知识库 - langchain

作者：Calvin
QQ: 179209347
Mail: 179209347@qq.com

介绍

笔记简介：

- 面向对象：深度学习初学者
- 依赖课程：**线性代数，统计概率**，优化理论，图论，离散数学，微积分，信息论

知乎专栏：

https://zhuanlan.zhihu.com/p/693738275

Github & Gitee 地址：

https://github.com/mymagicpower/AIAS/tree/main/deep_learning

https://gitee.com/mymagicpower/AIAS/tree/main/deep_learning

*** 版权声明：**

- 仅限用于个人学习
- 禁止用于任何商业用途

知识库产品选型

名称	说明	地址
🌟🌟🌟🌟🌟 LangChain	• 提供了标准的内存接口和内存实现，支持自定义大模型的封装。	• https://github.com/langchain-ai/langchain
Dust.tt	• 提供了简单易用的API，可以让开发者快速构建自己的LLM应用程序。	• https://dust.tt/
Semantic-kernel	• 轻量级SDK，可将AI大型语言模型（LLMs）与传统编程语言集成在一起。	• https://github.com/microsoft/semantic-kernel
fixie.ai	• 开放、免费、简单，多模态（images, audio, video...）	• https://github.com/fixie-ai • https://www.fixie.ai/
Brancher AI	• 在线平台，链接所有大模型，无代码快速生成应用, Langchain产品）	• https://www.brancher.ai/
🌟🌟🌟 Wenda	一个LLM调用平台。目标为针对特定环境的高效内容生成，同时考虑个人和中小企业的计算资源局限性，以及知识安全和私密性问题。	• https://github.com/l15y/wenda
MaxKB	• MaxKB 是一款基于 LLM 大语言模型的知识库问答系统。 MaxKB = Max Knowledge Base，旨在成为企业的最强大脑。	• https://github.com/1Panel-dev/MaxKB

Langchain介绍

- 基于大模型的知识库通用框架，适配多种主流大模型，实现基于大模型的知识库问答。
- https://github.com/langchain-ai/langchain

- LangChain是一个开源框架，为开发人员提供使用大型语言模型（LLM）创建应用程序所需的工具。从本质上讲，是一个提示编排工具，使团队更容易地进行各种提示的互动连接。
- LangChain最初是一个开源项目，但随着GitHub的星星积累增加，它很快成为由Harrison Chase领导的公司。
- LLM（如ChatGLM，Baichuan）为单个提示提供了一个完成的结果。
- 为了避免用户明确提供每个步骤并选择执行顺序，可以使用LLM（语言模型）在每个点产生下一步，并利用前一步的结果作为其上下文。LangChain框架可以实现这一点。它通过一系列提示来达到所需的结果。它为开发人员提供了一个简单的接口与LLM进行交互。可以说，LangChain就像是一个简化的包装器，用于利用LLM。

Langchain – 工作原理

要开始使用LangChain：

1. 首先需要创建一个语言模型。可以利用公开可用的语言模型，或者训练自己的模型。
2. 完成后，您可以开始使用LangChain开发应用程序。LangChain提供了许多工具和API，使得将语言模型与外部数据源连接、与其环境交互和开发复杂应用程序变得简单。
3. 它通过将一系列被称为链的组件链接在一起来创建工作流程。链中的每个链接都执行特定的操作，例如：
   - 格式化用户输入
   - 使用数据源
   - 引用语言模型
   - 处理语言模型的输出

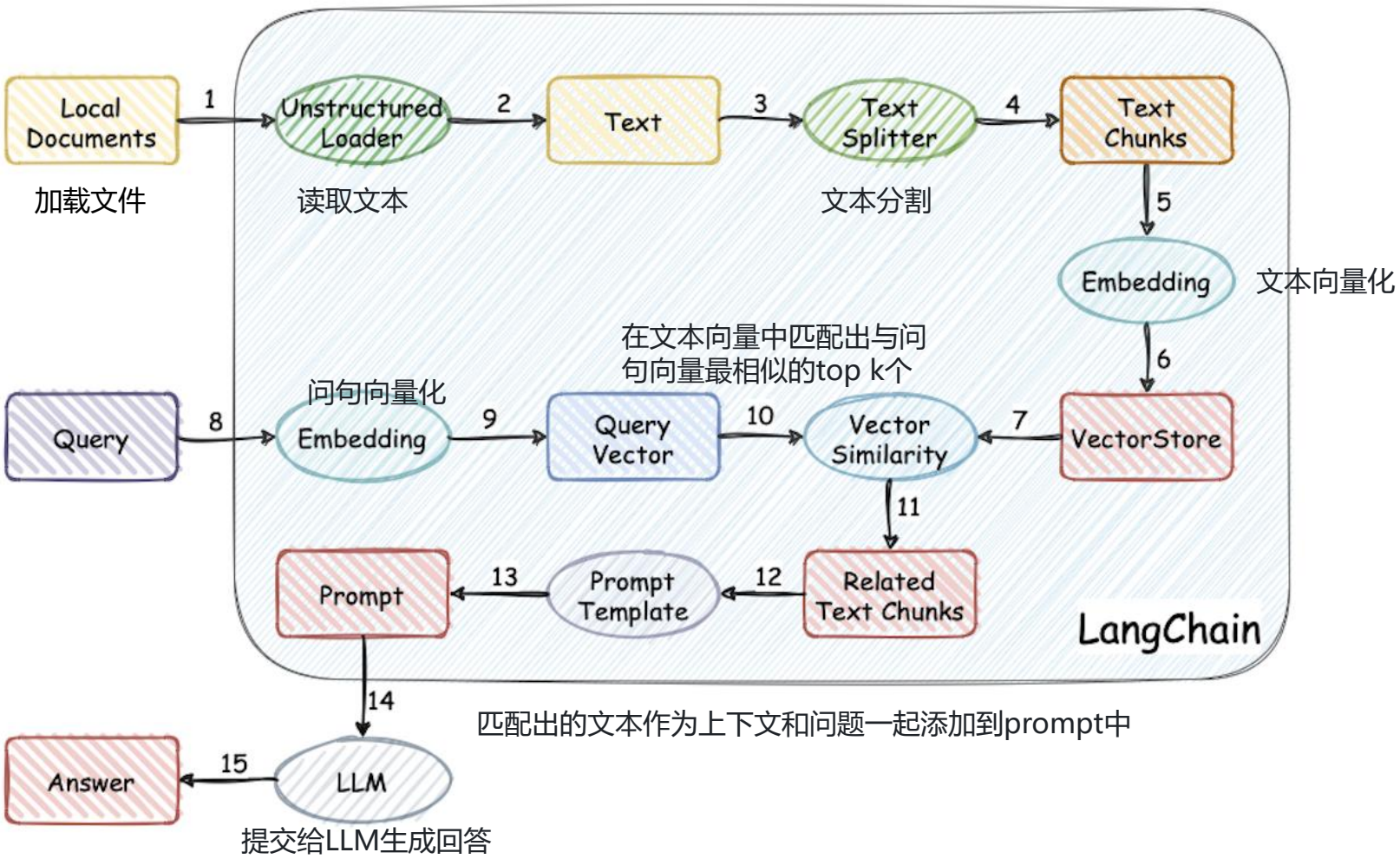
链中的链接按顺序连接在一起，一个链接的输出作为下一个链接的输入。通过将小操作链接在一起，链能够执行更复杂的任务。

