第 3 章

LangChain 的基础组件

随着 LLM 的快速发展，如何高效地构建 LLM 应用成为一个热点话题。LangChain 作为一个专为 LLM 应用开发而设计的框架，提供了一整套工具和组件来简化和加速开发流程，如图 3-1 所示。本章将深入探讨 LangChain 的核心概念和基础组件，包括模型、提示模板、索引、文档加载器、输出解析器等，并通过实例演示如何使用这些组件构建 LLM 应用。通过学习本章内容，读者将掌握使用 LangChain 进行 LLM 应用开发的基本技能，为后续章节中更复杂的主题打下坚实的基础。

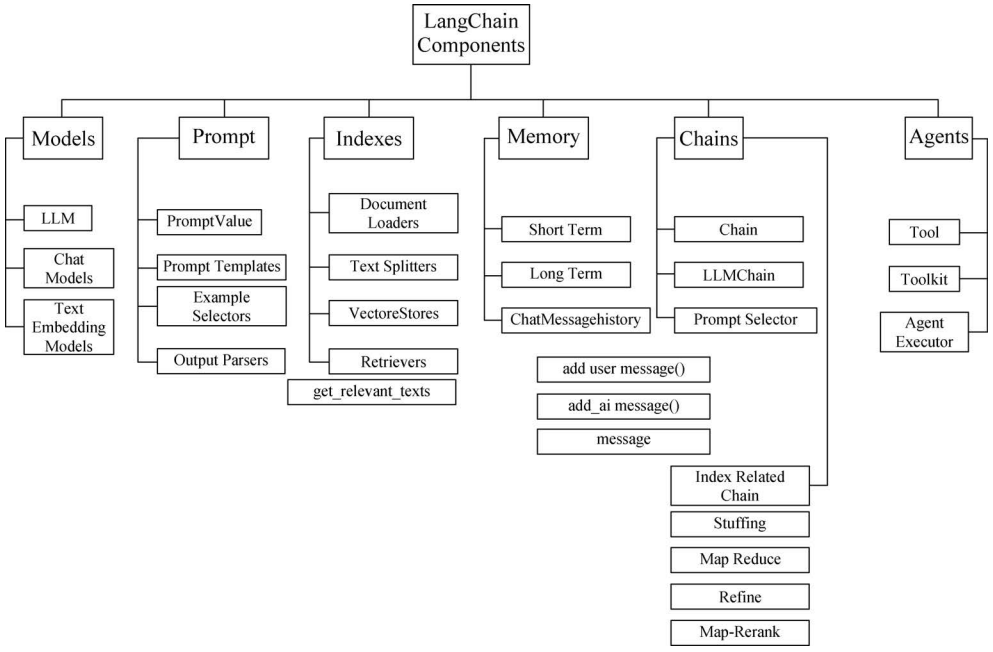

图 3-1　LangChain 的基础组件

3.1　快速入门案例

【例 3-1】 参考官网的案例，本节首先编写了一个类似于 hello world 的案例。这个案例只依赖于提示模板中的信息进行响应。然后，构建一个检索链，从单独的数据库中获取数据并将其传递到提示模板中。接下来，添加聊天历史记录，以创建对话检索链。这样可以以聊天的方式与这个 LLM 交互，因为它记住了之前的问题。最后，将构建一个代理——它利用 LLM 来确定是否需要获取数据来回答问题。这一案例展示了基础组件的一个基本应

用。代码请参考程序 3.1.py。

3.1.1 LLM 链

假设读者已经安装了 Gemma 本地模型，并确保 Ollama 服务器正在运行。如果没有运行，可以通过 ollama run gemma：2b 来加以运行。

```
from langchain_community.llms import Ollama
llm = Ollama(model = "gemma:2b")
```

接下来，可以调用它看看它是不是会给出一个很好的回应。

```
llm.invoke("how can langsmith help with testing?")
```

为了改善 LLM 的响应质量，可以使用提示模板来指导它。提示模板用于将原始用户输入转换为对 LLM 更友好的格式。下面是一个使用 ChatPromptTemplate 创建提示模板的示例。

```
from langchain_core.prompts import ChatPromptTemplate
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are world class technical documentation writer."),
    ("user", "{input}")
])
```

有了提示模板后，可以将其与 LLM 组合成一个简单的 LLM 链。

```
chain = prompt | llm
```

现在，可以调用这个链并问同样的问题。尽管它仍然可能无法准确回答问题，但它应该以一个技术写作人员更适当的语气进行回应。

```
chain.invoke({"input": "how can langsmith help with testing?"})
```

需要注意的是，ChatModel(以及由此构成的链)的输出是一条消息。为了更方便地处理输出，可以添加一个简单的输出解析器，将聊天消息转换为字符串。

```
from langchain_core.output_parsers import StrOutputParser
output_parser = StrOutputParser()
```

将输出解析器添加到之前的链中：

```
chain = prompt | llm | output_parser
```

现在，当我们调用链时，答案将是一个字符串(而不是 ChatMessage 对象)：

```
chain.invoke({"input": "how can langsmith help with testing?"})
```

这样，就成功地建立了一个基本的 LLM 链。

3.1.2 检索链

为了正确回答原始问题("how can langsmith help with testing?")，需要向 LLM 提供额外的上下文信息，这可以通过检索来实现。当用户拥有大量数据而无法直接传递给 LLM 时，检索就显得尤为重要。可以使用检索器获取最相关的部分，并将其传递给 LLM。

在这个过程中，将从检索器中查找相关文档，然后将它们传递给提示模板。检索器可以由任何数据源支持，例如，SQL 表、互联网等。在本例中，将填充一个向量存储，并将其用作

检索器。

首先，需要加载要索引的数据。为此，我们使用 WebBaseLoader。使用 WebBaseLoader 需要安装 BeautifulSoup。

```
pip install beautifulsoup4
```

安装完成后，导入并使用 WebBaseLoader。

```
from langchain_community.document_loaders import WebBaseLoader
loader = WebBaseLoader("https://docs.smith.langchain.com/user_guide")
docs = loader.load()
```

接下来，需要将数据索引到向量存储中。这需要两个组件：嵌入模型和向量存储。

对于嵌入模型，可以使用通过 API 访问的模型（如 OpenAI）或本地运行的模型（如 Ollama）。以下是使用 Ollama 嵌入模型的示例。

```
from langchain_community.embeddings import OllamaEmbeddings
embeddings = OllamaEmbeddings()
```

现在，可以使用这个嵌入模型将文档摄取到向量存储中。为了简单起见，将使用一个名为 FAISS 的本地向量存储。

首先，需要安装 FAISS 所需的包。

```
pip install faiss-cpu
```

然后，可以构建索引。

```
from langchain_community.vectorstores import FAISS
from langchain_text_splitters import RecursiveCharacterTextSplitter
text_splitter = RecursiveCharacterTextSplitter()
documents = text_splitter.split_documents(docs)
vector = FAISS.from_documents(documents, embeddings)
```

现在，已经在向量存储中建立了数据索引，可以创建一个检索链了。这个链将接收传入的问题，查找相关文档，然后将这些文档与原始问题一起传递给 LLM，并要求它回答原始问题。

设置一个链，它接收一个问题和检索到的文档，并生成答案。

```
from langchain.chains.combine_documents import create_stuff_documents_chain
prompt = ChatPromptTemplate.from_template("""根据提供的上下文回答以下问题：
<context>
{context}
</context>
问题：{input}""")
document_chain = create_stuff_documents_chain(llm, prompt)
```

也可以通过直接传递文档来手动运行这个链。

```
from langchain_core.documents import Document
document_chain.invoke({
    "input": "how can langsmith help with testing?",
    "context": [Document(page_content="langsmith can let you visualize test results")]
})
```

然而，这里希望文档首先来自刚刚设置的检索器。这样，对于给定的问题，可以使用检

索器动态选择最相关的文档并传递给链。

```
from langchain.chains import create_retrieval_chain
retriever = vector.as_retriever()
retrieval_chain = create_retrieval_chain(retriever, document_chain)
```

接下来可以调用这个检索链。它会返回一个字典，其中，LLM 的响应在 answer 键中。

```
response = retrieval_chain.invoke({"input": "how can langsmith help with testing?"})
print(response["answer"])
```

通过使用检索器提供相关上下文，得到的答案应该更加准确。这样就成功地建立了一个基本的检索链。

3.1.3 对话检索链

到目前为止，创建的链只能回答单个问题。在实际应用中，人们经常会构建聊天机器人，它需要能够处理多轮对话。那么，如何将检索链转换为可以回答后续问题的对话检索链呢？

下面仍然使用 create_retrieval_chain 函数，但需要做出以下两点改变。

(1) 检索方法不应该只考虑最近的输入，而应该将整个对话历史纳入考虑范围。

(2) 最终的 LLM 链同样应该考虑整个对话历史。

为了更新检索器，创建一个新的链。这个链将接收最近的输入(input)和对话历史(chat_history)，并使用 LLM 生成搜索查询。

```
from langchain.chains import create_history_aware_retriever
from langchain_core.prompts import MessagesPlaceholder
prompt = ChatPromptTemplate.from_messages([
    MessagesPlaceholder(variable_name = "chat_history"),
    ("user", "{input}"),
    ("user", "Given the above conversation, generate a search query to look up in order to get
information relevant to the conversation")
])
retriever_chain = create_history_aware_retriever(llm, retriever, prompt)
```

可以通过传入一个包含后续问题的对话历史来测试这个链。

```
from langchain_core.messages import HumanMessage, AIMessage
chat_history = [
    HumanMessage(content = "Can LangSmith help test my LLM applications?"),
    AIMessage(content = "Yes!")
]
retriever_chain.invoke({
    "chat_history": chat_history,
    "input": "Tell me how"
})
```

可以看到，这个链返回的是与在 LangSmith 中进行测试相关的文档。这是因为 LLM 生成了一个新的查询，将对话历史与后续问题结合在一起。有了这个新的检索器，就可以创建一个新的链来继续对话，并考虑检索到的文档。

```
prompt = ChatPromptTemplate.from_messages([
    ("system", "Answer the user's questions based on the below context:\n\n{context}"),
```

```
    MessagesPlaceholder(variable_name = "chat_history"),
    ("user", "{input}"),
])
document_chain = create_stuff_documents_chain(llm, prompt)
retrieval_chain = create_retrieval_chain(retriever_chain, document_chain)
```

现在就可以端到端地测试这个链了。

```
chat_history = [
    HumanMessage(content = "Can LangSmith help test my LLM applications?"),
    AIMessage(content = "Yes!")
]
retrieval_chain.invoke({
    "chat_history": chat_history,
    "input": "Tell me how"
})
```

可以看到，这个链给出了一个连贯的答案——我们已经成功地将检索链转换为对话检索链！通过考虑对话历史并使用检索器提供相关上下文，构建了一个能够进行多轮对话的聊天机器人。

3.1.4　代理

到目前为止，创建的都是链的示例，其中每个步骤都是预先确定的。接下来将创建一个代理，它可以使用 LLM 动态地决定采取什么步骤。

请注意，在这个示例中，展示的是如何使用 OpenAI 模型创建代理，因为本地模型的可靠性还不够高（但如果只是为了学习，可以对代码进行少量改动）。在构建代理时，首先要决定代理可以访问哪些工具。在本例中，将为代理提供以下两个工具。

（1）刚刚创建的检索器。这将让代理轻松回答有关 LangSmith 的问题。

（2）搜索工具。这将让代理轻松回答需要最新信息的问题。

首先，为刚刚创建的检索器设置一个工具。

```
from langchain.tools.retriever import create_retriever_tool
retriever_tool = create_retriever_tool(
     retriever,
     "langsmith_search",
     "Search for information about LangSmith. For any questions about LangSmith, you must use
this tool!",
)
```

这里将使用的搜索工具是 Tavily。使用 Tavily 需要一个 API 密钥（它们提供免费额度）。在它们的平台上创建 API 密钥后，需要将其设置为环境变量。

```
export TAVILY_API_KEY = …
```

如果不想设置 API 密钥，可以跳过创建这个工具。

```
from langchain_community.tools.tavily_search import TavilySearchResults
search = TavilySearchResults()
```

现在，可以创建一个工具列表，供代理使用。

```
tools = [retriever_tool, search]
```

有了这些工具，就可以创建一个代理来使用它们。下面简要介绍代理的创建过程。

首先安装 langchain hub：

```
pip install langchainhub
```

然后，可以使用 langchain hub 获取预定义的提示模板。

```
from langchain_openai import ChatOpenAI
from langchain import hub
from langchain.agents import create_openai_functions_agent
from langchain.agents import AgentExecutor
prompt = hub.pull("hwchase17/openai-functions-agent")
llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0)      # 如果基于本地模型，修改之
agent = create_openai_functions_agent(llm, tools, prompt)      # 如果基于本地模型，修改之
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)
```

现在可以调用这个代理，看看它如何响应。可以问它关于 LangSmith 的问题：

```
agent_executor.invoke({"input": "how can langsmith help with testing?"})
```

也可以问它与天气相关的问题：

```
agent_executor.invoke({"input": "what is the weather in SF?"})
```

通过使用代理，可以构建一个能够动态决定采取什么步骤的智能系统。代理可以根据问题的类型选择合适的工具，并利用这些工具生成最终的答案。

3.2 模型（Model I/O）

在开发任何语言模型应用时，模型本身无疑是最核心的元素。LangChain 为开发者提供了一系列工具和抽象，使得与语言模型的交互变得更加简单和高效。本节将重点介绍 LangChain 中的语言模型类型、与模型交互的最佳实践，以及用于构建模型输入和处理模型输出的辅助工具。

3.2.1 简介

LangChain 集成了两种主要类型的语言模型：大语言模型和聊天模型。它们的区别主要在于输入和输出的格式。LLM 接收字符串格式的提示作为输入，并生成字符串格式的完成作为输出，如 OpenAI 的 GPT-3；而聊天模型接收一个聊天消息列表作为输入，并返回一个 AI 消息作为输出，如 GPT-4 和 Anthropic 的 Claude-3。尽管聊天模型通常也是基于 LLM 构建的，但它们经过了专门的调整和优化，以更好地适应对话场景。

