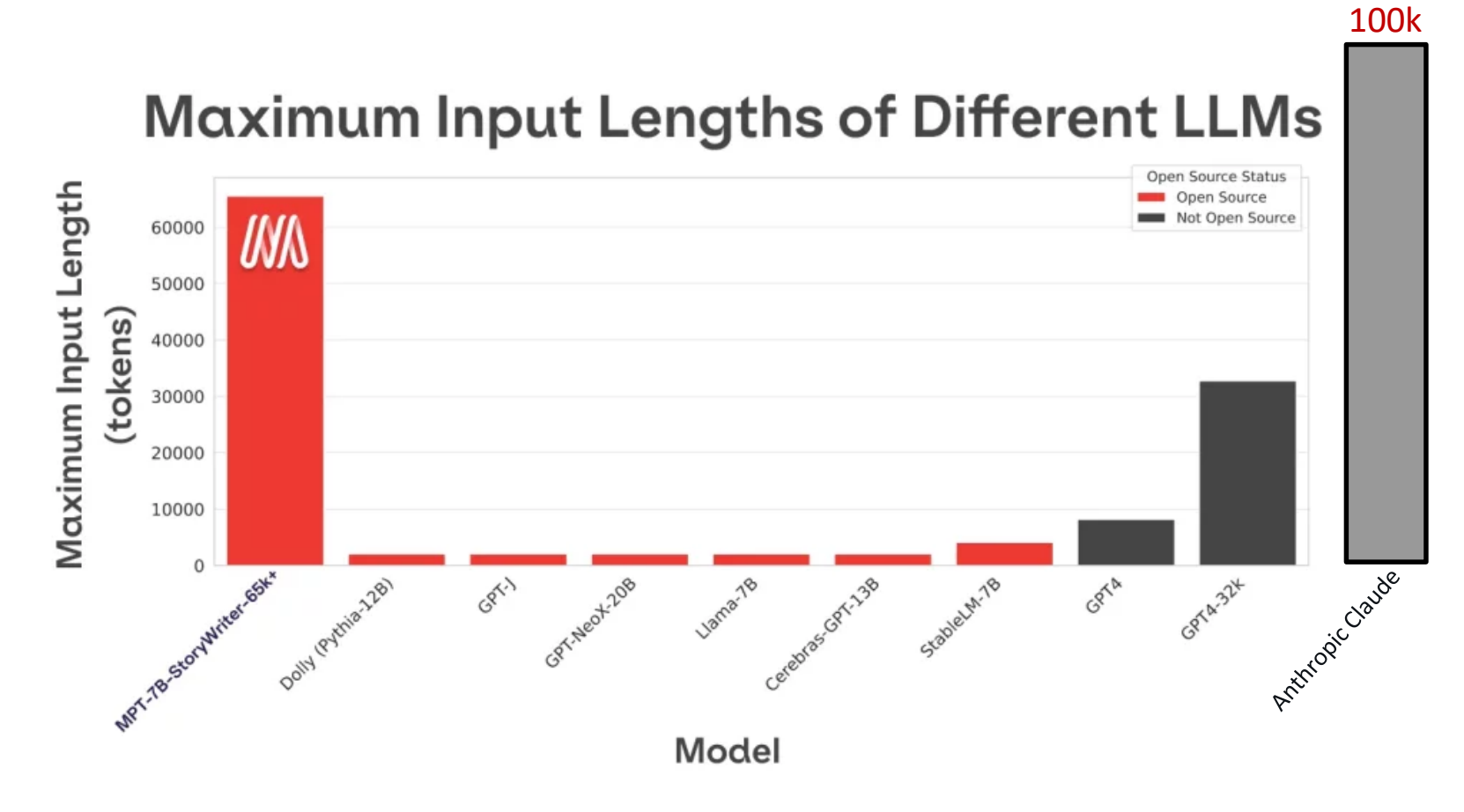
# 大模型使用2：Retrieval Augumented Generation

