# **大模型开发之LangChain**

官网：https://www.langchain.com/

LangChain是 \*\*2022年10月\*\* ，由哈佛大学的 \*\*Harrison Chase\*\* （哈里森·蔡斯）发起研发的一个开源框架，用于开发由大语言模型（LLMs）驱动的应用程序。

比如，搭建“智能体”（Agent）、问答系统（QA）、对话机器人、文档搜索系统、企业私有知识库等。

- LangChain 也是一套面向大模型的开发框架（SDK）

- LangChain 是 AGI 时代软件工程的一个探索和原型

LangChain ≠ LLMs

LangChain 之于 LLMs，类似 Spring 之于 Java

顾名思义，LangChain中的“Lang”是指language，即⼤语⾔模型，“Chain”即“链”，也就

是将⼤模型与外部数据&各种组件连接成链，以此构建AI应⽤程序。



\*\*有哪些大模型应用开发框架呢？\*\*



\*\*LangChain\*\* ：这些工具里出现最早、最成熟的，适合复杂任务分解和单智能体应用

\*\*LlamaIndex\*\* ：专注于高效的索引和检索，适合 RAG 场景。（注意不是Meta开发的）

\*\*LangChain4J\*\* ：LangChain还出了Java、JavaScript（\*\*LangChain.js\*\*）两个语言的版本，

LangChain4j的功能略少于LangChain，但是主要的核心功能都是有的

\*\*SpringAI/SpringAI Alibaba\*\* ：有待进一步成熟，此外只是简单的对于一些接口进行了封装

\*\*SemanticKernel\*\* ：也称为sk，微软推出的，对于C#同学来说，那就是5颗星

\*\*为什么需要LangChain？\*\*

在大语言模型（LLM）如 ChatGPT、Claude、DeepSeek 等快速发展的今天，开发者不仅希望能“使用”这些模型，还希望能 \*\*将它们灵活集成到自己的应用中\*\* ，实现更强大的对话能力、检索增强生成（RAG）、工具调用（Tool Calling）、多轮推理等功能

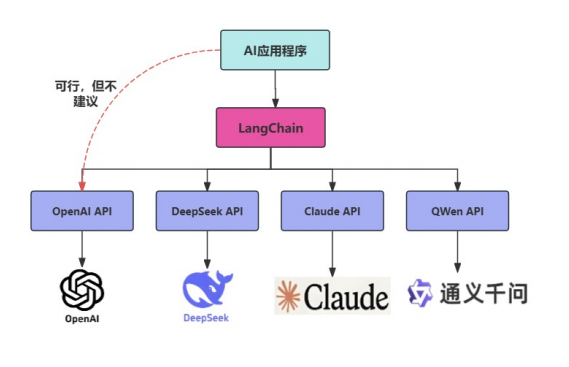
## 使用LangChain的好处

\*\*简化开发难度\*\*：更简单、更高效、效果更好

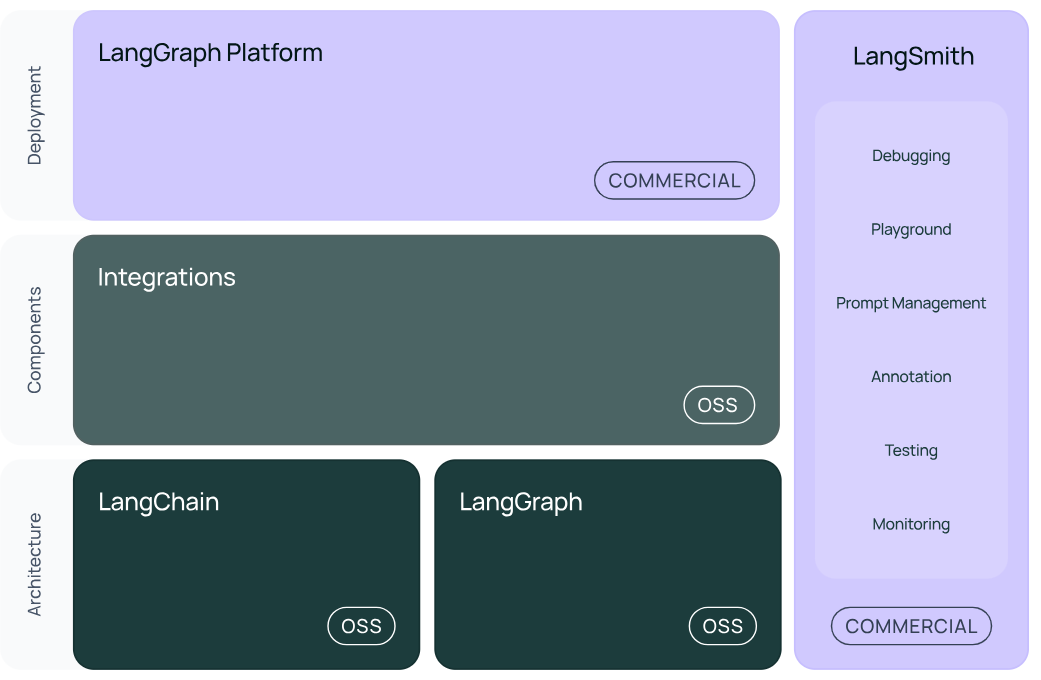
\*\*学习成本更低\*\*：不同模型的API不同，调用方式也有区别，切换模型时学习成本高。使用