在选择语言模型时，开发者需要仔细权衡不同模型的特点和适用场景。虽然 LangChain 提供了一致的接口来处理不同类型的模型，但这并不意味着所有模型都可以互换使用。不同的模型可能需要采用不同的提示策略和优化手段。例如，Anthropic 的模型更适合使用 XML 格式的提示，而 OpenAI 的模型则更适合 JSON 格式。此外，LangChain 提供的默认提示模板可能并不适用于所有模型。开发者需要根据实际使用的模型，对提示模板进行必要的调整和优化。

在聊天模型中，消息是一个非常重要的概念。聊天模型接收一个消息列表作为输入，并

返回一个 AI 生成的消息作为输出。每个消息都包含角色(如用户、助手等)和内容两个属性。内容可以是字符串、字典列表(用于多模态输入)等不同形式。此外,消息还可以携带额外的元数据,如上下文信息、特定于提供商的参数等,这些信息可以通过 additional_kwargs 属性传递。

提示模板是将用户输入转换为语言模型可接受格式的关键工具。在实际应用中,用户的原始输入通常需要经过一定的转换和处理,才能成为合适的模型输入。提示模板定义了这个转换过程,将用户输入与必要的上下文、指令等信息结合,生成最终的提示。LangChain 提供了多种提示模板的抽象,如 ChatPromptTemplate、HumanMessagePromptTemplate 等,方便开发者使用。

输出解析器用于将语言模型的原始输出转换为更易于处理和使用的格式。模型的输出可能是字符串或消息,其中包含以特定格式组织的信息,如逗号分隔列表、JSON 等。输出解析器负责提取和转换这些信息,使其更容易被下游的应用逻辑所使用。LangChain 提供了多种内置的输出解析器,如 StrOutputParser(用于字符串输出)、OpenAI Functions Parsers(用于处理 OpenAI 的函数调用)、Agent Output Parsers(用于将原始输出转换为代理可执行的动作)等。

LangChain 表达式语言(LCEL)是一种强大的工具,用于以声明式的方式组合语言模型应用的各个组件。通过使用“|”操作符,开发者可以将提示模板、语言模型、输出解析器等组件连接起来,形成一个完整的处理管道。这种声明式的组合方式使得应用的结构更加清晰和模块化,提高了代码的可读性和可维护性。LCEL 的实现依赖于 Runnable 接口,该接口定义了组件之间的统一输入和输出格式,使得组件可以无缝地连接和协作。

```
template = "Generate a list of 5 {text}.\n\n{format_instructions}"
chat_prompt = ChatPromptTemplate.from_template(template)
chat_prompt = chat_prompt.partial(format_instructions = output_parser.get_format_instructions())
chain = chat_prompt | chat_model | output_parser
chain.invoke({"text": "colors"})
```

在这个示例中,首先定义了一个提示模板,然后使用 ChatPromptTemplate 创建了一个聊天提示。接着,使用 partial()方法将输出解析器的格式化指令注入提示中。最后,使用“|”操作符将提示、聊天模型和输出解析器组合成一个完整的处理链,并调用 invoke()方法来执行这个链。

3.2.2　提示模板

提示模板是为语言模型生成提示的预定义方案。模板可能包括说明、少样本示例以及适合特定任务的上下文和问题。LangChain 提供了创建和使用提示模板的工具。LangChain 致力于创建与模型无关的模板,以便于跨不同语言模型重用现有模板。通常,语言模型期望提示要么是字符串,要么是聊天消息列表。

PromptTemplate 使用 PromptTemplate 为字符串提示创建模板。默认情况下,PromptTemplate 使用 Python 的 str.format 语法进行模板化。例如:

```
from langchain.prompts import PromptTemplate
prompt_template = PromptTemplate.from_template(
    "Tell me a {adjective} joke about {content}."
```

```
)
prompt_template.format(adjective = "funny", content = "chickens")
```

即“Tell me a funny joke about chickens”。模板支持任意数量的变量,包括没有变量的情况。例如:

```
from langchain.prompts import PromptTemplate
prompt_template = PromptTemplate.from_template("Tell me a joke")
prompt_template.format()
```

可以创建以任何方式格式化提示的自定义提示模板。

ChatPromptTemplate 聊天模型的提示是一个聊天消息列表。每个聊天消息都与内容相关联,以及一个名为角色的附加参数。例如,在 OpenAI Chat Completions API 中,聊天消息可以与 AI 助手、人类或系统角色相关联。像这样创建聊天提示模板:

```
from langchain_core.prompts import ChatPromptTemplate
chat_template = ChatPromptTemplate.from_messages(
    [
        ("system", "You are a helpful AI bot. Your name is {name}."),
        ("human", "Hello, how are you doing?"),
        ("ai", "I'm doing well, thanks!"),
        ("human", "{user_input}"),
    ]
)
messages = chat_template.format_messages(name = "Bob", user_input = "What is your name?")
```

ChatPromptTemplate.from_messages 接收各种消息表示形式。例如,除了使用上面的(type,content)这种二元组表示之外,还可以传递 MessagePromptTemplate 或 BaseMessage 的实例。

```
from langchain.prompts import HumanMessagePromptTemplate
from langchain_core.messages import SystemMessage
from langchain_openai import ChatOpenAI
chat_template = ChatPromptTemplate.from_messages(
        [
            SystemMessage(
                content = (
                    "You are a helpful assistant that re - writes the user's text to "
                    "sound more upbeat."
                )
            ),
            HumanMessagePromptTemplate.from_template("{text}"),
        ]
    )
    messages = chat_template.format_messages(text = "I don't like eating tasty things")
    print(messages)
    [SystemMessage(content = "You are a helpful assistant that re - writes the user's text to
sound more upbeat."), HumanMessage(content = "I don't like eating tasty things")]
```

LangChain 提供了一个用户友好的界面,用于将提示的不同部分组合在一起。可以对字符串提示或聊天提示执行此操作,以这种方式构建提示允许轻松重用组件。

(1) 字符串提示组合。使用字符串提示时,每个模板会被组合在一起。读者可以直接使用提示或字符串(列表中的第一个元素必须是提示)。

【例 3-2】　字符串提示组合(参考代码 3.2.py)。

```
from langchain.prompts import PromptTemplate
prompt = (
    PromptTemplate.from_template("Tell me a joke about {topic}")
    + ", make it funny"
    + "\n\n and in {language}"
)
PromptTemplate(input_variables = ['language', 'topic'], output_parser = None, partial_
variables = {}, template = 'Tell me a joke about {topic}, make it funny\n\nand in {language}',
template_format = 'f - string', validate_template = True)
prompt.format(topic = "sports", language = "spanish")
```

(2) 聊天提示组合。聊天提示由一个消息列表组成。纯粹为了开发人员体验,我们添加了一种方便的方式来创建这些提示。在此管道中,每个新元素都是最终提示中的一条新消息。例如:

```
from langchain_core.messages import AIMessage, HumanMessage, SystemMessage
First, let's initialize the base ChatPromptTemplate with a system message. It doesn't have
to start with a system, but it's often good practice
prompt = SystemMessage(content = "You are a nice pirate")
```

然后,可以创建一个管道,将其与其他消息或消息模板组合在一起。当没有要格式化的变量时,使用 Message;当有要格式化的变量时,使用 MessageTemplate。也可以只使用一个字符串(注意:将自动被推断为 HumanMessagePromptTemplate)。

```
new_prompt = (
    prompt + HumanMessage(content = "hi") + AIMessage(content = "what?") + "{input}"
)
```

以下代码将创建一个 ChatPromptTemplate 类的实例。

```
new_prompt.format_messages(input = "I said hi")
[SystemMessage(content = 'You are a nice pirate', additional_kwargs = {}),
HumanMessage(content = 'hi', additional_kwargs = {}, example = False),
AIMessage(content = 'what?', additional_kwargs = {}, example = False),
HumanMessage(content = 'i said hi', additional_kwargs = {}, example = False)]
```

(3) 少样本提示模板是一种利用少量示例来动态生成提示的技术。它可以帮助语言模型更好地理解任务,并根据具体输入生成相关的响应。接下来,使用本地大模型 Gemma 来演示如何构建和使用少样本提示模板。少样本提示模板的核心思想是利用一些预先定义的示例来告知模型如何完成任务。这些示例通常包含一个输入和一个相应的输出。当我们给模型一个新的输入时,模型会根据这些示例来推断出所需的输出格式和风格。

构建少样本提示模板通常需要以下步骤。

① 准备一组有代表性的示例,每个示例包含一个输入和一个输出。

② 定义一个示例格式化函数,用于将示例转换为字符串形式。

③ 创建一个 FewShotPromptTemplate 对象,传入示例和格式化函数。

④ 使用 FewShotPromptTemplate 对象的 format()方法,传入新的输入,生成最终的提示。

【例 3-3】 少样本提示模板(参考代码 3.3.py)。

```
from langchain.prompts.few_shot import FewShotPromptTemplate
from langchain.prompts.prompt import PromptTemplate
from langchain_community.llms import Ollama as OllamaLLM
examples = [
    {
        "sentence": "今天天气真不错,我想出去走走。",
        "keywords": "天气, 出去走走"
    },
    {
        "sentence": "我正在学习编程,希望能尽快找到一份相关的工作。",
        "keywords": "学习编程, 找工作"
    },
    {
        "sentence": "医生建议我每天坚持锻炼,并且要保证充足的睡眠。",
        "keywords": "坚持锻炼, 充足睡眠"
    }
]
example_prompt = PromptTemplate(
    input_variables=["sentence", "keywords"],
    template="句子: {sentence}\n关键词: {keywords}"
)
prompt = FewShotPromptTemplate(
    examples=examples,
    example_prompt=example_prompt,
    suffix="句子: {input}\n关键词:",
    input_variables=["input"],
)
model = OllamaLLM(model="gemma:2b")
input_sentence = "我打算周末去海边旅行,好好放松一下。"
final_prompt = prompt.format(input=input_sentence)
print(final_prompt)
output = model.invoke(final_prompt)
print(output)
```

在上述程序中,首先构建了一系列示例,每个示例都由一句话及其相应的关键词组成。目的在于训练模型识别并提取句子中的核心信息。随后,程序初始化了一个名为 example_prompt 的 PromptTemplate 对象,该对象规定了示例的呈现格式。具体来说,格式要求每个示例以“句子:”为前缀,紧接着是具体的句子内容,换行后以“关键词:”开始,并列出相应的关键词。进一步,构建了 FewShotPromptTemplate 对象,它采用三个关键参数:examples 为已准备好的示例集合;example_prompt 为定义示例展示方式的模板;suffix 为附加在所有示例后面的部分,通常用于引入新的输入变量。接下来,程序中创建了一个名为 Gemma 的模型实例,该模型负责根据给出的示例和新输入生成关键词。定义了一个 input_sentence,表示待提取关键词的新句子。通过调用 prompt.format()方法并传入 input_sentence,生成了完整的提示文本,其中包括所有已格式化的示例及新加入的句子。将此提示文本输入 Gemma 模型后,模型依据提供的示例和新句子输出相应的关键词。程序最终展示了模型生成的关键词结果。

(4) PipelinePrompt 是一种用于构建复杂提示模板的强大工具。它主要由两部分组成:最终提示和管道提示。最终提示是整个管道的输出,而管道提示则是一系列中间步骤,

每个步骤都由一个字符串名称和一个提示模板组成。在执行过程中，每个管道提示都会被格式化，然后将格式化后的结果作为具有相同名称的变量传递给下一个提示模板，直到生成最终的提示。

【例 3-4】 PipelinePrompt 演示（参考代码 3.4.py）。

```
from langchain.prompts.pipeline import PipelinePromptTemplate
from langchain.prompts.prompt import PromptTemplate
from langchain_community.llms import Ollama as OllamaLLM
#定义最终提示模板
final_template = """{greeting}
{body}
{signature}"""
final_prompt = PromptTemplate.from_template(final_template)
#定义管道提示模板
greeting_template = "尊敬的{name}女士/先生:"
greeting_prompt = PromptTemplate.from_template(greeting_template)
body_template = """我们诚挚地邀请您参加将于{date}在{location}举办的{event}。作为{industry}领域的佼佼者，您的到来将为本次活动增光添彩。
本次活动的主题是"{theme}"，我们相信这个主题一定能引起您的兴趣。我们准备了精彩的议程和演讲嘉宾阵容，期待与您和业界同仁共同探讨{industry}的未来发展方向。
如果您需要任何进一步的信息或协助，请随时与我们联系。我们非常期待您的参与！"""
body_prompt = PromptTemplate.from_template(body_template)
signature_template = """此致
敬礼
{sender}
{title}, {company}"""
signature_prompt = PromptTemplate.from_template(signature_template)
#组合管道提示
pipeline_prompts = [
    ("greeting", greeting_prompt),
    ("body", body_prompt),
    ("signature", signature_prompt),
]
#创建 PipelinePromptTemplate
pipeline_prompt = PipelinePromptTemplate(
    final_prompt = final_prompt, pipeline_prompts = pipeline_prompts
)
print(pipeline_prompt.input_variables)
#格式化 PipelinePromptTemplate
input_data = {
    "name": "张",
    "date": "2023 年 9 月 1 日",
    "location": "北京",
    "event": "全球人工智能峰会",
    "industry": "人工智能",
    "theme": "AI 赋能，智创未来",
    "sender": "李明",
    "title": "市场部经理",
    "company": "ABC 科技有限公司"
}
final_prompt_str = pipeline_prompt.format( ** input_data)
print(final_prompt_str)
#使用 Gemma 模型生成个性化邮件
```

```
model = OllamaLLM(model = "gemma:2b")
result = model.invoke(final_prompt_str)
print(result)
```

