every input  
  
Word: happy  
Antonym: sad  
  
Word: tall  
Antonym: short  
  
Word: big  
Antonym:

FewShotPromptTemplate 是一种非常有用的范例，可以控制 LLMs 的输出并引导其响应。

在实际的应用开发中, 我们的示例相当复杂, 现实是我们在提示中还要加入大量的历史聊天信息, 可是大型语言模型可以传递的标记数量是有限的，大多数模型最多可以传递 1024 到 16000 个标记。

Langchain 提供了控制提示长度的方法, 有兴趣的读者可以在官方文档中查看 LengthBasedExampleSelector 和 SemanticSimilarityExampleSelector 两种高阶的示例选择器, Langchain 还支持自定义示例选择器, 自己动手做一个示例选择器也是个不错的选择。 # 2.2.3 扩展提示模板

LangChain 提供了极其灵活的提示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模板。在这种情况下，LangChain 支持你可以创建一个自定义的提示模板。

为了个性化 LLM 应用，你可能需要将 LLM 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 LLM 进行组合。做法是从提示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模板。LangChain 以两种方式支持这一点：部分格式化字符串值和部分格式化返回字符串值的函数。这两种不同的方式支持不同的用例。

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。

通常，将提示存储为文件，而不是 Python 代码，更为方便。这样可以方便地共享、存储和版本控制提示。LangChain 中支持的文件类型有 JSON 和 YAML, 框架理论上 LangChain 想要支持一切的文件类型。

# 2.2.4 示例提示选择器

示例提示对模型输出结果产生的影响是非常显著的。这点在实际操作中的反馈是明确无误的。但问题在于，我们可能有大量这样的示例，我们不可能全部输入给模型。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。

以金融财报的摘要为例，如果举例的摘要过长，成本会变得很昂贵。大量、冗长的例子可能会占用模型可处理的 Token 数量，这使得模型无法充分理解和处理用户的真正输入，从而影响输出质量。

幸运的是，LangChain 的示例提示选择器（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：

from langchain.prompts import PromptTemplate  
from langchain.prompts import FewShotPromptTemplate  
from langchain.prompts.example\_selector import LengthBasedExampleSelector  
  
examples = [  
 {"word": "happy", "antonym": "sad"},  
 {"word": "tall", "antonym": "short"},  
 {"word": "energetic", "antonym": "lethargic"},  
 {"word": "sunny", "antonym": "gloomy"},  
 {"word": "windy", "antonym": "calm"},  
]  
  
example\_prompt = PromptTemplate(  
 input\_variables=["word", "antonym"],  
 template="Word: {word}\nAntonym: {antonym}",  
)  
  
example\_selector = LengthBasedExampleSelector(  
 examples=examples,   
 example\_prompt=example\_prompt,   
 max\_length=25,  
)  
  
dynamic\_prompt = FewShotPromptTemplate(  
 example\_selector=example\_selector,  
 example\_prompt=example\_prompt,  
 prefix="Give the antonym of every input",  
 suffix="Word: {adjective}\nAntonym:",   
 input\_variables=["adjective"],  
)  
  
print(dynamic\_prompt.format(adjective="big"))

todo: 返回的结果

# 提示

一切都始于 Prompts 。

事实上, Langchain 的一切都是围绕 Prompts 构建的, Prompts 对于你使用 Langchain 所做的一切至关重要。

让我们稍微谈谈 Prompts 是什么以及在 Langchain 中的 Prompts 模板的概念。

我们跟聊天机器人沟通的唯一方式是输入 Prompts。

什么是 Prompts ？Prompts 指的是模型的输入,　我们发给大语言模型的内容。

在 Langchain 中, Prompts 基本上是让模型在某些输入文本的 **条件限制** 下生成文本的过程。这些模型通常只是继续预测下一个标记, 通常是下一个单词, 然后通过这种方式组装文本。

Prompts 本身对输出结果有很大的影响, 以至于这导致了一个全新的领域, 人们现在称之为 Prompts 工程。

早期的 Prompts 非常简单, 你可能会问一些简单的问题, 比如乾隆是什么时候皇帝? 希望大语言模型在其权重中学到足够的知识来回答这个问题, 并生成一个输出结果。

随着模型的不断改进, Prompts 从这些简单的形式转变为更加复杂的形式, 现在我们看到人们给出了包含整个上下文和各种信息的 Prompts 。

从 GPT 论文中, 我们可以看到其中有不同类型的 Prompts , 论文中包含了许多不同的 Prompts 。

你还可以看到这个示例中我们有所谓的样本（few shot）模板, 即我们不仅仅告诉 Prompts 一些内容, 还会给它一些示例, 然后让它生成一个新的答案或示例。