Langchain – 优点

优点	说明
可扩展性	• LangChain可用于创建能够处理大量数据的应用程序。
适应性	• 该框架的适应性使其可以用于开发各种应用，从聊天机器人到问答系统。
可扩展性	• 开发人员可以向框架添加自己的功能和功能，因为它是可扩展的。
易于使用	• LangChain提供了一个高级API，用于将语言模型连接到各种数据源和构建复杂的应用程序。
开源	• LangChain是一个开源框架，可以免费使用和修改。
-完善的文档	• 文档详尽且易于理解。
集成性	• LangChain可以与各种框架和库集成。
活跃的社区	• 有一个庞大而活跃的LangChain用户和开发者社区，可以为您提供帮助和支持。

Langchain – 流程

- 基于大模型的知识库通用框架，适配多种主流大模型，实现基于大模型的知识库问答。
- https://github.com/langchain-ai/langchain



- 本地知识库可以提升大模型的准确性
- 本地知识库可以约束回复内容的范围

原理是将长文本分解成一些短的片段，本地查询后，将片段带入提示词(promot)，然后让大模型进行上下文学习，拿到新知识。再组织提问什么的。

流程:

- 加载文件
- 读取文本
- 文本分割
- 文本向量化
- 问句向量化
- 在文本向量中匹配出与问句向量最相似的top k个
- 匹配出的文本作为上下文和问题一起添加到prompt中
- 提交给LLM生成回答

Langchain – 用例

用例	说明
自治代理（autonomous agents）	• 长时间运行的代理会采取多步操作以尝试完成目标。 AutoGPT 和 BabyAGI就是典型代表。
代理模拟（agent simulations）	• 将代理置于封闭环境中观察它们如何相互作用，如何对事件作出反应，是观察它们长期记忆能力的有趣方法。
个人助理（personal assistants）	• 主要的 LangChain 使用用例。个人助理需要采取行动、记住交互并具有您的有关数据的知识。
问答（question answering）	• 第二个重大的 LangChain 使用用例。仅利用这些文档中的信息来构建答案，回答特定文档中的问题。
聊天机器人（chatbots）	• 由于语言模型擅长生成文本，因此它们非常适合创建聊天机器人。
查询表格数据（tabular）	• 如果您想了解如何使用 LLM 查询存储在表格格式中的数据（csv、SQL、数据框等），请阅读此页面。
代码理解（code）	• 如果您想了解如何使用 LLM 查询来自 GitHub 的源代码，请阅读此页面。
与 API 交互（apis）	• 使LLM 能够与 API 交互非常强大，以便为它们提供更实时的信息并允许它们采取行动。
提取（extraction）	• 从文本中提取结构化信息。
摘要（summarization）	• 将较长的文档汇总为更短、更简洁的信息块。一种数据增强生成的类型。
评估（evaluation）	• 生成模型是极难用传统度量方法评估的。 一种新的评估方式是使用语言模型本身进行评估。LangChain 提供一些用于辅助评估的提示/链。

拖拽式图形界面构建工具 - 构建定制的LLM流程

向量引擎	简介	后端框架	前端框架	许可证	链接
Flowise	• 一个强大的自然语言处理平台，可以帮助开发者构建智能对话系统。它提供了丰富的语义理解和对话管理功能，可以处理复杂的对话场景，并支持多轮对话和上下文理解。Flowise 还提供了一些预训练的模型和工具，可以加速对话系统的开发过程。	X	React	Aache2.0	https://github.com/FlowiseAI/Flowise
LangFlow	• 一个对话流程设计工具，它专注于对话流程的可视化设计和管理。LangFlow 提供了直观的界面和工具，可以帮助开发者设计和管理对话流程，包括对话节点、条件逻辑和回复等。LangFlow 还支持多语言和多渠道的对话流程设计。	python	React	MIT	https://github.com/logspace-ai/langflow
VectorVein	• 向量脉络是受到 LangChain 以及 langflow 的启发而开发的无代码 AI 工作流软件，旨在结合大语言模型的强大能力并让用户通过简单的拖拽即可实现各类日常工作流的智能化和自动化。	python	Vue	非商用	https://github.com/AndersonBY/vector-vein

- Flowise适合构建复杂的对话系统，具备强大的语义理解和对话管理功能。
- LangFlow适合设计和管理对话流程，注重可视化和易用性。
- VectorVein支持Vue, 但是商用可能需要联系作者付费使用。