杨沐昀

语言技术研究中心
哈尔滨工业大学

# 为什么需要RAG?

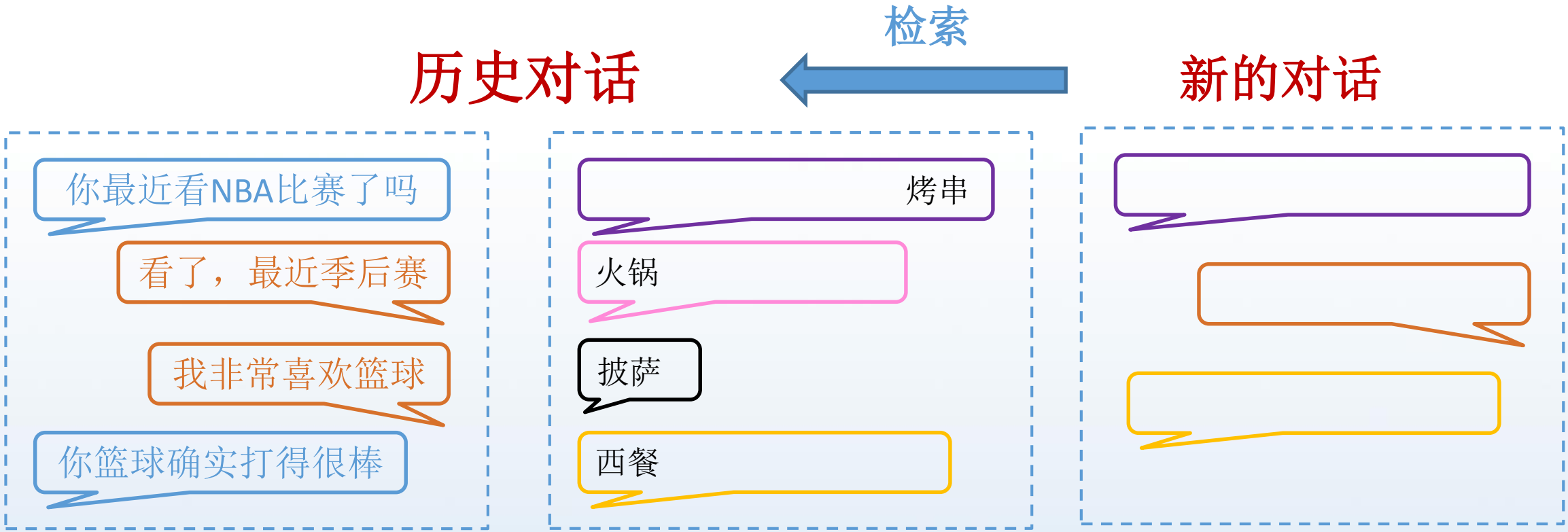
- 大语言模型可以处理的文本长度有限。




*https://www.aidemos.info/mpt-7b-new-llm-from-mosaicml/*

# 为什么需要RAG?

- 历史信息需要有效存储。




*https://www.aidemos.info/mpt-7b-new-llm-from-mosaicml/*

# RAG也有人称为综合应用框架

ChatGPT 所取得的巨大成功，使得越来越多的开发者希望利用OpenAI 提供的API或私有化模型开发基于大语言模型的应用程序。然而，即使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，帮助大家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。

# 1.1 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。

# 1.1 LangChain 框架核心模块

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）。

LangChain 中的模型**输入/输出（Model I/O）**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下图所示，主要包含以下部分：Prompts、Language Models及Output Parsers。用户原始输入与模型和示例进行组合，然后输入大语言模型，再根据大语言模型的返回结果进行输出或者结构化处理。