在 Langchain 中, 所有这些都由 Prompts 模板来处理, 因此了解 Prompts 的不同部分非常重要, 因为你将在 Prompts 中构建不同的部分。

在 Langchain 中, 你将构建一个 Prompts , 并在其中注入不同的内容。例如, 这是一个简单的 Prompts , 以一个上下文开始, 告诉模型你是一个起名字助手, 可以帮助用户根据所提供的业务描述创建业务名称, 然后给出一些示例, 以特殊的方式给出这些示例, 这样模型就可以学习生成与示例相似的内容。然后, 我们可以传递问题或任务, 并为输出结果进行预处理。

不仅 Langchain 中经常使用, 也在许多其他 Prompts 系统中使用。这些 Prompts 系统, 基本上是在占位符注入一些用户输入的信息, 这是用于回复的一些要点, 再次注入一些信息。把 Prompts 发给语言模型, 语言模型给我们一个回答。

这样可以针对许多不同的任务进行操作, 可以用于分类, 也可以用于其他各种不同的任务。

在实际的应用开发中, 开发者一定是想要动态插入提示, 而不是硬编码的。还想要有多个灵活的值, 而不只有一句提示。LangChain 提供了 PromptTemplate 基本类对象来构建使用多个值的提示。

LangChain 还提供了多个类和函数, 构建了很多不同实例的 PromptTemplate, 让开发者使用提示更加容易, 满足这些实际开发的多样需求。

这一章的重要概念包括提示模板、输出解析器、示例选择器和控制提示长度。

我们首先来理解提示模板 PromptTemplate。

# 2.2 提示

新的大语言模型的编程方式是通过提示（prompt）进行的。“提示” 是 Langchain 与大模型交互的最重要组件。Langchain 中的提示输入通常由多个部分构成。LangChain 提供了类和函数，以方便构造和操作提示。

## 提示的定义

什么是提示？提示指的是模型的输入,　我们发给大语言模型的内容。

在 Langchain 中, 提示是让模型在输入的提示文本 **条件限制** 下生成文本的过程。这些模型通常是继续预测下一个标记, 通常是下一个单词, 然后通过这种方式组装文本输出。

## 你的第一个提示

在编程世界里，我们常说 “Hello World” 是每个程序员的第一步。现在，对于使用大语言模型编程的我们，“你好”, 一个 emoj 表情或者任何一句话，也是我们的第一步。因为无论我们的任务多么复杂，一切都始于一个 prompt。

还记得你第一次使用一个聊天机器人时，你敲打键盘的第一句话是什么吗？是不是 “你好” ？

|  |
| --- |
| 图 2-1 |

图 2-1

事实上，这个看似简单的“你好”，在编程世界里就像完成了“Hello World”的编程那样具有重要意义。这是因为在这个场景中，“你好”是一个提示。

你的第一个提示代表你已进入 AI 编程的美丽新世界。

提示对大语言模型的输出结果有很大的影响, 以至于这导致了一个全新的领域, 人们现在称之为提示工程。

## 提示的演化

早期的提示非常简单, 你可能会问一些简单的问题, 比如乾隆是什么时候皇帝? 希望大语言模型在它之前数据集学到足够的知识来回答这个问题, 并生成一个输出结果。

随着模型的不断改进, 提示从这些简单的形式转变为更加复杂的形式, 现在我们看到人们给出了包含整个上下文和各种信息的提示。还有少数示例提示模板（Few-shot）, 即我们不仅仅告诉提示一些内容, 还会给它一些示例, 然后让它生成一个新的答案或示例。

从 GPT 论文中, 我们可以看到其中有不同类型的提示, 论文中包含了许多不同的提示。

随着技术的不断进步和人工智能领域的深入研究，如今的提示已经越来越复杂和工程化。尤其是在参数化模型输入和样本示例这两个方面，都做出了巨大的努力来提高提示的质量，以实现更好地控制大语言模型的目标。

## 模型对提示的影响

我们现在进入了模型开发的白热阶段，也就是“千模大战”, 未来可能会有更多模型提供商加入, 那其他的模型对提示会有什么影响？

LangChain 构造和操作的提示适用于其他模型，提示对于其他模型的影响，取决于模型的规模和质量。如果你想使用免费的 Hugging Face 模型中的一个，我们可以用 LangChain 做相同的事情。

