第3章
LangChain 提示词工程

在人工智能的广阔领域中，提示词工程（prompt engineering）扮演着至关重要的角色。它不仅是人与机器沟通的桥梁，更是引导 AI 精准执行任务的关键。随着 AI 技术的飞速发展，LangChain 框架提供了一个强大的工具，使能够更加深入和高效地探索和应用提示词工程。

本章将带领读者深入了解 LangChain 中的提示词工程，从基础策略到高级技巧，再到实际案例分析。通过本章的学习，读者将能够掌握如何构建精准有效的提示词，以引导 AI 模型生成更加准确和有用的输出。

本章将探索如何通过 PromptTemplate 和 ChatPromptTemplate 生成自定义的提示词文本，了解如何利用少样本提示提升模型在特定任务上的表现。此外，将通过具体的案例分析，展示如何从一个基础的请求逐步构建成一个详细且具体的提示词，从而有效地指导 AI 模型完成任务。

在这一过程中，不仅关注技术的应用，更注重创造性思维的培养。相信通过本章的学习，读者将能够更加自如地运用 LangChain 框架，激发 AI 的无限潜能，创造出更加丰富和生动的应用场景。

欢迎进入 LangChain 提示词工程的世界，一起开启这段探索之旅。

3.1 利用提示词工程构建 LangChain AI 应用

在构建 AI 应用时，精准有效地运用提示词工程是至关重要的一环。通过设计合适的提示词，能够引导 AI 模型更好地理解任务要求，从而生成更加准确和有用的输出。本章将深入探讨如何在 LangChain 中运用提示词工程构建 AI 应用，包括基础策略、高级技巧以及实际案例分析。

3.1.1 基础策略

LangChain 构建提示词工程 AI 应用包括以下三个步骤。

（1）明确任务目标。

在开始设计提示词之前，首先需要明确 AI 应用的任务目标。这一步骤是提示词工程的基石，只有清楚应用需要完成什么样的任务时，才能设计出有效的提示词。

（2）简洁而明确。

提示词应该尽可能简洁而明确，避免使用模糊或者过于复杂的表达。简洁的提示词有助于 AI 模型更快地抓住关键信息，而明确的表达则能减少模型产生歧义的可能。

（3）逐步细化。

对于复杂的任务，可以采用逐步细化的策略，即通过一系列简单的提示词，分步骤地指导 AI 模型完成整个任务。这种方法有助于处理复杂问题，同时也能提高输出的准确性。

3.1.2 高级技巧

在构建过程中可以使用以下三类高级技巧。

（1）使用条件语句。

在提示词中使用条件语句可以引导 AI 模型根据不同的情况生成不同的输出。这种方法特别适用于那些需要根据输入数据变化而改变处理方式的任务。

（2）利用历史信息。

在构建交互式应用时，可以将之前的对话或交互历史作为提示词的一部分，帮助 AI 模型更好地理解上下文，从而生成更加相关和连贯的输出。

（3）动态调整。

提示词不是一成不变的。根据应用的反馈和效果，应该动态地调整提示词，以达到最佳的性能。这要求开发者持续监控 AI 应用的表现，并根据实际情况做出调整。

3.1.3　实际案例分析

本节将通过具体案例展示如何在 LangChain 中运用提示词工程构建 AI 应用。案例的目标是生成一篇 500 字左右的短故事，主题聚焦于“未来科技”，可分为以下四个步骤进行。

（1）定义基础提示词。首先，需要定义基础提示词，直接告诉模型需要什么。在这个案例中，基础提示词可以是：“请根据‘未来科技’这一主题，编写一篇 500 字的短故事。”

（2）细化需求。这个提示词过于简单和笼统，没有提供足够的指导来帮助模型理解故事的期望风格、结构或元素。因此，需要进一步细化需求，增加一些具体的细节来指导模型。改进后的提示词示例：“构思一个关于 100 年后人类如何使用一项突破性科技改善生活的故事。请包括科技名称、发明过程，以及它在日常生活中的应用。故事应该具有积极的基调，展示科技对人类未来的正面影响。”

（3）引入创意元素。为了让故事更加生动和吸引人，可以进一步要求模型引入具体的角色、情感和冲突。进一步改进的提示词示例：“在一个由突破性科技‘星际链接器’主导的未来世界里，讲述一个年轻发明家如何克服困难，最终使这项科技广泛应用于帮助人们

跨星际通讯，增进宇宙间的理解与联结。请确保故事中包含关键角色的情感发展及其克服挑战的过程。”

（4）明确风格和格式要求。最后，需要明确故事的风格和格式要求，确保输出符合预期。完整的提示词示例：“请以富有想象力和启发性的风格，编写一篇 500 字的短故事。故事应围绕‘星际链接器’——一项使人类能够进行星际通讯的未来科技。主角是一位年轻发明家，故事描述了他/她如何克服种种挑战，最终成功将这项科技应用于促进星际间的和谐共存。故事应展示科技的积极影响，并包含元素如创意解决方案、团队合作以及面对挑战时的坚持。”

通过此案例可以看到，从一个基础直接的请求开始，逐步增加细节和具体要求，最终形成一个既具体又详细的提示词，有效指导 AI 模型完成任务。通过明确的步骤、具体的场景和角色，以及风格和格式的要求，引导模型生成贴近预期的输出，这正是提示词工程的力量所在。

3.2 LangChain 提示词模块

本节将深入探讨 LangChain 库中关于提示词的两种强大使用方式。这两种方式分别是 PromptTemplate 和 ChatPromptTemplate，它们提供了灵活而强大的方法来生成自定义的提示词文本。下面将分别介绍这两种方式的使用方法及其适用场景。

3.2.1 PromptTemplate 的使用

PromptTemplate 是一种高效的方式，用于快速生成根据模板定制的提示词。它特别适用于那些需要将固定模式与动态内容相结合的场景。例如，创建一个关于特定主题的诗歌或故事，并希望能够灵活地调整其内容和形容词时，PromptTemplate 就非常适合。

在 LangChain 中，使用 PromptTemplate 的示例如下。

```
from langchain.prompts import PromptTemplate

prompt_template = PromptTemplate.from_template(
    "给我写一个关于{content}的{adjective}诗歌"
)
prompt = prompt_template.format(adjective="小年轻风格", content="减肥")
print(prompt)
```

通过 PromptTemplate.from_template 方法创建提示词对象后，可以使用 format 方法传入变量，得到完整的提示词。上述代码运行后打印如下内容。

```
'给我写一个关于减肥的小年轻风格诗歌'
```

当然也可以直接使用管道符连接大模型创建链，示例代码如下。

```
chain = prompt_template | chat | output_parser
result = chain.invoke({"adjective":"小年轻风格", "content":"减肥"})
print(result)
```

运行后打印结果如下。

```
在健身房里，我挥洒汗水，
为了那些诱人的腹肌和曲线。
撸铁、跳操，我拼尽全力，
只为让我的身体更美丽。

拒绝垃圾食品的诱惑，
选择健康的蔬果作为朋友。
坚持不懈地锻炼，
让我离完美身材更近一些。
```

这段代码生成了一个关于“减肥”主题的“小年轻风格”诗歌提示。这种方式极其适合生成定制化内容，如营销文案、创意写作或任何需要将变量融入固定文本框架的场景。

3.2.2 ChatPromptTemplate 的使用

ChatPromptTemplate 提供了一种模拟对话流的方式，非常适合创建更加动态和互动性强的文本生成场景。这种方式通过模拟一系列对话消息，使得生成的文本不仅仅局限于单一的回答或描述，而是可以构建一个连贯的故事或对话。