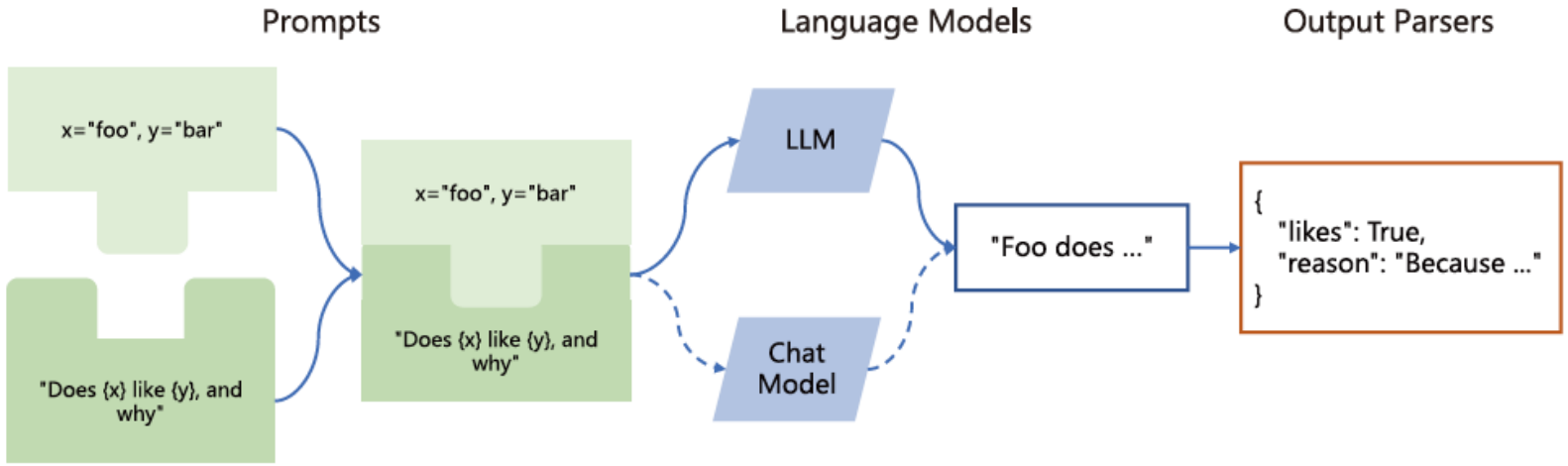



图 7.4 LangChain 模型输入/输出模块的基本流程

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，通常由模板、示例和用户输入组成。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，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。

比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。

比如 PydanticOutput-Parser 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，比如这里所使用的 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!')
```

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（Data connection）**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7.5 所示。

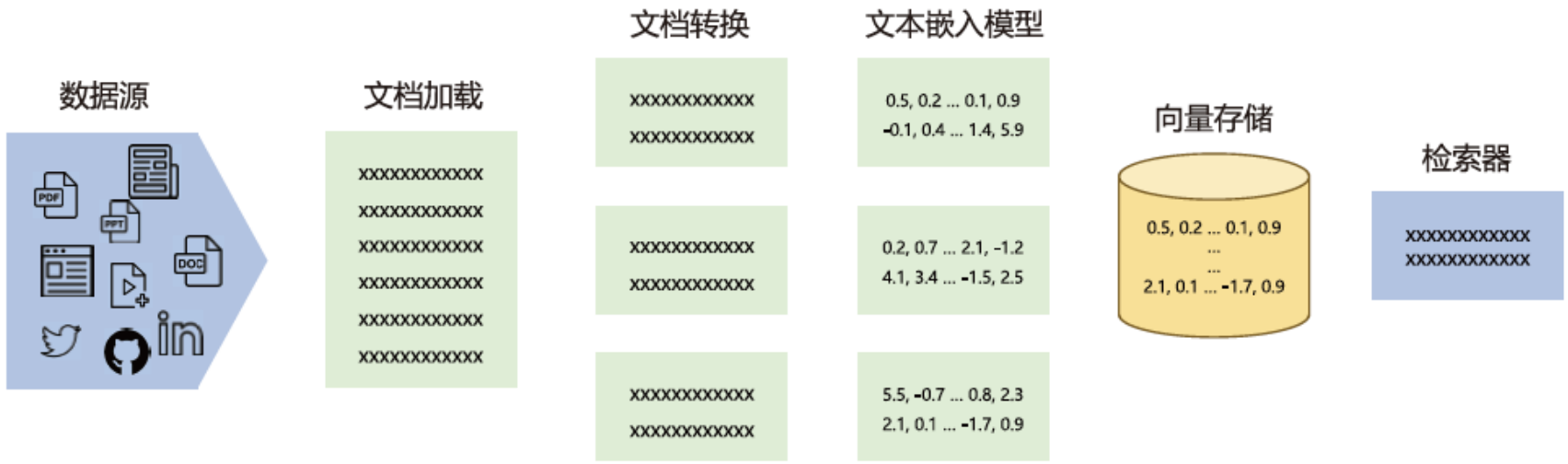



图 7.5　LangChain 数据连接模块的基本框架

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txt 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()
```

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])
```

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]
```

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
        """
        ...
```

```python
    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
        """
        ...
```

Retrievers（检索器）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Retriever 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")
```