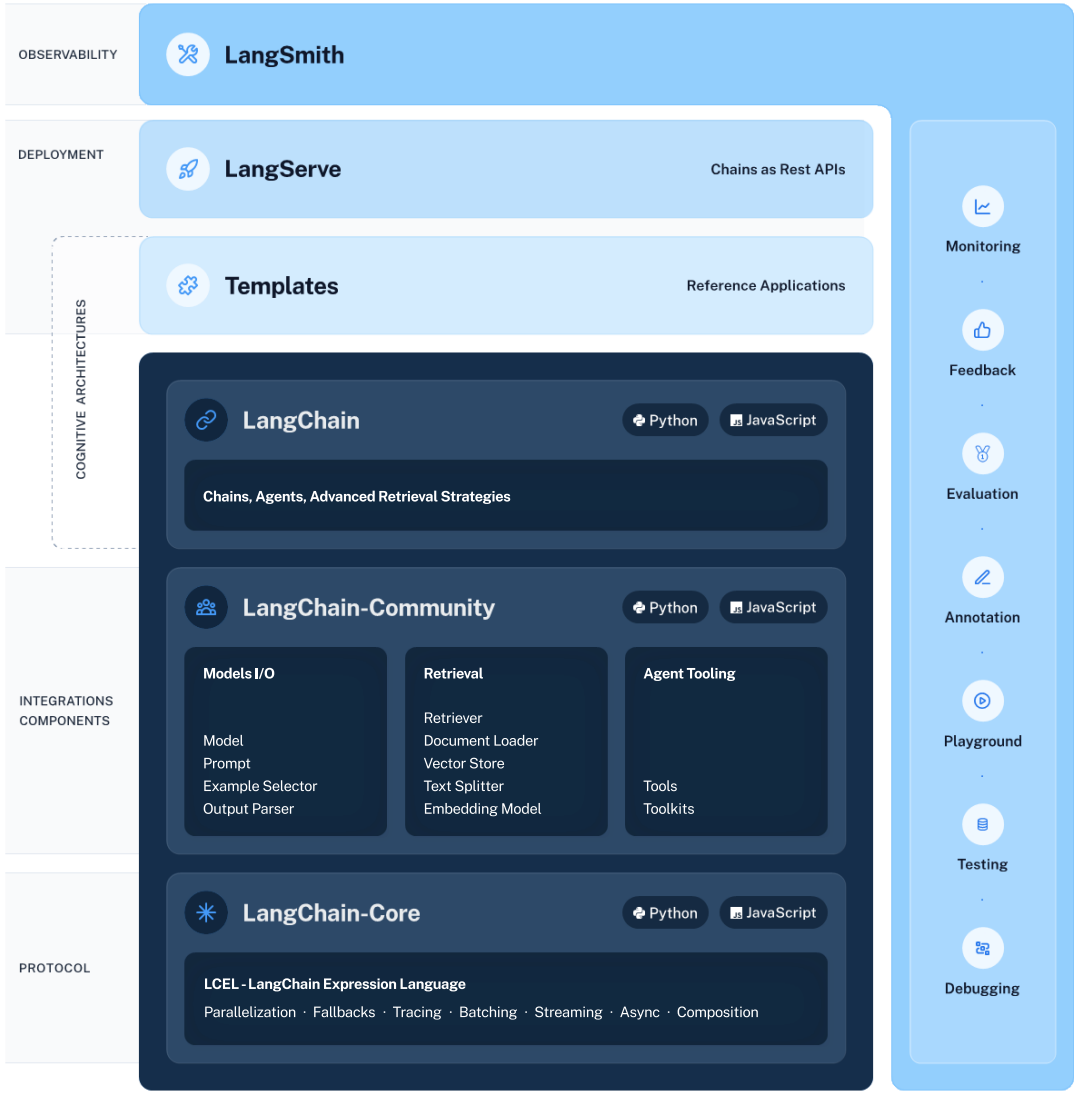
Langchain架构

LangChain是一个基于语言模型开发应用程序的框架。它能够实现以下功能：

1. 上下文感知：将语言模型与上下文信息（提示指令、少量示例、内容等）连接起来，使其能够根据提供的上下文进行回答。
2. 推理：依赖语言模型进行推理，根据提供的上下文决定如何回答以及采取什么行动等。

该框架由几个部分组成：

- LangChain库：Python和JavaScript库。包含接口和集成多种组件的功能，提供基本的运行时环境，用于将这些组件组合成链和代理，并提供现成的链和代理实现。
- LangChain模板：一系列易于部署的参考架构，适用于各种任务。
- LangServe：用于将LangChain链作为REST API部署的库。
- LangSmith：开发者平台，可让您对任何基于LLM框架构建的链进行调试、测试、评估和监控，并与LangChain无缝集成。



Langchain – 核心组件

模型输入/输出（Model I/O）

- 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。

数据连接（Retrieval）

- 与特定应用程序的数据进行交互的接口，包含：
  - ✓ 文档加载器（Document loaders）
  - ✓ 文本分割（Text Splitting）
  - ✓ 文本嵌入模型（Text embedding models）
  - ✓ 向量存储（Vector stores）
  - ✓ 检索器（Retrievers）
  - ✓ 索引（Indexing）

智能体（Agents）

- 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。

链（Chains）

- 提示词 + 大语言模型 + 输出解析
- 用于复杂应用的调用序列；

记忆（memory）

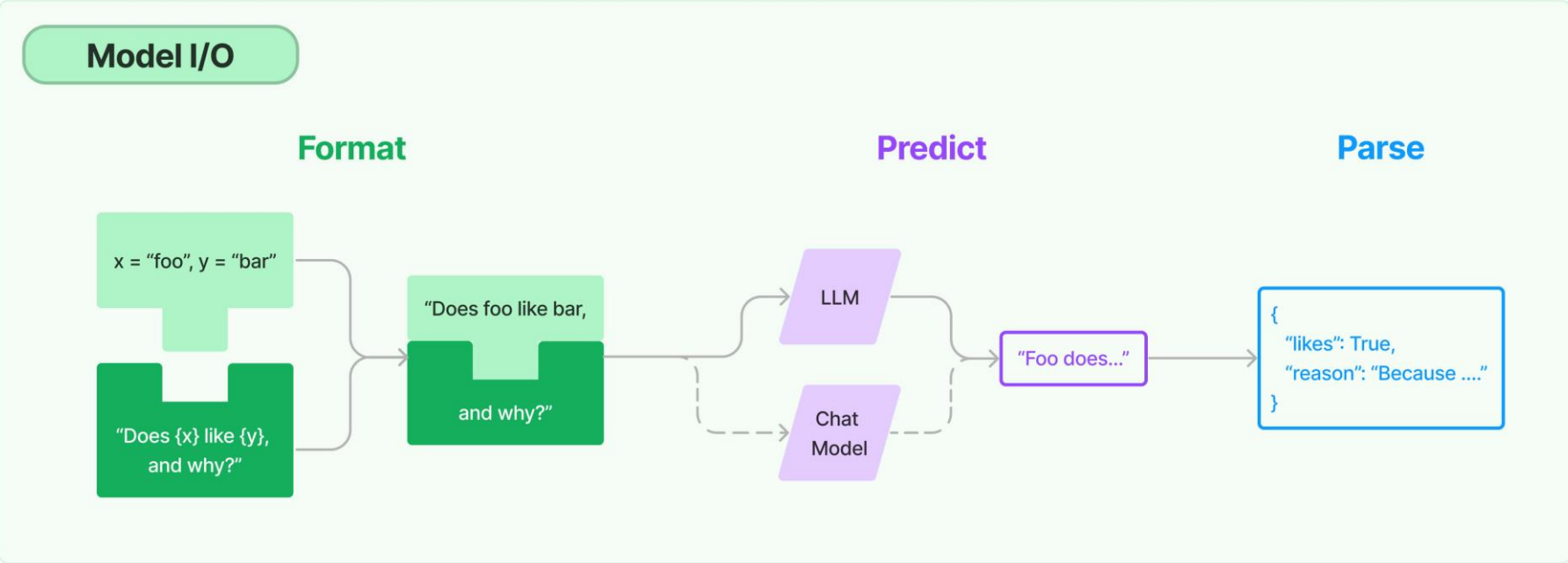
- 记忆是在链/代理调用之间保持状态的概念。
- LangChain 提供了一个标准的记忆接口、一组记忆实现及使用记忆的链/代理示例。