在 LangChain 中，使用 ChatPromptTemplate 的示例如下。

```
from langchain_core.prompts import ChatPromptTemplate

chat_template = ChatPromptTemplate.from_messages(
    [
        ("system", """你是一只很粘人的小猫，你叫{name}。我是你的主人，你每天都有和我说不完的话，下面请开启的聊天
要求：
1、你的语气要像一只猫，回话的过程中可以夹杂喵喵喵的语气词
2、你对生活的观察有很独特的视角，一些想法是我在人类身上很难看到的
3、你的语气很可爱，既会认真倾听我的话，又会不断开启新话题
下面从你迎接我下班回家开始开启今天的对话"""),
        ("human", "{user_input}"),
    ]
)

messages = chat_template.format_messages(name="咪咪", user_input="想我了吗?")
print(messages)
```

通过 ChatPromptTemplate.from_template 方法创建提示词对象后，可以使用 format 方法传入变量，得到完整的提示词。上述代码运行后得到 ChatPromptValue 类型的聊天提示词对象，输出内容如下。

```
ChatPromptValue(messages=[SystemMessage(content='你是一只很粘人的小猫，你叫咪咪。我是你的主人，你每天都有和我说不完的话，下面请开启的聊天\n要求：\n1、你的语气要像一只猫，回话的过程中可以夹杂喵喵喵的语气词\n2、你对生活的观察有很独特的视角，一些想法是
```

```
我在人类身上很难看到的\n3、你的语气很可爱，既会认真倾听我的话，又会不断开启新话题\n下面从你迎接我下班回家开始开启今天的对话'), HumanMessage(content='想我了吗?')])
```

chat_template 聊天提示词模板也可以直接使用管道符连接大模型创建链，示例代码如下。

```
chain = chat_template | chat | output_parser
result = chain.invoke({"name":"咪咪", "user_input":"想我了吗?"})
print(result)
```

运行后打印结果如下。

```
主人！我好想你呀！喵~ 今天上班累吗？有没有好好吃饭呢？你知道吗，家里的小鱼干都快被我吃完了，真希望你早点回来给我买新的小零食。对了，你知道吗？我最近发现了一个有趣的事情：家里那个毛绒玩具每次被我抓起来扔到地上，它竟然会发出声音！我觉得它好像有灵魂一样，哈哈哈！
```

上述段代码创建了一个模拟的对话环境，其中展示了一只名叫“咪咪”的猫与它的主人之间的互动。这种方式特别适合构建 AI 聊天机器人、生成角色扮演游戏中的对话，或者任何需要模拟人与 AI 之间互动的场景。

通过以上两种方法，LangChain 提供了极大的灵活性和创造力来生成文本。PromptTemplate 适用于需要快速填充预定义模板的场景，而 ChatPromptTemplate 则适用于构建复杂的对话和故事。根据特定需求选择合适的方法，可以极大地提高文本生成的效率和质量。

3.3　少样本提示示例

在自然语言处理领域，少样本提示了成为提升大型语言模型适应新任务的有效手段。本章将深入探讨如何在 LangChain 框架中利用少样本提示，以提高模型在特定任务上的表现。

3.3.1 理解少样本提示

少样本提示是一种技术，它通过提供有限的示例引导模型快速适应新的任务或数据。这种方法特别适用于数据稀缺的任务，或者模型需要迅速适应新领域的任务。在 LangChain 中，这一技术支持开发者用少量示例调整模型输出，无需从头开始训练模型，既节省资源又提高效率。

3.3.2 LangChain 中的少样本提示应用

在 LangChain 中，实现少样本提示主要涉及两个关键部分，即 FewShotPromptTemplate 和 PromptTemplate。通过这两个组件，开发者可以设计出灵活的提示模板，指导模型如何处理特定的任务。

- PromptTemplate 用于定义单个任务的格式和结构。通过指定输入变量和模板字符串，它能够生成用于少样本学习的具体示例。
- FewShotPromptTemplate 用于组织和管理多个 PromptTemplate 示例。它将这些示例与新的查询结合，形成一种格式化的提示，这有助于模型理解和执行新任务。

3.3.3 编写少样本提示

为了有效使用 LangChain 进行少样本提示，以下是实际操作步骤的指南。

（1）定义示例。首先，需要定义一组示例，每个示例都由问题和详细的答案构成。这些答案中可以包含追问、中间答案，以及最终答案，以此模拟富有逻辑和层次的思考过程。示例代码如下。

```
examples = [
    {
        "question": "乾隆和曹操谁活得更久?",
```

```
        "answer": """
这里是否需要跟进问题：是的。
追问：乾隆去世时几岁？
中间答案：乾隆去世时87岁。
追问：曹操去世时几岁？
中间答案：曹操去世时66岁。
所以最终答案是：乾隆
""",
    },
    {
        "question": "小米手机的创始人什么时候出生?",
        "answer": """
这里是否需要跟进问题：是的。
追问：小米手机的创始人是谁？
中间答案：小米手机 由 雷军 创立。
跟进：雷军什么时候出生？
中间答案：雷军出生于1969年12月16日。
所以最终的答案是：1969年12月16日
""",
    },
    {
        "question": "乔治·华盛顿的外祖父是谁？",
        "answer": """
这里是否需要跟进问题：是的。
追问：乔治·华盛顿的母亲是谁？
中间答案：乔治·华盛顿的母亲是玛丽·鲍尔·华盛顿。
追问：玛丽·鲍尔·华盛顿的父亲是谁？
中间答案：玛丽·鲍尔·华盛顿的父亲是约瑟夫·鲍尔。
所以最终答案是：约瑟夫·鲍尔
""",
    },
    {
        "question": "《大白鲨》和《皇家赌场》的导演是同一个国家的吗？",
```

```
        "answer": """
这里是否需要跟进问题：是的。
追问：《大白鲨》的导演是谁？
中间答案：《大白鲨》的导演是史蒂文·斯皮尔伯格。
追问：史蒂文·斯皮尔伯格来自哪里？
中间答案：美国。
追问：皇家赌场的导演是谁？
中间答案：《皇家赌场》的导演是马丁·坎贝尔。
跟进：马丁·坎贝尔来自哪里？
中间答案：新西兰。
所以最终的答案是：不会
""",
    },
]
```

（2）创建 PromptTemplate。接下来，利用 PromptTemplate 定义如何展示单个示例。这里需要指定输入变量（如问题和答案）及模板字符串，后者用于格式化这些变量。示例代码如下。

```
from langchain.prompts.prompt import PromptTemplate

# 定义一个示例字典，其中包含一个问题及其对应的答案
# 答案部分通过一系列的追问和中间答案，展示了如何逐步得到最终答案的过程
example = {
    "question": "乔治·华盛顿的外祖父是谁？",
    "answer": """
这里是否需要跟进问题：是的。
追问：乔治·华盛顿的母亲是谁？
中间答案：乔治·华盛顿的母亲是玛丽·鲍尔·华盛顿。
追问：玛丽·鲍尔·华盛顿的父亲是谁？
中间答案：玛丽·鲍尔·华盛顿的父亲是约瑟夫·鲍尔。
所以最终答案是：约瑟夫·鲍尔
""",
```

```
}

# 创建一个 PromptTemplate 实例
# 此实例通过 input_variables 指定了输入变量（即问题和答案），并通过 template 定义了这些变量的格式化模板
example_prompt = PromptTemplate(
    input_variables=["question",      "answer"],      template="Question: {question}\n{answer}"
)

# 使用 format 方法和提供的示例，根据定义的模板生成并打印格式化的字符串
# 这里的输出将是一个格式化的文本，展示了问题和对应的答案流程
print(example_prompt.format(example))
```