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（Chain）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)
```

# LangChain 框架核心模块--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”（Memory）**。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7.6 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。

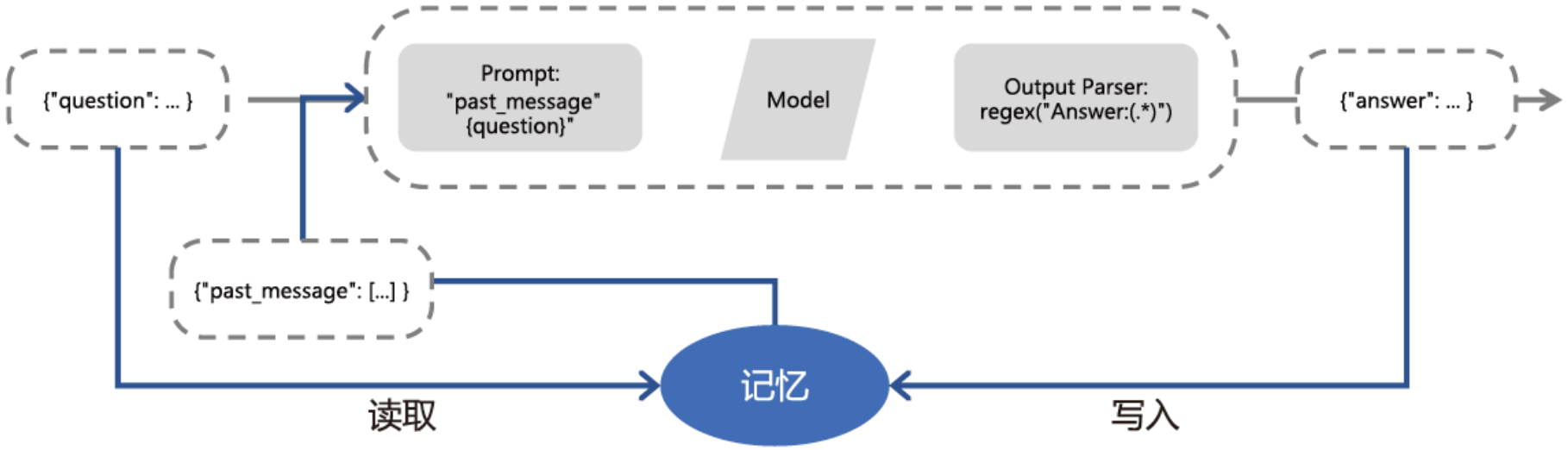



图 7.6 LangChain 记忆模块的基本框架

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 injecto
])

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：

```
 > Entering new LLMChain chain...
Prompt after formatting:
System: You are a chatbot having a conversation with a human.
Human: Hi there my friend
AI: Hello! How can I assist you today, my friend?
Human: Not too bad - how are you?

> Finished chain.

"I'm an AI chatbot, so I don't have feelings, but I'm here to help and chat with you! Is there
something specific you would like to talk about or any questions I can assist you with?"
```

**智能体（Agent）**的核心思想是使用大语言模型来选择要执行的一系列动作。

在链中，操作序列是硬编码在代码中的。

在智能体中，需要将大语言模型用作推理引擎，以确定要采取哪些动作，以及以何种顺序采取这些动作。

智能体通过将大语言模型与动作列表结合，自动选择最佳的动作序列，从而实现自动化决策和行动。智能体可以用于许多不同类型的应用程序，例如自动化客户服务、智能家居等。

# 1.1 LangChain 框架核心模块--智能体

LangChain 现实的智能体仅是简化方案。LangChain中的智能体由如下几个核心组件构成：

- **Agent：**决定下一步该采取什么操作的类，由大语言模型和提示词驱动。提示词可以包括智能体的个性（有助于使其以某种方式做出回应）、智能体的背景上下文（有助于提供所要求完成的任务类型的更多上下文信息）、激发更好的推理的提示策略。
- **Tools：**智能体调用的函数。这里有两个重要的考虑因素，一是为智能体提供正确的工具访问权限；二是用对智能体最有帮助的方式描述工具。
- **Toolkits：**一组旨在一起使用以完成特定任务的工具集合，具有方便的加载方法。通常一个工具集合中有3 ~ 5 个工具。
- **AgentExecutor：**智能体的运行空间，这是实际调用智能体并执行其所选操作的部分。除了AgentExecutor 类，LangChain 还支持其他智能体运行空间，包括Plan-and-execute Agent、Baby AGI、Auto GPT 等。

以下代码给出了利用搜索增强模型对话能力的智能体的实现：