LangChain，可以以统一、规范的方式进行调用，有更好的移植性。

\*\*现成的链式组装\*\*：LangChain提供了一些 \*\*现成的链式组装\*\* ，用于完成特定的高级任务。让复杂的逻辑变得 \*\*结构化、易组合、易扩展\*\*



# LangChain架构设计



**结构1：LangChain**

**langchain**：构成应用程序认知架构的Chains，Agents，Retrieval strategies等

构成应⽤程序的链、智能体、RAG。

**langchain-community**：第三方集成⽐如：Model I/O、Retrieval、Tool & Toolkit；合作伙伴包 langchain-openai，langchain anthropic等。

**langchain-Core**：基础抽象和LangChain表达式语言 (LCEL)

<B>  
   小结：LangChain，就是AI应用组装套件，封装了一堆的API。langchain框架不大，但是里面琐碎的知识点特别多。就像玩乐高，提供了很多标准化的乐高零件（比如，连接器、轮子等）  
</B>

**结构2：LangGraph**

LangGraph可以看做基于LangChain的api的进一步封装，能够协调多个Chain、Agent、Tools完成更复杂的任务，实现更高级的功能。

**结构3：LangSmith**

<https://docs.smith.langchain.com/>

**链路追踪。**提供了6大功能，涉及Debugging (调试)、Playground (沙盒)、Prompt Management (提示管理)、Annotation (注释)、Testing (测试)、Monitoring (监控)等。与LangChain无缝集成，帮助你从原型阶段过渡到生产阶段。

正是因为LangSmith这样的⼯具出现，才使得LangChain意义更⼤，要不仅靠⼀些API（当然也可以不⽤，⽤原⽣的API），⽀持不住LangChain的热度。

**结构4：LangServe**

将LangChain的可运行项和链部署为REST API，使得它们可以通过网络进行调用。

Java怎么调用langchain呢？就通过这个langserve。将langchain应用包装成一个rest api，对外暴露服务。同时，支持更高的并发，稳定性更好。