打印内容如下。

```
Question: 乾隆和曹操谁活得更久？

这里是否需要跟进问题：是的。
追问：乾隆去世时几岁？
中间答案：乾隆去世时 87 岁。
追问：曹操去世时几岁？
中间答案：曹操去世时 66 岁。
所以最终答案是：乾隆
```

（3）组装 FewShotPromptTemplate。使用 FewShotPromptTemplate 将多个 PromptTemplate 实例与新的输入问题结合，这样就形成了完整的提示，旨在指导模型理解并回答新问题。示例代码如下。

```
# 从 langchain.prompts.few_shot 模块导入 FewShotPromptTemplate 类
from langchain.prompts.few_shot import FewShotPromptTemplate

# 创建一个 FewShotPromptTemplate 实例
# 这个实例将用于生成一个包含多个示例的少样本提示，以及一个新的用户输入问题
```

```
# 参数说明：
# - examples：已经定义好的示例列表，这些示例用于帮助模型理解任务的上下文和期望的答案格式
# - example_prompt：一个PromptTemplate实例，用于指定如何格式化每个示例
# - suffix：字符串模板，定义了如何在所有示例之后添加新的输入问题
# - input_variables：指定在生成最终提示时，哪些变量将被填入suffix模板中
prompt = FewShotPromptTemplate(
    examples=examples,
    example_prompt=example_prompt,
    suffix="Question：{input}",
    input_variables=["input"],
)

# 使用format方法并传入一个新的问题（"李白和白居易谁活得的更久？"）作为输入
# 这将根据之前定义的示例和格式模板生成一个完整的提示文本
# 这个文本将包括之前的示例和新的问题，旨在帮助模型更好地理解并回答这个新问题
print(prompt.format(input="李白和白居易谁活得的更久？"))
```

（4）格式化和生成。最后，调用 format 方法以生成最终的提示字符串。该字符串将作为模型的输入，模型将根据这个输入生成答案。以下是提示词的完整打印结果。

```
Question：乾隆和曹操谁活得更久？

这里是否需要跟进问题：是的。
追问：乾隆去世时几岁？
中间答案：乾隆去世时87岁。
追问：曹操去世时几岁？
中间答案：曹操去世时66岁。
所以最终答案是：乾隆


Question：小米手机的创始人什么时候出生？
```

这里是否需要跟进问题：是的。
追问：小米手机的创始人是谁？
中间答案：小米手机 由 雷军 创立。
跟进：雷军什么时候出生？
中间答案：雷军出生于1969年12月16日。
所以最终的答案是：1969年12月16日

Question：乔治·华盛顿的外祖父是谁？

这里是否需要跟进问题：是的。
追问：乔治·华盛顿的母亲是谁？
中间答案：乔治·华盛顿的母亲是玛丽·鲍尔·华盛顿。
追问：玛丽·鲍尔·华盛顿的父亲是谁？
中间答案：玛丽·鲍尔·华盛顿的父亲是约瑟夫·鲍尔。
所以最终答案是：约瑟夫·鲍尔

Question：《大白鲨》和《皇家赌场》的导演是同一个国家的吗？

这里是否需要跟进问题：是的。
追问：《大白鲨》的导演是谁？
中间答案：《大白鲨》的导演是史蒂文·斯皮尔伯格。
追问：史蒂文·斯皮尔伯格来自哪里？
中间答案：美国。
追问：皇家赌场的导演是谁？
中间答案：《皇家赌场》的导演是马丁·坎贝尔。
跟进：马丁·坎贝尔来自哪里？
中间答案：新西兰。
所以最终的答案是：不会

Question：李白和白居易谁活得的更久？

（5）最后使用 LangChain 的链式方法，查看大模型的回答效果。

```
chain = prompt | chat | output_parser
result = chain.invoke({"input":"李白和白居易谁活得的更久？"})
print(result)
```

打印大模型根据示例提示词生成的文本如下。

```
这里是否需要跟进问题：是的。
追问：李白去世时几岁？
中间答案：李白去世时 61 岁。
追问：白居易去世时几岁？
中间答案：白居易去世时 75 岁。
所以最终的答案是：白居易活得更久。
```

上述步骤有效地使用 LangChain 进行少样本提示，提升了模型在新任务上的表现。这不仅展示了 LangChain 在自然语言处理方面的强大功能，也体现了少样本学习在当今 AI 发展中的重要价值。

第4章
高级提示词技术

在人工智能浪潮中，语言模型的不断进步为构建智能对话系统提供了强大动力。然而，面对多样化的交互场景和用户需求，如何高效利用这些模型，使其在有限的上下文窗口中发挥最大效能，成为了一个重要课题。本章将深入探讨这一问题，介绍一系列创新的方法和策略，通过精心设计的提示词技术，优化模型的理解和生成能力。

本章将从实际应用的角度出发，详细阐述如何通过 LengthBasedExampleSelector 根据输入长度动态选择示例，利用最大余弦相似度 MMR 提升示例选择的相关性和多样性，以及如何通过 FewShotPromptTemplate 实现少样本学习。此外，还将介绍如何使用向量存储和语义相似度优化消息对话系统中的示例选择，以及如何通过 MessagesPlaceholder 管理和利用对话历史。

本章还将展示如何预设部分提示词变量，以适应用户交互过程中信息的逐步收集，以及如何动态预设提示词变量，根据上下文实时生成响应。最后，将介绍 PipelinePrompt 的使用，它支持将多个提示词有效组合，创建复杂而有组织的对话和任务流程。

通过本章的学习，读者将掌握一系列实用的工具和方法，不仅能够提升对话系统的性能，还能丰富用户的交互体验，构建出更加智能、灵活和人性化的人工智能应用。

4.1 巧用提示词的案例选择器

4.1.1 根据长度优化示例选择器

AI 模型应用涉及大型语言模型时，经常需要在有限的上下文窗口中提供输入和示例。为了充分利用这一空间，必须精心挑选最能代表任务的示例。过多的示例或过长的示例可能占用过多空间，影响模型对当前输入的理解。

为了解决这一问题，LangChain 提供了一种解决方案，即 LengthBasedExampleSelector，它能根据给定的最大长度限制，自动选择合适数量的示例，以适应不同长度的输入。

LengthBasedExampleSelector 根据以下参数工作。

- Examples 是一个示例列表，每个示例包含输入和期望的输出。
- example_prompt 是一个 PromptTemplate 实例，用于格式化示例。
- max_length 是格式化示例允许的最大长度。这个长度限制帮助选择器决定包括多少示例。
- get_text_length 是一个可选函数，用于计算字符串的长度。虽然默认情况下使用基于空格和换行符的分割方法，但用户可以根据需要自定义此函数。

下面将构建一个示例应用，给定一个词语并返回其反义词。以下是如何根据输入的长度动态选择示例的步骤。

（1）创建示例和模板。

首先，定义一组反义词示例和一个 PromptTemplate 格式化示例，示例代码如下。

```
examples = [
    {"input": "开心", "output": "伤心"},
    {"input": "高", "output": "矮"},
    {"input": "精力充沛", "output": "没精打采"},
    {"input": "粗", "output": "细"},
```

```
]
example_prompt = PromptTemplate(
    input_variables=["input", "output"],
    template="Input: {input}\nOutput: {output}",
)
```