```
from langchain.agents import Tool
from langchain.agents import AgentType
from langchain.memory import ConversationBufferMemory
from langchain.chat_models import ChatOpenAI
from langchain.utilities import SerpAPIWrapper
from langchain.agents import initialize_agent

search = SerpAPIWrapper()
tools = [
    Tool(
        name = "Current Search",
        func=search.run,
        description="useful for when you need to answer questions about current events
                     or the current state of the world"
    ),
]

memory = ConversationBufferMemory(memory_key="chat_history", return_messages=True)
llm = ChatOpenAI(openai_api_key=OPENAI_API_KEY, temperature=0)
agent_chain = initialize_agent(
    tools,
    llm,
    agent=AgentType.CHAT_CONVERSATIONAL_REACT_DESCRIPTION,
    verbose=True,
    memory=memory
)
```

注意，此处在agent 类型选择时使用了“CHAT_CONVERSATIONAL_REACT_DESCRIPTION”，模型将使用ReAct 逻辑生成。根据上面定义的智能体，使用如下调用方式：

```
agent_chain.run(input="what's my name?")
```

给出如下回复：

```
> Entering new AgentExecutor chain...
{
    "action": "Final Answer",
    "action_input": "Your name is Bob."
}
> Finished chain.


'Your name is Bob.'
```

但是，如果换一种需要利用当前知识的用户输入，并给出如下调用方式：

```
agent_chain.run(input="whats the weather like in pomfret?")
```

给出如下回复：

```
> Entering new AgentExecutor chain...
{
    "action": "Current Search",
    "action_input": "weather in pomfret"
}
Observation: Cloudy with showers. Low around 55F. Winds S at 5 to 10 mph.
             Chance of rain 60%. Humidity76%.
Thought:{
    "action": "Final Answer",
    "action_input": "Cloudy with showers. Low around 55F. Winds S at 5 to 10 mph.
                     Chance of rain 60%. Humidity76%."
}

> Finished chain.

'Cloudy with showers. Low around 55F. Winds S at 5 to 10 mph. Chance of rain 60%. Humidity76%.'
```

# 1.1 LangChain 框架核心模块--回调

LangChain 提供了**回调（Callback）**系统，允许连接到大语言模型应用程序的各个阶段。这对于日志记录、监控、流式处理和其他任务处理非常有用。可以通过使用 API 中提供的 callbacks 参数订阅这些事件。CallbackHandlers 是实现 CallbackHandler 接口的对象，每个事件都可以通过一个方法订阅。当事件被触发时，CallbackManager 会调用相应事件所对应的处理程序，代码示例如下：

```python
class BaseCallbackHandler:
    """ 基本回调处理程序，可用于处理来自LangChain的回调 """

    def on_llm_start(
        self, serialized: Dict[str, Any], prompts: List[str], **kwargs: Any
    ) -> Any:
        """ 在LLM开始运行时运行 """

    def on_chat_model_start(
        self, serialized: Dict[str, Any], messages: List[List[BaseMessage]], **kwargs: Any
    ) -> Any:
        """ 在聊天模型开始运行时运行 """

    def on_llm_new_token(self, token: str, **kwargs: Any) -> Any:
        """ 在新的LLM词元上运行，仅在启用了流式处理时可用 """

    def on_llm_end(self, response: LLMResult, **kwargs: Any) -> Any:
        """ 在LLM结束运行时运行 """
```

```python
    def on_llm_error(
        self, error: Union[Exception, KeyboardInterrupt], **kwargs: Any
    ) -> Any:
        """ 在LLM出现错误时运行 """

    def on_chain_start(
        self, serialized: Dict[str, Any], inputs: Dict[str, Any], **kwargs: Any
    ) -> Any:
        """ 在链开始运行时运行 """

    def on_chain_end(self, outputs: Dict[str, Any], **kwargs: Any) -> Any:
        """ 在链结束运行时运行 """

    def on_chain_error(
        self, error: Union[Exception, KeyboardInterrupt], **kwargs: Any
    ) -> Any:
        """ 在链出现错误时运行 """

    def on_tool_start(
        self, serialized: Dict[str, Any], input_str: str, **kwargs: Any
    ) -> Any:
        """ 在工具开始运行时运行 """

    def on_tool_end(self, output: str, **kwargs: Any) -> Any:
```

# 1.1 LangChain 框架核心模块--回调

