编码为向量 h_c，作为查询向量，用于在记忆库 M 中搜索与之最相关的记忆片段。在此框架下，编码器 $E(\cdot)$ 可根据具体应用需求替换为任何适合的模型，从而灵活适配不同的场景。

通过持久的记忆存储和记忆检索，智能体记忆存储能力可以得到极大的提升。然而，在需要更具人类化记忆行为的场景中（如 AI 伴侣等），引入记忆更新机制尤为重要。忘记不重要且长时间未被调用的记忆片段，可以使智能体的行为更加自然。

记忆遗忘机制受到艾宾浩斯遗忘曲线理论的启发，并遵循以下基本原则：（1）遗忘速率：记忆保留率会随时间迅速下降，除非通过有意识的复习进行强化；（2）时间与记忆衰减：遗忘曲线在开始时陡峭，表明大部分信息会在学习后的数小时或数天内被遗忘，随后遗忘速率逐渐减缓；（3）间隔效应：重新学习比首次学习更容易，定期复习可以重置遗忘曲线，减缓遗忘速率，从而提高记忆保持能力。

艾宾浩斯遗忘曲线采用指数衰减模型表示：$R = e^{(-t/S)}$，其中 R 代表记忆保留率，即信息被保留的比例；t 是从学习到现在经过的时间；S 是记忆强度，受学习深度和重复次数等因素影响。为简化记忆更新过程，将 S 模型化为离散值，并在某项记忆首次出现在对话中时将其初始化为 1。当某项记忆在对话中被调用时，其在记忆中的存留时间会延长，将 S 增加 1，并将 t 重置为 0，从而降低遗忘的概率。需要注意，这是一种探索性且高度简化的记忆更新模型，而实际的记忆过程更复杂，会受到多种因素的影响。遗忘曲线在不同人群和不同信息类型中表现各异。

8.4 大模型智能体实践

大模型智能体的构建方式多样化，主要包括手工编写代码、使用框架开发以及采用低代码平台三种方式：1）手工编写代码，开发者通过直接编写代码，可以灵活地设计模型结构、任务流程和外部接口。这种方式通常需要较高的技术能力和充足的开发时间，但能够实现高度定制化，适用于复杂场景或特定需求的智能体开发；2）使用框架开发，基于现有的开发框架（如 LangChain、Haystack、AutoGPT、LLaMA Index 等）进行智能体构建，框架通常提供了模块化的工具和组件，包括记忆管理、检索增强生成等功能。开发者可以利用这些框架快速构建智能体，同时保留一定的灵活性，适合中等复杂度的应用场景；3）低代码/零代码平台（如 Coze、Microsoft Copilot Studio、Hugging Face AutoTrain 等）为非技术用户提供了便捷的开发方式，只需少量编程甚至无需编程即可搭建智能体。这种方式降低了开发门槛，适合快速验证概念或简单应用的实现，但定制化能力有限。

本节将分别介绍编写代码、使用框架开发以及采用低代码平台三种方式构建大模型智能体方法。

8.4.1 手工编写代码

手工编写代码是一种构建大模型智能体的方式之一，适合对系统有完全掌控需求的开发者。这种方法通过从零开始直接编写代码，使开发者能够灵活地设计模型结构、任务流程以及外部接口。

手工编写代码提供了最大的自由度，可以根据具体需求对智能体进行深度优化和定制。这种方法尤其适用于复杂场景或特定领域的智能体开发，例如需要实现垂直领域的功能、整合独特的业务逻辑或优化性能的场景。

本节以辩论和角色扮演为例，介绍手工编写代码实现大模型智能体样例。

1. 辩论

人类之间的交流大多是以语言为媒介完成的，基于大语言模型实现的智能体，可以完成谈判、辩论等基于语言的多轮交流。在每一轮中，每个智能体都会表达自己的观点，同时收集其他智能体的观点，以此作为下一轮生成的参考；直至多个智能体达成共识才结束上述辩论循环。研究表明，当多个智能体以“针锋相对（Tit for Tat）”的状态表达自己的观点时，单个智能体可以从其他智能体处获得充分的外部反馈，以此纠正自己的“扭曲思维”；当检测到自己的观点与其他智能体的观点出现矛盾时，智能体会仔细检查每个步骤的推理和假设，进一步改进自己的解决方案。

以解决数学问题的任务（数据集可以从 GitHub 上 OpenAI 的 grade-school-math 项目中获取）为例，最简单的交互实现可大致分为以下步骤。

（1）对于每个任务，用户首先描述任务的基本需求：

```
question = "Jimmy has $2 more than twice the money Ethel has. \
            If Ethal has $8, how much money is Jimmy having?"  # 用户提出问题
agent_contexts = [[{"role": "user", "content": """Can you solve the following math
                                                 problem? {} Explain your reasoning.
                                                 Your final answer should be a single
                                                 numerical number, in the form
                                                 \\boxed{{answer}}, at the end of your
                                                 response.""".format(question)}]
for agent in range(agents)]  # 为每一个智能体构造输入提示
```