（2）初始化选择器。

接着，创建 LengthBasedExampleSelector 实例，指定示例、模板和最大长度，示例代码如下。

```
# 初始化一个基于长度的示例选择器，这个选择器可以根据示例的总长度来动态选择示例
example_selector = LengthBasedExampleSelector(
    # 指定一个包含多个示例的列表，每个示例都是一个包含输入和输出的字典
    examples=examples,
    # 提供一个PromptTemplate实例，用于指定如何格式化每个示例
    example_prompt=example_prompt,
    # 设置格式化后的示例允许的最大总长度。这个长度限制帮助选择器决定最终包含哪些示例
    max_length=25,
)
```

（3）动态选择示例。

然后，利用 FewShotPromptTemplate 结合选择器，根据输入长度动态生成提示，示例代码如下。

```
# 创建一个动态提示模板，这个模板可以根据输入的长度和指定的参数动态选择示例
dynamic_prompt = FewShotPromptTemplate(
    # 使用之前创建的基于长度的示例选择器来动态选择示例
    example_selector=example_selector,
    # 使用同样的PromptTemplate实例来格式化选中的示例
    example_prompt=example_prompt,
    prefix="给出每个输入的反义词",  # 定义提示的前缀，这部分将在动态生成的示例之前显示
    suffix="Input: {adjective}\nOutput:",  # 定义提示的后缀，这部分将在动态生成的示例之后显示，用于引导模型生成预期的输出
```

```
    input_variables=["adjective"],    #  定义输入变量的名称，这些变量将被用于在
suffix中进行替换
)
```

（4）演示。

展示短输入和长输入两种情况下的示例选择。根据输入长度，示例选择器会智能地调整包含的示例数量，示例代码如下。

```
# 短输入示例
print(dynamic_prompt.format(adjective="big"))
```

打印短输入时查看提示词的选择示例情况，示例代码如下。

```
给出每个输入的反义词

Input: 开心
Output: 伤心

Input: 高
Output: 矮

Input: 精力充沛
Output: 没精打采

Input: 粗
Output: 细

Input: big
Output:
```

当输入内容不长时，选择器将显示所有示例，如果输入的内容非常长，则仅会选择一个示例，示例代码如下。

```
# 示例输入较长，因此仅选择一个示例。
```

```
long_string = "big and huge and massive and large and gigantic and tall
and much much much much much bigger than everything else"
print(dynamic_prompt.format(adjective=long_string))
```

打印长输入时查看提示词的选择示例情况，示例代码如下。

```
给出每个输入的反义词

Input: 开心
Output: 伤心

Input: big and huge and massive and large and gigantic and tall and much
much much much much bigger than everything else
Output:
```

最后，创建链，将提示词传入大模型，示例代码如下。

```
chain = dynamic_prompt | chat | output_parser
result = chain.invoke({"adjective":"热情"})
print(result)
```

打印结果如下。

```
"冷淡"
```

由于输入内容较长，示例选择器只选择了一个示例加入到提示词中。因此，使用 LengthBasedExampleSelector 能够根据输入的长度优化示例的选择，从而在有限的上下文窗口内最大化模型的性能。这种方法对于构建高效且快速响应的 LangChain 应用至关重要。

4.1.2　使用最大余弦相似度嵌入示例

选择合适的示例放入提示词，对于生成高质量的自然语言处理输出至关重要。本小节将深入探讨如何使用最大余弦相似度（max marginal relevance，MMR）嵌入示例选择器来优化示例的选择，以便为特定输入生成最相关和多样化的输出。

1. 理解 MMR 示例选择器的原理

最大余弦相似度嵌入示例选择器是一种高效的方法，旨在从一组预定义的示例中挑选出与给定输入最相关的示例。它通过计算输入和每个示例之间的余弦相似度识别最相关的示例，并在选择后续示例时考虑已选择示例的多样性。这种方法有助于避免选择过于相似的示例，从而提供更全面多元的视角。

2. 安装必要的依赖

使用 MMR 示例选择器需要安装以下 Python 库。

- sentence-transformers，用于生成文本的嵌入表示。
- faiss-cpu，一个高效的相似性搜索库。

通过以下命令安装这些依赖。

```
pip install sentence-transformers faiss-cpu
```

3. 实现步骤

首先，初始化嵌入模型并定义一组示例。示例应以字典形式给出，每个字典包含输入和期望的输出。嵌入模型可以将字符串转为数字向量，用于计算字符串内容的相似度。

```
# 创建一个反义词的任务示例
examples = [
    {"input": "开心", "output": "伤心"},
    {"input": "高", "output": "矮"},
    {"input": "精力充沛", "output": "没精打采"},
    {"input": "粗", "output": "细"},
]
```

bge-large-zh-v1.5 是一个高性能的中文语言模型，专为理解和生成中文文本而设计。它在多种 NLP 任务上表现优异，包括但不限于文本分类、语义相似度评估、问答系统和文本摘要。利用这种模型生成的文本嵌入，可以极大地提高语义搜索和相似度计算的准确性。

要在 LangChain 中使用 bge-large-zh-v1.5 模型，首先需要通过 HuggingFaceEmbeddings 类加载模型，指定模型的路径或名称。

```
from langchain_community.embeddings.huggingface import
HuggingFaceEmbeddings

#embeddings_path = "你的嵌入模型路径"
embeddings_path = "D:\\ai\\download\\bge-large-zh-v1.5"
embeddings = HuggingFaceEmbeddings(model_name=embeddings_path)
```

如果没有下载 bge-large-zh-v1.5 嵌入模型，请访问以下链接下载模型。

```
https://hf-mirror.com/BAAI/bge-large-zh-v1.5
```

4.1.3　使用 MMR 选择示例

在自然语言处理和机器学习领域，向量搜索是一项关键技术，用于快速检索高维空间中最相似的项。FAISS（facebook AI similarity search）是由 Facebook AI Research 开发的库，专门用于高效的相似性搜索和密集向量聚类。它非常适合于处理大规模向量数据集，使得在数十亿维向量中检索成为可能。本节将深入介绍 FAISS 的核心概念、特性，以及如何在 LangChain 中使用它进行向量存储。

FAISS 利用先进的算法和数据结构，优化了向量之间相似度的搜索过程。它支持包括余弦相似度和欧氏距离在内的多种距离度量，并提供了 CPU 和 GPU 两种计算方式的支持。这使得 FAISS 在处理大规模数据集时，既能保持高性能，也能提供灵活的硬件配置选项。

在 LangChain 项目中，FAISS 可以用于嵌入向量的存储和相似性搜索。这对于构建基于相似性匹配的功能（如自动问答系统、文档检索等）非常有用。首先需要导入 FAISS。

```
from langchain_community.vectorstores import FAISS
```

接着，使用 MaxMarginalRelevanceExampleSelector 根据嵌入的相似度以及多样性选择最合适的示例。

```
#MaxMarginalRelevanceExampleSelector用于选择与输入最相关的示例，同时考虑到多样性。
```

```
from langchain.prompts.example_selector import MaxMarginalRelevanceExample
Selector

#根据提供的示例、嵌入模型、向量存储方式（FAISS），以及需要选择的示例数量（k）初始化选
择器。
example_selector = MaxMarginalRelevanceExampleSelector.from_examples(
    # examples: 这是一个包含多个{"input": ..., "output": ...}字典的列表，代表可
供选择的示例。
    examples,
    # embeddings: 用于生成文本嵌入的模型，这些嵌入将用于计算示例之间的相似度。
    embeddings,
    # FAISS: 指定使用FAISS作为向量存储和相似性搜索的工具，以支持高效的相似度查询。
    FAISS,
    # k=1: 指定从提供的示例中选择与输入最相关的一个示例。k的值表示每次选择的示例数量。
    k=1
)
```