# **大模型应用开发**

### ****基于RAG架构的开发****

**背景：**

大模型的知识冻结

大模型幻觉

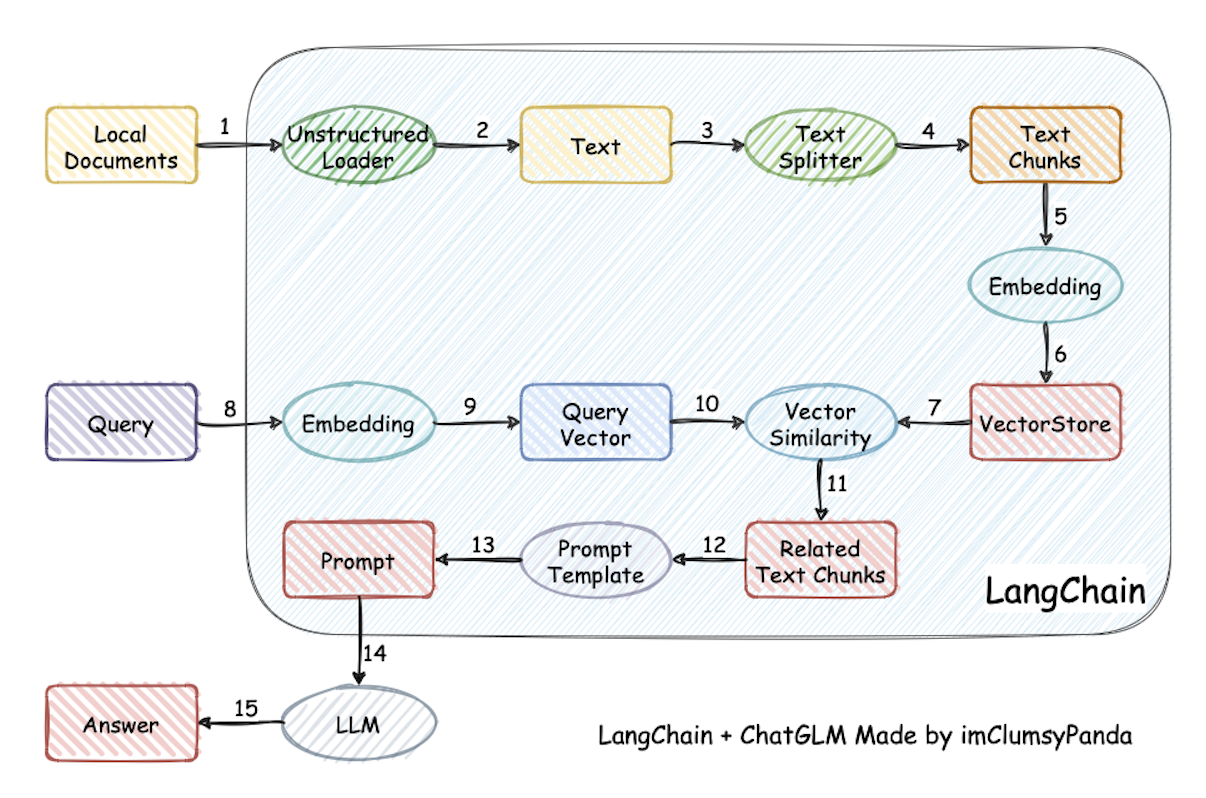
而RAG就可以非常精准的解决这两个问题。

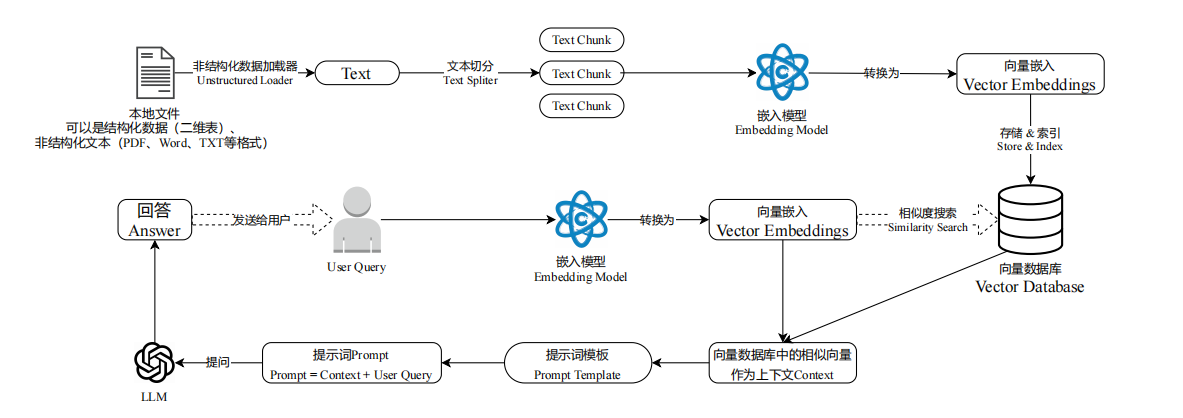
**举例：**

LLM在考试的时候面对陌生的领域，答复能力有限，然后就准备放飞自我了。而此时RAG给了一些提示和思路，让LLM懂了开始往这个提示的方向做，最终考试的正确率从60%到了90%！