回调（Callback）

- 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。

模型输入/输出（Model I/O）

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。



模型输入/输出（Model I/O）- Prompts

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.prompts import PromptTemplate

prompt_template = PromptTemplate.from_template(
    "Tell me a {adjective} joke about {content}."
)
prompt_template.format(adjective="funny", content="chickens")
```

```
'Tell me a funny joke about chickens.'
```

模型输入/输出（Model I/O）- Language Models

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。

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。

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。

```python
class CustomLLM(LLM):
    n: int

    @property
    def _llm_type(self) -> str:
        return "custom"

    def _call(
        self,
        prompt: str,
        stop: Optional[List[str]] = None,
        run_manager: Optional[CallbackManagerForLLMRun] = None,
        **kwargs: Any,
    ) -> str:
        if stop is not None:
            raise ValueError("stop kwargs are not permitted.")
        return prompt[: self.n]

    @property
    def _identifying_params(self) -> Mapping[str, Any]:
        """Get the identifying parameters."""
        return {"n": self.n}
```

模型输入/输出（Model I/O）- Output Parsers

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。

- 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。
- 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。

```python
from langchain.output_parsers import PydanticOutputParser
from langchain.prompts import PromptTemplate
from langchain_core.pydantic_v1 import BaseModel, Field, validator
from langchain_openai import OpenAI

model = OpenAI(model_name="gpt-3.5-turbo-instruct", temperature=0.0)


# Define your desired data structure.
class Joke(BaseModel):
    setup: str = Field(description="question to set up a joke")
    punchline: str = Field(description="answer to resolve the joke")

    # You can add custom validation logic easily with Pydantic.
    @validator("setup")
    def question_ends_with_question_mark(cls, field):
        if field[-1] != "?":
            raise ValueError("Badly formed question!")
        return field


# Set up a parser + inject instructions into the prompt template.
parser = PydanticOutputParser(pydantic_object=Joke)

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

# And a query intended to prompt a language model to populate the data structure.
prompt_and_model = prompt | model
output = prompt_and_model.invoke({"query": "Tell me a joke."})
parser.invoke(output)
```

模型输入/输出（Model I/O）- Output Types

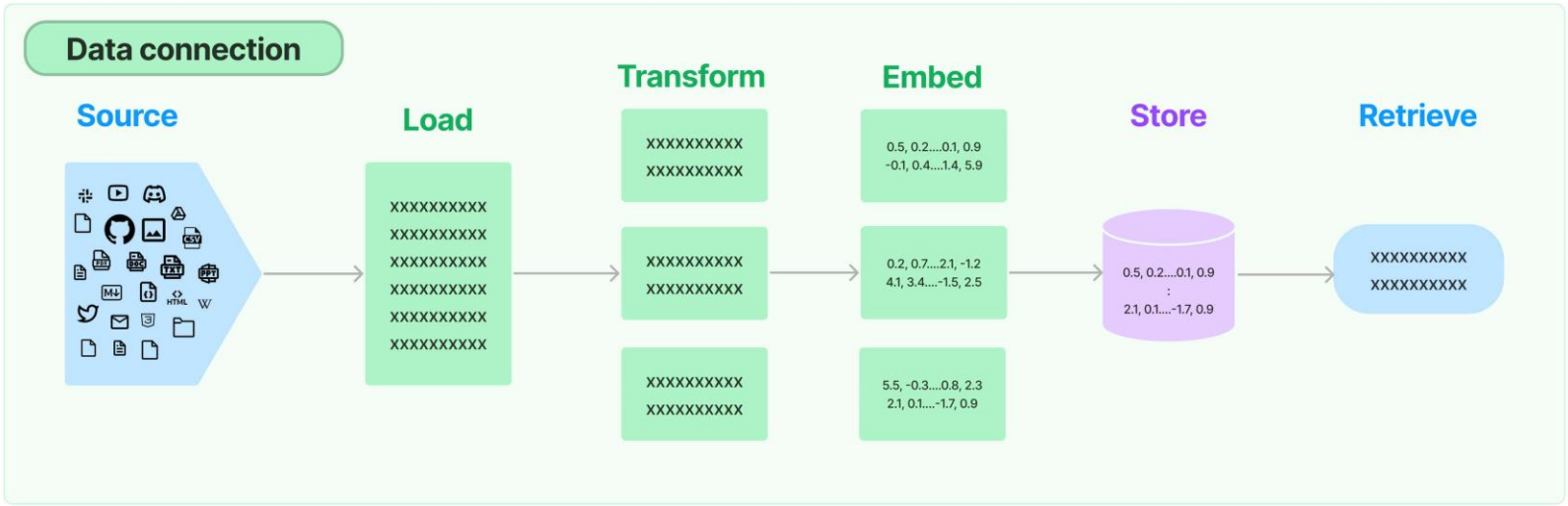
Name	Supports Streaming	Has Format Instructions	Calls LLM	Input Type	Output Type
OpenAITools		(Passes tools to model)		Message (with tool_choice)	JSON object
OpenAIFunctions	✅	(Passes functions to model)		Message (with function_call)	JSON object
JSON	✅	✅		str | Message	JSON object
XML	✅	✅		str | Message	dict
CSV	✅	✅		str | Message	List[str]
OutputFixing			✅	str | Message	
RetryWithError			✅	str | Message	
Pydantic		✅		str | Message	pydantic.BaseModel
YAML		✅		str | Message	pydantic.BaseModel
PandasDataFrame		✅		str | Message	dict
Enum		✅		str | Message	Enum
Datetime		✅		str | Message	datetime.datetime
Structured		✅		str | Message	Dict[str, str]

数据连接（Retrieval）

许多LLM应用需要用户特定的数据，而这些数据并不是模型训练集的一部分。实现这一目标的主要方法是通过检索增强生成（Retrieval Augmented Generation，RAG）。在这个过程中，外部数据被检索出来，然后在生成步骤中传递给LLM。LangChain为RAG应用提供了从简单到复杂的所有构建模块。

模块通过以下方式提供组件来加载、转换、存储和查询数据：

- 文档加载器（Document loaders）
- 文本分割（Text Splitting）
- 文本嵌入模型（Text embedding models）
- 向量存储（Vector stores）
- 检索器（Retrievers）
- 索引（Indexing）



数据连接（Retrieval）- 文档加载器（Document loaders）

文档加载器从许多不同的来源加载文档。LangChain提供了100多个不同的文档加载器，以及与其他主要提供商（如AirByte和Unstructured）的集成。LangChain提供了加载各种类型文档（HTML、PDF、代码）以及从各种位置（私有S3存储桶、公共网站）加载文档的集成功能。以下是一个最简单的从文件中读取文本来加载数据的 json 的示例：