（2）每个智能体按一定顺序依次发言：

```
for i, agent_context in enumerate(agent_contexts):  # 每一个智能体
    completion = openai.ChatCompletion.create(      # 发言
              model="gpt-3.5-turbo-0301",  # 选择模型
              messages=agent_context,      # 智能体的输入
              n=1)
    content = completion["choices"][0]["message"]["content"]  # 提取智能体生成的文本内容
    assistant_message = {"role": "assistant", "content": content}  # 修改角色为智能体
    agent_context.append(assistant_message)                # 将当前智能体的发言添加至列表
```

（3）每个智能体接收来自其他智能体的发言，并重新思考：

```
for i, agent_context in enumerate(agent_contexts):  # 对每一个智能体
    if round != 0:  # 第一轮不存在来自其他智能体的发言
        # 获取除自己以外，其他智能体的发言
        agent_contexts_other = agent_contexts[:i] + agent_contexts[i+1:]
        # construct_message()函数：构造提示用作智能体的下一轮输入
        message = construct_message(agent_contexts_other, question, 2*round - 1)
        agent_context.append(message)  # 将当前智能体的下一轮输入添加至列表
```

（4）重复步骤（2）和步骤（3），直至多个智能体达成一致意见或迭代达到指定轮次。完整的实现代码如下：

```
agents = 3            # 指定参与的智能体个数
rounds = 2            # 指定迭代轮次上限
question = "Jimmy has $2 more than twice the money Ethel has. \
           If Ethal has $8, how much money is Jimmy having?"  # 用户提出问题
agent_contexts = [[{"role": "user", "content": """Can you solve the following math
                                                 problem? {} Explain your reasoning.
                                                 Your final answer should be a single
                                                 numerical number, in the form
                                                 \\boxed{{answer}}, at the end of your
                                                 response.""".format(question)}]
                  for agent in range(agents)]  # 为每一个智能体构造输入提示

for round in range(rounds):                                  # 对每一轮迭代
    for i, agent_context in enumerate(agent_contexts):  # 对每一个智能体
        if round != 0:  # 第一轮不存在来自其他智能体的发言
            # 获取除自己以外，其他智能体的发言
            agent_contexts_other = agent_contexts[:i] + agent_contexts[i+1:]
            # construct_message()函数：构造提示用作智能体的下一轮输入
            message = construct_message(agent_contexts_other, question, 2*round - 1)
            agent_context.append(message)  # 将当前智能体的下一轮输入添加至列表
        completion = openai.ChatCompletion.create(      # 进行发言
                  model="gpt-3.5-turbo-0301",  # 选择模型
                  messages=agent_context,      # 智能体的输入
                  n=1)
        content = completion["choices"][0]["message"]["content"]  # 提取智能体生成的文本内容
        assistant_message = {"role": "assistant", "content": content}  # 修改角色为智能体
        agent_context.append(assistant_message)                # 将当前智能体的发言添加至列表

        print(assistant_message['content'])
```

本例中，多个智能体之间达成一致意见，不仅按照指定格式给出了正确的答案，更增强了答案的可靠性，具体输出如下：

```
# 第一轮输出
We know that Jimmy has $2 more than twice the money Ethel has.
Twice the money Ethel has is $8 x 2 = $16.
Two more than $16 is $16 + $2 = $18.
Therefore, Jimmy has $18.
Answer: $\boxed{18}$.

We know that Jimmy has $2 more than twice the money Ethel has.
Twice the money Ethel has is $8*2=<<8*2=16>>16.
Adding $2 to this, we get that Jimmy has $16+$2=$\boxed{18}$.

Twice the money Ethel has is $8\cdot 2=16$.
Jimmy has $2$ more than that, so his total is $16+2=\boxed{18}$.

# 第二轮输出
Based on the solutions provided by other agents, I also arrive at the answer:
Jimmy has twice the money Ethel has, which is $8*2=$16,
and he also has $2 more than that, which is $16+$2=$\boxed{18}$.

Yes, based on the information provided and the solutions given by other agents, Jimmy has $18.
Answer: $\boxed{18}$.

Given that Ethel has $8 and Jimmy has $2 more than twice Ethel's money,
we can calculate Jimmy's money as follows.
Twice Ethel's money is $8 \times 2 = $16.
Adding $2 to this, we get that Jimmy has $16 + $2 = $\boxed{18}$.
```

2. 角色扮演

角色扮演（Role-Playing）是指在事先设计的情景中自然地扮演某个角色。通过构造特定的提示，大语言模型有能力扮演不同的角色——无论是一个五年级的小学生，还是一个计算机领域的专家。令人意想不到的是，扮演特定角色的大语言模型能够激发其内部独特的领域知识，产生比没有指定角色时更好的答案。角色扮演在赋予智能体个体优势和专业技能的同时，更在多个智能体的协作交流中体现出了极大的价值，大大提高了多智能体系统的问题解决效率。