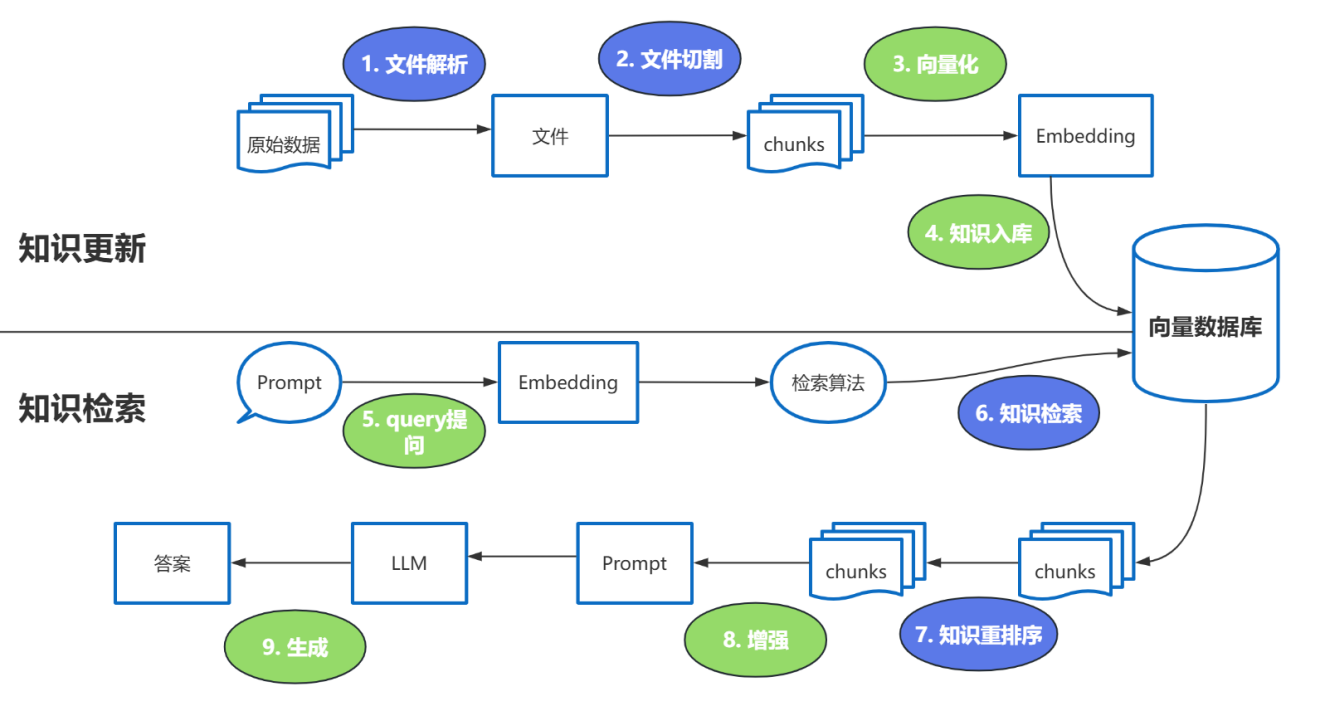
**何为RAG？**

Retrieval-Augmented Generation（检索增强生成）





强调一下难点的步骤：



这些过程中的难点：1、文件解析 2、文件切割 3、知识检索 4、知识重排序

**Reranker的使用场景：**

适合：追求 **回答高精度** 和 **高相关性** 的场景中特别适合使用 Reranker，例如专业知识库或者客服系统等应用。

不适合：引入reranker会增加召回时间，增加检索延迟。服务对 **响应时间要求高** 时，使用reranker可能不合适。

**这里有三个位置涉及到大模型的使用：**

第3步向量化时，需要使用EmbeddingModels。

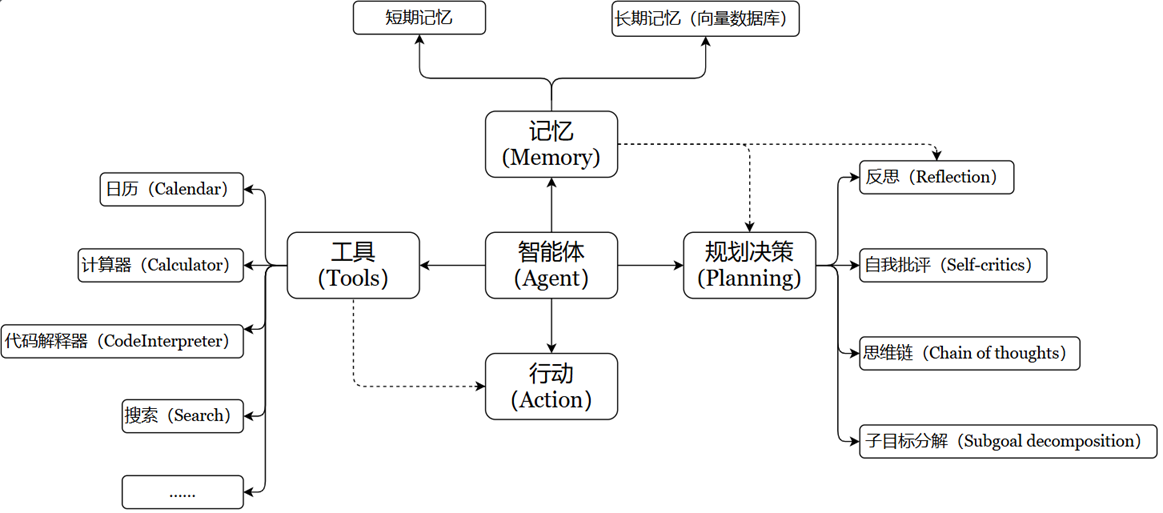
第7步重排序时，需要使用RerankModels。

第9步生成答案时，需要使用LLM。

### ****基于Agent架构的开发****

充分利用 LLM 的推理决策能力，通过增加 **规划** 、 **记忆** 和 **工具** 调用的能力，构造一个能够独立思考、逐步完成给定目标的智能体。

OpenAI的元老翁丽莲(Lilian Weng)于2023年6月在个人博客首次提出了现代AI Agent架构。



一个数学公式来表示：

**Agent = LLM + Memory + Tools + Planning + Action**

⽐如，打⻋到西藏玩。  
⼤脑中枢：规划⾏程的你  
规划：步骤1：规划打⻋路线，步骤2：定饭店、酒店，。。。  
调⽤⼯具：调⽤MCP或FunctionCalling等API，滴滴打⻋、携程、美团订酒店饭店  
记忆能⼒：沟通时，要知道上下⽂。⽐如定酒店得知道是西藏路上的酒店，不能聊着聊着忘  
了最初的⽬的。  
能够执⾏上述操作。说走就走，不能纸上谈兵。

**智能体核心要素被细化为以下模块：**

**1、大模型（LLM）作为“大脑”**：提供推理、规划和知识理解能力，是AI Agent的决策中枢。

⼤脑主要由⼀个⼤型语⾔模型 LLM 组成，承担着信息处理和决策等功能， 并可以呈现推理和规划的过程，能很好地应对未知任务。

**2、记忆（Memory）**

记忆机制能让智能体在处理重复⼯作时调⽤以前的经验，从而避免⽤⼾进⾏⼤量重复交互。

**短期记忆**：存储单次对话周期的上下文信息，属于临时信息存储机制。受限于模型的上下文窗口长度。

ChatGPT：⽀持约8k token的上下⽂  
GPT4：⽀持约32k token的上下⽂  
最新的很多⼤模型：都⽀持100万、1000万 token的上下⽂ （相当于2000万字⽂本或20小时视频）  
⼀般情况下模型中 token 和字数的换算⽐例⼤致如下：  
1 个英⽂字符 ≈ 0.3 个 token。  
1 个中⽂字符 ≈ 0.6 个 token。

**长期记忆**：可以横跨多个任务或时间周期，可存储并调用核心知识，非即时任务。

长期记忆，可以通过**模型参数微调（固化知识）**、**知识图谱（结构化语义网络）**或**向量数据库**

**（相似性检索）**方式实现。

**3、工具使用（Tool Use）**：调用外部工具（如API、数据库）扩展能力边界。

**4、规划决策（Planning）**：通过任务分解、反思与自省框架实现复杂任务处理。例如，利用思维链（Chain of Thought）将目标拆解为子任务，并通过反馈优化策略。

**5、行动（Action）**：实际执行决策的模块，涵盖软件接口操作（如自动订票）和物理交互（如机器人执行搬运）。比如：检索、推理、编程等。

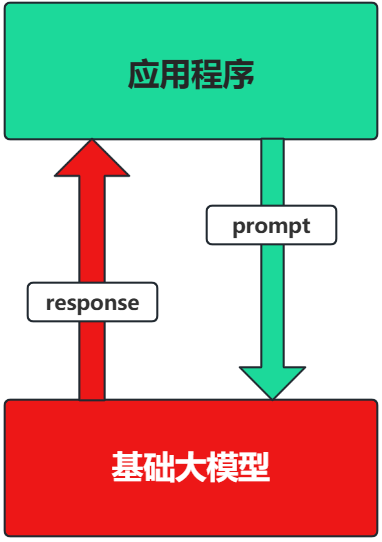
智能体会形成完整的计划流程。例如先读取以前⼯作的经验和记忆，之后规划⼦⽬标并使⽤相应⼯具去处理问题，最后输出给⽤⼾并完成反思。

### ****大模型应用开发的4个场景****

**场景1：纯 Prompt**

Prompt是操作大模型的唯一接口

当人看：你说一句，ta回一句，你再说一句，ta再回一句...