上述示例展示了一个分步骤构建复杂提示模板的过程，其主要步骤如下。

首先，定义了一个综合提示模板 final_prompt，该模板由三个关键部分组成：问候语、邮件正文和签名。这一步骤为我们构建一个具有明确结构的邮件内容奠定了基础。

随后，为了实现这一结构，设计了三个专用的管道提示模板：greeting_prompt 生成问候语，body_prompt 负责邮件正文，而 signature_prompt 则用于创建签名。每个模板都定义了自己所需的输入变量，确保了每一部分都能独立并准确地表达其预定的信息。

接着，将这些管道提示模板依序组织到一个列表 pipeline_prompts 中，这一列表将作为生成最终邮件内容的蓝图。为了将这些独立的部分融合成一个连贯的整体，创建了 PipelinePromptTemplate 对象，并将最终提示模板及管道提示模板列表传入。通过这一操作，将各个部分按顺序拼接起来，形成了一个完整的邮件内容生成流程。通过查询 pipeline_prompt.input_variables，得以确认构建这一邮件所需的全部输入变量，这一步骤确保了我们在生成过程中不遗漏任何必要的信息。

接下来，准备了包含所需所有输入变量及其对应值的字典 input_data。通过执行 pipeline_prompt.format(** input_data)，所有输入变量被逐一填充到相应的管道提示模板中，进而生成了一串完整的提示文本。

最后，将这一生成的提示文本交给 Gemma 模型处理，模型据此生成了一封具有个性化特征的邮件内容。

通过这一示例，得以窥见 PipelinePrompt 的强大功能。它让我们能够将复杂的提示模板拆解为若干个简单的组件，每个组件都可以独立格式化并组合。这种模块化的方法极大地简化了复杂提示模板的构建和维护工作。更重要的是，PipelinePrompt 提供了极高的灵活性，允许我们根据不同的应用场景定制管道的结构和内容。

在深入探索中文语言模型的应用时，构建精准而有效的提示词模板显得尤为关键。以下是几种精心设计的模板实例，旨在引导模型更准确地理解和回应各种查询。

(1) 简单的问答模板。

通过定义明确的问答对，可以培养模型对具体问题给出精确答案的能力。例如，创建一个模板，其中包含一系列问题及其对应答案，最后搭配一个待解答的问题。这种模式不仅适用于静态知识点的查询，也能够促进模型对上下文信息的理解和应用。

```
from langchain.prompts import PromptTemplate
template = """
```

根据以下问题和答案，回答最后的问题。

问题：中国的首都是哪里？

答案：中国的首都是北京。

问题：上海有哪些著名的旅游景点？

答案：上海有东方明珠电视塔、外滩、豫园、南京路步行街等著名景点。

问题：中国台湾的最高山峰是哪座山？

答案：中国台湾的最高山峰是玉山，海拔 3952 米。

问题：{input}
答案："""

```
prompt = PromptTemplate(input_variables=["input"], template=template)
final_prompt = prompt.format(input="长江三峡都包括哪些景点?")
```

(2) 角色扮演模板。

利用角色扮演模板，能够模拟具体角色的语言风格和知识体系，从而实现更加自然和专业的对话体验。例如，设定一个场景，其中用户扮演学生，模型扮演一位中国历史老师，用通俗易懂的语言讲解历史知识。

```
from langchain.prompts import ChatPromptTemplate, HumanMessagePromptTemplate,
SystemMessagePromptTemplate
chat_prompt = ChatPromptTemplate.from_messages([
    SystemMessagePromptTemplate.from_template("你是一位中国历史老师,要用通俗易懂的语言向
学生讲解历史知识。"),
    HumanMessagePromptTemplate.from_template("{input}")
])
output = chat_prompt.format_prompt(input="请讲讲秦始皇统一中国的过程。").to_messages()
print(output)
```

(3) 动态选择示例的少样本提示。

在少样本学习场景中，选择与查询语义相近的示例对于提升模型性能至关重要。这种方法利用了语义相似度选择器和向量存储技术，从一组预定义的问题和答案中选择与输入最为相似的示例。这种动态选择机制使模型能够针对特定查询，利用最相关的上下文信息来提高答案的准确性和相关性。例如，从涵盖中国古典文学、历史人物和著名故事的示例库中选择最合适的示例，以回答“三国演义中，刘备的结拜兄弟是谁?”这样的问题。

【例 3-5】 动态选择示例的少样本提示(参考代码 3.5.py)。

```
from langchain.prompts import FewShotPromptTemplate,
PromptTemplate, SemanticSimilarityExampleSelector
from langchain_community.vectorstores import Chroma
from langchain_community.embeddings import OllamaEmbeddings
#示例集合
examples = [
    {"input": "红楼梦的作者是谁?", "output": "曹雪芹"},
    {"input": "《西游记》中唐僧的徒弟都有谁?", "output": "孙悟空、猪八戒、沙僧,以及白龙马。"},
    #假设在这里添加更多的示例…
]
#初始化示例选择器
example_selector = SemanticSimilarityExampleSelector.from_examples(
    examples,
    OllamaEmbeddings(model="gemma:2b"),
    Chroma,
    k=1
)
#构建少样本提示模板
few_shot_prompt = FewShotPromptTemplate(
    example_selector=example_selector,
    example_prompt=PromptTemplate(input_variables=["input", "output"], template="问题:
{input}\n答案:{output}"),
    prefix="根据以下示例问题和答案,回答最后的问题:\n",
```

```
    suffix = "\n 问题:{input}\n 答案:",
    input_variables = ["input"],
)
#定义一个新的问题
new_question = "三国演义中,刘备的结拜兄弟是谁?"
#格式化新问题以生成完整的提示
formatted_prompt = few_shot_prompt.format(input = new_question)
print(formatted_prompt)
```

在这段代码中,formatted_prompt 将包含一个完整的提示,它基于少数示例和新问题生成。这个提示可以被用来指导一个语言模型生成对新问题的答案。注意,这个程序假定存在一套工作流来处理和利用 OpenAIEmbeddings 和 Chroma,这通常需要访问相应的 API 和服务。此外,这段代码是概念性的示例,它展示了如何使用 LangChain 的工具来构建少样本学习的应用,但它不包括与真实模型交互的部分。在实际应用中,需要将生成的提示传给一个支持的大语言模型来获取答案。

设计高效的提示模板是开发语言模型应用的核心环节,它决定了模型输出的质量和准确性。为了创建一个高效的提示模板,开发者需要精心设计模板的内容,确保它既简洁又能提供足够的上下文信息。这包括明确输入变量的定义、精心规划模板的结构,并确保输入变量能够被有效地整合到模板中。在不同的应用场景下,例如聊天模型,选取恰当的消息类型(如 HumanMessage、SystemMessage 和 AIMessage)对于满足特定任务的需求至关重要。

选取适合的示例对于模型的性能有着显著影响,尤其是在进行少样本学习时。开发者可以采用多种方法来优化示例的选择,包括手动挑选、基于语义相似度的自动挑选,或利用向量存储技术。LangChain 为开发者提供了多样化的工具,包括 ChatPromptTemplate、FewShotPromptTemplate 和 PipelinePromptTemplate 等,使得构建既复杂又灵活的语言模型应用成为可能。

提示模板的设计是一个动态的迭代过程,需要基于模型的实际输出效果进行持续的优化。这一过程可能涉及人工评估、对比实验等多种方法,目的是不断提高模板的性能。综上所述,开发高品质的提示模板需要考虑诸多因素,包括任务的具体需求、模型的特性以及示例的选取等。幸运的是,LangChain 提供的一系列工具和组件可以帮助开发者以更高效、灵活的方式进行工作,大大简化了这一过程。

3.2.3 聊天模型

在 LangChain 框架内,聊天模型被视为核心组件之一,它区别于传统的仅基于纯文本输入和输出的语言模型。聊天模型的独特之处在于,它接收聊天消息作为输入,并以聊天消息的形式返回输出,为用户提供了更加自然和互动的交流体验。LangChain 通过集成多个模型提供商,如 OpenAI、Cohere、Hugging Face 等,提供了一个统一的接口,使得与这些不同模型的交互变得无缝且灵活。此外,LangChain 支持多种使用模式,包括同步、异步、批处理和流式模式,并引入了缓存等附加功能,以优化模型的使用效率和成本。

聊天模型作为语言模型的一种特殊形式,采用了与传统模型略有不同的接口设计。其核心在于将输入和输出都视为“聊天消息”,而非简单的文本串。这种基于消息的接口设计,让聊天模型能够更好地应用于对话场景。LangChain 支持的消息类型包括 AIMessage、HumanMessage、SystemMessage、FunctionMessage 和 ChatMessage,其中,ChatMessage 允

许接收任意角色参数，但在大多数场景下，开发者只需关注 HumanMessage、AIMessage 和 SystemMessage 即可。

随着技术的进步，越来越多的聊天模型开始提供函数调用 API，使得模型不仅能处理文本交流，还能基于描述的函数及其参数返回结构化的输出。这种功能极大地扩展了聊天模型的应用范围，使其能够更加灵活地与外部工具和系统集成，执行复杂任务。

LangChain 提供了许多实用程序，使函数调用变得容易。即它带有将函数绑定到模型的简单语法；用于将各种类型的对象格式化为预期函数模式的转换器；用于从 API 响应中提取函数调用的输出解析器；用于从模型获取结构化输出的链，建立在函数调用之上。下面主要介绍前两种方式。

第一种方式是函数绑定。许多模型都实现了辅助方法，用于处理不同函数对象的格式化和绑定。下面以 Pydantic 函数模式为例，演示如何将其与本地大模型 Gemma 进行绑定。注意，本示例使用了 OpenAI 的 API。

```
from langchain_core.pydantic_v1 import BaseModel, Field
#请注意,这里的 docstrings 至关重要,因为它们将与类名一起传递给模型
class Multiply(BaseModel):
    """将两个整数相乘。"""
    a: int = Field(..., description="第一个整数")
    b: int = Field(..., description="第二个整数")
```

可以使用 ChatOpenAI.bind_tools()方法来处理将 Multiply 转换为 OpenAI 函数并将其绑定到模型(即每次调用模型时都传递它)。

```
from langchain_openai import ChatOpenAI
llm = ChatOpenAI(model="gpt-3.5-turbo-0125", temperature=0)
llm_with_tools = llm.bind_tools([Multiply])
llm_with_tools.invoke("3 * 12 是多少?")
```

进一步地，可以添加一个工具解析器，从生成的消息中提取工具调用到 JSON。

```
from langchain_core.output_parsers.openai_tools import JsonOutputToolsParser
tool_chain = llm_with_tools | JsonOutputToolsParser()
tool_chain.invoke("3 * 12 是多少?")
```

或者返回原始的 Pydantic 类：

```
from langchain_core.output_parsers.openai_tools import PydanticToolsParser
tool_chain = llm_with_tools | PydanticToolsParser(tools=[Multiply])
tool_chain.invoke("3 * 12 是多少?")
```

如果想强制使用某个工具(并且只使用一次)，可以设置 tool_choice 参数。

```
llm_with_multiply = llm.bind_tools([Multiply], tool_choice="Multiply")
llm_with_multiply.invoke("如果你想的话,可以编造一些数字,但我不强迫你")
```

如果需要直接访问函数模式，LangChain 有一个内置的转换器，可以将 Python 函数、Pydantic 类和 LangChain 工具转换为 OpenAI 格式的 JSON 模式，相关方法可以参考 LangChain 的帮助文档。

LangChain 为聊天模型提供了可选的缓存层，包括内存缓存和 SQL 缓存等，这么做有以下两点好处。其一是如果经常多次请求相同的完成，它可以通过减少对 LLM 提供商的 API 调用次数来节省资金。其二是它可以通过减少对 LLM 提供商的 API 调用次数来加

速应用程序。下面的代码片段演示了这么做的好处，注意请在 Jupyter 中运行程序。

```
from langchain.globals import set_llm_cache
from langchain_openai import ChatOpenAI
llm = ChatOpenAI()
#内存缓存
%%time
from langchain.cache import InMemoryCache
set_llm_cache(InMemoryCache())
#第一次，它还不在缓存中，所以应该需要更长时间
llm.predict("讲个笑话")
%%time
#第二次，它在缓存中，所以速度更快
llm.predict("讲个笑话")
```

LangChain 为开发者提供了一个功能强大且灵活的框架，使人们能够轻松地创建和定制聊天模型。通过定义多种消息类型，如 SystemMessage、HumanMessage、AIMessage 等，LangChain 帮助人们清晰地区分聊天过程中的不同角色和内容，从而更好地组织和管理聊天流程。此外，LangChain 的函数调用功能允许在聊天过程中动态地调用外部函数或工具，大大扩展了聊天模型的能力，使其能够与外部系统交互完成更复杂的任务。

LangChain 还为聊天模型定义了标准接口，如 ChatModel 和 BaseChatModel，便于将自定义的聊天模型集成到 LangChain 生态系统中，并利用其提供的丰富工具和功能。支持流式传输和异步编程的特性，使得 LangChain 非常适合需要低延迟和高交互性的应用场景，如实时对话系统。内置的缓存机制不仅提高了系统响应速度，还有助于节省 API 调用次数，降低成本。更重要的是，LangChain 允许开发者自定义聊天模型的实现细节，提供了极大的灵活性来满足特定需求。

例如，在客服聊天系统中，可以利用 SystemMessage 来定义客服角色的行为规范，使用 HumanMessage 和 AIMessage 来模拟客户与客服之间的互动，并通过函数调用接入客户信息数据库和订单系统。在写作助手应用中，则可以利用流式传输让用户实时查看内容生成，通过缓存机制个性化调整 LLM 的输出以适应用户的写作历史和偏好。

总而言之，LangChain 提供了一整套完备的工具和框架，以支持构建功能全面、灵活可扩展的聊天模型。开发者可以借此发挥创造力，设计出满足实际需求的智能对话系统，为用户带来优质的交互体验。随着不断的实践和优化，我们有信心能够打造出表现卓越的聊天应用。