```
    """ 在工具结束运行时运行 """

def on_tool_error(
    self, error: Union[Exception, KeyboardInterrupt], **kwargs: Any
) -> Any:
    """ 在工具出现错误时运行 """

def on_text(self, text: str, **kwargs: Any) -> Any:
    """ 在任意文本上运行 """

def on_agent_action(self, action: AgentAction, **kwargs: Any) -> Any:
    """ 在代理动作上运行 """

def on_agent_finish(self, finish: AgentFinish, **kwargs: Any) -> Any:
    """ 在代理结束时运行 """
```

LangChain 在langchain/callbacks 模块中提供了一些内置的处理程序，其中最基本的处理程序是StdOutCallbackHandler，它将所有事件记录到stdout 中，代码示例如下：

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

handler = StdOutCallbackHandler()
llm = OpenAI()
prompt = PromptTemplate.from_template("1 + {number} = ")

# 构造函数回调
# 首先，在初始化链时显式设置StdOutCallbackHandler
chain = LLMChain(llm=llm, prompt=prompt, callbacks=[handler])
chain.run(number=2)

# 使用详细模式标志：然后，使用verbose标志来实现相同的结果
chain = LLMChain(llm=llm, prompt=prompt, verbose=True)
chain.run(number=2)

# 请求回调：最后，使用请求的callbacks来实现相同的结果
chain = LLMChain(llm=llm, prompt=prompt)
chain.run(number=2, callbacks=[handler])
```

大语言模型虽然可以很好地回答很多领域的各种问题，但是由于其知识是通过语言模型训练及指令微调等方式注入模型参数中的，因此针对本地知识库中的内容，大语言模型很难通过此前的方式有效地进行学习。通过LangChain 框架，则可以有效地融合本地知识库内容与大语言模型的知识问答能力。