```
from langchain_community.document_loaders import JSONLoader
```

```
import json
from pathlib import Path
from pprint import pprint


file_path='./example_data/facebook_chat.json'
data = json.loads(Path(file_path).read_text())
```

```
pprint(data)
```

```
{'image': {'creation_timestamp': 1675549016, 'uri': 'image_of_the_chat.jpg'},
 'is_still_participant': True,
 'joinable_mode': {'link': '', 'mode': 1},
 'magic_words': [],
 'messages': [{'content': 'Bye!',
               'sender_name': 'User 2',
               'timestamp_ms': 1675597571851},
              {'content': 'Oh no worries! Bye',
               'sender_name': 'User 1',
               'timestamp_ms': 1675597435669},
              {'content': 'No Im sorry it was my mistake, the blue one is not '
                          'for sale',
               'sender_name': 'User 2',
               'timestamp_ms': 1675596277579},
              {'content': 'I thought you were selling the blue one!',
               'sender_name': 'User 1',
               'timestamp_ms': 1675595140251},
              {'content': 'Im not interested in this bag. Im interested in the '
                          'blue one!',
               'sender_name': 'User 1',
```

数据连接（Retrieval）- 文本分割（Text Splitting）

在检索过程中，只获取文档中相关部分是一个关键步骤。这涉及多个转换步骤来为检索准备文档。其中一个主要步骤是将大型文档分割成较小的块。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）进行优化的逻辑。

Name	Splits On	Adds Metadata	Description
Recursive	A list of user defined characters		递归地分割文本。递归地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 specific characters	✅	基于HTML特定字符进行文本分割。特别是，这会添加关于该块来自何处的相关信息（基于HTML）。
Markdown	Markdown specific characters	✅	基于Markdown特定字符进行文本分割。特别是，这会添加关于该块来自何处的相关信息（基于Markdown）。
Code	Code (Python, JS) specific characters		基于编程语言特定字符进行文本分割。提供15种不同的编程语言可供选择。
Token	Tokens		基于标记进行文本分割。有几种不同的标记计量方法。
Character	A user defined character		基于用户定义的字符进行文本分割。这是一种比较简单的方法。
[Experimental] Semantic Chunker	Sentences		首先按句子进行分割，然后如果它们在语义上足够相似，就将相邻的句子合并在一起。取自Greg Kamradt。

数据连接（Retrieval）- 文本嵌入模型（Text embedding models）

检索的另一个关键部分是为文档创建嵌入。嵌入捕捉文本的语义含义，使您能够快速高效地找到其他与文本相似的部分。LangChain提供与25多个不同的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方法。LangChain提供了一个标准接口，方便您轻松切换模型。
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。

```
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])
```

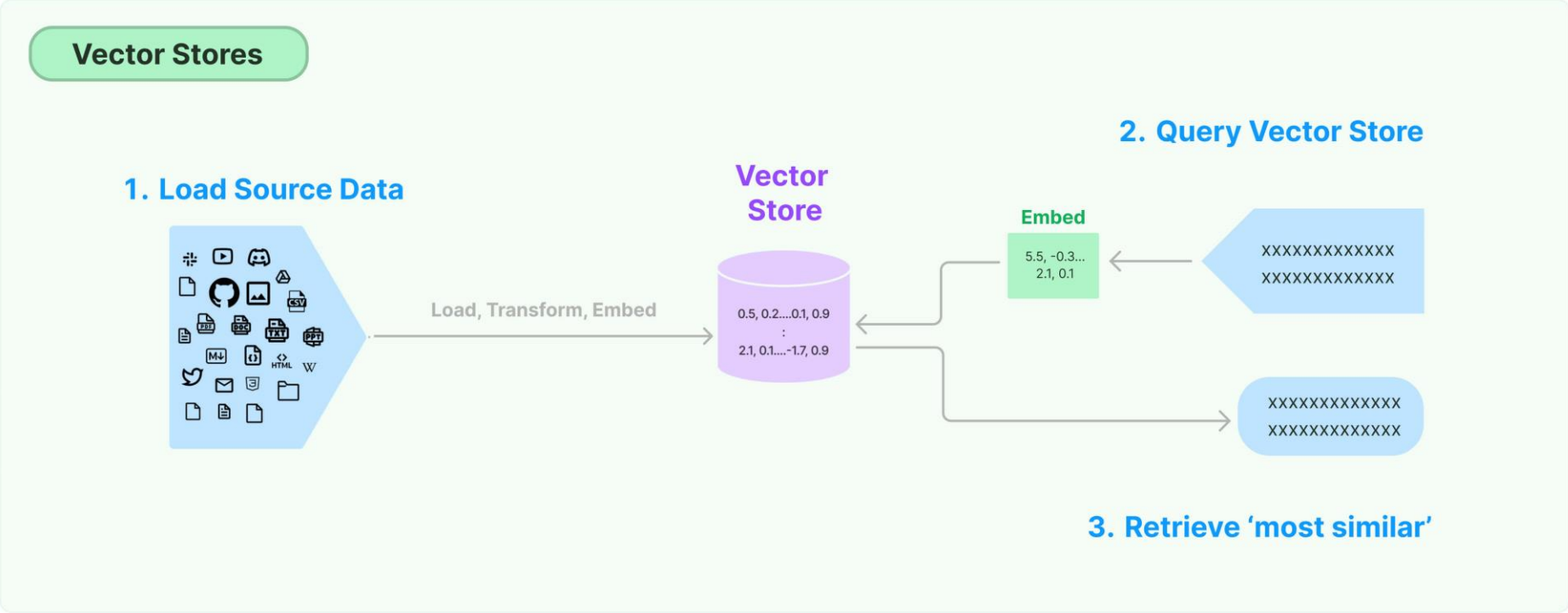
```
(5, 1536)
```

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