3.2.4 大语言模型

LLM 是 LangChain 的核心组件。LangChain 本身并不提供 LLM，而是为与许多不同的 LLM 交互提供了一个标准接口。具体来说，该接口接收一个字符串作为输入并返回一个字符串。

目前有许多 LLM 提供商，如 OpenAI、Cohere、Hugging Face 等。LLM 类旨在为所有这些提供商提供一个标准的交互接口，使得人们可以方便地切换和比较不同的 LLM。

如果使用 OpenAI，那么需要安装 OpenAI 的 Python 包：pip install openai。访问 OpenAI 的 API 需要一个 API 密钥。可以通过创建一个账户并访问 OpenAI 官网获取密钥。拿到密钥后，需要将其设置为环境变量：export OPENAI_API_KEY＝"你的 API 密

钥"。如果不想设置环境变量，也可以在初始化 OpenAI LLM 类时直接通过参数 openai_api_key 传入密钥：

```
from langchain_openai import OpenAI
llm = OpenAI(openai_api_key = "你的 API 密钥")
```

否则，可以直接初始化，不需要任何参数。

```
from langchain_openai import OpenAI
llm = OpenAI()
```

如果使用 Gemma 本地大模型，需要下载 Gemma 模型的权重文件。可以从 Gemma 官方仓库下载最新的模型权重。下载完成后，将权重文件放到工作目录下。然后可以初始化 Gemma LLM。

```
from langchain_community.llms import Ollama as OllamaLLM
llm = OllamaLLM(model_path = "path/to/gemma/weights")
```

这里的 model_path 参数指定了 Gemma 权重文件的路径。

使用 Gemma LLM 有了 Gemma LLM 实例，就可以像使用其他 LLM 一样调用它：llm.invoke("请写一段励志的话")。结果会显示一段励志的文本(略)。

如果想使用自己的 LLM 或者 LangChain 尚未支持的其他 LLM 包装器，可以通过创建自定义 LLM 实现。自定义 LLM 需要实现以下三个方法和属性，其中两个是必要的，一个是可选的。

(1) _call：一个必要方法，接收字符串输入、可选的停止词，并返回字符串。

(2) _llm_type：一个必要属性，返回字符串，仅用于记录日志。

(3) _identifying_params：一个可选属性，用于帮助打印该类的属性，应返回一个字典。

下面的代码展示了如何实现一个简单的自定义大语言模型(LLM)，这个自定义模型非常基础，它仅返回输入字符串的前 n 个字符。这种类型的 LLM 可以用于测试或演示目的，帮助理解如何在 LangChain 框架下创建和使用自定义 LLM。

【例 3-6】 自定义大语言模型(代码参见 3.6.py)。

```
from typing import Any, List, Mapping, Optional
from langchain_core.callbacks.manager import CallbackManagerForLLMRun
from langchain_core.language_models.llms import LLM
class CustomLLM(LLM):
    """一个简单的自定义 LLM,返回输入的前 n 个字符。"""
    n: int  #定义一个属性 n,用来指定返回字符的数量
    @property
    def _llm_type(self) -> str:
        """返回 LLM 的类型,用于日志记录和调试。"""
        return "自定义 LLM"
    def _call(
        self,
        prompt: str,
        stop: Optional[List[str]] = None,
        run_manager: Optional[CallbackManagerForLLMRun] = None,
        **kwargs: Any,
    ) -> str:
        """处理输入,返回前 n 个字符。"""
        if stop is not None:
```

```
            raise ValueError("停止词参数不被允许。")        # 如果提供了停止词参数,则
                                                          # 抛出异常
        return prompt[: self.n]                           # 返回输入字符串的前 n 个字符
    @property
    def _identifying_params(self) -> Mapping[str, Any]:
        """返回用于识别 LLM 的参数,有助于日志记录和调试。"""
        return {"n": self.n}
# 使用自定义 LLM
llm = CustomLLM(n=10)
response = llm.invoke("这是一个测试输入")
print(response)                                           # 输出: 这是一个测试
# 打印 LLM 的详细信息
print(llm)
```

以上代码展示了一个自定义 LLM 的基本结构和实现方式。这个自定义 LLM 利用了 LangChain 框架中的 LLM 基类,通过重写_call()方法来实现具体的逻辑。此外,它通过_llm_type 和_identifying_params 属性提供了关于 LLM 类型和参数的信息,这对于调试和日志记录非常有用。

下面继续探讨如何创建一个更复杂的自定义 LLM——CustomGemma。这个自定义 LLM 模拟了使用 Gemma 模型,并允许动态设置生成参数,例如,温度(temperature)和最大长度(max_length)。这种方式为使用 LLM 提供了更多灵活性和控制力。

```
class CustomGemma(LLM):
    """一个自定义的 Gemma LLM,允许动态设置生成参数。"""
    model_path: str               # Gemma 模型权重文件的路径
    temperature: float = 0.7      # 控制生成结果的多样性
    max_length: int = 512         # 限制生成文本的最大长度

    @property
    def _llm_type(self) -> str:
        return "自定义 Gemma"
    def _call(
        self,
        prompt: str,
        stop: Optional[List[str]] = None,
        run_manager: Optional[CallbackManagerForLLMRun] = None,
        **kwargs: Any,
    ) -> str:
        from gemma import generate # 从 Gemma 库导入 generate 函数
        # 如果模型不同,请注意使用不同的导入形式
        # 调用 Gemma 的 generate()函数,传入相应的参数
        return generate(
            prompt,
            model_path=self.model_path,
            temperature=self.temperature,
            max_length=self.max_length,
            stop_words=stop,
        )
    @property
    def _identifying_params(self) -> Mapping[str, Any]:
        """返回用于识别 LLM 的参数,有助于日志记录和调试。"""
        return {
            "model_path": self.model_path,
```

```
            "temperature": self.temperature,
            "max_length": self.max_length,
        }
#使用自定义 Gemma LLM
    custom_llm = CustomGemma(model_path = "path/to/gemma/weights", temperature = 0.5, max_
length = 256)                          #此处需要修改
response = custom_llm.invoke("写一首关于春天的诗")
print(response)
```

输出会是一首关于春天的诗，具体内容取决于Gemma模型的生成能力和配置的参数。这个例子中的CustomGemma类展现了如何创建一个自定义的大语言模型(LLM)，它可以与特定的模型如Gemma进行交互。这个类通过重写_call()方法来实现与Gemma模型的交互，其中，generate()函数是假定的Gemma模型的生成接口，它根据提供的输入prompt和其他参数来生成文本。通过在_call()方法中调用这个函数，CustomGemma能够实现生成文本的功能。此外，通过设置temperature和max_length参数，开发者可以控制生成文本的多样性和长度，从而得到更加符合需求的输出。

重要的是，CustomGemma类还包含_identifying_params属性，它返回一个字典，描述了这个自定义LLM的关键配置。这有助于日志记录和调试，因为开发者可以快速识别LLM使用的配置参数。

读者需要格外注意，上述代码并不能直接运行成功。如果想运行上述代码，一定要考虑生成函数的不同表示，以及模型权重文件所在的路径。如果读者追求简单，可以使用Transformer等基础模型来替代上述模型。

3.2.5 输出解析器

输出解析器在LangChain框架中扮演了关键角色，其主要职责是将语言模型生成的原始输出转换成更加结构化、易于处理的格式。LangChain精心设计了一系列输出解析器，每种解析器针对不同的需求和场景，提供了独特的处理能力。例如，OpenAITools和OpenAIFunctions解析器专门处理OpenAI的函数调用输出，而对于标准的数据格式，如JSON、XML和CSV，相应的解析器能够将输出内容转换成对应格式。在输出处理过程中可能遇到的错误，OutputFixing和RetryWithError解析器能够自动触发LLM重试或修复操作。此外，Pydantic和YAML解析器允许将输出映射到用户定义的数据模型中，而Pandas DataFrame、Enum、Datetime和Structured解析器则针对特定领域的解析需求提供了专门的功能。

许多输出解析器支持流式传输功能，这意味着可以边生成边处理语言模型的输出，无须等待整个输出内容全部生成。此外，大多数解析器还提供了详细的格式说明，引导语言模型按照预期格式生成输出。虽然在某些情况下，解析器可能需要重新调用LLM以修复或重试输出，但整体上，输出解析器的使用显著提升了语言模型应用的实用性和效率。

根据具体任务的需求，开发者可以灵活选择和搭配不同的输出解析器，最大化语言模型的潜能。输出解析器尤其在我们期望从模型获得结构化信息而非纯文本时显得尤为重要。简而言之，输出解析器是构建和优化语言模型响应的强大工具。

每个输出解析器需实现以下主要方法。

(1) 获取格式说明：返回一个字符串的方法，其中包含关于如何格式化语言模型输出

的说明。

(2) 解析：一种方法，它接收一个字符串(假定是语言模型的响应)并将其解析为某种结构。

还有一个可选项“使用提示解析”，它也是一种方法，它接收一个字符串(假定是语言模型的响应)和一个提示(假定是生成此类响应的提示)，并将其解析为某种结构。主要在OutputParser想要以某种方式重试或修复输出并需要来自提示的信息以执行此操作的情况下提供提示。

【例 3-7】 输出解析器类型 PydanticOutputParser(参考代码 3.7.py，3.7.1.py)。

```
from langchain_community.llms import Ollama as OllamaLLM
from langchain.output_parsers import PydanticOutputParser
from langchain.prompts import PromptTemplate
from langchain_core.pydantic_v1 import BaseModel, Field, validator
import json
model = OllamaLLM(model = "gemma:2b")
class Joke(BaseModel):
    setup: str = Field(description = "笑话的问题部分")
    punchline: str = Field(description = "笑话的答案部分")
    @validator("setup")
    def question_ends_with_question_mark(cls, field):
        if field[ - 1] != "?":
          raise ValueError("问题格式错误，必须以问号结尾!")
        return field
parser = PydanticOutputParser(pydantic_object = Joke)
prompt = PromptTemplate(
    template = "请创作一个笑话，并严格按照以下 JSON 格式返回:\n{format_instructions}\n\n
只需给出 setup 和 punchline 的具体内容，不要有额外的解释或说明。",
    input_variables = ["query"],
    partial_variables = {"format_instructions": parser.get_format_instructions()},
)
prompt_and_model = prompt | model
output = prompt_and_model.invoke(input = {"query": ""})
#检查输出是否为有效的 JSON 格式
try:
    json_output = json.loads(output)
except (json.JSONDecodeError, TypeError):
    print("模型生成的输出无法解析为有效的 JSON 格式:")
    print(output)
else:
    parser.invoke(json_output)
```

上述代码首先初始化了一个名为 OllamaLLM 的大语言模型实例，指定使用模型 gemma:2b。然后，定义了一个名为 Joke 的 Pydantic 模型，用来描述一个笑话，其中包含笑话的设置部分(setup)和笑话的解答部分(punchline)。通过 validator 确保每个笑话的设置部分都以问号结束。接着，使用 PydanticOutputParser 创建一个输出解析器，它能将模型的输出转换成 Joke 模型的实例。通过 PromptTemplate 构造了一个提示模板，这个模板要求模型生成的笑话严格遵循一定的 JSON 格式，其中包含如何格式化输出的具体指导，这些指导是通过 parser.get_format_instructions()获取的。然后，将提示模板和模型链接起来，并使用空查询调用这个组合，尝试生成一个符合要求的笑话。最后，尝试将模型的输出解析为

JSON 格式，如果成功，进一步使用 PydanticOutputParser 解析器解析这个 JSON 输出，否则打印出错信息。

在某些情况下，读者可能希望实现自定义解析器来将模型输出构造为自定义格式。有两种方法可以实现自定义解析器：第一种是在 LCEL 中使用 RunnableLambda 或 RunnableGenerator(建议大多数用例使用此方法)；第二种是通过继承输出解析的基类之一，这是困难的方式。这两种方法之间的区别主要是表面的，主要体现在触发哪些回调(例如，on_chain_start 与 on_parser_start)以及在 LangSmith 等跟踪平台中可视化 runnable lambda 与解析器的方式。

3.3　文 档 检 索

许多大语言模型(LLM)应用程序需要特定于用户的数据，而这些数据并不是模型训练集的一部分。实现这一点的主要方法是通过检索增强生成(RAG)。在这个过程中，外部数据被检索，然后再生成步骤传递给 LLM。

3.3.1　关键模块

LangChain 提供了 RAG 应用程序的所有构建块——从简单到复杂。文档的这一部分涵盖了与检索步骤相关的所有内容，如数据的获取。尽管这听起来很简单，但实际上可能有些复杂。如图 3-2 所示，这包括以下几个关键模块。

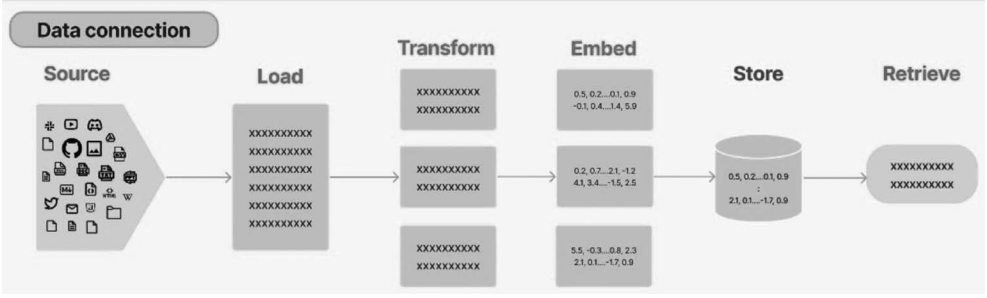

图 3-2　Retrieval 环节的关键模块

(1) 文档加载器。文档加载器从许多不同的来源加载文档。LangChain 提供了 100 多种不同的文档加载器，以及与该领域的其他主要提供商(如 AirByte 和 Unstructured)的集成。LangChain 提供了集成，可以从所有类型的位置(私有 S3 存储桶、公共网站)加载所有类型的文档(HTML、PDF、代码)。