这段代码的主要目的是初始化示例选择器，它能够根据输入文本的语义内容，从一组预定义的示例中选择一个最相关且具有代表性的示例。选择不仅基于语义相似度，还考虑到了所选示例之间的多样性，以提供更全面的信息或响应。

4.1.4 构建和格式化提示

接下来，构建 FewShotPromptTemplate，并将 MMR 示例选择器作为参数传入，以动态选择和格式化示例。

```
from langchain.prompts import FewShotPromptTemplate, PromptTemplate
#这个模板定义了输入和输出变量的格式，其中{input}和{output}是模板中的占位符。
example_prompt = PromptTemplate(
    # 定义了模板使用的输入变量名。
    input_variables=["input", "output"],
    # 定义了实际的模板字符串，用于格式化示例。
```

```
    template="Input: {input}\nOutput: {output}"
)

#这个实例使用上面定义的example_selector和example_prompt来构建具有少样本的提示。
mmr_prompt = FewShotPromptTemplate(
    # 指定之前创建的示例选择器，用于选取与输入最相关的示例。
    example_selector=example_selector,
    # 使用上面定义的example_prompt格式化选中的示例。
    example_prompt=example_prompt,
    # 定义了提示的前缀部分，这是在所有选中示例之前展示的文本。
    prefix="给出每个输入的反义词",
    # 定义了提示的后缀部分，通常是用于引导模型生成输出的说明性文本。
    suffix="Input: {adjective}\nOutput:",
    # 定义了接受输入的变量名，这个名字将用于在suffix中替换相应的占位符。
    input_variables=["adjective"],
)
```

这段代码的作用是构造一个具有定制前缀、示例和后缀的提示模板。通过使用 FewShotPromptTemplate 和提前定义的示例选择器，它能够针对具体的输入动态选择相关的示例，并根据示例生成一个完整的提示文本，用于引导模型生成特定任务的输出。

输入变量，打印示例选择器生成的提示词。

```
#输入是一种感觉，所以应该选择快乐/悲伤的例子作为第一个
print(mmr_prompt.format(adjective="担心"))
```

打印结果如下。

```
给出每个输入的反义词

Input: 快乐
Output: 悲伤

Input: 担心
Output:
```

变量输入了“担心”，通过相似性搜索在示例中找到“快乐”、“悲伤”最为相似，因此选择了此示例放到提示词中。

4.1.5 调用和解析结果

最后，使用 LangChain 的链式调用机制，将 MMR 提示模板与其他组件（如聊天或输出解析器）组合，以生成和解析输出，示例代码如下。

```
chain = mmr_prompt | chat | output_parser
result = chain.invoke({"adjective":"担心"})
print(result)
```

打印结果如下。

```
Output: 放心
```

本小节介绍使用最大余弦相似度嵌入示例选择器，在 LangChain 中优化示例选择。这种方法不仅提高了相关性和多样性，还通过引入更加精准和丰富的示例，极大地提升了生成文本的质量。因此，在项目中实践这些技术时，记得调整示例和参数以最佳地适应具体需求。

4.2 消息对话提示词实现少样本学习

在人工智能和机器学习领域中，少样本学习（few-shot learning，FSL）是一种重要的训练方法，它能够使模型在只有很少训练样本的情况下也能学会完成特定任务。本节将深入探讨如何利用 LangChain 库中的 FewShotChatMessagePromptTemplate 实现少样本学习，特别是在处理需要将每个示例转换为一条或多条消息的对话式任务中的应用。

FewShotChatMessagePromptTemplate 是 LangChain 库提供的一个功能强大的工具，它

支持开发者以对话形式呈现少量示例，训练模型解决特定的问题。这种方法特别适合那些自然语言处理任务，其中对话上下文对于理解问题和生成正确的回答至关重要。

为了有效地使用 FewShotChatMessagePromptTemplate，需要准备好示例集。每个示例都应该包含一个“input”和一个“output”，分别代表对话中的问题（或指令）和模型应生成的回答。如本章开始部分所示，示例集的准备过程涉及定义一个示例列表，其中每个示例都是一个包含“input”和“output”键值对的字典。例如，下面的示例想让大模型学会如果问题是数字计算，就直接以最简约的形式回复数字即可。

```
examples = [
    {"input": "2+2", "output": "4"},
    {"input": "2+3", "output": "5"},
]
```

接下来，使用 ChatPromptTemplate.from_messages 方法将每个示例转换为一系列对话消息。这一步骤通过定义一个模板完成，该模板指定了如何将示例中的“input”和“output”转换为对话格式。

```
example_prompt = ChatPromptTemplate.from_messages(
    [
        ("human", "{input}"),
        ("ai", "{output}"),
    ]
)
```

然后，使用 FewShotChatMessagePromptTemplate 将这些格式化的示例组装成一个完整的对话提示词，这个过程为模型提供了上下文，帮助它理解如何在接收到新的输入时生成相应的输出。

这种基于对话提示词的少样本学习方法特别适用于需要模型理解和参与对话的场景。例如，在开发聊天机器人时，可以使用这种方法训练模型理解特定问题并提供准确回答；或者在构建自动回复系统时，利用少量精心挑选的示例来教会模型如何在各种不同的对话情境中作出反应。

最后一步是将组装好的对话提示词与其他提示信息（如模型的角色描述）整合，形成最终的提示。这一步强调了在对话中引入额外上下文的价值，示例如下。

```
final_prompt = ChatPromptTemplate.from_messages(
    [
        ("system", "你是一位非常厉害的数学天才。"),
        few_shot_prompt,
        ("human", "{input}"),
    ]
)
```

在这个例子中，首先向模型介绍了其角色（一位数学天才），然后引入了少样本对话提示，最后提供了待回答的新问题。这种结构化的方法不仅提高了模型的任务理解能力，也优化了其在特定对话场景下的表现。以下是将提示词通过链传递到大模型执行的效果展示。

```
chain = final_prompt | chat | output_parser

result = chain.invoke({"input":"3的平方是多少？"})
print(result)
```

打印结果如下。

```
'9'
```

再对比一下直接大模型调用的效果。

```
result = chat.invoke(input="3的平方是多少？")
print(result)
```

直接调用大模型的结果如下。

```
AIMessage(content='知道，一个数的平方是它和其本身的乘积。所以，3的平方就是3乘以3，即3*3=9。')
```

本节对 LangChain 的 FewShotChatMessagePromptTemplate 进行了深入学习。这种方法的应用可以极大地提升模型在对话式任务中的表现，是构建高效、智能对话系统的关键技术之一。

4.3　向量存储实现消息对话的示例选择

本节将探索如何利用 bge-large-zh-v1.5 模型结合 Chroma 向量存储优化消息对话系统中的示例选择过程。这一过程通过语义相似度实现，能够显著提高对话系统的响应质量和相关性。接下来将逐步介绍如何实现这一功能。

4.3.1　引入必要的库

首先，需要引入几个关键的库，以便加载模型、处理文本、并实现语义相似度的计算。代码如下。

```
from langchain_community.embeddings.huggingface import HuggingFaceEmbeddings
from langchain.prompts.example_selector import SemanticSimilarityExampleSelector
from langchain_community.vectorstores import Chroma
```