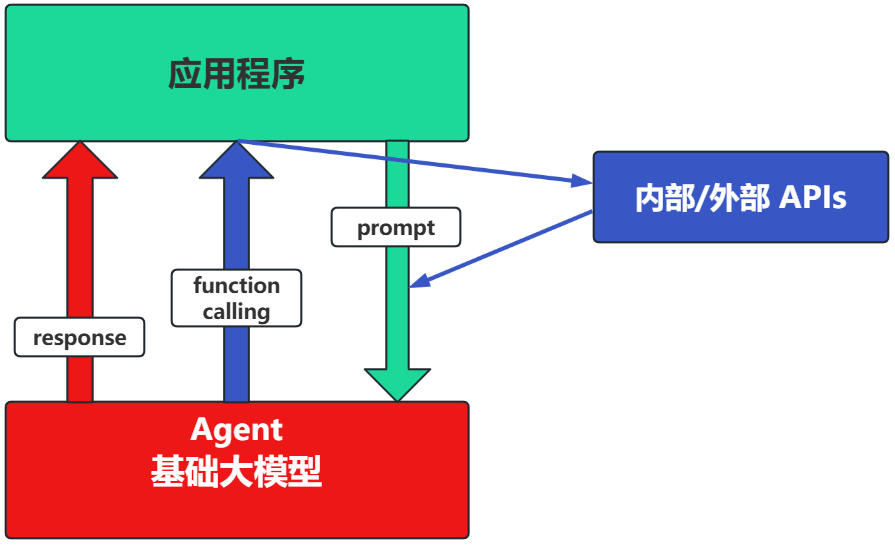


**场景2：Agent + Function Calling**

Agent：AI 主动提要求

Function Calling：需要对接外部系统时，AI 要求执行某个函数

当人看：你问 ta「我明天去杭州出差，要带伞吗？」，ta 让你先看天气预报，你看了告诉ta，ta再告诉你要不要带伞



**场景3：RAG (Retrieval-Augmented Generation)**

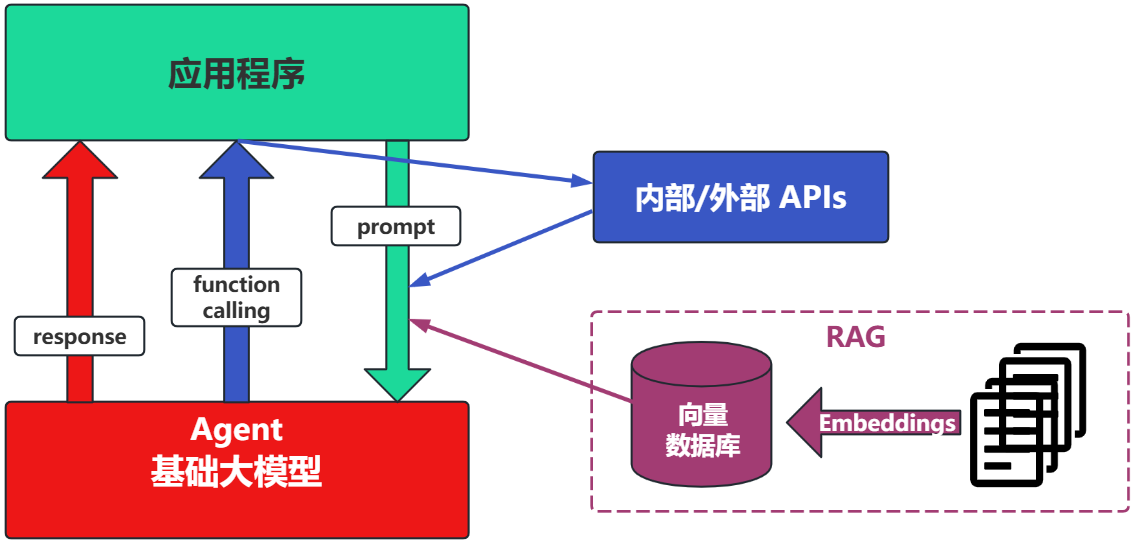
RAG：需要补充领域知识时使用

Embeddings：把文字转换为更易于相似度计算的编码。这种编码叫向量

向量数据库：把向量存起来，方便查找

向量搜索：根据输入向量，找到最相似的向量

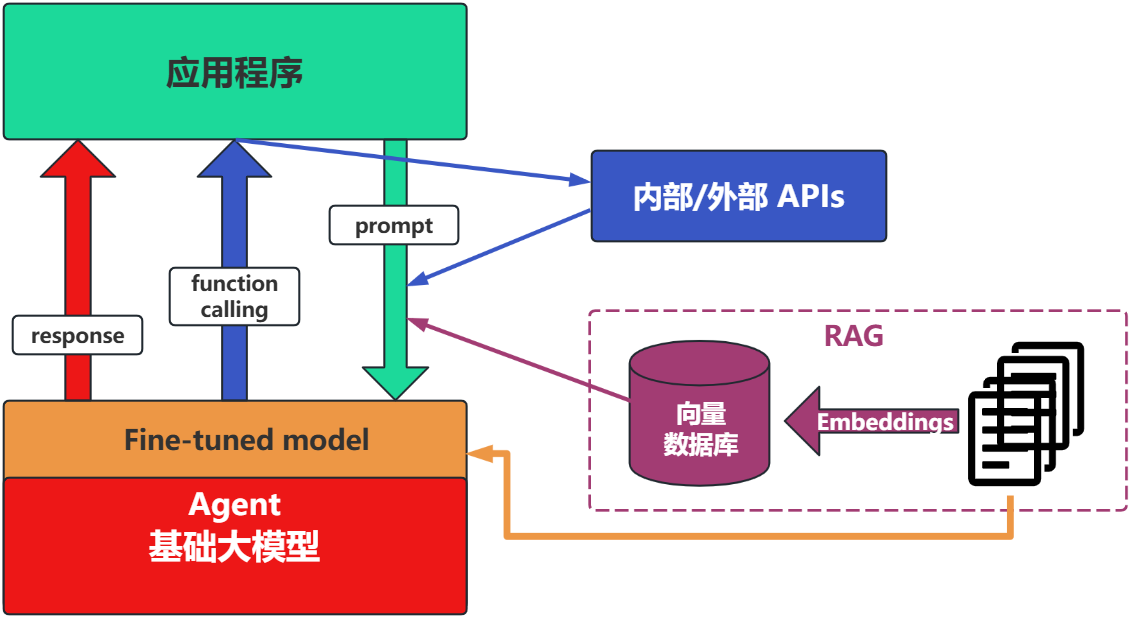
举例：考试答题时，到书上找相关内容，再结合题目组成答案



这个在智能客服上用的最广泛。

**场景4：Fine-tuning(精调/微调)**

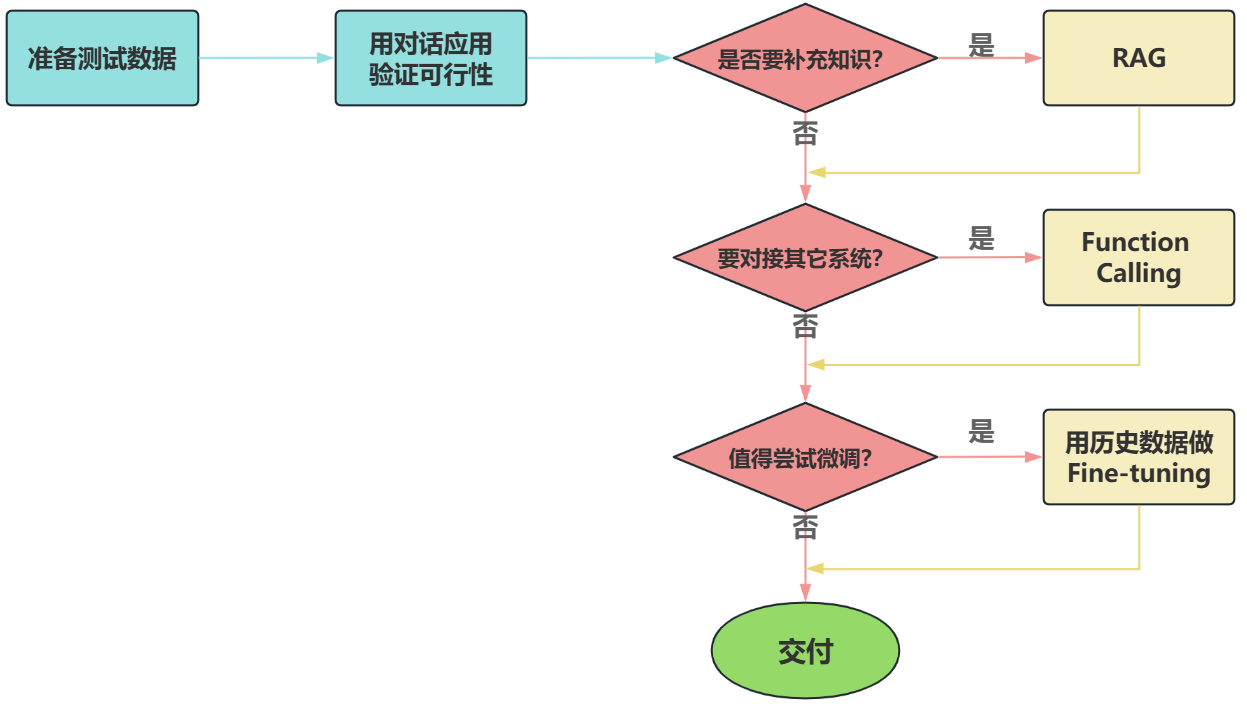
举例：努力学习考试内容，长期记住，活学活用。



特点：成本最高；在前面的方式解决不了问题的情况下，再使用。

**如何选择**

面对一个需求，如何开始，如何选择技术方案？下面是个常用思路：



# **LangChain 的核心组件**

**一个问题引发的思考**

**如果要组织一个AI应用，开发者一般需要什么？**

第1，提示词模板的构建，不仅仅只包含用户输入。