以下是使用 Google T5-Flan-XL 模型的代码：

from langchain.llms import HuggingFaceHub

在这里，我只是设置了一个 Hugging Face 版本的 Google T5-Flan-XL 模型。这不是最大的模型，最大的模型是 XXL，但是这个模型的规模肯定没有 OpenAI 的 text-davinci-003 模型大，不过你会看到我们仍然可以得到一些连贯的回应。

llm\_hf = HuggingFaceHub(  
 repo\_id="google/flan-t5-xl",  
 model\_kwargs={"temperature":0.9 }  
)

你会注意到，这个回应不同，如果我们再次运行它，我们仍然得到相同的东西，但每个模型都给出了不同的回应，但基本上做的事情都是一样的。

text = "Why did the chicken cross the road?"  
  
print(llm\_hf(text))

It was hungry.

## Langchain 里的提示

Langchain 对于你开发应用的提示功能非常重要。通过它，你可以更有效地指导你的应用来实现预期的功能。而且，一旦模板构建好，你只需要提供关键的信息。让我们通过一个具体的实例来说明这一点。

假设你正在开发一个专门为电影爱好者设计的聊天机器人，该机器人的目标是提供详细的电影信息。此时，Langchain 的提示功能就可以派上用场。你可以建立一个模板，像这样：

* 提示：“” “请告诉我《{电影名}》的导演是谁？” “”
* 提示：“” “《{电影名}》的主要演员有谁？” “”
* 提示：“” “《{电影名}》的剧情简介是什么？” “”
* 提示：“” “《{电影名}》的评分是多少？” “”

在模板建立好之后，用户只输入电影名，比如《阿凡达》或者《泰坦尼克号》。此时，模型会接收到 Langchain 提供的结构化提示输入，如“请告诉我《阿凡达》的导演是谁？”或“《泰坦尼克号》的主要演员有谁？”等。

然后，模型会根据这些提示，生成相应的输出。由于 Langchain 的提示非常具体和结构化，模型生成的答案质量通常较高，符合用户的预期。你可以将更多的精力集中在用户和业务上，而不必担心如何指导模型生成满足用户需求的答案。因为一旦模板建立好，Langchain 的提示功能就会帮你做这件事。

在 Langchain 中, 所有这些都由提示模板来处理, 因此了解提示的组成部分非常重要, 对于你使用 Langchain 所做的一切至关重要。

Langchain 里的提示模板主要 2 个功能是：

### 1. 参数化模型输入

参数化模型输入是通过将模型输入转化为可动态的参数形式，从而实现模型输入的灵活调整。具体来说，它允许我们根据特定的需求和环境条件，来动态地调整模型的输入。这样一来，我们可以根据特定的任务需求，通过调整输入参数，使模型生成满足我们需求的输出。例如，我们正在开发一个聊天机器人，并希望给它一个具有个性的名字，比如 “小智” 或 “问答达人” 等。我们可以将机器人的名字作为输入参数，通过 Langchain 里的提示模板构建一个动态的提示。“小智” 或 “问答达人” 就是我们输入的参数，Langchain 里的提示模板接受这个参数，构造成一个完整的提示输入给大语言模型,。

这是最基本的示例，但它使我们能够动态更改带有用户输入的提示。

这个代码中, {botname} 是要被替换的占位符。

# Import prompt and define PromptTemplate  
from langchain import OpenAI  
from langchain import PromptTemplate  
  
template = """  
从现在开始，你的名字是：{botname}。每次打招呼的时候，请回应：“你好，我是{botname}聊天机器人，有什么可以帮到你？”。  
"""  
  
prompt = PromptTemplate(  
 input\_variables=["botname"],  
 template=template,  
)  
# Run LLM with PromptTemplate  
llm = OpenAI(openai\_api\_key="YOUR\_OPENAI\_API\_KEY")  
llm(prompt.format(botname=""))

当我们传入 botname="小智" 时, PromptTemplate 方法, 将 “小智” 注入到提示中，这个聊天机器人就以“小智”的身份回应你的问题。

### 2. 动态选择包含在提示中的示例

“示例提示模板”（又称为 “少数示例学习” 或 “Few-shot learning”）已经成为当前提升大语言模型性能的一个重要策略。让我们通过一个具体的例子来了解一下这种方法的工作原理。

假设我们正在开发一个聊天机器人，该机器人需要回答用户关于天气的各种问题。这时，我们可以使用示例提示模板或者少数示例学习的方法来训练和优化我们的机器人。

首先，我们可以为聊天机器人提供一些关于如何回答天气相关问题的示例。这些示例包括如下内容：

User：“今天北京的天气怎么样？” Chatbot：“我不确定，让我查一下。今天北京的天气预报是晴朗。”