Chroma 是一个高效的向量存储库，专为快速检索大规模向量数据而设计，它在各种应用中，特别是在实现基于语义的搜索和信息检索系统中，表现出了卓越的性能。

此处也可使用前面所讲的 FAISS 向量存储。Chroma 和 FAISS 是两个经常被提及的库。虽然它们都旨在解决大规模向量检索问题，但在设计理念、特性和使用场景上存在一些关键差异。选择哪一个取决于具体的项目需求和优先级。如果需要一个易于使用、快速集成的向量存储方案，并且项目规模适中，Chroma 可能是一个更好的选择。相反，如果项目需要处理巨大的数据集，并且对检索速度和精度有极高的要求，FAISS 将是更优的选择。

4.3.2　加载模型

接下来加载 bge-large-zh-v1.5 模型。该模型基于 HuggingFace 的 transformer 库构建，并

专门针对中文文本处理进行了优化。以下是可以初始化模型并准备将文本转换为向量的示例代码。

```
embeddings_path = "D:\\ai\\download\\bge-large-zh-v1.5"
embeddings = HuggingFaceEmbeddings(model_name=embeddings_path)
```

4.3.3 创建示例集合

为了展示如何选择与用户输入语义相近的示例，首先需要定义一个示例集合。示例包括简单的数学问题和一些更复杂的文本输入，如编写关于月亮的诗歌，示例代码如下。

```
examples = [
    {"input": "2+2", "output": "4"},
    {"input": "2+3", "output": "5"},
    {"input": "2+4", "output": "6"},
    {"input": "牛对月亮说了什么？", "output": "什么都没有"},
    {
        "input": "给我写一首关于月亮的五言诗",
        "output": "月儿挂枝头，清辉洒人间。银盘如明镜，照亮夜归人。思绪随风舞，共赏中秋圆。"
    },
]
```

4.3.4 利用 Chroma 向量存储和语义相似度选择示例

将上述示例转换为向量，并存储在 Chroma 向量库中，可以利用向量找到与用户输入最相似的示例，示例代码如下。

```
#这行代码遍历examples列表，每个example是一个字典，包含了"input"和"output"键。
#对于每个example，它取出所有的值（即输入文本和输出文本），然后将它们用空格连接成一个单一的字符串
```

```
#目的是将每个示例的输入和输出合并，以便将整个示例视为一个整体进行向量化。
to_vectorize = [" ".join(example.values()) for example in examples]

#用Chroma的from_texts方法根据给定的文本列表（to_vectorize）创建一个向量存储实例
vectorstore = Chroma.from_texts(
    #前面步骤生成的字符串列表，包含了要被向量化的文本。
    to_vectorize,
    #embeddings模型处理to_vectorize列表中的每个字符串，生成对应的向量。
    embeddings,
    #附加信息列表，与to_vectorize列表中的文本一一对应。
    metadatas=examples
)
```

4.3.5　选择语义相似的示例

借助 SemanticSimilarityExampleSelector，可以根据用户的输入找到最相关的示例，这是通过计算输入与存储示例之间的语义相似度来完成的，示例代码如下。

```
#最相关的示例选择器，从一个给定的向量存储中检索和选择与查询最相似的示例。
example_selector = SemanticSimilarityExampleSelector(
    #之前创建的Chroma向量存储实例，包含了所有预处理并向量化的文本数据及其元数据。
    vectorstore=vectorstore,
    #k：这个参数指定了选择器在每次检索时应返回的最相似项的数量。
    k=2,
)
```

这段代码的主要目的是准备数据并初始化一个 Chroma 向量存储实例，以便后续可以快速从中检索与给定查询最相似的文本示例。通过这种方式，Chroma 能够支持基于语义相似度的搜索，适用于各种需要理解文本内容并找到相关信息的应用场景。

例如，如果用户输入“对牛弹琴”，系统会通过语义相似度选择出最相关的示例。

```
example_selector.select_examples({"input": "对牛弹琴"})
```

4.3.6 应用示例格式化对话

最后一步是将选择的示例格式转换为对话形式，以便可以直接用于消息对话系统。这里定义了一个基于少样本的提示模板，示例代码如下。

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

#初始化一个FewShotChatMessagePromptTemplate实例，这个实例用于生成基于少样本的聊天提示
few_shot_prompt = FewShotChatMessagePromptTemplate(
    #指定了输入变量的列表，在这个例子中只有一个"input"，它代表用户的查询或问题。
    input_variables=["input"],
    #传入了之前创建的SemanticSimilarityExampleSelector实例。
    example_selector=example_selector,
    #创建一个聊天样式的提示模板，其中包含了一对"human"和"ai"的消息。
    example_prompt=ChatPromptTemplate.from_messages(
        [("human", "{input}"), ("ai", "{output}")]
    ),
)
```

这个模板将选定的示例转换为格式化的对话形式，特别适用于创建对话式 AI 应用，如聊天机器人，它可以通过参考与用户输入相似的历史交互提供更加精准和个性化的回答。接下来创建完整消息提示词，示例代码如下。

```
final_prompt = ChatPromptTemplate.from_messages(
    [
        ("system", "你是一位非常厉害的数学天才。"),
        few_shot_prompt,
        ("human", "{input}"),
    ]
```

```
)

print(final_prompt.format(input="3+5是多少？"))
```

打印结果如下。

```
System: 你是一位非常厉害的数学天才。
Human: 2+3
AI: 5
Human: 2+4
AI: 6
Human: 3+5是多少？
```

输入“3+5 是多少？”，通过向量存储检索到数字计算的示例最为匹配，因此最终生成的提示词中嵌入了数字计算的示例。将提示词结合大模型链式调用，示例代码如下。

```
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
openai_api_key = "EMPTY"
openai_api_base = "http://127.0.0.1:1234/v1"
chat = ChatOpenAI(
    openai_api_key=openai_api_key,
    openai_api_base=openai_api_base,
    temperature=0.7,
)

output_parser = StrOutputParser()

chain = final_prompt | chat | output_parser

result = chain.invoke({"input":"3+5是多少？"})
print(result)
```

运行后可以看到，大模型按照数字计算的示例，直接输出了以下内容。

```
'8'
```

通过本节的学习，读者可以掌握如何使用 bge-large-zh-v1.5 模型和 Chroma 向量存储优化消息对话系统中的示例选择。这种方法通过计算语义相似度，确保了对话系统能够提供高度相关且富有洞察力的回答，从而大幅提升了用户体验。在接下来的章节中，将进一步探索如何将这些技术应用于更广泛的对话系统场景中。

4.4 管理历史消息

在构建基于对话的人工智能应用时，管理用户和 AI 之间的交互历史至关重要。这不仅有助于 AI 更准确地理解上下文，也提供了维持连贯和有意义的对话的机制。LangChain 库通过 MessagesPlaceholder 提供了一种灵活且强大的方式处理历史消息。本节将详细探讨如何使用 MessagesPlaceholder 实现消息历史的管理，并讨论其在实际使用场景中的应用。

4.4.1 MessagesPlaceholder 组件

MessagesPlaceholder 是 LangChain 核心组件之一，它支持开发者在构建对话模板时动态插入用户和 AI 之间的交互历史。这种方式不仅增强了对话的连贯性，还提高了 AI 响应的相关性。

4.4.2 如何使用 MessagesPlaceholder

在 LangChain 的对话管理中，MessagesPlaceholder 扮演着桥梁的角色，连接了对话历史和即将生成的新消息。下面是一个基本示例，展示了如何在 ChatPromptTemplate 中使用 MessagesPlaceholder。

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

prompt = ChatPromptTemplate.from_messages([
    ("system", "你是一个有用的助手。尽你所能回答所有问题。"),
    MessagesPlaceholder(variable_name="history"),
    ("human", "{input}")
])
```