```
[0.0053587136790156364,
 -0.0004999046213924885,
 0.038883671164512634,
 -0.003001077566295862,
 -0.00900818221271038]
```

数据连接（Retrieval）- 向量存储（Vector stores）

随着嵌入的兴起，出现了对数据库支持这些嵌入的高效存储和搜索的需求。LangChain提供与50多个不同的向量存储的集成，从开源本地存储到云托管的专有存储，让您可以选择最适合您需求的方法。LangChain提供了一个标准接口，方便您轻松切换向量存储。
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。



数据连接（Retrieval）- 检索器（Retrievers）

一旦数据存储在数据库中，仍然需要检索它。LangChain支持许多不同的检索算法。LangChain支持易于入门的基本方法，即简单的语义搜索。然而，还在此基础上添加了一系列算法以提高性能。其中包括：

- 父文档检索器：这允许您为每个父文档创建多个嵌入，使您能够查找较小的块但返回更大的上下文。
- 自查询检索器：用户的问题通常包含对不仅仅是语义的内容的引用，而是表示某些逻辑的元数据过滤器。自查询允许您从查询中解析出查询的语义部分以及其他存在的元数据过滤器。
- 集成检索器：有时您可能希望从多个不同的来源或使用多个不同的算法检索文档。集成检索器使您可以轻松实现这一点。

```
from langchain_community.document_loaders import TextLoader

loader = TextLoader("../../state_of_the_union.txt")
```

```
from langchain.text_splitter import CharacterTextSplitter
from langchain_community.vectorstores import FAISS
from langchain_openai 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")
```

数据连接（Retrieval）- 索引（Indexing）

LangChain索引API将您的数据从任何来源同步到向量存储中：

- 避免将重复内容写入向量存储
- 避免重新编写未更改的内容
- 避免在未更改的内容上重新计算嵌入

在索引内容时，对每个文档计算哈希，并将以下信息存储在记录管理器中：

- 文档哈希（页面内容和元数据的哈希）
- 写入时间
- 源ID - 每个文档应包含元数据中的信息，以便我们确定该文档的最终来源。

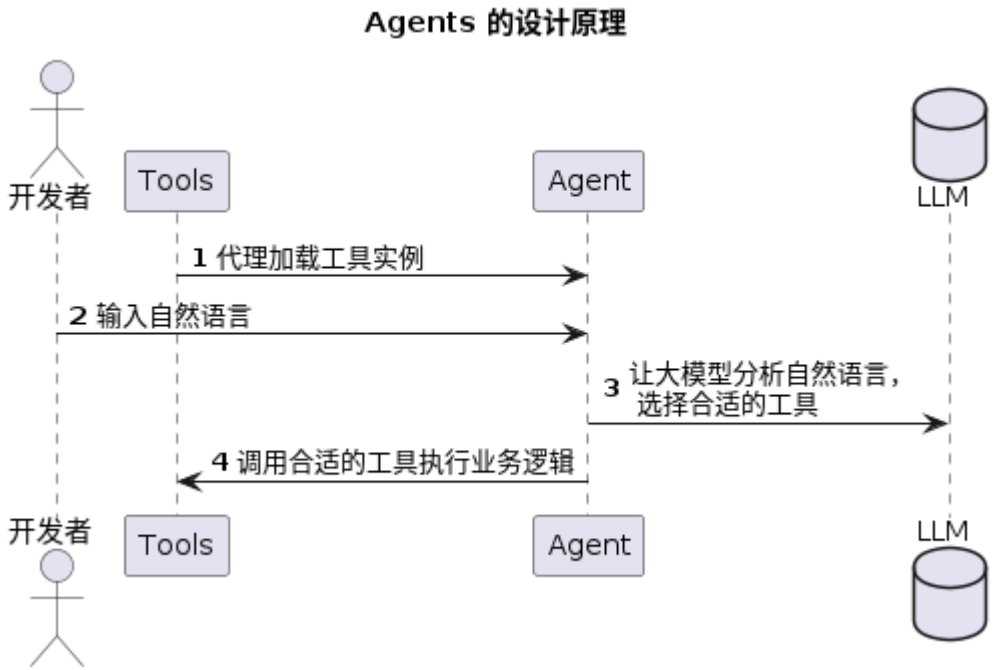
Cleanup Mode	De-Duplicates Content	Parallelizable	Cleans Up Deleted Source Docs	Cleans Up Mutations of Source Docs and/or Derived Docs	Clean Up Timing
None	✅	✅	❌	❌	-
Incremental	✅	✅	❌	✅	Continuously
Full	✅	❌	✅	✅	At end of indexing

智能体（Agents）

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。

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。
- **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。
- **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。



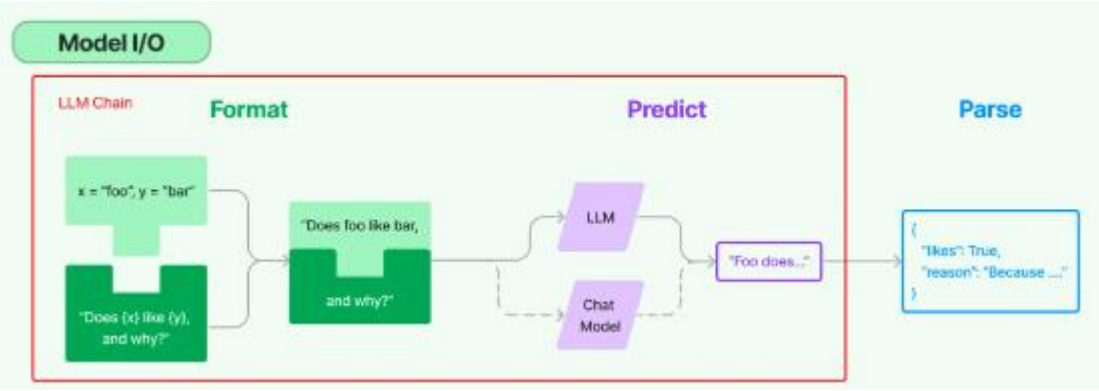
智能体（Agents）- 类型

Agent Type	Intended Model Type	Supports Chat History	Supports Multi-Input Tools	Supports Parallel Function Calling	Required Model Params	When to Use
OpenAI Tools	Chat	✅	✅	✅	tools	如果您正在使用最新的OpenAI模型（从1106开始）
OpenAI Functions	Chat	✅	✅		functions	如果您正在使用OpenAI模型，或者是经过微调以进行函数调用并公开与OpenAI相同的函数参数的开源模型
XML	LLM	✅				如果您正在使用Anthropic模型或其他擅长XML的模型
Structured Chat	Chat	✅	✅			如果您需要支持具有多个输入的工具
JSON Chat	Chat	✅				如果您正在使用擅长JSON的模型
ReAct	LLM	✅				如果您正在使用简单的模型
Self Ask With Search	LLM					如果您正在使用简单的模型并且只有一个搜索工具

链（Chains）

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。

下面是一个简单的链，由PromptTemplate和LLM组成，它使用提供的输入键值格式化提示模板，将格式化的字符串传递给LLM，并返回LLM的输出。