(2) 文本分割器。检索的一个关键部分是仅获取文档的相关部分。这涉及几个转换步骤，以准备用于检索的文档。这里的主要步骤之一是将大型文档拆分(或分块)为较小的块。LangChain 提供了几种用于执行此操作的转换算法，以及针对特定文档类型(代码、markdown 等)优化的逻辑。

(3) 文本嵌入模型。检索的另一个关键部分是为文档创建嵌入。嵌入捕获文本的语义含义，使能够快速有效地查找语义相似的其他文本片段。LangChain 提供了与 25 种不同嵌入提供商和方法的集成，从开源到专有 API，可以选择最适合需求的嵌入模型。LangChain

提供标准接口，允许在模型之间轻松切换。

(4) 向量存储。随着嵌入的兴起，出现了对支持高效存储和搜索这些嵌入数据库的需求。LangChain 提供了与 50 多个不同向量存储的集成，从开源的本地存储到云托管的专有存储，可以选择最适合需求的存储。LangChain 公开了一个标准接口，允许在向量存储之间轻松切换。

(5) 检索器。数据进入数据库后，仍然需要检索它。LangChain 支持许多不同的检索算法，这是我们添加最多价值的地方之一。LangChain 支持易于上手的基本方法，即简单的语义搜索。但是，我们还在此基础上添加了一系列算法来提高性能，包括：

① 父文档检索器。允许为每个父文档创建多个嵌入，从而可以查找较小的块但返回较大的上下文。

② 自查询检索器。用户问题通常包含对某些内容的引用，这些内容不仅是语义的，而且表达了一些最好表示为元数据过滤器的逻辑。自查询允许从查询中存在的其他元数据过滤器中解析查询的语义部分。

③ 集成检索器。有时可能希望使用多个不同的源或使用多个不同的算法检索文档。

(6) 索引。LangChain 索引 API 将数据从任何来源同步到向量存储中，从而可以避免将重复的内容写入向量存储，避免重写未更改的内容，避免在未更改的内容上重新计算嵌入。

下面详细介绍各关键模块。

3.3.2 文档加载器

文档加载器是数据处理和分析流程中的关键组件，主要职责是从各种数据源中提取文本数据及其相关元数据，并将这些数据转换为结构化的文档格式。文档通常包含一段文本和与之关联的元数据，如作者、发布日期或任何其他相关信息。这样的机制允许复杂的数据处理和分析工作在一个统一和标准化的数据结构上进行，提高了后续处理的效率和可靠性。

文档加载器支持多种类型的数据源，包括但不限于简单的文本文件(.txt)、网络页面的文本内容，以及 YouTube 视频的字幕等。这种多样性使得文档加载器能够适应各种数据获取需求，无论数据存储在哪里或以何种格式存在。

加载器的核心功能之一是 load 方法，它能够从指定的数据源中读取数据，并将其转换成一系列文档对象。此外，许多文档加载器还支持懒加载(lazy load)机制，这意味着数据只有在实际需要时才被加载到内存中，从而优化了内存使用和提高了处理速度，特别是在处理大规模数据集时。

以下是一个具体示例，展示了如何使用 CSV 格式的文档加载器。

```
from langchain_community.document_loaders import CSVLoader
#实例化 CSVLoader,指定数据文件路径
loader = CSVLoader('example_data.csv')
#使用 load()方法加载数据,转换为文档对象集合
documents = loader.load()
# 此时,'documents'包含从 example_data.csv 文件中加载的数据
#每一行数据被转换成一个独立的文档对象,便于后续处理和分析
```

上述过程不仅简化了从 CSV 文件中读取数据的步骤，而且将数据以文档对象的形式组织起来，使得每个数据项都拥有一致的接口和结构。通过这种方式，开发者可以轻松实施更

复杂的数据处理策略，如文本分析、信息提取和数据挖掘等，无论数据的原始格式如何，文档加载器都为数据的进一步处理提供了一个清晰、灵活的起点。

3.3.3　文本分割器

在加载文档到自己的应用程序之后，通常需要对其进行某种形式的转换以更好地适应应用程序的需求。一个典型的场景是将较长的文档分割成更小的块，以适应模型的上下文窗口限制。LangChain 提供了多种内置的文档转换工具，使得对文档进行分割、合并、过滤和操作变得简单而直接。这些工具的存在极大地简化了文档预处理的过程，为后续的文本处理和分析提供了便利。

处理长文本时，将文本切割成小块是一个必要的步骤，尽管这听起来简单，但实际上包含不少潜在的复杂性。理想的分割策略是将语义相关的文本片段保持在一起，而“语义相关”的含义则可能根据文本的具体类型而变化。文本分割器的基本工作原理是首先将文本拆分成小的、语义上有意义的单元（如句子），然后将这些小单元组合成更大的块，直到达到预定的大小，同时在新的文本块创建时引入一些重叠，以维持块之间的语义连贯性。

LangChain 通过提供多种类型的文本分割器来支持文本的自定义分割策略，允许用户根据具体需求调整如何分割文本以及如何测量块的大小。所有这些分割器都可以在 langchain-text-splitters 包中找到，提供了多种选择以适应不同的应用场景。通过这些工具，LangChain 旨在简化文本处理流程，帮助开发者更高效地构建和优化他们的语言模型应用，如表 3-1 所示。可以使用 Greg Kamradt 创建的 Chunkviz 实用程序来评估文本分割器。Chunkviz 是一个很好的工具，用于可视化文本分割器的工作方式。它能够展示文本是如何被分割的，并帮助调整分割参数。

表 3-1　分割器的特征

名　称	分割依据	添加元数据	描　述
Recursive	用户定义的字符列表		递归地分割文本。递归分割文本的目的是试图将相关的文本片段保持在一起。这是开始分割文本的推荐方法
HTML	HTML 特定字符	是	根据 HTML 特定字符分割文本。值得注意的是，这会添加关于该块来自何处的相关信息（基于 HTML）
Markdown	Markdown 特定字符	是	根据 Markdown 特定字符分割文本。值得注意的是，这会添加关于该块来自何处的相关信息（基于 Markdown）
Code	代码（Python、JS）特定字符		根据编码语言特定的字符分割文本。可以选择 15 种不同的语言
Token	Tokens		根据 tokens 分割文本。有几种不同的方法来测量 tokens
Character	用户定义的字符		根据用户定义的字符分割文本。这是较简单的方法之一

3.3.4　文本嵌入模型

Embeddings 类是一个与文本嵌入模型交互的工具类，旨在为多个嵌入模型提供商（如

OpenAI、Cohere、Hugging Face 等)提供一个统一的接口。利用这个类,可以为文本生成向量表示,这一点非常有用,因为它使人们能够在向量空间中处理文本,进行如语义搜索等操作,从而在向量空间中找到最相似的文本片段。

在 LangChain 中,基础的 Embeddings 类提供了两种方法:一种是 embed_documents,用于处理多个文本输入;另一种是 embed_query,专门用于处理单个文本输入。这样设计是因为一些嵌入模型提供商对于文档(即搜索对象)和查询(即搜索词)使用了不同的嵌入策略。以下是使用 Gemma 模型进行嵌入创建的示例代码。

```
from langchain_community.embeddings import OllamaEmbeddings
embeddings_model = OllamaEmbeddings(model = "gemma:2b")
```

然后,可以调用 embed_documents()方法为一系列文本创建嵌入,这将返回一个包含对应每个输入文本的嵌入向量的列表。

如图 3-3 所示,一个典型的工作流程包括加载源数据(Load Source Data)、检索向量存储(Query Vector Store)以及获取最相似的结果(Retrieve 'most similar')。接下来,通过安装必要的包、加载文档、创建嵌入并将其存储到 Chroma 向量存储中的示例,我们展示了如何实现这一流程。首先是安装向量数据库,向量存储负责嵌入向量的存储和搜索,是处理嵌入数据及其向量搜索的关键组件。

```
pip install chromadb        #安装向量数据库
```

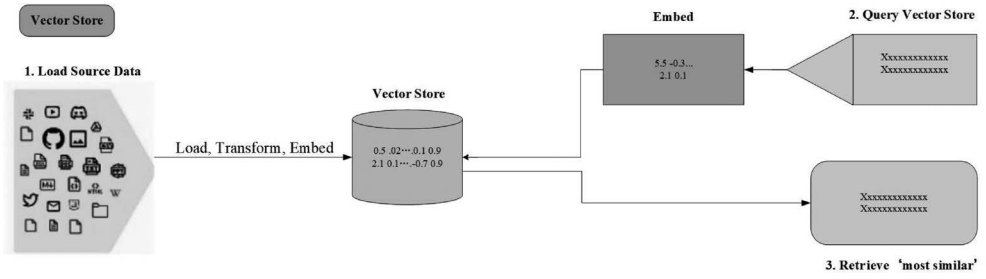

图 3-3 向量存储检索的典型工作流程

加载文档后,通过文本分割、创建嵌入并加载到向量存储的过程,构建了一个填充了嵌入向量的 Chroma 实例。通过 similarity_search 方法,可以根据文本查询检索最相似的文档,或者使用 similarity_search_by_vector 方法通过嵌入向量而非文本查询来检索相似文档,这在已预先计算了查询嵌入向量的情况下非常有用。这些功能展示了 LangChain 在构建灵活、高效的聊天模型应用中的强大能力,为开发者提供了丰富的工具和框架,以实现复杂的语言处理任务。

【例 3-8】 构建一个能够响应查询并找出最相关文档的系统(代码:new3.5textembeded)。

```
from langchain_community.embeddings import OllamaEmbeddings
#初始化嵌入模型
embeddings_model = OllamaEmbeddings(model = "gemma:2b")
#需要被嵌入的文本列表
texts = [
    "你好!",
    "哦,你好!",
    "你叫什么名字?",
```

```
    "我的朋友都叫我小明",
    "你好小明!"
]
#为文本列表创建嵌入
embeddings = embeddings_model.embed_documents(texts)
from langchain_community.vectorstores import Chroma
from langchain_community.document_loaders import TextLoader
from langchain_text_splitters import CharacterTextSplitter
#假设有一些原始文档需要加载和分割
raw_documents = TextLoader(doc.txt').load()
text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
documents = text_splitter.split_documents(raw_documents)
#创建并填充 Chroma 向量存储
db = Chroma.from_documents(documents, embeddings_model)
#用户查询文本
query = "小明喜欢什么运动?"
#使用 Chroma 向量存储检索最相似的文档
docs = db.similarity_search(query)
#打印最相似文档的内容
print(docs[0].page_content)
```

3.3.5　检索器

检索器是一个接口，给定非结构化查询，返回文档。它比向量存储更通用。检索器不需要能够存储文档，只需要能够返回（或检索）文档。向量存储可以用作检索器的主干，但还有其他类型的检索器。检索器接收字符串查询作为输入，并返回文档列表作为输出。LangChain 提供了几种高级检索类型，如表 3-2 所示。这个表总结了 LangChain 提供的各种高级检索类型，包括它们的索引要求、是否使用 LLM、适用场景以及工作原理。

表 3-2　检索类型列表

名　称	索引类型	使用 LLM	何时使用	描　述
Vectorstore	Vectorstore	否	当有大量文本数据需要快速、高效检索，且对语义理解要求不是特别高时	将文本转换为向量表示，通过计算向量相似度来检索相关文档。适合处理大规模数据，检索速度快，但可能忽略上下文语义
ParentDocument	Vectorstore＋Documentstore	否	当文档结构复杂，包含多个小的信息片段，但需要保持整体上下文时	将文档分割成小块进行索引，但在检索时返回完整的父文档。这种方法平衡了精确检索和保持上下文的需求
Multi Vector	Vectorstore＋Documentstore	有时在索引期间使用	当单一向量表示不足以捕捉文档的全部重要特征时	为每个文档创建多个向量表示，可能包括文本摘要、假设问题等。这种方法提高了检索的多样性和准确性，但增加了索引复杂度

续表

名　称	索引类型	使用 LLM	何时使用	描　述
Self Query	Vectorstore	是	当用户查询需要复杂的解释和转换，或者检索需要考虑元数据时	使用 LLM 将用户输入转换为语义查询和元数据过滤器。这种方法能更好地理解用户意图，提高检索精度
Contextual Compression	任何	有时	当检索结果包含大量冗余或不相关信息时	在检索后进行额外的处理，从检索到的文档中提取最相关的信息。这种方法可以大大提高返回信息的质量和相关性
Time-Weighted Vectorstore	Vectorstore	否	当文档的时效性很重要，需要考虑内容的新近程度时	结合语义相似性和时间因素进行检索。这种方法适合新闻、社交媒体等时效性强的内容检索
Multi-Query Retriever	任何	是	当面对复杂、多方面的查询，单一查询可能无法全面捕捉用户意图时	使用 LLM 从原始查询生成多个相关查询，然后综合这些查询的结果。这种方法可以提高检索的全面性和深度
Ensemble	任何	否	当单一检索方法无法满足复杂需求，需要结合多种方法的优势时	组合多个检索器的结果。这种方法可以平衡不同检索策略的优缺点，提高整体检索性能
Long-Context Reorder	任何	否	当使用能处理长文本的大语言模型，需要优化输入顺序以提高模型性能时	重新排序检索到的文档，使最相关的内容位于开头和结尾。这种方法利用了大语言模型对文本开头和结尾部分的特殊关注，以提高处理效果

3.3.6　索引

在 LangChain 中，索引 API 提供了一种强大的机制，允许开发者从任何来源加载文档并与向量存储同步。这一过程旨在优化存储管理，避免重复内容的写入、未更改内容的重写以及对未更改内容的嵌入重算，从而节省时间和成本，同时提升向量搜索的效果。索引 API 的设计考虑了文档在经历多个转换步骤（如文本分割）后，仍能保持与原始源文档的一致性，确保数据的准确性和可靠性。