CAMEL 是角色扮演的经典应用实例，该框架实现了两个智能体的交互，其中一个智能体作为用户，另一个智能体作为助手。此外，CAMEL 中还允许用户自由选择是否需要设置任务明确智能体与评论智能体，任务明确智能体专门负责将人类给出的初始任务提示细致化，评论智能体则负责评价交互的内容，一方面引导交互向正确的方向进行，另一方面判定任务目标是否已达成。CAMEL 中定义了一个 RolePlaying 类，可以指定两个智能体的具体身份，给定任务提示，给出相关参数等。在实际使用过程中，可以直接调用此类来完成任务。以股票市场的机器人开发任务为例，代码示例如下：

```
role_play_session = RolePlaying(                                    # 直接调用核心类
    assistant_role_name="Python Programmer",                        # 指定助手智能体的具体身份
    assistant_agent_kwargs=dict(model=model_type),                  # 传递助手智能体的相关参数
    user_role_name="Stock Trader",                                  # 指定用户智能体的具体身份
    user_agent_kwargs=dict(model=model_type),                       # 传递用户智能体的相关参数
    task_prompt="Develop a trading bot for the stock market",       # 给定初始任务提示
    with_task_specify=True,                                         # 选择是否需要进一步明确任务
    task_specify_agent_kwargs=dict(model=model_type),               # 传递任务明确智能体的相关参数
)
```

其中，智能体的系统消息由框架自动生成，可以手动打印相关内容，命令如下：

```
print(f"AI Assistant sys message:\n{role_play_session.assistant_sys_msg}\n")
print(f"AI User sys message:\n{role_play_session.user_sys_msg}\n")
```

本示例中打印的内容如下：

```
AI Assistant sys message:
BaseMessage(role_name='Python Programmer',
            role_type=<RoleType.ASSISTANT: 'assistant'>,
            meta_dict={'task': 'Develop a Python trading bot for a stock trader ... ',
                       'assistant_role': 'Python Programmer', 'user_role': 'Stock Trader'},
            content='Never forget you are a Python Programmer and I am a Stock Trader.
                     Never flip roles! ...
                     Here is the task: ...
                     Never forget our task! ...
                     Unless I say the task is completed,
                     you should always start with: Solution: <YOUR_SOLUTION>...
                     Always end <YOUR_SOLUTION> with: Next request.')

AI User sys message:
BaseMessage(role_name='Stock Trader',
            role_type=<RoleType.USER: 'user'>,
            meta_dict={'task': 'Develop a Python trading bot for a stock trader ... ',
                       'assistant_role': 'Python Programmer', 'user_role': 'Stock Trader'},
            content='Never forget you are a Stock Trader and I am a Python Programmer.
                     Never flip roles! ...
                     Here is the task: ...
                     Never forget our task! ...
                     When the task is completed,
                     you must only reply with a single word <CAMEL_TASK_DONE>.
                     Never say <CAMEL_TASK_DONE> unless my responses have solved your task.')
```

基于 RolePlaying 的结构，可以直接调用 step() 函数完成角色扮演的交互过程，获取两个智能体的新一轮输出：

```
assistant_response, user_response = role_play_session.step(input_assistant_msg)
```

角色扮演的完整对话实现如下：

```
while n < chat_turn_limit:  # 迭代轮次限制
    # 获取两个智能体的新一轮输出
    assistant_response, user_response = role_play_session.step(input_assistant_msg)

    # 判断两个智能体是否结束对话
    if assistant_response.terminated:
        print("AI Assistant terminated. "
              f"Reason: {assistant_response.info['termination_reasons']}.")
        break
    if user_response.terminated:
        print("AI User terminated. "
              f"Reason: {user_response.info['termination_reasons']}.")
        break

    # 打印角色扮演的对话内容
    print(f"AI User:\n{user_response.msg.content}\n")
    print(f"AI Assistant:\n{assistant_response.msg.content}\n")

    # 根据用户智能体的反馈判断任务是否完成
    if "CAMEL_TASK_DONE" in user_response.msg.content:
        break

    input_assistant_msg = assistant_response.msg  # 更新角色扮演的下一轮输入
    n += 1  # 进行下一轮迭代
```

当用户智能体认为任务已完成，或迭代达到指定轮次时，对话将终止。本例中的部分对话如下：

AI User:
Instruction: Install the necessary Python libraries for developing a trading bot.
Input: None

AI Assistant:
Solution: To develop a trading bot in Python, we need to install several libraries that will help us with data analysis, machine learning, and executing trades. Here are the essential libraries you need to install:

1. Pandas: Pandas is a powerful library for data manipulation and analysis. It provides data structures and functions to efficiently handle and analyze large datasets.

To install Pandas, you can use the following command:

```
pip install pandas
```

...

Next request.

AI User:
Instruction: Import the necessary libraries for developing the trading bot.
Input: None

AI Assistant:
Solution: To import the necessary libraries for developing the trading bot, you need to include the following import statements in your Python script:

```python
import pandas as pd
import numpy as np
import sklearn
import matplotlib.pyplot as plt
import alpaca_trade_api as tradeapi
```

...

Next request.

AI User:
Instruction: Connect to the Alpaca API using your API keys.
Input: Alpaca API key and secret key.