第2，模型调用与返回，参数设置，返回内容的格式化输出。

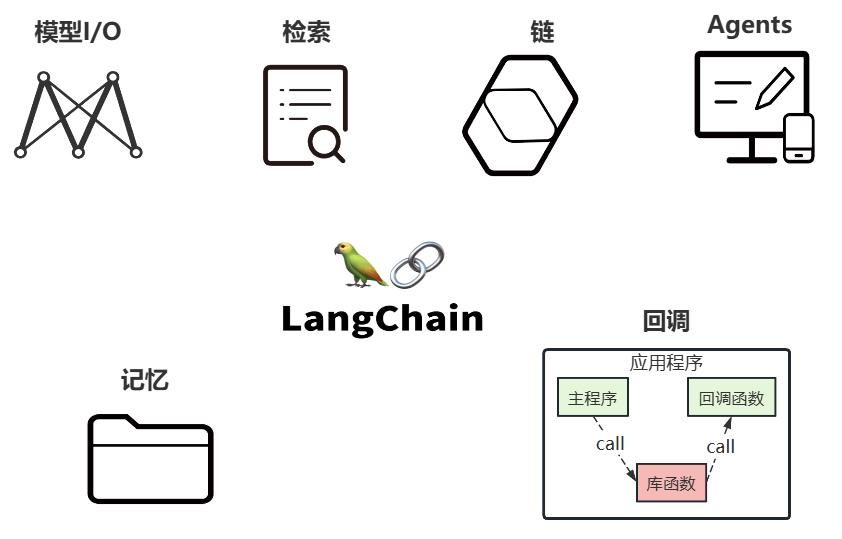
第3，知识库查询，这里会包含文档加载，切割，以及转化为词嵌入（Embedding）向量。

第4，其他第三方工具调用，一般包含天气查询、Google搜索、一些自定义的接口能力调用。

第5，记忆获取，每一个对话都有上下文，在开启对话之前总得获取到之前的上下文吧？

### ****核心组件的概述****

LangChain的核心组件涉及六大模块，这六大模块提供了一个全面且强大的框架，使开发者能够创建复杂、高效且用户友好的基于大模型的应用。



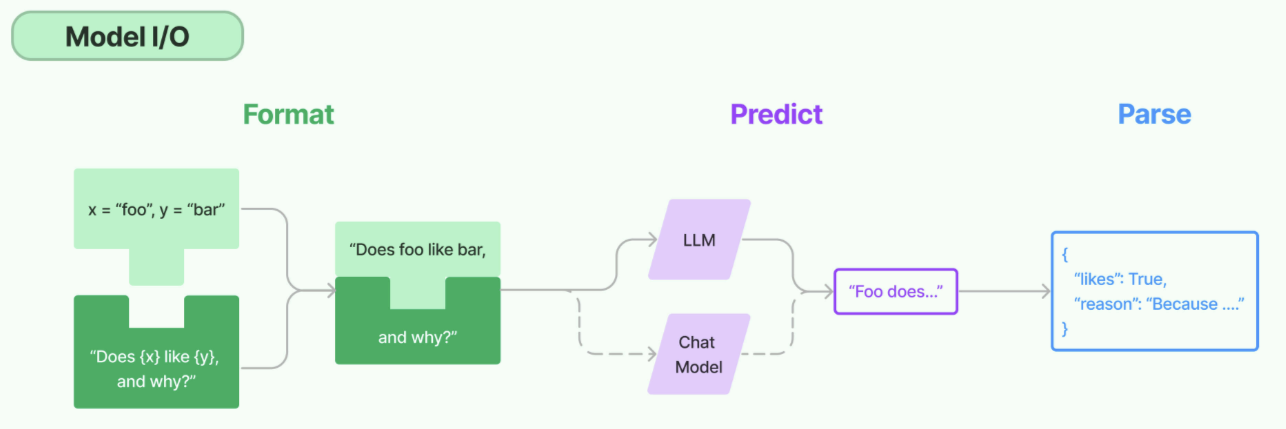
### ****核心组件的说明****

#### ****核心组件1：Model I/O****

这个模块使⽤最多，也最简单

Model I/O：标准化各个大模型的输入和输出，包含输入模版，模型本身和格式化输出。

以下是使用语言模型从输入到输出的基本流程。



以下是对每一块的总结：

**Format(格式化)** ：即指代Prompts Template，通过模板管理大模型的输入。将原始数据格式化成模型可以处理的形式，插入到一个模板问题中，然后送入模型进行处理。

**Predict(预测)** ：即指代Models，使用通用接口调用不同的大语言模型。接受被送进来的问题，然后基于这个问题进行预测或生成回答。

**Parse(生成)** ：即指代Output Parser 部分，用来从模型的推理中提取信息，并按照预先设定好的模版来规范化输出。比如，格式化成一个结构化的JSON对象。

#### ****核心组件2：Chains****

Chain："链条"，用于将多个模块串联起来组成一个完整的流程，是 LangChain 框架中最重要的模块。

例如，一个 Chain 可能包括一个 Prompt 模板、一个语言模型和一个输出解析器，它们一起工作以处理用户输入、生成响应并处理输出。

**常见的Chain类型：**

* **LLMChain** ：最基础的模型调用链
* **SequentialChain** ：多个链串联执行
* **RouterChain** ：自动分析用户的需求，引导到最适合的链
* **RetrievalQA** ：结合向量数据库进行问答的链

#### ****核心组件3：Memory****

Memory：记忆模块，用于保存对话历史或上下文信息，以便在后续对话中使用。

**常见的 Memory 类型：**

* **ConversationBufferMemory** ：保存完整的对话历史
* **ConversationSummaryMemory** ：保存对话内容的精简摘要（适合长对话）
* **ConversationSummaryBufferMemory** ：混合型记忆机制，兼具上面两个类型的特点
* **VectorStoreRetrieverMemory** ：保存对话历史存储在向量数据库中