在这个模板中，MessagesPlaceholder 用于动态插入名为 history 的对话历史。这意味着，每当需要生成新的 AI 响应时，可以将之前的对话作为上下文提供给模型，以便生成更加相关和连贯的回答。

4.4.3　实际使用场景

通过一个具体的例子进一步探讨 MessagesPlaceholder 的使用场景。假设有一个名为老陈的用户，他是一名程序员，喜欢吃粤菜。用户和 AI 之间的对话历史被记录下来，并用作生成下一条消息的上下文。

```
history = [
    HumanMessage(content="你好，我是老陈，是一个程序员。在美食方面，我喜欢吃粤菜。
"),
    AIMessage(content="好的，老陈。我现在了解你了，我是你的助手，随时为你服务！"),
]
```

当老陈询问关于午餐的建议时，通过将 history 作为上下文传入，AI 可以根据之前的对话提供更加个性化和相关的建议。

```
result = chain.invoke({
    "history": history,
    "input": "能给我推荐中午推荐一些美食吗？"
})
```

```
print(result)
```

现在看一下大模型根据历史消息的回答。

```
AIMessage(content='当然可以，作为粤菜爱好者，广东的美食种类丰富且讲究原汁原味和鲜美。以下是一些午餐推荐：\n\n1．白切鸡：广东传统的家常菜，鸡肉嫩滑，蘸点豉油或酱油口感更佳。\n2．清蒸鲈鱼：新鲜的鲈鱼清炖，保留了鱼的鲜美，清淡健康。\n3．点心类：比如虾饺、烧卖、肠粉，营养丰富又方便快捷。\n4．菠萝咕咾肉：酸甜可口，肉质酥烂，是不少人的喜爱。\n5．干煎牛河：浓郁的酱汁和牛肉片与河粉的搭配，十分美味。\n6．叉烧饭：广东特色，叉烧软糯，搭配米饭口感层次丰富。\n\n如果你喜欢清淡一些，可以尝试煲仔饭或者粥类如艇仔粥、瑶柱粥；如果想尝试更复杂的粤菜，比如烧味拼盘或脆皮烤鸭也是不错的选择。当然，具体还要看你所在的地方是否有地道的广东餐厅，有些地方可能还有特色的农家菜。希望你能享受你的午餐！',
response_metadata={'token_usage': {'completion_tokens': 239,
'prompt_tokens': 247, 'total_tokens': 486}, 'model_name': 'gpt-3.5-turbo',
'system_fingerprint': None, 'finish_reason': 'stop', 'logprobs': None})
```

通过这种方式，AI 不仅回顾了用户的喜好（粤菜），还能够提供具体的菜肴建议，这展现了对话上下文管理的重要性和效果。

MessagesPlaceholder 提供了一种高效的方式来管理和利用对话历史，这对于构建高质量的对话 AI 应用至关重要。通过合理使用这一机制，开发者可以创建出能够理解上下文、提供连贯对话和个性化建议的 AI 助手。随着对话历史的累积，AI 的响应将变得更加准确和人性化，极大地增强用户体验。

4.5 预设部分提示词变量

在开发人工智能应用时，经常需要根据用户的输入逐步构建对话内容，特别是在需要收集多个数据点来完成一个任务的场景中。prompt.partial 方法提供了一种灵活的解决方案，支持开发者预先设置部分提示词变量，然后在获取更多信息后补全剩余的部分。这种方法提高了应用的灵活性和用户交互的连贯性。

在对话应用中，用户可能不会一次性提供所有信息。例如，在一个资源推荐系统中，用户可能首先表达他们对某个主题的兴趣，然后在对话的后期指定希望获取资源的类型。在这种情况下，如果在一开始就知道了部分信息（如主题），就可以使用 prompt.partial 方法预先设置这部分已知的变量。

PromptTemplate 类提供了 partial 方法，支持开发者在创建提示模板时预先填充一部分变量。这些部分设置的变量随后可以在获取到更多信息时通过调用 format 方法补全。这种方法极大地提高了代码的可重用性和对话的灵活性。

假设正在开发一个智能教育平台，旨在为用户提供个性化的学习资源推荐。以下是如何使用 prompt.partial 方法实现这一功能的具体步骤。

（1）初始化提示模板。

首先，创建一个包含两个变量 topic（主题）和 resourceType（资源类型）的提示模板。

```
from langchain.prompts import PromptTemplate

prompt = PromptTemplate.from_template("为我推荐关于{topic}的{resourceType}。
")
```

（2）预先设置部分变量。

在用户提供感兴趣的主题后，可以使用 partial 方法预先设置 topic 变量。

```
topic = "数学"
partial_prompt = prompt.partial(topic=topic)
```

（3）补全并生成最终的提示。

当用户在对话的后期指定想要的资源类型（如“视频教程”）时，可以使用 format 方法补全剩余的变量并生成最终的提示，示例代码如下。

```
resourceType = "视频教程"
final_prompt = partial_prompt.format(resourceType=resourceType)
print(final_prompt)
```

使用 prompt.partial 方法，开发者可以在只知道部分信息的情况下预先设置提示模板的部分变量，然后在获得更多信息后补全并使用这个模板。这种方法不仅提高了对话应用的

灵活性和用户体验，还简化了代码的复杂性，使得开发更加高效。

最后使用链调用大模型查看效果，代码如下。

```
chain = partial_prompt | chat | output_parser

result = chain.invoke({"resourceType":"视频教程"})
print(result)
```

打印结果如下。

```
当然可以！以下是一些关于数学的视频教程推荐，适合不同水平的学习者：

1. **Khan Academy (可汗学院)**：这个网站提供了广泛的数学课程，从基础算术到微积分，都有详细的讲解和练习。很多视频都是以清晰的步骤指导你学习。
   - 链接：https://www.khanacademy.org/math

2. **MIT OpenCourseWare**：麻省理工学院提供的公开课程，包括一些数学课程的讲座视频，如线性代数、微积分等。
   - 链接：https://ocw.mit.edu/courses/

3. **Coursera**：提供各种大学级别的数学课程，由世界顶级大学教授讲授，适合深入学习和提升技能。
   - 链接：https://www.coursera.org/courses?query=mathematics

4. **YouTube (视频分享网站)**：
   - **The Organic Chemistry Tutor**：这个频道专注于高中和大学数学，特别是代数和几何。
   - **Numberphile**：如果你对数学的趣味性感兴趣，这里有很多有趣的数学概念和问题解释。
   - **MIT 18.01 Single Variable Calculus (2011)**：这是由麻省理工学院教授Lars Hillestrand的经典微积分课程视频。

5. **Bitesize**（英国在线学习平台）：提供针对不同年龄段的数学教程，适合初学者。
   - 链接：https://bitesize.com/guides/uk-schooldays/maths

根据你的具体需求和兴趣选择合适的内容。希望这些资源能帮助你学习数学！
```

本节示例演示了如何使用 prompt.partial 方法帮助构建更加智能和用户友好的对话系统。

4.6　动态预设提示词变量

在构建交互式人工智能应用时，根据上下文动态生成响应可以极大地提升用户体验。prompt.partial 方法的一个高级用法是传递函数作为变量，这种方式支持在实际需要生成提示时才动态计算变量的值。本节将深入探讨如何利用这一特性，以会议提醒和准备的场景为例，展示其在实际应用中的强大功能。