AI Assistant:
Solution: To connect to the Alpaca API using your API keys, you can use the following code:

```python
import alpaca_trade_api as tradeapi

api_key = "YOUR_API_KEY"
```

8.4.2 LangChain 框架

大语言模型的调用相对简单，仍需要完成大量的定制开发工作，包括 API 集成、交互逻辑、数据存储等。为了解决这个问题，从 2022 年开始，多家机构和个人陆续推出了大量开源项目，帮助开发者快速创建基于大语言模型的端到端应用程序或流程，其中较为著名的是 LangChain 框架。LangChain 框架是一种利用大语言模型的能力开发各种下游应用的开源框架，旨在为各种大语言模型应用提供通用接口，简化大语言模型应用的开发难度。它可以实现数据感知和环境交互，即能够使语言模型与其他数据源连接起来，并允许语言模型与其环境进行交互。

本节将重点介绍 LangChain 框架以及其核心模块组成。

1. LangChain 框架核心模块

使用 LangChain 框架的核心目标是连接多种大语言模型（如 ChatGPT、LLaMA 等）和外部资源（如 Google、Wikipedia、Notion 及 Wolfram 等），提供抽象组件和工具以在文本输入和输出之间进行接口处理。大语言模型和组件通过“链（Chain）”连接，使得开发人员可以快速开发原型系统和应用程序。LangChain 的主要价值体现在以下几个方面。

（1）组件化：LangChain 框架提供了用于处理大语言模型的抽象组件，以及每个抽象组件的一系列实现。这些组件具有模块化设计，易于使用，无论是否使用 LangChain 框架的其他部分，都可以方便地使用这些组件。

（2）现成的链式组装：LangChain 框架提供了一些现成的链式组装，用于完成特定的高级任务。这些现成的链式组装使得入门变得更加容易。对于更复杂的应用程序，LangChain 框架也支持自定义现有链式组装或构建新的链式组装。

（3）简化开发难度：通过提供组件化和现成的链式组装，LangChain 框架可以大大简化大语言模型应用的开发难度。开发人员可以更专注于业务逻辑，而无须花费大量时间和精力处理底层技术细节。

LangChain 提供了以下 6 种标准化、可扩展的接口，并且可以外部集成：**模型输入/输出**（Model I/O），与大语言模型交互的接口；**数据连接**（Data Connection），与特定应用程序的数据进行交互的接口；**链**（Chain），用于复杂应用的调用序列；**记忆**（Memory），用于在链的多次运行之间持久化应用程序状态；**智能体**（Agent），语言模型作为推理器决定要执行的动作序列；**回调**（Callback），用于记录和流式传输任何链式组装的中间步骤。下文中的介绍和代码基于 LangChain V0.0.248 版本（2023 年 7 月 31 日发布）。

2. 模型输入/输出

LangChain 中的模型输入/输出模块是与各种大语言模型进行交互的基本组件，是大语言模型应用的核心元素。该模块的基本流程如图8.8 所示，主要包含以下部分：Prompts、Language Models 及 Output Parsers。将用户的原始输入与模型和示例进行组合输入大语言模型，再根据大语言模型的返回结果进行输出或者结构化处理。

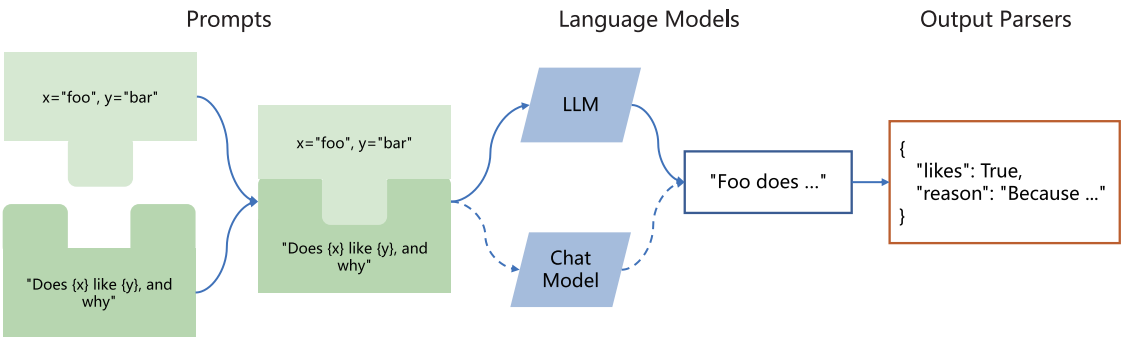

图 8.8 LangChain 模型输入/输出模块的基本流程

Prompts 部分的主要功能是提示词模板、提示词动态选择和输入管理。提示词是指输入模型的内容。该输入通常由模板、示例和用户输入组成。LangChain 提供了几个类和函数，使得构建和处理提示词更加容易。LangChain 中的 PromptTemplate 类可以根据模板生成提示词，它包含了一个文本字符串（模板），可以根据从用户处获取的一组参数生成提示词。以下是一个简单的示例：

```
from langchain import PromptTemplate

template = """\
You are a naming consultant for new companies.
What is a good name for a company that makes {product}?
"""

prompt = PromptTemplate.from_template(template)
prompt.format(product="colorful socks")
```

通过上述代码，可以获取最终的提示词“You are a naming consultant for new companies. What is a good name for a company that makes colorful socks?”