```python
chat = ChatOpenAI(temperature=0)
prompt_template = "Tell me a {adjective} joke"
llm_chain = LLMChain(
    llm=chat,
    prompt=PromptTemplate.from_template(prompt_template)
)

llm_chain(inputs={"adjective":"corny"})
```

输出：

```
{'adjective': 'corny',
 'text': 'Why did the tomato turn red? Because it saw the salad dressing!'}
```

链（Chains） - 类型

Chain Constructor	Function Calling	Other Tools	When to Use
create_stuff_documents_chain			这个链条接收一个文档列表，并将它们格式化为一个提示，然后将该提示传递给一个LLM（语言模型）。它会传递所有的文档，所以你需要确保它适应你正在使用的LLM的上下文窗口。
create_openai_fn_runnable	✅		如果你想使用OpenAI的函数调用来可选地结构化输出响应，你可以传入多个函数供其调用，但它不一定要调用它们。
create_structured_output_runnable	✅		如果你想使用OpenAI的函数调用来强制LLM以特定函数回应，你只能传入一个函数，链条将始终返回该响应。
load_query_constructor_runnable			可以用于生成查询。你必须指定一个允许的操作列表，然后返回一个可运行的对象，将自然语言查询转换为这些允许的操作。
create_sql_query_chain		SQL Database	如果你想从自然语言构建一个SQL数据库的查询。
create_history_aware_retriever		Retriever	这个链条接收对话历史，然后使用它来生成一个搜索查询，该查询被传递给底层的检索器。
create_retrieval_chain		Retriever	这个链条接收一个用户查询，然后将其传递给检索器来获取相关文档。这些文档（和原始输入）然后被传递给LLM生成一个回应。

其它 Legacy Chains：

- MapRerankDocumentsChain
- ConstitutionalChain
- LLMChain
- ElasticsearchDatabaseChain
- FlareChain
- ArangoGraphQAchain
- GraphCypherQAchain
- FalkorDBGraphQAchain
- HugeGraphQAchain
- KuzuQAChain
- NebulaGraphQAchain
- NeptuneOpenCypherQAchain
- GraphSparqlChain
- LLMMath
- LLMCheckerChain
- ……

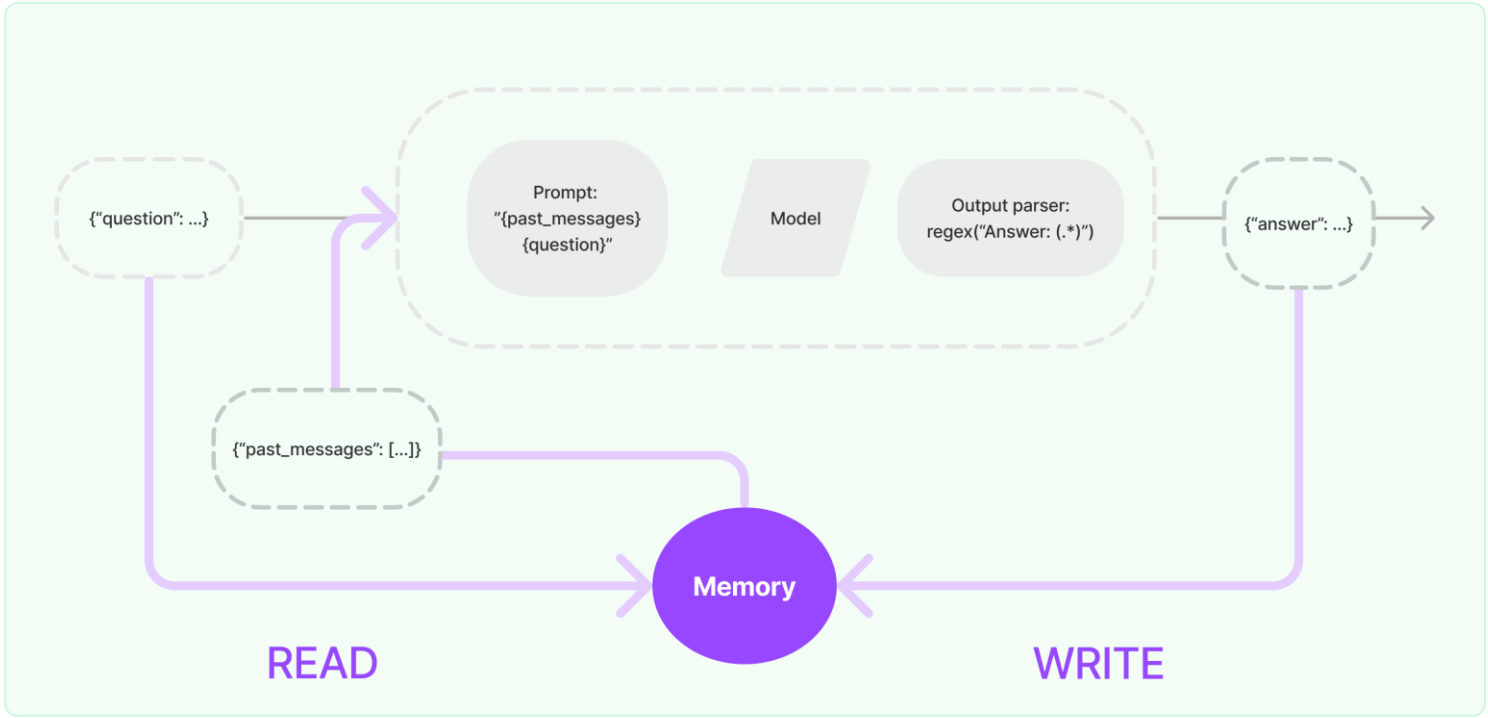
记忆（memory）

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。

一个记忆系统需要支持两个基本操作：

读取和写入。每个链定义了一些核心执行逻辑，需要特定的输入。其中一些输入直接来自用户，但其中一些输入可以来自内存。在给定的运行中，一个链将与其记忆系统交互两次。

- 在接收到初始用户输入之后，但在执行核心逻辑之前，一个链将从其记忆系统中读取并增强用户输入。
- 在执行核心逻辑之后，但在返回答案之前，一个链将把当前运行的输入和输出写入内存，以便在未来的运行中可以引用它们。



记忆（memory）- 类型

- Conversation Buffer
- Conversation Buffer Window
- Entity
- Conversation Knowledge Graph
- Conversation Summary
- Conversation Summary Buffer
- Conversation Token Buffer
- Backed by a Vector Store

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。

记忆（memory）- Memory in LLMChain

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。最重要的步骤是正确设置提示。在下面的提示中，我们有两个输入键：一个用于实际输入，另一个用于来自Memory类的输入。重要的是，我们确保PromptTemplate 和 ConversationBufferMemory中的键匹配（chat_history）。

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")
```