LangChain 索引工作原理基于记录管理器（RecordManager），该管理器负责追踪文档被写入向量存储的情况。在索引过程中，每个文档都会被计算出一个哈希值，并将文档哈希、写入时间和源 ID 等信息存储在记录管理器中。这些信息有助于在以后的索引操作中判断文档是否已更改或已被删除，从而实现内容的有效管理。如表 3-3 所示，索引 API 支持不同的删除模式，如无清理（None）、增量（Incremental）和完整（Full），以适应不同的应用场景和需求，提供灵活的数据管理策略。

表 3-3　索引 API 删除模式

清理模式	去重内容	可并行化	清理已删除的源文档	清理源文档和/或派生文档的变体	清理时间
None	✓	✓	✗	✗	—
Incremental	✓	✓	✗	✓	连续
Full	✓	✗	✓	✓	索引结束时

为了展示 LangChain 索引 API 的基本工作流程，以下是使用本地 Gemma 模型进行索引操作的示例。首先，初始化向量存储和嵌入模型，然后配置记录管理器，并选定合适的删除模式进行文档索引。这个过程涉及加载原始文档、将文档分割成块、为每个块创建嵌入并加载到 Chroma 向量存储中。通过选择合适的删除模式，可以有效地管理存储内容，避免旧版本的冗余，确保向量存储中的数据既准确又高效。如下示例强调了 LangChain 在处理索引任务时的灵活性和效率，使得构建基于文本嵌入的应用变得更加简单和直观。

【例 3-9】 LangChain 索引 API 的基本工作流程(参考代码 new3.6index)。

```
from langchain.indexes import SQLRecordManager, index
from langchain_core.documents import Document
from langchain_community.vectorstores import Chroma
from langchain_community.embeddings import OllamaEmbeddings
#初始化向量存储和嵌入模型
collection_name = "test_index"
embedding = OllamaEmbeddings(model_path="path/to/gemma/model")      #依据自己的实际设置
vectorstore = Chroma(collection_name=collection_name, embedding_function=embedding)
#初始化记录管理器
namespace = f"chroma/{collection_name}"
record_manager = SQLRecordManager(namespace, db_url="sqlite:///record_manager_cache.sql")
record_manager.create_schema()
#示例文档
doc1 = Document(page_content="小明喜欢踢足球", metadata={"source": "xiaoming.txt"})
doc2 = Document(page_content="小红喜欢打篮球", metadata={"source": "xiaohong.txt"})
#使用不同的删除模式将文档索引到向量存储中
index([doc1, doc2], record_manager, vectorstore, cleanup="incremental", source_id_key="source")
```

通过上述步骤，读者不仅可以优化向量存储的内容管理，还能根据实际需要灵活选择删除模式，确保数据的新鲜度和搜索结果的相关性。这一流程示例突出了 LangChain 在索引管理方面的强大功能和灵活性，为开发基于向量搜索的应用提供了坚实的基础。

索引 API 提供了一种智能的方式来同步源数据与向量存储。通过跟踪文档的变化并自动处理重复、更新和删除操作，索引 API 确保了向量存储始终保持最新和相关，从而提高了检索质量和效率。

在实践中，读者可以根据具体的任务需求，灵活选择和组合不同的检索组件。例如，可以使用 Gemma 等开源模型来创建高质量的嵌入向量；使用 Chroma、FAISS 等高性能的向量存储来支持海量数据的实时检索；使用自查询检索器、上下文压缩检索器等高级检索算法来处理复杂的用户查询。

3.4 代　　理

代理(Agents)的核心概念围绕着利用语言模型作为决策引擎,以动态地选择和排序一系列的行动。与在链(Chains)结构中行动序列被预设在代码里不同,代理使用语言模型的推理能力来判断执行哪些具体的行动,以及这些行动的执行顺序。在设计代理的过程中,几个重要的概念需要被充分理解:代理(Agents)本身,它们是执行行动的实体;代理执行器(AgentExecutor),负责实施代理决定的行动;工具(Tools),代理可利用的操作或功能单元;以及工具包(Toolkits),一组工具的集合,为代理执行任务提供支持。这些元素共同构成了代理的基础架构,使得代理能够在复杂环境中做出智能决策并执行任务。

3.4.1 核心思想

在 LangChain 中,代理(Agents)与链(Chains)的概念相辅相成,但有一个根本的区别:在链中,动作的序列是在代码中硬编码的,而在代理中,语言模型则充当推理引擎,用来决定哪些动作应当被采取以及它们的执行顺序。这种设计使得代理能够根据上下文和先前的动作结果动态地做出决策,从而提供更加灵活和智能的行为。

代理的实现涉及以下核心组件。

(1) AgentAction:这是一个数据类,用于表示代理应该执行的动作。它包括一个指定应调用的工具名称的 tool 属性,以及一个为该工具提供输入数据的 tool_input 属性。

(2) AgentFinish:表示代理的终态,当代理准备好向用户返回结果时使用。它通常包含一个包含代理最终输出的 return_values 键值对映射,这里面通常会有一个名为 output 的键,其值是代理向用户返回的响应字符串。

(3) 中间步骤(Intermediate Steps):这些代表了代理先前执行的动作和在本次代理运行中得到的相应输出。为了确保代理了解它已经完成了哪些工作,将这些信息传递给未来的迭代是非常重要的。这些步骤的类型是 List[Tuple[AgentAction, Any]],其中,observation 目前保留为 Any 类型,以便提供最大的灵活性。在实践中,这通常是一个字符串。

Agent 类负责决定下一个动作,通常依赖于语言模型、提示模板和输出解析器。根据不同的应用场景,代理可能需要不同的推理提示风格、不同的输入编码方式和不同的输出解析策略。LangChain 允许轻松构建自定义代理,以及利用内置代理类型进行扩展。

代理的输入是一个键值对映射,其中只有一个键是必需的:intermediate_steps,它对应于前述的中间步骤。通常,PromptTemplate 会负责将这些键值对转换成适合传递给 LLM 的格式。代理的输出则是下一个要执行的动作,或者是最终发送给用户的响应(AgentActions 或 AgentFinish)。

AgentExecutor 是代理的运行时环境,负责实际调用代理,执行它选择的动作,将动作的输出反馈给代理并进行下一轮迭代。虽然这个过程在表面上看起来简单,但 AgentExecutor 为开发者处理了诸多复杂问题,如处理不存在的工具、工具执行错误、输出无法解析为工具调用等情况,同时在所有级别(包括代理决策和工具调用)提供日志记录和可观测性。

图 3-4 给出了一个一般的代理的执行过程。在观察(Observation)阶段,代理通过输入接口接收外部的触发,例如,用户的提问或系统的请求,随后对这些输入进行解析,提取出关键信息,作为后续处理的基础。紧接着,代理进入思考(Thought)阶段,它利用预设的规则、知识库或机器学习模型来分析所观察到的信息,旨在确定如何响应观察到的情况。这个过程可能包括理解用户意图、推断所需工具以及提取运行工具所需的参数等子步骤。最后,在行动(Action)阶段,代理基于思考阶段的结果执行具体行动,这可能涉及填充参数、执行工具、生成响应以及将响应输出给用户或系统等子步骤。整个流程在 LangChain 框架下得到了高效的实现,允许代理以高度智能化的方式处理和响应各种情况。

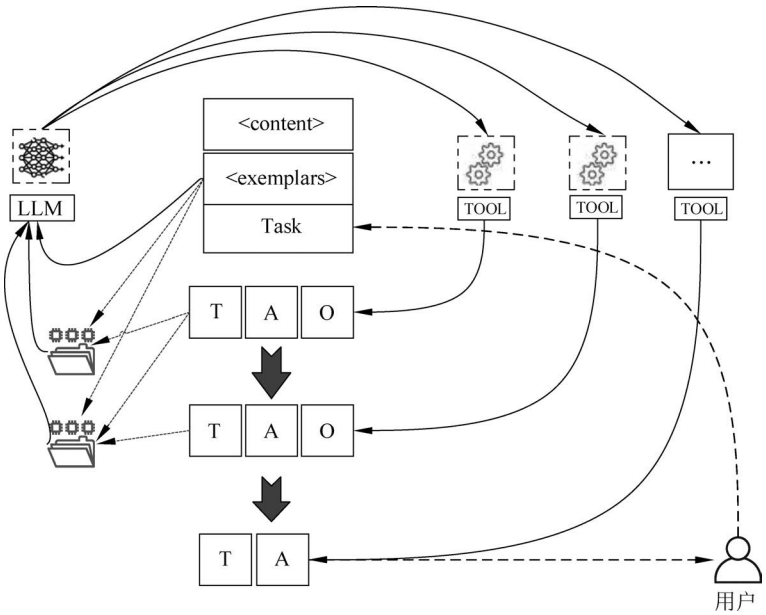

图 3-4　代理的执行过程

以下是一个使用 Agent 和 AgentExecutor 的代码示例,以展示如何实现和运行代理。

```
#初始化代理和 AgentExecutor
agent = Agent(…)                                          #假设 Agent 已经被正确初始化
agent_executor = AgentExecutor(agent = agent, tools = […]) #传入代理和工具列表
#执行代理,获取动作或最终结果
result = agent_executor.execute(input_data)
#根据结果进行处理
if isinstance(result, AgentAction):
      #执行动作
      …
elif isinstance(result, AgentFinish):
      #处理最终结果
      …
```

通过动态决策和执行,代理提供了一种更加灵活和智能的方式来处理复杂的交互流程。它允许应用根据上下文和先前的交互来动态调整行为,使得构建复杂的对话系统和交互式应用变得可能。代理的这种能力,特别是在与用户的实时互动、自动化工作流程处理和智能决策支持等场景中,显得尤为重要。

为了进一步优化代理的实现和运行效率，LangChain 提供了一系列工具和策略来管理和优化代理的决策过程。这包括对代理行为的详细日志记录，以及可观察性的提高，使开发者能够更容易地追踪和调试代理的行为。此外，AgentExecutor 的设计考虑到了错误处理和异常管理，确保了在执行代理决策和工具调用过程中的稳定性和可靠性。

在实践中，构建一个高效的代理需要对目标应用的具体需求有深入的理解，包括用户的交互方式、所需的自动化程度，以及如何最有效地利用可用的工具和资源。开发者可以根据这些需求，利用 LangChain 提供的代理框架和 API，设计出能够智能响应用户输入、自动执行任务并提供有用反馈的代理。通过迭代开发和测试，不断调整代理的推理逻辑和动作序列，可以逐步提高代理的性能，使其更好地服务于应用和用户。

3.4.2 代理类型

根据 LangChain 提供的分类，代理可以基于它们对预期模型类型、是否支持聊天历史、是否支持多输入工具、是否支持并行函数调用等特征进行选择。例如，OpenAI Tools 代理支持聊天模型，并能处理聊天历史、多输入工具及并行函数调用，适合使用最新 OpenAI 模型的场景。而 XML 代理则适用于语言模型，特别是擅长处理 XML 格式的 Anthropic 模型。不同代理的选择依赖于具体的应用需求、模型能力及任务复杂度等多个因素。

在选择代理类型时，应综合考虑模型能力、任务复杂性、效率要求、可用工具和定制需求等。例如，对于需要聊天历史支持的复杂多轮交互任务，选择如 OpenAI Tools 这样的代理会更加合适。而对于效率极其重要的场景，可以考虑使用支持并行函数调用的代理以加快处理速度。另外，根据任务需求选择支持特定格式输出（如 XML 或 JSON）的代理，或者根据可用工具选择支持多输入的代理也是重要的考虑因素。通过充分了解各类代理的特性和适用场景，开发者可以做出明智的选择，以期代理能在实际应用中发挥最大的价值。

例如，在构建一个客服聊天机器人时，我们的目标是实现一个能够理解用户查询并提供有用反馈的系统。在这种场景下，代理需要支持聊天历史，以便理解上下文和之前的对话内容。OpenAI Tools 代理就非常适合这种应用，因为它不仅支持聊天历史，还能处理多输入工具和并行函数调用，从而提高处理效率。使用最新的 OpenAI 模型，可以让聊天机器人理解复杂的用户查询，并利用外部工具（如数据库查询）来提供准确的答案。

再如，对于一个内容生成工具，如自动撰写新闻稿或撰写代码注释，可能会倾向于使用擅长特定格式的代理。例如，如果我们的内容生成工具需要生成结构化的 XML 格式数据，那么 XML 代理就非常合适。这是因为 Anthropic 模型等擅长处理 XML 格式的模型可以直接以 XML 格式理解和生成内容，从而提高生成内容的准确性和效率。

而对于数据分析和报告的场景，可能需要代理来帮助整理和总结数据，然后生成报告。在这种情况下，Structured Chat 代理可能是一个好选择，因为它支持具有多个输入的工具，允许代理同时调用数据查询和数据处理工具，然后将结果整合成易于理解的报告。此外，支持并行函数调用的特性也能加快数据处理和报告生成的速度。

3.4.3 工具

在 LangChain 框架中，工具（Tools）是构建智能代理（Agents）不可或缺的一部分，它们为代理提供了与外部世界交互的能力。工具的设计理念是提供一个清晰定义的接口，使代

理能够执行特定的操作或访问特定的服务。每个工具通常包括以下几个关键要素。

(1) 工具名称：为工具指定一个唯一的标识符，方便在代理中引用。

(2) 工具描述：简要说明工具的功能和用途，帮助理解工具的作用。

(3) 工具输入的 JSON 模式：定义工具期望的输入格式，使得输入的参数符合预期结构。

(4) 调用的函数：指定当工具被触发时，应该执行的函数或操作。

(5) 结果处理方式：指明工具的执行结果是否应直接返回给用户。

这些要素共同构成了一个工具的基本框架，使得在代理决策过程中可以精确地指定并执行动作。简单的工具输入有助于大语言模型(LLM)更高效地使用工具，而名称、描述和 JSON 模式的清晰定义则确保了语言模型能够准确理解和调用工具。