如果有大量的示例,则可能需要选择将哪些示例包含在提示词中。LangChain 中提供了 Example Selector 以提供各种类型的选择，包括 LengthBasedExampleSelector、MaxMarginalRelevanceExampleSelector、SemanticSimilarityExampleSelector、NGramOverlapExampleSelector 等，可以提供按照句子长度、最大边际相关性、语义相似度、n-gram 覆盖率等多种指标进行选择的方式。例如，基于句子长度的筛选器的功能是这样的：当用户输入较长时，该筛选器可以选择简洁的模板，而面对较短的输入则选择详细的模板。这样做可以避免输入总长度超过模型的限制。

Language Models 部分提供了与大语言模型的接口，LangChain 提供了两种类型的模型接口和集成：LLM，接收文本字符串作为输入并返回文本字符串；Chat Model，由大语言模型支持，但接收聊天消息（Chat Message）列表作为输入并返回聊天消息。在 LangChain 中，LLM 指纯文本补

全模型，接收字符串提示词作为输入，并输出字符串。OpenAI 的 GPT-3 是 LLM 实现的一个实例。Chat Model 专为会话交互设计，与传统的纯文本补全模型相比，这一模型的 API 采用了不同的接口方式：它需要一个标有说话者身份的聊天消息列表作为输入，如“系统”、“AI”或“人类”。作为输出，Chat Model 会返回一个标为“AI”的聊天消息。GPT-4 和 Anthropic 的 Claude 都可以通过 Chat Model 调用。以下是利用 LangChain 调用 OpenAI API 的代码示例：

```
from langchain.chat_models import ChatOpenAI
from langchain.schema import (AIMessage, HumanMessage, SystemMessage)

chat = ChatOpenAI(
    openai_api_key="...",
    temperature=0,
    model='gpt-3.5-turbo'
)
messages = [
    SystemMessage(content="You are a helpful assistant."),
    HumanMessage(content="Hi AI, how are you today?"),
    AIMessage(content="I'm great thank you. How can I help you?"),
    HumanMessage(content="I'd like to understand string theory.")
]

res = chat(messages)
print(res.content)
```

上例中，HumanMessage 表示用户输入的消息，AIMessage 表示系统回复用户的消息，SystemMessage 表示设置的 AI 应该遵循的目标。程序中还会有 ChatMessage，表示任务角色的消息。上例调用了 OpenAI 提供的 gpt-3.5-turbo 模型接口，可能返回的结果如下：

```
Sure, I can help you with that. String theory is a theoretical framework in physics that
attempts to reconcile quantum mechanics and general relativity. It proposes that the
fundamental building blocks of the universe are not particles, but rather tiny,
one-dimensional "strings" that vibrate at different frequencies. These strings are
incredibly small, with a length scale of around 10^-35 meters.

The theory suggests that there are many different possible configurations of these
strings, each corresponding to a different particle. For example, an electron might
be a string vibrating in one way, while a photon might be a string vibrating in a
different way.
    ...
```

Output Parsers 部分的目标是辅助开发者从大语言模型输出中获取比纯文本更结构化的信息。

Output Parsers 包含很多具体的实现，但是必须包含如下两个方法。

（1）获取格式化指令（Get format instructions），返回大语言模型输出格式化的方法。

（2）解析（Parse）接收的字符串（假设为大语言模型的响应）为某种结构的方法。

还有一个可选的方法：带提示解析（Parse with prompt），接收字符串（假设为语言模型的响应）和提示（假设为生成此响应的提示）并将其解析为某种结构的方法。例如，PydanticOutputParser 允许用户指定任意的 JSON 模式，并通过构建指令的方式与用户输入结合，使得大语言模型输出符合指定模式的 JSON 结果。以下是 PydanticOutputParser 的使用示例：

```
from langchain.prompts import PromptTemplate, ChatPromptTemplate, HumanMessagePromptTemplate
from langchain.llms import OpenAI
from langchain.chat_models import ChatOpenAI

from langchain.output_parsers import PydanticOutputParser
from pydantic import BaseModel, Field, validator
from typing import List

model_name = 'text-davinci-003'
temperature = 0.0
model = OpenAI(model_name=model_name, temperature=temperature)

# 定义期望的数据结构
class Joke(BaseModel):
    setup: str = Field(description="question to set up a joke")
    punchline: str = Field(description="answer to resolve the joke")

    # 使用Pydantic轻松添加自定义验证逻辑
    @validator('setup')
    def question_ends_with_question_mark(cls, field):
        if field[-1] != '?':
            raise ValueError("Badly formed question!")
        return field

# 设置解析器并将指令注入提示模板
parser = PydanticOutputParser(pydantic_object=Joke)

prompt = PromptTemplate(
    template="Answer the user query.\n{format_instructions}\n{query}\n",
    input_variables=["query"],
    partial_variables={"format_instructions": parser.get_format_instructions()}
)

# 这是一个旨在提示大语言模型填充数据结构的查询
joke_query = "Tell me a joke."
_input = prompt.format_prompt(query=joke_query)

output = model(_input.to_string())

parser.parse(output)
```

如果是能力足够强的大语言模型，例如这里使用的 text-davinci-003 模型，就可以返回如下格式的输出：

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

3. 数据连接

许多大语言模型应用需要使用用户特定的数据，这些数据不是模型训练集的一部分。为了支持上述应用的构建，LangChain 数据连接模块通过以下方式提供组件来加载、转换、存储和查询数据：Document loaders、Document transformers、Text embedding models、Vector stores 及 Retrievers。LangChain 数据连接模块的基本框架如图8.9 所示。