基于LangChain 的知识库问答系统框架如图所示，主要包含以下几个步骤：

（1）收集领域知识数据构造知识库，这些数据应当能够尽可能地全面覆盖问答需求。

（2）对知识库中的非结构数据进行文本提取和文本分割，得到文本块。

（3）利用嵌入向量表示模型给出文本块的嵌入表示，并利用向量数据库进行保存。

（4）根据用户输入信息的嵌入表示，通过向量数据库检索得到最相关的文本片段，将提示词模板与用户提交问题及历史消息合并输入大语言模型。

（5）将大语言模型结果返回给用户。

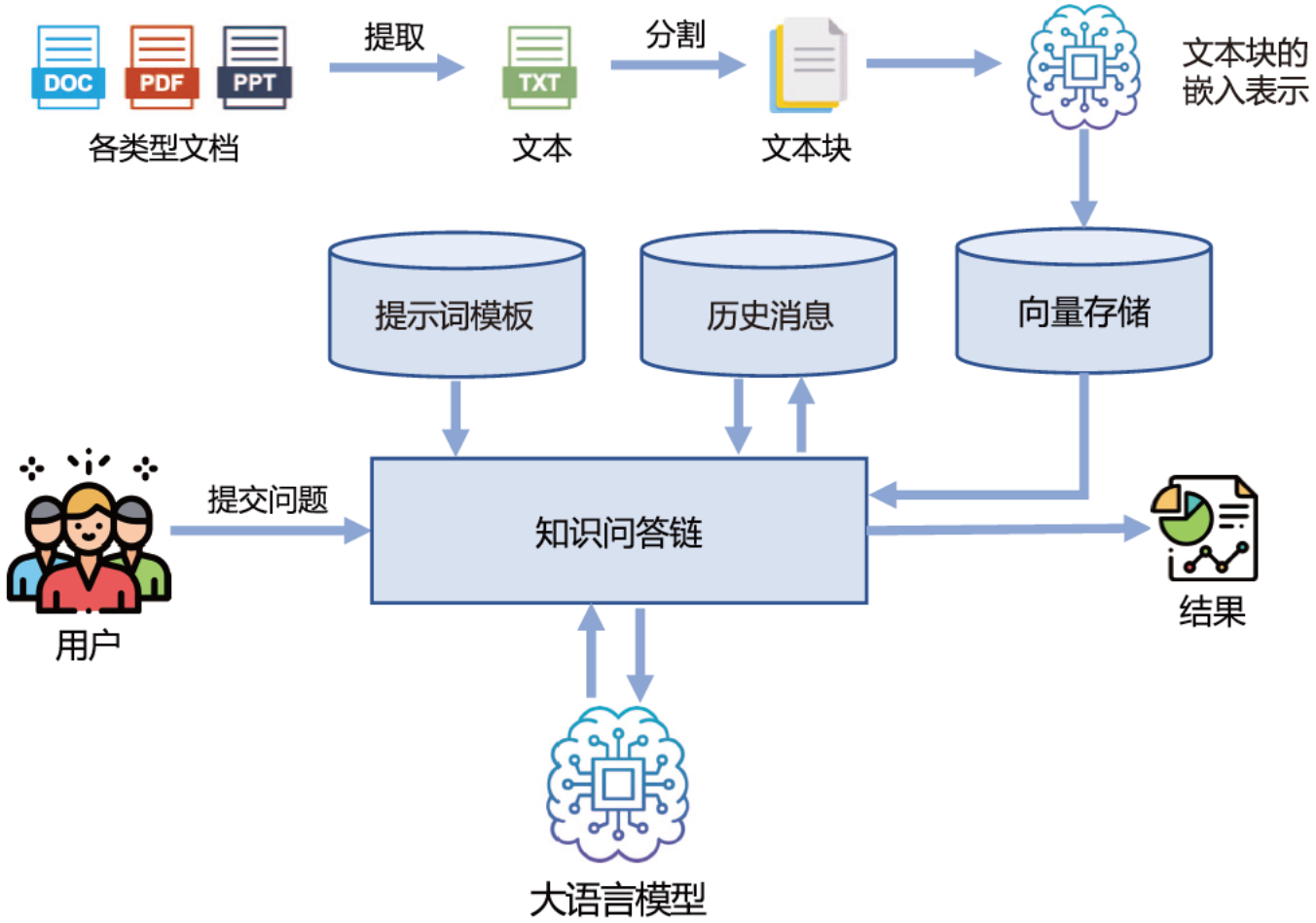



图 7.7　LangChain 知识库问答系统框架

```
from langchain.document_loaders import DirectoryLoader
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.text_splitter import CharacterTextSplitter
from langchain.vectorstores import Chroma
from langchain.chains import ChatVectorDBChain, ConversationalRetrievalChain
from langchain.chat_models import ChatOpenAI
from langchain.chains import RetrievalQA

# 从本地读取相关数据
loader = DirectoryLoader(
    './Langchain/KnowledgeBase/',  glob='**/*.pdf',  show_progress=True
)
docs = loader.load()

# 将文本进行分割
text_splitter = CharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=0
)
docs_split = text_splitter.split_documents(docs)

# 初始化OpenAI Embeddings
embeddings = OpenAIEmbeddings()

# 将数据存入Chroma向量存储
vector_store = Chroma.from_documents(docs, embeddings)
# 初始化检索器，使用向量存储
retriever = vector_store.as_retriever()


system_template = """
Use the following pieces of context to answer the users question.
If you don't know the answer, just say that you don't know, don't try to make up an answer.
Answering these questions in Chinese.
-----------
{question}
-----------
{chat_history}
"""
```

```
# 构建初始消息列表
messages = [
  SystemMessagePromptTemplate.from_template(system_template),
  HumanMessagePromptTemplate.from_template('{question}')
]

# 初始化Prompt对象
prompt = ChatPromptTemplate.from_messages(messages)

# 初始化大语言模型，使用OpenAI API
llm=ChatOpenAI(temperature=0.1,  max_tokens=2048)

# 初始化问答链
qa = ConversationalRetrievalChain.from_llm(llm,retriever,condense_question_prompt=prompt)

chat_history = []
while True:
  question = input(' 问题: ')
  # 开始发送问题chat_history为必须参数，用于存储历史消息
  result = qa({'question': question, 'chat_history': chat_history})
  chat_history.append((question, result['answer']))
  print(result['answer'])
```