下面以基于本地大模型 Llama2 的自定义工具为例，构建一个工具来查询 Llama2 模型并获取相关信息。假设目标是创建一个工具，它能够根据用户的查询，使用 Llama2 模型生成相关的文本信息。这个工具可以用于多种场景，如自动回答用户的问题、生成相关内容等。我们之所以切换了本地大模型，是希望读者了解这些是可以自由切换的。

【例 3-10】 工具的使用(代码参见 3.8llama2.py)。

```
from langchain.tools import BaseTool
from pydantic import BaseModel, Field
from langchain_community.llms import Ollama as OllamaLLM
class Llama2QAInput(BaseModel):
    query: str = Field(description="向 LLAMA2 模型提问的问题字符串。")
    class Config:
        json_schema_extra = {
            "example": {
                "query": "LangChain 是什么?"
            }
            }
class Llama2QATool(BaseTool):
    name = "llama2_qa"
    description = "一个使用 LLAMA2 模型回答问题的工具。输入应该是一个问题字符串。"
    llm: OllamaLLM = Field(..., description="用于回答问题的 LLAMA2 模型实例。")
    def _run(self, query: str) -> str:
        return self.llm(query)
    async def _arun(self, query: str) -> str:
        return self.llm(query)
    def run(self, tool_input: Llama2QAInput) -> str:
        query = tool_input.query
        return self._run(query)
llm = OllamaLLM(model="llama2")
tool = Llama2QATool(llm=llm)
input_data = Llama2QAInput(query="什么是 Python?")
result = tool.run(input_data)
print(result)
```

在这个案例中，程序中定义了一个名为“llama2_qa”的工具，作为工具的唯一标识符。程序中为工具提供了一个简要的描述：“一个使用 LLAMA2 模型回答问题的工具。输入应该是一个问题字符串。”，这有助于理解工具的功能和用途。程序定义了一个名为 Llama2QAInput 的 Pydantic 模型类，用于指定工具期望的输入格式。该模型包含一个名为

query 的字符串字段，表示向 LLAMA2 模型提问的问题。程序中的 Llama2QATool 类定义了_run()和_arun()方法，分别用于同步和异步调用 LLAMA2 模型来生成回答。run()方法则负责将 Llama2QAInput 实例转换为_run()方法所需的参数。程序中的工具直接返回 LLAMA2 模型生成的答案字符串，无须额外的后处理。

通过这种方式，可以轻松地将复杂的模型调用封装为 LangChain 中的工具，进而在代理中使用这些工具来构建更加智能和灵活的应用。这个案例展示了如何基于本地大模型 Llama2 创建和使用自定义工具，使得开发者能够充分利用现有的模型资源，为用户提供高质量的智能服务。

工具的概念在 LangChain 中有广泛的应用场景，不仅限于内置工具的使用。开发者可以根据实际需要定义自定义工具，扩展代理的能力。此外，工具包(Toolkits)提供了一组协同工作的工具集合，为解决特定问题提供了方便的解决方案。将工具与 OpenAI 函数结合，可以进一步增强工具的通用性和灵活性，实现从简单的查询到复杂的数据处理等多种功能。工具在 LangChain 中扮演着桥接代理与外部世界的角色，通过精心设计和应用工具，可以显著提升代理的智能化水平和实用性。

3.4.4 案例分析

本案例分析中，将探索如何构建一个智能代理，该代理整合了两种不同的工具来提供信息检索功能：一种是基于 Gemma 大模型的本地检索器，另一种是在线搜索工具。我们的目标是使代理能够根据用户的查询，在本地索引和在线资源中查找相关信息。

(1) 本地检索器。首先，需要建立一个本地检索器，它将基于 Gemma 大模型创建索引并执行检索操作。下面将从一些源数据开始，如自己的数据库或文档集合。接着，将使用 Gemma 模型来嵌入这些文档，并使用 FAISS(一种高效的相似性搜索库)来建立向量索引，以便可以快速检索与查询最相关的文档。

```
# 引入必要的库
from langchain.llms import Ollama as OllamaLLM
from langchain.document_loaders import TextLoader
from langchain.vectorstores import FAISS
from langchain.text_splitters import RecursiveCharacterTextSplitter
#初始化 Gemma 模型和 FAISS 向量存储
gemma_model = OllamaLLM(model = "gemma:2b")
faiss_vector_store = FAISS()
#加载并嵌入文档
loader = TextLoader("path/to/your/data")
docs = loader.load()
splitter = RecursiveCharacterTextSplitter(chunk_size = 1000, chunk_overlap = 200)
split_docs = splitter.split_documents(docs)
vectors = gemma_model.embed_documents([doc.page_content for doc in split_docs])
#将嵌入向量存储到 FAISS 中
faiss_vector_store.add_documents(vectors)
```

(2) 在线搜索工具。对于在线搜索工具，可以利用已有的搜索引擎 API。假设有一个 API 密钥用于访问这项服务，可以定义一个简单的工具来发送查询并接收结果。

```
#假设已有在线搜索工具 API 封装
from your_search_tool_api_wrapper import OnlineSearchAPIWrapper
```

```
online_search_tool = OnlineSearchAPIWrapper(api_key = "your_api_key")
```

(3) 创建代理。现在已经定义了两个工具：本地检索器和在线搜索工具，接下来需要创建一个智能代理来决定如何使用这些工具。

```
#使用 Gemma 模型和定义的工具创建代理
from langchain.agents import BasicAgent
#定义代理决策逻辑
def agent_decision(input_text):
    if "本地" in input_text:
        #如果查询包含"本地",则使用本地检索器
        return faiss_vector_store.search(input_text.replace("本地", ""))
    else:
        #否则,默认使用在线搜索工具
        return online_search_tool.search(input_text)
#创建代理实例
agent = BasicAgent(decision_logic = agent_decision)
```

(4) 运行代理并分析结果。通过定义的代理，可以对不同的查询进行处理，并根据查询的内容决定是调用本地检索器还是在线搜索工具。

```
#示例查询
queries = ["本地如何上传数据集", "San Francisco 的天气如何"]
for query in queries:
    result = agent.invoke({"input": query})
    print(f"查询: {query}\n 结果: {result}\n")
```

这个案例展示了如何通过整合不同的工具和智能决策逻辑来构建一个功能强大的智能代理。代理根据输入的查询内容，自动选择最合适的工具进行信息检索，无论是从本地数据源中查找还是通过在线搜索。这种方法不仅提高了信息检索的灵活性和准确性，也展示了如何利用 LangChain 框架和大模型技术来构建复杂的智能系统。

3.5　链

在 LangChain 框架中，链(Chains)扮演着至关重要的角色，它们是将不同功能模块串联起来，以实现复杂工作流程的基石。通过 LCEL(LangChain 表达式语言)，开发者可以灵活构建出满足特定需求的链，从而在智能应用中实现高效的数据处理和决策支持。

(1) 问答链(QA Chain)。假设需要构建一个问答系统，该系统能够从大量文档中检索到与用户问题最相关的信息，并基于这些信息提供答案。可以使用以下步骤构建一个问答链。

```
from langchain.chains import create_retrieval_chain
from langchain.llms import Ollama as OllamaLLM
from langchain.vectorstores import FAISS
#初始化检索器和语言模型
retriever = FAISS.load_local("path/to/faiss_index")
llm = OllamaLLM(model = "gemma:2b")
#构建问答链
qa_chain = create_retrieval_chain(retriever, llm)
#执行问答链
query = "如何提高编程能力?"
```

```
result = qa_chain.invoke(query)
print(result)
```

(2) 对话式文档问答链(Conversational Retrieval Chain)。在一些场景中,我们希望系统能够理解并记忆与用户的对话历史,以便在提供答案时考虑上下文信息。这就需要构建一个对话式的文档问答链:

```
from langchain.chains import ConversationalRetrievalChain
from langchain.llms import Ollama as OllamaLLM
from langchain.vectorstores import FAISS
#初始化检索器和语言模型
retriever = FAISS.load_local("path/to/faiss_index")
llm = OllamaLLM(model = "gemma:2b")
#构建对话式问答链
conv_qa_chain = ConversationalRetrievalChain(retriever = retriever, llm = llm)
#模拟对话
chat_history = []
while True:
    query = input("用户: ")
    result = conv_qa_chain.invoke({"question": query, "chat_history": chat_history})
    chat_history.append((query, result["answer"]))
    print(f"助手: {result['answer']}")
```

(3) 多功能智能助手链。LangChain 的灵活性允许我们构建更为复杂的链,如一个集成了信息检索、数据分析、实时计算等多种功能的智能助手链。

```
from langchain.chains import MultiFunctionChain
from langchain.llms import Ollama as OllamaLLM
from langchain.vectorstores import FAISS
from langchain.tools import MathCalculator
#初始化组件
retriever = FAISS.load_local("path/to/faiss_index")
llm = OllamaLLM(model = "gemma:2b")
math_calculator = MathCalculator()
#构建多功能智能助手链
multi_func_chain = MultiFunctionChain(retriever = retriever, llm = llm, tools = [math_calculator])
#运行链以处理不同类型的查询
queries = ["北京的历史", "2 加 2 等于多少", "如何学习 Python"]
for query in queries:
    result = multi_func_chain.invoke(query)
    print(f"查询: {query}\n 结果: {result}\n")
```

随着 LangChain 生态的不断成熟和发展,开发者将能够利用更多预定义的链模板和组件,以及通过 LCEL 构建的自定义链,来创建功能更为丰富、响应更为灵活的智能应用。这些链不仅在特定场景下提供了极大的便利,而且它们的可组合性和可扩展性也意味着未来智能应用的可能性将变得无限广阔。

3.6 记　　忆

记忆(Memory)在构建语言模型应用中扮演着重要的角色,特别是在对话系统中。这种记忆功能使得对话系统能够参照过去的对话信息,从而更自然地与用户进行互动。

LangChain 提供了多样的工具来实现记忆功能，支持从简单的对话历史记录到复杂的世界模型构建。

（1）聊天记忆与聊天历史。

如图 3-5 所示，记忆系统需要支持两个基本操作：读取和写入。一般地，每个链都定义了一些核心执行逻辑，这些逻辑需要某些输入。其中一些输入直接来自用户，但有些输入可能来自记忆。在给定的运行中，链将与其记忆系统交互两次。第一次是在接收初始用户输入之后，但在执行核心逻辑之前，链将从其记忆系统中读取数据并扩充用户输入。第二次是在执行核心逻辑之后，但在返回答案之前，链会将当前运行的输入和输出写入记忆，以便将来的运行可以引用它们。

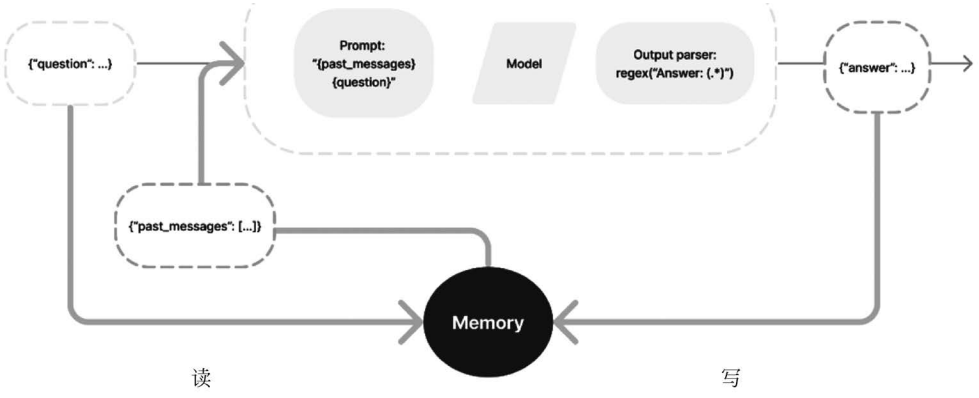

图 3-5　记忆系统的操作流程

在实现记忆功能时，一个核心考虑是如何有效地存储和查询对话的历史信息。LangChain 提供了 ConversationBufferMemory 等工具，使得开发者能够方便地存储用户和 AI 之间的互动记录。此外，通过使用本地大模型如 Gemma，开发者可以构建出更加智能的记忆系统，这些系统不仅能够回忆过去的对话内容，还能理解和提取其中的关键信息，如实体及其属性和关系。下面给出一个示例程序。

```
from langchain.memory import ConversationBufferMemory
from langchain.llms import Ollama as OllamaLLM
#初始化记忆存储和语言模型
memory = ConversationBufferMemory()
llm = OllamaLLM(model="gemma:2b")
#向记忆中添加对话信息
memory.chat_memory.add_user_message("你好!")
memory.chat_memory.add_ai_message("有什么可以帮到你的吗?")
#演示如何加载记忆变量
print(memory.load_memory_variables({}))
```

（2）定制记忆系统。

对于更高级的应用，LangChain 允许开发者定制自己的记忆系统。例如，开发者可以设计一个记忆系统，该系统基于 NLP 技术提取对话中提到的实体，并构建实体之间的关系模型。以下示例展示了如何基于 Gemma 模型和 spacy 库构建一个能够理解和记忆实体信息的记忆系统。

【例 3-11】 定制记忆系统(参考代码 3.9.py)。

```
from langchain.schema import BaseMemory
from langchain.llms import Ollama as OllamaLLM
import spacy
nlp = spacy.load("zh_core_web_sm")
class EntityMemory(BaseMemory):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.entity_map = {}
    def extract_entities(self, text):
        doc = nlp(text)
        return [ent.text for ent in doc.ents]
    def update_memory(self, question, answer):
        entities = self.extract_entities(question)
        for entity in entities:
            if entity not in self.entity_map:
                self.entity_map[entity] = []
            self.entity_map[entity].append(answer)
#示例: 使用实体记忆系统
memory = EntityMemory()
llm = OllamaLLM(model="gemma:2b")
```

在这个记忆系统中,使用 spacy 提取问题中的实体,然后将答案存储在与每个实体关联的列表中。这样,就可以跟踪每个实体的相关答案,并在需要时检索这些信息。然后,创建了一个 EntityMemory 的实例 memory,以及一个使用 Gemma:2b 模型的 OllamaLLM 实例 llm。这为使用实体记忆系统提供了基础设置。