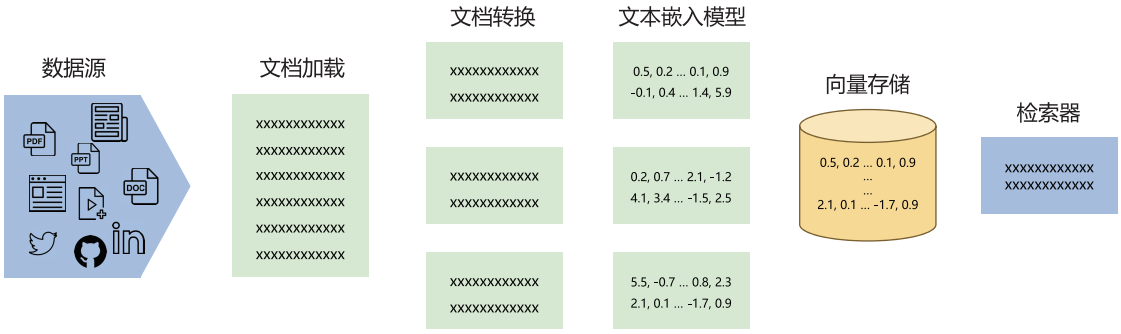

图 8.9　LangChain 数据连接模块的基本框架

Document loaders（文档加载）旨在从数据源中加载数据构建 Document。LangChain 中的 Document 包含文本和与其关联的元数据。LangChain 中包含加载简单文本文件的文档加载器，用于加载任何网页文本内容的加载器。以下是一个最简单的从文件中读取文本来加载数据的 Document 的示例：

```
from langchain.document_loaders import TextLoader

loader = TextLoader("./index.md")
loader.load()
```

根据上述示例获得的 Document 内容如下：

```
[
    Document(page_content='---\nsidebar_position: 0\n---\n# Document loaders\n\nUse document
    loaders to load data from a source as `Document`\'s. A `Document` is a piece of text\n and
    associated metadata. For example, there are document loaders for loading a simple `.txt`
    file, for loading the text\ncontents of any web page, or even for loading a transcript of
    a YouTube video.\n\nEvery document loader exposes two methods:\n1. "Load": load documents
    from the configured source\n2. "Load and split": load documents from the configured source
    and split them using the passed in text splitter\n\nThey optionally implement:\n\n
    3. "Lazy load": load documents into memory lazily\n',
    metadata={'source': '../docs/docs_skeleton/docs/modules/data_connection/document_loaders/
    index.md'})
]
```

Document transformers（文档转换）旨在处理文档，以完成各种转换任务，如将文档格式转化为 Q&A 形式、去除文档中的冗余内容等，从而更好地满足不同应用程序的需求。一个简单的文档转换示例是将长文档分割成较短的部分，以适应不同模型的上下文窗口大小。LangChain 中有许多内置的文档转换器，使拆分、合并、过滤文档及其他文档操作都变得很容易。以下是对长文档进行拆分的代码示例：

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

# 这是一个长文档，可以拆分处理
with open('../../wiki_computer_science.txt') as f:

text_splitter = RecursiveCharacterTextSplitter(
    # 为了显示，设置一个非常小的块尺寸
    chunk_size = 100,
    chunk_overlap  = 20,
    length_function = len,
    add_start_index = True,
)

texts = text_splitter.create_documents([state_of_the_union])
print(texts[0])
print(texts[1])
```

根据以上示例可以获得如下输出结果：

```
page_content='Computer science is the study of computation, information, and automation.
Members of Congress and' metadata={'start_index': 0}
page_content='and automation.
Computer science spans theoretical disciplines (such as algorithms,
                                                theory of computation, and information theory)'
metadata={'start_index': 60}
```

Text embedding models（文本嵌入模型）旨在将非结构化文本转换为嵌入表示。基于文本的嵌入表示可以进行语义搜索，查找最相似的文本片段。Embeddings 类则用于与文本嵌入模型进行交互，并为不同的嵌入模型提供统一的标准接口，包括 OpenAI、Cohere 等。LangChain 中的 Embeddings 类公开了两个方法：一个用于文档嵌入表示，另一个用于查询嵌入表示。前者输入多个文本，后者输入单个文本。之所以将它们作为两个单独的方法，是因为某些嵌入模型为文档和查询采用了不同的嵌入策略。以下是使用 OpenAI 的 API 接口完成文本嵌入的代码示例：

```
from langchain.embeddings import OpenAIEmbeddings
embeddings_model = OpenAIEmbeddings(openai_api_key="...")

embeddings = embeddings_model.embed_documents(
    [
        "Hi there!",
        "Oh, hello!",
        "What's your name?",
        "My friends call me World",
        "Hello World!"
    ]
)
len(embeddings), len(embeddings[0])

embedded_query = embeddings_model.embed_query("What was the name mentioned in this session?")
embedded_query[:5]
```

执行上述代码可以得到如下输出：

```
(5, 1536)
[0.0053587136790156364,
 -0.0004999046213924885,
 0.038883671164512634,
 -0.003001077566295862,
 -0.00900818221271038]
```

Vector Stores（向量存储）是存储和检索非结构化数据的主要方式之一。它首先将数据转化为嵌入表示，然后存储生成的嵌入向量。在查询阶段，系统会利用这些嵌入向量来检索与查询内容“最相似”的文档。向量存储的主要任务是保存这些嵌入向量并执行基于向量的搜索。LangChain 能够与多种向量数据库集成，如 Chroma、FAISS 和 Lance 等。以下为使用 FAISS 向量数据库的代码示例：