User：“明天上海会下雨吗？” Chatbot：“让我为你查看明天上海的天气预报。不，明天上海不会下雨。”

这些示例将帮助机器人理解用户可能会提出的天气相关问题，以及如何回答这些问题。然后，当用户实际提问时，机器人 (Chatbot) 就可以参考这些已经学习过的示例，来生成合适的回答。这就是少数示例学习的工作原理。

通过使用这种方法，我们可以大大提高聊天机器人的性能。因为它允许我们以一种非常直接而灵活的方式，为机器人提供具体的指导。这使得我们的机器人不仅可以更准确地回答用户的问题，而且还可以更好地适应各种不同的环境和情境。因此，示例提示模板已经成为当前提升大语言模型性能的一个重要工具。

在实际开发中，我们通过将这两种方法结合起来，我们可以更好地控制大语言模型的表现。这是因为我们不仅可以根据需求调整模型的输入，还可以选择最适合的训练数据作为输入。这就提供了一种强大的工具，可以帮助我们实现更精细、更有目标性的模型控制。

# 2.3.1 输出列表格式

数组或者列表是程序世界最基本的数据格式, 这种格式在大语言模型开发中, 也可以发挥不少作用.

你可以想象大语言模型就像一个非常聪明的机器人，它可以回答很多问题。但是，有时候这个机器人会说一些很复杂或者不正确的话，就像它在猜你想要知道什么，或者想要显得自己很聪明一样。

我们可以让这个机器人把它的回答写在一张“清单”上，就像你写购物清单一样。这样，我们就可以很容易地看到它的每个答案，而不需要去理解一大堆复杂的话。这个“清单”还可以帮助我们更好地使用这个机器人的答案，比如我们可以把这个名单给到其他机器人让它们帮忙处理。

使用“清单”这样的方式，可以让我们更好地使用这个聪明的机器人，让它变得更有用，更容易理解。

在软件世界中, 最常见的“清单”就是列表格式。

我们通过快速入门的代码, 来看看 Langchain 是如何将模型的回答输出为“清单”式的列表格式。

下面是最简单的列表输出解析器代码：

导入库, 实例化列表输出解析器对象（CommaSeparatedListOutputParser）, 预期我们获得列表的结果:

from langchain.output\_parsers import CommaSeparatedListOutputParser  
from langchain.prompts import PromptTemplate, ChatPromptTemplate, HumanMessagePromptTemplate  
from langchain.llms import OpenAI  
from langchain.chat\_models import ChatOpenAI  
  
output\_parser = CommaSeparatedListOutputParser()

获取格式化指令:

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}  
)

这里我们引入 OpenAI 模型:

model = OpenAI(temperature=0)

实例化一个提示模板:

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

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:

['Vanilla',  
 'Chocolate',  
 'Strawberry',  
 'Mint Chocolate Chip',  
 'Cookies and Cream']

从快速入门代码, 我们使用了输出解析器的 2 大方法: 格式化 output\_parser.get\_format\_instructions() 和 解析 output\_parser.parse()

接下来, 我们进入到语言模型应用开发中，最常用的一种格式： JSON 对象。

# 2.3.2 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!')

细心的读者会注意到, 我们这次仍旧是使用了输出解析器的 2 大方法: 格式化 output\_parser.get\_format\_instructions() 和 解析 output\_parser.parse()。

掌握这两种方法，我们就能掌握输出解析器的使用方法。 # 2.3.4 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。 # 2.3 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。比如在提示结尾加上“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。

这就是输出解析器的用武之地。

输出解析器是一组工具，它们的主要功能是处理和格式化模型的输出。这个工具组包括了几个部分，对于我们的需求来说，最关键的部分是 Structured Output Parser。这个工具可以把我们之前作为字符串返回的模型输出，转化为可以在代码中直接使用的数据结构。

输出解析器是帮助结构化语言模型响应的类。一个输出解析器主要需要实现两种方法：

“get\_format\_instructions”：这是一种返回字符串的方法，该字符串包含如何格式化语言模型输出的指示。 “parse”：这是一种接受字符串（模型的响应）并将其解析为某种结构的方法。

本书中, 我们选取最常用的列表和 JSON 格式的输出解析器, 通过代码来解释输出解析器的必要性。Langchain 还提供了 Datetime parser, Enum parser , Structured output parser 等类型的输出解析器。 # 本章小结

本章，我们深入探讨了大语言模型的输入输出（Model 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。