注意,要运行这段代码,读者需要确保已经安装了所需的依赖项,特别是 spacy 和 zh_core_web_sm 模型。如果还没有安装,可以运行以下命令。

```
pip install spacy
python -m spacy download zh_core_web_sm
```

(3) 链与记忆的整合。

LangChain 支持将记忆功能与链(Chains)整合,以实现更复杂的对话流程。以下是一个示例,展示了如何将 EntityMemory 集成到对话链中,使得 AI 能够根据之前的对话内容和提取的实体信息来生成更准确的回答。

```
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate
#初始化提示模板
template = PromptTemplate(template="根据之前的对话和实体信息,回答问题: {question}")
llm_chain = LLMChain(llm=llm, prompt=template, memory=memory)
#使用链和记忆进行对话
result = llm_chain.invoke(question="张三最喜欢的运动是什么?")
print(result)
```

LangChain 为开发聊天机器人和其他对话式应用提供了丰富的记忆功能,通过与提示模板、链条等组件的灵活组合,使得应用能够实现更加自然和连贯的多轮对话。记忆在构建智能对话系统中扮演着不可或缺的角色,它不仅能够降低加入记忆能力的难度,还让开发者更加专注于应用的核心逻辑。此外,LangChain 还提供了接口让开发者定制自己的记忆实现,从而大大增强了开发的灵活性。

在 LangChain 中，记忆是指在链条或代理执行之间持久保持的状态，为对话和交互式应用开发者提供了关键的优势。例如，通过将聊天历史上下文存储在记忆中，可以随着时间的推移提高 LLM 响应的连贯性和相关性。此外，记忆中的信息可以减少对 LLM 的重复调用，从而降低 API 使用成本，同时为代理或链条提供所需的上下文。

LangChain 提供了多种记忆选项，包括 ConversationBufferMemory、ConversationBufferWindowMemory、ConversationKGMemory 等，用于存储模型历史中的所有消息或仅保留最近的消息。此外，LangChain 还集成了多种数据库选项，如 SQL 选项（Postgres 和 SQLite）、NoSQL 选择（MongoDB 和 Cassandra）、内存数据库 Redis，以及托管的云服务 AWS DynamoDB，用于持久存储。专为记忆服务器设计的 Remembrall 和 Motörhead 等工具提供了优化的对话上下文。选择合适的记忆方法取决于持久性需求、数据关系、规模和资源等因素，但在对话和交互式应用中强健地保持状态是至关重要的。

3.7　回　　调

在构建语言模型应用时，常常需要在模型运行的不同阶段执行一些额外的操作，如记录日志、监控进度、流式传输结果等。LangChain 提供了一个强大的回调（Callbacks）系统，让人们能够方便地在语言模型、链、工具、代理等组件的各个阶段插入自定义的处理逻辑。本节详细介绍 LangChain 的回调机制以及如何使用它来增强应用。

（1）回调处理器。

要使用 LangChain 的回调功能，首先需要定义一个或多个回调处理器（CallbackHandler）。回调处理器是一个实现了特定接口的类，其中包含在不同事件发生时会被调用的方法。以下是一些常见的回调事件。

- on_llm_start：当语言模型开始运行时触发。
- on_llm_end：当语言模型运行结束时触发。
- on_llm_error：当语言模型运行出错时触发。
- on_chain_start：当链开始运行时触发。
- on_chain_end：当链运行结束时触发。
- on_tool_start：当工具开始运行时触发。
- on_tool_end：当工具运行结束时触发。
- on_agent_action：当代理执行动作时触发。

下面是一个简单的回调处理器示例，它在语言模型生成每个新 token 时打印出该 token。

```
from langchain.callbacks.base import BaseCallbackHandler
class MyHandler(BaseCallbackHandler):
    def on_llm_new_token(self, token: str, **kwargs) -> None:
        print(f"New token generated: {token}")
```

（2）使用回调处理器。

定义好回调处理器后，可以通过以下两种方式将其附加到语言模型应用的不同组件上。

① 构造时回调（Construct callbacks）：在初始化组件时，通过 callbacks 参数传入回调

处理器列表。在这种方式下，回调将在该组件的整个生命周期内生效。

```
from langchain_community.llms import Ollama as OllamaLLM
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate
handler = MyHandler()
llm = Gemma(callback=[handler])
prompt = PromptTemplate()
chain = LLMChain(llm=llm, prompt=prompt, callbacks=[handler])
```

② 请求时回调(Request callbacks)：在调用组件的 run()/call()/apply() 等方法时，通过 callbacks 参数传入回调处理器列表。在这种方式下，回调仅在该次请求中生效。

```
chain.run("问题", callbacks=[handler])
```

(3) 异步回调。

如果使用了异步的语言模型或链，则需要定义异步版本的回调处理器，即继承 AsyncCallbackHandler 类。

```
import asyncio
from langchain.callbacks.base import AsyncCallbackHandler
class MyAsyncHandler(AsyncCallbackHandler):
    async def on_llm_start(self, serialized: Dict[str, Any], prompts: List[str], **kwargs:
Any) -> None:
        await asyncio.sleep(1)
        print("异步回调, LLM 开始运行")
```

使用异步回调处理器可以避免在异步运行流程中阻塞事件循环。

(4) 流式传输。

回调的一个常见应用是实现语言模型的流式传输，即在模型生成 responses 的过程中实时返回中间结果。可以通过传入一个自定义的回调处理器并在 on_llm_new_token()方法中处理每个新生成的 token 来达到此目的。

下面的例子展示了如何使用 Gemma 语言模型和自定义回调处理器来实现流式传输。

```
from langchain_community.llms import Ollama as OllamaLLM
from langchain.callbacks.streaming_stdout import StreamingStdOutCallbackHandler
llm = Gemma(streaming=True, callbacks=[StreamingStdOutCallbackHandler()])
prompt = """
对爱好长跑的孩子说些鼓励的话，让他能坚持下去训练。以富有诗意的语言书写一段话，不少于
150 字。
"""
llm(prompt)
```

运行上述代码，将看到模型生成的文本被一个词一个词地实时打印出来，而不是等到生成完整段落后才一次性返回。

(5) 记录日志。

另一个常见的回调应用是记录日志。除了使用内置的 StdOutCallbackHandler 将日志打印到控制台，还可以使用 FileCallbackHandler 将日志写入文件，或者自定义日志格式和存储方式。

下面的示例展示了如何将语言模型和链的运行日志记录到文件中。

```
from langchain_community.llms import Ollama as OllamaLLM
```

```
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate
from langchain.callbacks.file import FileCallbackHandler
handler = FileCallbackHandler("gemma.log")
llm = Gemma(callbacks=[handler])
prompt = PromptTemplate(
    input_variables=["question"],
    template="启动我的{question}需要哪些步骤?"
)
chain = LLMChain(llm=llm, prompt=prompt, callbacks=[handler])
chain.run("航天火箭")
```

运行后可以打开 gemma.log 文件查看详细的日志记录，其中包含 prompt 内容、生成过程、最终输出等信息。

(6) 多个回调处理器。

在实际项目中，可能需要同时使用多个回调处理器，例如，一个用于流式传输、一个用于记录日志。LangChain 支持传入一个回调处理器列表，系统会自动调用列表中的每一个处理器。

```
from langchain.agents import initialize_agent
from langchain.callbacks.base import BaseCallbackHandler
class CallbackOne(BaseCallbackHandler):
    def on_chain_start(self, serialized, inputs, **kwargs):
        print("CallbackOne - Chain 开始运行")
class CallbackTwo(BaseCallbackHandler):
    def on_tool_end(self, output, **kwargs):
        print("CallbackTwo - 工具执行结束")
agent = initialize_agent(
    ..., #其他初始化参数
    callbacks=[CallbackOne(), CallbackTwo()]
)
agent.run("帮我订一张从北京到上海的机票")
```

通过使用回调，可以在不修改核心逻辑的情况下，方便地扩展语言模型应用的功能，添加日志记录、流式传输、进度监控等附加特性。在实践中，可以根据实际需求选择使用内置的回调处理器，如 StdOutCallbackHandler、FileCallbackHandler 等，也可以通过继承 BaseCallbackHandler 或 AsyncCallbackHandler 来实现自定义的回调处理器。灵活运用回调机制，可以让语言模型应用更加健壮和易于管理。

【例 3-12】 回调函数综合演示(代码参见 3.10huidiao.py)。

```
import asyncio
from typing import Any, Dict, List
from langchain_community.llms import Ollama as OllamaLLM
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate
from langchain.callbacks.base import BaseCallbackHandler
from langchain.callbacks.streaming_stdout import StreamingStdOutCallbackHandler
from langchain.callbacks.file import FileCallbackHandler
from langchain.schema import LLMResult
#定义同步回调处理器
class SyncHandler(BaseCallbackHandler):
    def on_llm_start(self, serialized: Dict[str, Any], prompts: List[str], **kwargs: Any) -> None:
        """
```

```
            当 LLM 开始运行时触发
            :param serialized: 序列化后的 LLM 参数
            :param prompts: 输入的提示信息列表
            :param kwargs: 其他参数
            """
            print(f"LLM 开始运行, Prompts: {prompts}")
        def on_llm_end(self, response: LLMResult, **kwargs: Any) -> None:
            """
            当 LLM 运行结束时触发
            :param response: LLM 的响应结果
            :param kwargs: 其他参数
            """
            print(f"LLM 运行结束, Response: {response}")
        def on_llm_new_token(self, token: str, **kwargs: Any) -> None:
            """
            当 LLM 生成新的 token 时触发
            :param token: 新生成的 token
            :param kwargs: 其他参数
            """
            print(f"New token generated: {token}", end="", flush=True)
        def on_llm_error(self, error: Exception, **kwargs: Any) -> None:
            """
            当 LLM 运行出错时触发
            :param error: 异常对象
            :param kwargs: 其他参数
            """
            print(f"LLM 运行出错: {error}")
    #初始化 Gemma 模型
    llm = OllamaLLM(
        model="gemma:2b",
        callbacks=[SyncHandler(), StreamingStdOutCallbackHandler()],
        verbose=True,
    )
    #定义提示模板
    prompt = PromptTemplate(
        input_variables=["product"],
        template="请帮我撰写一段关于{product}的产品介绍,不少于 100 字。",
    )
    #创建 LLMChain
    chain = LLMChain(
        llm=llm,
        prompt=prompt,
        callbacks=[SyncHandler(), FileCallbackHandler("gemma.log")],
    )
    #运行 LLMChain
    chain.run("智能手表")
```

在这个案例中,展示了如何使用回调函数与本地大模型 Gemma 进行交互,涵盖了多个关键环节。首先,定义了同步回调处理器 SyncHandler,它实现了 on_llm_start()和 on_llm_end()方法,这些方法分别在 LLM(大语言模型)开始和结束运行时被触发,以便进行必要的操作。接着,引入了异步回调处理器 AsyncHandler,该处理器通过实现 on_llm_new_token()和 on_llm_error()方法,能够在 LLM 生成新的 token 和遇到运行错误时进行相应的响应。进一步地,初始化了 Gemma 模型,并通过 callbacks 参数传入 SyncHandler、AsyncHandler 以及 StreamingStdOutCallbackHandler,这样做启用了流式传输和详细输出,增强了用户与模

型交互的实时性和透明度。然后，创建了 LLMChain，将 Gemma 模型、提示模板和回调处理器结合起来，其中，FileCallbackHandler 用于将运行日志写入文件，这是监控模型运行状态和输出的有效方式。最后，通过运行 LLMChain 并传入 product 参数，触发了回调函数，实现了对模型运行过程的全面控制。

运行该案例的结果表明，在控制台上可以实时打印出生成的 token，同时还能打印出 LLM 开始和结束运行的信息。如果 LLM 运行出现错误，相应的错误信息也会被立即打印到控制台。此外，生成的结果被有效地写入了 gemma.log 文件，方便用户后续的查阅和分析。

通过这个案例，用户可以深入了解如何利用 LangChain 的回调机制与本地大模型 Gemma 进行互动，实现了日志记录、流式传输、异步处理等多种功能。这种机制为用户提供了高度的自定义能力，可以根据实际需求调整回调处理器，满足不同的应用场景。

小　结

本章全面介绍了 LangChain 框架的基础组件，展示了如何利用这些组件高效地构建 LLM 应用。首先通过一个快速入门案例，演示了如何使用 LLM 链、检索链、对话检索链和代理来构建基本的 LLM 应用。接着，详细探讨了 LangChain 中的关键组件，包括模型与提示模板、文档加载器与文本分割器、文本嵌入与向量存储、检索器与索引等。此外，本章还介绍了如何使用 LangChain 构建代理、链条和记忆系统，以实现更加智能和复杂的 LLM 应用。最后，讨论了回调机制在 LLM 应用开发中的重要作用。通过本章的学习，读者应该掌握了使用 LangChain 进行 LLM 应用开发的基本技能和思路。

思　考　题

一、简答题

1. 在 LangChain 中什么是链？
2. 什么是代理？
3. 什么是记忆？为什么需要它？
4. LangChain 中有哪些类型的工具？
5. LangChain 是如何工作的？
6. Gemma 等本地大模型与 OpenAI 等 API 模型相比，在应用开发中有哪些优势？
7. 在构建检索增强的问答系统时，应如何选择和优化文档切分策略？
8. 你认为未来工具(Tool)将在 LLM 应用开发中扮演怎样的角色？

二、实践题

使用 Gemma 本地 LLM 和维基百科的数据，构建一个能够回答关于中国历史人物的问题的智能助手。要求：

1. 使用维基百科的中文数据源，通过文档加载器和文本分割器，构建向量索引。
2. 使用 Gemma 的 embedding 接口生成文本嵌入向量。
3. 使用问答链实现检索增强的问答功能，并基于上下文完成多轮对话。
4. 使用回调实现聊天记录，并允许将聊天记录导出为 Markdown 文件。