```
from langchain.document_loaders import TextLoader
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.text_splitter import CharacterTextSplitter
from langchain.vectorstores import FAISS

# 加载文档，将其分割成块，对每个块进行嵌入表示，并将其加载到向量存储中
raw_documents = TextLoader('../../../state_of_the_union.txt').load()
text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
documents = text_splitter.split_documents(raw_documents)
db = FAISS.from_documents(documents, OpenAIEmbeddings())

# 进行相似性搜索
query = "What did the president say about Ketanji Brown Jackson"
docs = db.similarity_search(query)
print(docs[0].page_content)
```

Retrievers（检索器）是一个接口，其功能是基于非结构化查询返回相应的文档。检索器不需要存储文档，只需要能根据查询要求返回结果即可。检索器可以使用向量存储的方式执行操作，也可以使用其他方式执行操作。LangChain 中的 BaseRetriever 类定义如下：

```python
from abc import ABC, abstractmethod
from typing import Any, List
from langchain.schema import Document
from langchain.callbacks.manager import Callbacks

class BaseRetriever(ABC):
    ...
    def get_relevant_documents(
        self, query: str, *, callbacks: Callbacks = None, **kwargs: Any
    ) -> List[Document]:
        """ 检索与查询内容相关的文档
        Args:
            query: 相关文档的字符串
            callbacks: 回调管理器或回调列表
        Returns:
            相关文档的列表
        """
        ...

    async def aget_relevant_documents(
        self, query: str, *, callbacks: Callbacks = None, **kwargs: Any
    ) -> List[Document]:
        """ 异步获取与查询内容相关的文档
        Args:
            query: 相关文档的字符串
            callbacks: 回调管理器或回调列表
        Returns:
            相关文档的列表
        """
        ...
```

它的使用非常简单，可以通过 get_relevant_documents 方法或通过异步调用 aget_relevant_ documents 方法获得与查询文档最相关的文档。基于向量存储的检索器（Vector store-backed retriever）是使用向量存储检索文档的检索器。它是向量存储类的轻量级包装器，与检索器接口契合，使用向量存储实现的搜索方法（如相似性搜索和 MMR）来查询使用向量存储的文本。以下是一个基于向量存储的检索器的代码示例：

```
from langchain.document_loaders import TextLoader
loader = TextLoader('../../../state_of_the_union.txt')

from langchain.text_splitter import CharacterTextSplitter
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings

documents = loader.load()
text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
texts = text_splitter.split_documents(documents)
embeddings = OpenAIEmbeddings()
db = FAISS.from_documents(texts, embeddings)

retriever = db.as_retriever()
docs = retriever.get_relevant_documents("what did he say about ketanji brown jackson")
```

4. 链

虽然独立使用大语言模型能够应对一些简单任务，但对于更加复杂的需求，可能需要将多个大语言模型进行链式组合，或与其他组件进行链式调用。LangChain 为这种“链式”应用提供了 Chain 接口，并将该接口定义得非常通用。作为一个调用组件的序列，其中还可以包含其他链。基本接口实现非常简单，代码示例如下：

```
class Chain(BaseModel, ABC):
    """ 所有链应该实现的基本接口"""

    memory: BaseMemory
    callbacks: Callbacks

    def __call__(
        self,
        inputs: Any,
        return_only_outputs: bool = False,
        callbacks: Callbacks = None,
    ) -> Dict[str, Any]:
        ...
```

链允许将多个组件组合在一起，创建一个单一的、连贯的应用程序。例如，可以创建一个链，接收用户输入，使用 PromptTemplate 对其进行格式化，然后将格式化后的提示词传递给大语言模型。也可以通过将多个链组合在一起或将链与其他组件组合来构建更复杂的链，代码示例如下：

```
from langchain.chat_models import ChatOpenAI
from langchain.prompts.chat import (
    ChatPromptTemplate,
    HumanMessagePromptTemplate,
)
human_message_prompt = HumanMessagePromptTemplate(
        prompt=PromptTemplate(
            template="What is a good name for a company that makes {product}?",
            input_variables=["product"],
        )
    )
chat_prompt_template = ChatPromptTemplate.from_messages([human_message_prompt])
chat = ChatOpenAI(temperature=0.9)
chain = LLMChain(llm=chat, prompt=chat_prompt_template)
print(chain.run("colorful socks"))
```

除了上例中的 LLMChain，LangChain 中的链还包含 RouterChain、SimpleSequentialChain、SequentialChain、TransformChain 等。RouterChain 可以根据输入数据的某些属性/特征值，选择调用哪个子链（Subchain）。SimpleSequentialChain 是最简单的序列链形式，其中的每个步骤具有单一的输入/输出，上一个步骤的输出是下一个步骤的输入。SequentialChain 是连续链的更一般的形式，允许多个输入/输出。TransformChain 可以引入自定义转换函数，对输入进行处理后再输出。以下是使用 SimpleSequentialChain 的代码示例：

```
from langchain.llms import OpenAI
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate

# 这是一个LLMChain，根据一部剧目的标题来撰写简介
llm = OpenAI(temperature=.7)
template = """You are a playwright. Given the title of play, it is your
job to write a synopsis for that title.