记忆（memory）- Memory in LLMChain

执行代码可以得到如下输出结果：

```
> Entering new LLMChain chain...
Prompt after formatting:
You are a chatbot having a conversation with a human.


Human: Hi there my friend
Chatbot:

> Finished chain.
```

在此基础上继续执行如下语句：

```
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:
You are a chatbot having a conversation with a human.

Human: Hi there my friend
AI:  Hi there! How can I help you today?
Human: Not too bad - how are you?
Chatbot:

> Finished chain.
```

```
" I'm doing great, thanks for asking! How are you doing?"
```

回调（Callback）

LangChain提供了一个回调系统，允许在LLM应用程序的各个阶段进行钩子处理。这对于日志记录、监控、流式传输和其他任务非常有用。可以通过在整个API中使用可用的回调参数来订阅这些事件。

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:
    """Base callback handler that can be used to handle callbacks from langchain."""

    def on_llm_start(
        self, serialized: Dict[str, Any], prompts: List[str], **kwargs: Any
    ) -> Any:
        """Run when LLM starts running."""

    def on_chat_model_start(
        self, serialized: Dict[str, Any], messages: List[List[BaseMessage]], **kwargs: Any
    ) -> Any:
        """Run when Chat Model starts running."""

    def on_llm_new_token(self, token: str, **kwargs: Any) -> Any:
        """Run on new LLM token. Only available when streaming is enabled."""

    def on_llm_end(self, response: LLMResult, **kwargs: Any) -> Any:
        """Run when LLM ends running."""

    def on_llm_error(
        self, error: Union[Exception, KeyboardInterrupt], **kwargs: Any
    ) -> Any:
        """Run when LLM errors."""

    def on_chain_start(
        self, serialized: Dict[str, Any], inputs: Dict[str, Any], **kwargs: Any
    ) -> Any:
        """Run when chain starts running."""

    def on_chain_end(self, outputs: Dict[str, Any], **kwargs: Any) -> Any:
        """Run when chain ends running."""

    def on_chain_error(
        self, error: Union[Exception, KeyboardInterrupt], **kwargs: Any
    ) -> Any:
        """Run when chain errors."""
```

```python
def on_tool_start(
    self, serialized: Dict[str, Any], input_str: str, **kwargs: Any
) -> Any:
    """Run when tool starts running."""

def on_tool_end(self, output: str, **kwargs: Any) -> Any:
    """Run when tool ends running."""

def on_tool_error(
    self, error: Union[Exception, KeyboardInterrupt], **kwargs: Any
) -> Any:
    """Run when tool errors."""

def on_text(self, text: str, **kwargs: Any) -> Any:
    """Run on arbitrary text."""

def on_agent_action(self, action: AgentAction, **kwargs: Any) -> Any:
    """Run on agent action."""

def on_agent_finish(self, finish: AgentFinish, **kwargs: Any) -> Any:
    """Run on agent end."""
```

回调（Callback）

LangChain提供了一些内置的处理程序。这些处理程序在langchain/callbacks模块中可用。最基本的处理程序是StdOutCallbackHandler，它简单地将所有事件记录到标准输出(stdout)中。

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

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

# Constructor callback: First, let's explicitly set the StdOutCallbackHandler when initializing our chain
chain = LLMChain(llm=llm, prompt=prompt, callbacks=[handler])
chain.invoke({"number":2})

# Use verbose flag: Then, let's use the `verbose` flag to achieve the same result
chain = LLMChain(llm=llm, prompt=prompt, verbose=True)
chain.invoke({"number":2})

# Request callbacks: Finally, let's use the request `callbacks` to achieve the same result
chain = LLMChain(llm=llm, prompt=prompt)
chain.invoke({"number":2}, {"callbacks":[handler]})
```



```
> Entering new LLMChain chain...
Prompt after formatting:
1 + 2 =

> Finished chain.


> Entering new LLMChain chain...
Prompt after formatting:
1 + 2 =

> Finished chain.


> Entering new LLMChain chain...
Prompt after formatting:
1 + 2 =

> Finished chain.
```

安装配置简介

https://python.langchain.com/docs/get_started/quickstart

langchain实际上是python的一个开发包，所以可以通过pip或者conda两种方式来安装：

1. pip安装：
pip install langchain

2. conda安装：
conda install langchain -c conda-forge

3. 默认情况下上面的安装方式是最简单的安装，还有很多和langchain集成的modules并没有安装进来，如果你希望安装common LLM providers的依赖模块，那么可以通过下面的命令：
pip install langchain[llms]

4. 如果你想安装所有的模块，那么可以使用下面的命令：
pip install langchain[all]

5. 因为langchain是开源软件，所以你也可以通过源代码来安装,下载好源代码之后，通过下面的命令安装即可：
pip install -e .


Thank
You