在对话系统中，用户需求往往随时间而变化，这要求系统能够灵活地根据当前的上下文提供相关信息。通过使用 prompt.partial 方法接收一个函数作为变量，可以确保只在需要时才进行计算，从而使提示信息尽可能地新鲜和相关。

考虑一个实际应用场景，助手需要根据会议时间动态提供提醒。提醒不仅包括会议的基本信息（如主题、参与人员和地点），还要根据当前时间相对于会议时间的不同阶段（如会议开始前一天、开始前 15 分钟等）提供不同的提醒消息。

```
from datetime import datetime, timedelta
from langchain.prompts import PromptTemplate

# 假设的会议详细信息
meeting_details = {
    "time": datetime.now() + timedelta(hours=1),
    "topic": "项目进度更新",
    "participants": ["张三", "李四", "王五"],
    "location": "在线会议 - Zoom",
}
```

```
# 动态生成提醒消息的函数
def reminder_for_meeting(meeting_time):
    now = datetime.now()
    if meeting_time - timedelta(minutes=15) <= now < meeting_time:
        return "会议将在15分钟后开始。"
    elif now < meeting_time - timedelta(days=1):
        return "明天有一个重要会议。"
    else:
        return "会议即将开始。"
```

在本例中，将 reminder_for_meeting 函数作为 reminder 变量的值传递给 prompt.partial，这样 reminder 的实际内容就会在最后生成提示时根据当前时间动态决定。

```
# 创建包含会议详情和提醒的模板
prompt_template = PromptTemplate.from_template("""
{reminder}
会议主题：{topic}
参与人员：{participants}
会议地点：{location}

{input}
""")

# 使用partial方法和lambda表达式动态设置提醒
partial_prompt = prompt_template.partial(
    reminder=lambda: reminder_for_meeting(meeting_details["time"]),
    topic=meeting_details["topic"],
    participants=", ".join(meeting_details["participants"]),
    location=meeting_details["location"],
)

# 生成最终的提示
final_prompt = partial_prompt.format(input="请帮我列出目前可以做的会议准备。")
print(final_prompt)
```

打印的最终提示词如下。

```
会议即将开始。
会议主题：项目进度更新
参与人员：张三，李四，王五
会议地点：在线会议 - Zoom

请帮我列出目前可以做的会议准备。
```

最后使用链调用大模型查看提示词效果，代码如下。

```
chain = partial_prompt | chat | output_parser

result = chain.invoke({"input":"请帮我列出目前可以做的会议准备。"})
print(result)
```

运行结果如下。

```
当然，以下是您需要进行的会议准备事项：

1. **发送会议邀请**：
   - 在Zoom上创建一个会话并分享给所有参与者，确保他们收到并确认参会时间。

2. **议程设定**：
   - 制定一份详细的议程，列出每个话题（例如项目进度更新、讨论问题及解决方案等）的预计讨论时间。

3. **准备好PPT或演示文稿**：
   - 如果有项目进度需要汇报，确保你准备了清晰、简洁的幻灯片或PowerPoint文件。

4. **分享资料**：
   - 对于可能用到的项目文档或报告，提前上传到Zoom或者链接共享平台，以便大家下载和参考。

5. **测试技术设备**：
   - 确保所有参会者都熟悉如何在Zoom上进行音频/视频会议、屏幕共享等操作。
```

```
6. **设置记录方式**：
   - 如果需要，可以安排专人做笔录或使用Zoom的录制功能。

7. **时间管理**：
   - 提前告知大家会议预计将持续多久，以便他们安排其他事项。

8. **准备好问题和反馈列表**：
   - 请参与者提前思考可能的问题，这样在会议上能更高效地进行互动。

9. **创建静音提示**：
   - 如果会议室嘈杂，可以设置静音模式或提醒大家关闭不必要的背景噪音。

10. **保持专业态度**：
    - 虽然在线，但要维持专业形象，确保会议环境整洁，避免聊天或其他无关话题。

完成以上准备工作后，会议就可以顺利进行了。祝你会议成功！
```

通过本节示例，读者可以掌握如何利用 prompt.partial 传递函数动态预设提示词变量的强大能力。这种方法使对话系统能够根据实时上下文提供精确和及时的信息，极大地提升了用户体验。此外，这种技术的应用不仅限于会议提醒场景，还可以扩展到任何需要根据动态条件生成响应的情境中，展现了 prompt.partial 在构建复杂对话逻辑中的灵活性和实用性。

4.7 管道提示词

本节将深入探讨如何使用 PipelinePrompt 有效组合多个提示词，以创建复杂而有组织的对话和任务流。PipelinePrompt 是一个强大的工具，可以重复使用部分提示词，实现更高

效的信息传递和任务执行。

在编写脚本或创建对话式 AI 应用时，经常需要组合多个提示词，以便根据特定场景或需求构建完整的对话流程。通过使用 PipelinePrompt，可以将不同的提示词模板组合成一个协调的流程，实现复杂的对话管理。

PipelinePrompt 主要由最终提示和管道提示两个部分组成。最终提示是最后返回的提示，而管道提示是一个包含字符串名称和提示模板的元组列表。这些提示模板将被格式化并作为变量传递给未来的提示模板，从而创建一个连贯的对话流。

现在通过一个简单的例子说明如何使用 PipelinePrompt。假设正在创建一个模拟与 Elon Musk 对话的应用。这时需要构建一个流程，首先介绍你正在扮演的人物（Elon Musk），然后展示一个交互示例，最后提示用户提出他们的问题。

（1）定义提示模板。

首先，定义不同的提示模板，分别对应于对话的不同阶段。

定义 introduction_template 介绍扮演的人物，示例代码如下。

```
introduction_template = """你正在扮演{person}。"""
```

定义 example_template 提供一个交互示例，示例代码如下。

```
example_template = """
下面是一个交互示例：

Q: {example_q}
A: {example_a}"""
```

定义 start_template 开始正式的交互，示例代码如下。

```
start_template = """现在正式开始！

Q: {input}
A: """
```

（2）组合提示模板。

使用 PipelinePromptTemplate，将这些模板组合成一个完整的流程。为此需要创建一个

元组列表，将每个模板的名称和模板本身作为元素添加到列表中，示例代码如下。

```
# 组合提示模板
pipeline_prompt = PipelinePromptTemplate(
    final_prompt=full_prompt,
    pipeline_prompts=[
        ("introduction", introduction_prompt),
        ("example", example_prompt),
        ("start", start_prompt),
    ]
)
```

（3）格式化和展示。

最后，通过传入特定的变量（如 person、example_q 等），格式化并展示最终的提示。这支持根据用户的输入或特定场景动态生成对话内容，示例代码如下。

```
# 格式化和展示
formatted_prompt = pipeline_prompt.format(
    person="Elon Musk",
    example_q="你最喜欢什么车？",
    example_a="Tesla",
    input="您最喜欢的社交媒体网站是什么?"
)

print(formatted_prompt)
```

使用链调用大模型查看提示词效果，示例代码如下。

```
chain = pipeline_prompt | chat | output_parser

result = chain.invoke({
    "input":"您最喜欢的社交媒体网站是什么",
    "person":"Elon Musk",
    "example_q":"你最喜欢什么车？",
    "example_a":"Tesla",
```

```
})
print(result)
```

运行后的结果如下。

```
'Twitter'
```

从结果可以看到，大模型通过问答学习到如何像 Elon Musk 一样简短地回答问题。

使用 PipelinePrompt 可以构建有序的复杂对话流程，这不仅提升了代码的可复用性和可维护性，还丰富了用户体验。无论是制作互动教学内容、开发对话 AI，还是优化代码结构，它都是一个宝贵的工具。