Title: {title}
Playwright: This is a synopsis for the above play:"""
prompt_template = PromptTemplate(input_variables=["title"], template=template)
synopsis_chain = LLMChain(llm=llm, prompt=prompt_template)

# 这是一个LLMChain，根据剧目简介来撰写评论
llm = OpenAI(temperature=.7)
template = """You are a play critic from the New York Times. Given the synopsis of play,
it is your job to write a review for that play.

Play Synopsis:
{synopsis}
Review from a New York Times play critic of the above play:"""
prompt_template = PromptTemplate(input_variables=["synopsis"], template=template)
review_chain = LLMChain(llm=llm, prompt=prompt_template)

# 这是总体链，按顺序运行这两个链
from langchain.chains import SimpleSequentialChain
overall_chain = SimpleSequentialChain(chains=[synopsis_chain, review_chain], verbose=True)
```

5. 记忆

大多数大语言模型应用都使用对话方式与用户交互。对话中的一个关键环节是能够引用和参考之前对话中的信息。对于对话系统来说，最基础的要求是能够直接访问一些过去的消息。在更复杂的系统中还需要一个能够不断更新的事件模型，其能够维护有关实体及其关系的信息。在 LangChain 中，这种能存储过去交互信息的能力被称为“记忆”。LangChain 中提供了许多用于向系统添加记忆的方法，可以单独使用，也可以无缝整合到链中使用。

LangChain 记忆模块的基本框架如图8.10 所示。记忆系统需要支持两个基本操作：读取和写入。每个链都根据输入定义了核心执行逻辑，其中一些输入直接来自用户，但有些输入可以来源于记忆。在接收到初始用户输入，但执行核心逻辑之前，链将从记忆系统中读取内容并增强用户输

入。在核心逻辑执行完毕并返回答复之前，链会将这一轮的输入和输出都保存到记忆系统中，以便在将来使用它们。

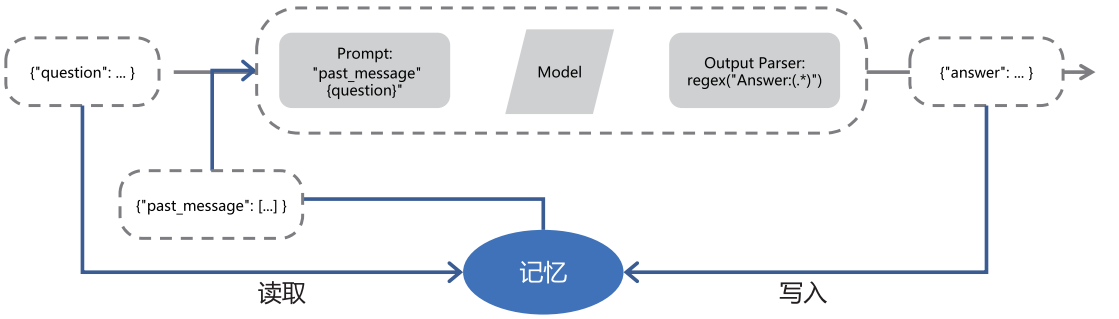

图 8.10 LangChain 记忆模块的基本框架

LangChain 中提供了多种对记忆方式的支持，ConversationBufferMemory 是记忆中一种非常简单的形式，它将聊天消息列表保存到缓冲区中，并将其传递到提示模板中，代码示例如下：

```
from langchain.memory import ConversationBufferMemory

memory = ConversationBufferMemory()
memory.chat_memory.add_user_message("hi!")
memory.chat_memory.add_ai_message("whats up?")
```

这种记忆系统非常简单，因为它只记住了先前的对话，并没有建立更高级的事件模型，也没有在多个对话之间共享信息，其可用于简单的对话系统，例如问答系统或聊天机器人。对于更复杂的对话系统，需要更高级的记忆系统来支持更复杂的对话和任务。将 ConversationBufferMemory 与 ChatModel 结合到链中的代码示例如下：

```
from langchain.chat_models import ChatOpenAI
from langchain.schema import SystemMessage
from langchain.prompts import ChatPromptTemplate, HumanMessagePromptTemplate, MessagesPlaceholder

prompt = ChatPromptTemplate.from_messages([
    SystemMessage(content="You are a chatbot having a conversation with a human."),
    MessagesPlaceholder(variable_name="chat_history"), # Where the memory will be stored.
    HumanMessagePromptTemplate.from_template("{human_input}"), # Where the human input will injectd
])

memory = ConversationBufferMemory(memory_key="chat_history", return_messages=True)

llm = ChatOpenAI()

chat_llm_chain = LLMChain(
    llm=llm,
    prompt=prompt,
    verbose=True,
    memory=memory,
)

chat_llm_chain.predict(human_input="Hi there my friend")
```

执行上述代码可以得到如下输出结果：

```
> Entering new LLMChain chain...
Prompt after formatting:
System: You are a chatbot having a conversation with a human.
Human: Hi there my friend

> Finished chain.

'Hello! How can I assist you today, my friend?'
```

在此基础上继续执行如下语句：

```
chat_llm_chain.predict(human_input="Not too bad - how are you?")
```

可以得到如下输出结果：