#### ****核心组件4：Agents****

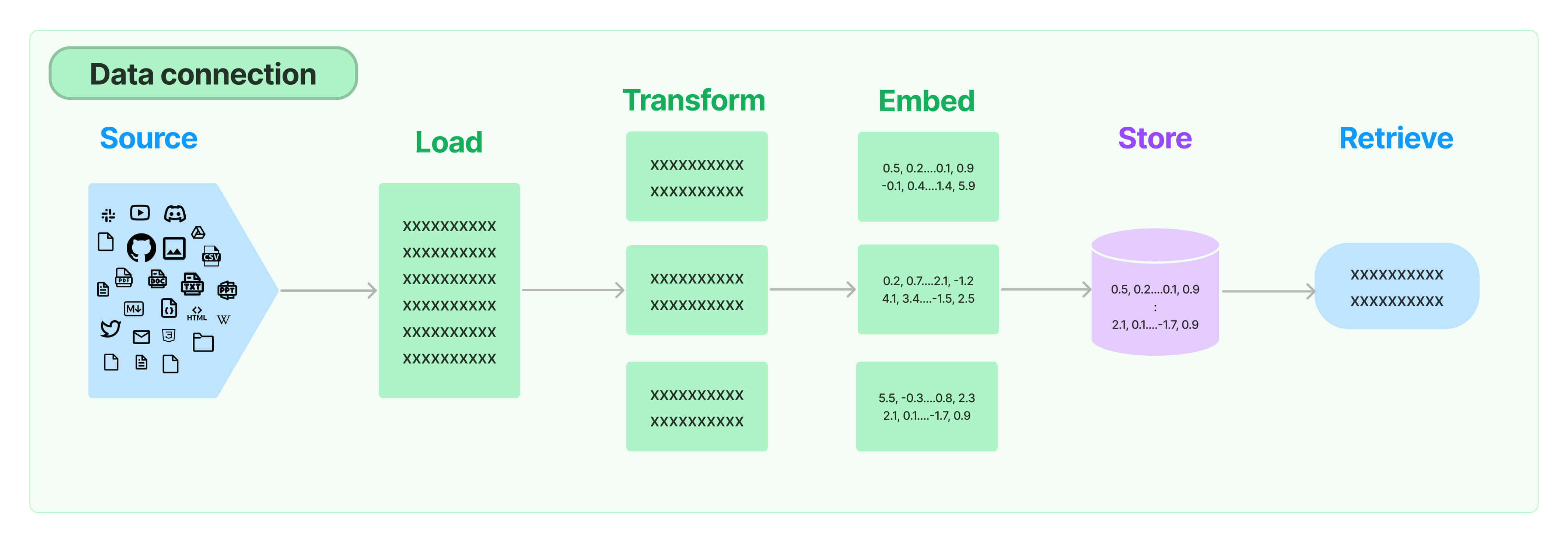
Agents，对应着智能体，是 LangChain 的高阶能力，它可以自主选择工具并规划执行步骤。

**Agent 的关键组成：**

* **AgentType** ：定义决策逻辑的工作流模式
* **Tool** ：是一些内置的功能模块，如API调用、搜索引擎、文本处理、数据查询等工具。Agents通过这些工具来执行特定的功能。
* **AgentExecutor** ：用来运行智能体并执行其决策的工具，负责协调智能体的决策和实际的工具执行。

#### ****核心组件5：Retrieval****

Retrieval：对应着RAG，检索外部数据，然后在执行生成步骤时将其传递到 LLM。步骤包括文档加载、切割、Embedding等



* **Source** ：数据源，即大模型可以识别的多种类型的数据：视频、图片、文本、代码、文档等。
* **Load** ：负责将来自不同数据源的非结构化数据，加载为文档(Document)对象
* **Transform** ：负责对加载的文档进行转换和处理，比如将文本拆分为具有语义意义的小块。
* **Embed** ：将文本编码为向量的能力。一种用于嵌入文档，另一种用于嵌入查询
* **Store** ：将向量化后的数据进行存储
* **Retrieve** ：从大规模文本库中检索和查询相关的文本段

#### ****核心组件6：Callbacks****

Callbacks：回调机制，允许连接到 LLM 应用程序的各个阶段，可以监控和分析LangChain的运行情况，比如日志记录、监控、流传输等，以优化性能。

回调函数，对于程序员们应该都不陌⽣。这个函数允许我们在LLM的各个阶段使⽤各种各样的  
“钩⼦”，从而达实现⽇志的记录、监控以及流式传输等功能。

**小结**

Model I/O模块：使用最多，也最简单  
​  
Chains 模块： 最重要的模块  
​  
Retrieval模块、Agents模块：大模型的主要落地场景  
​  
在这个基础上，其它组件要么是它们的辅助，要么只是完成常规应用程序的任务。

**文档（以 Python 版为例）**

* 功能模块：<https://python.langchain.com/docs/get_started/introduction>
* API 文档：<https://api.python.langchain.com/en/latest/langchain_api_reference.html>
* 三方组件集成：<https://python.langchain.com/docs/integrations/platforms/>
* 官方应用案例：<https://python.langchain.com/docs/use_cases>
* 调试部署等指导：<https://python.langchain.com/docs/guides/debugging>

conda create -n langchain-learn python=3.10

## Section01 模型I/O

把不同的模型，统一封装成一个接口，方便更换模型而不用重构代码。

**按照模型功能的不同：**

非对话模型（LLMs、Text Model）

对话模型（Chat Models）（ **推荐** ）

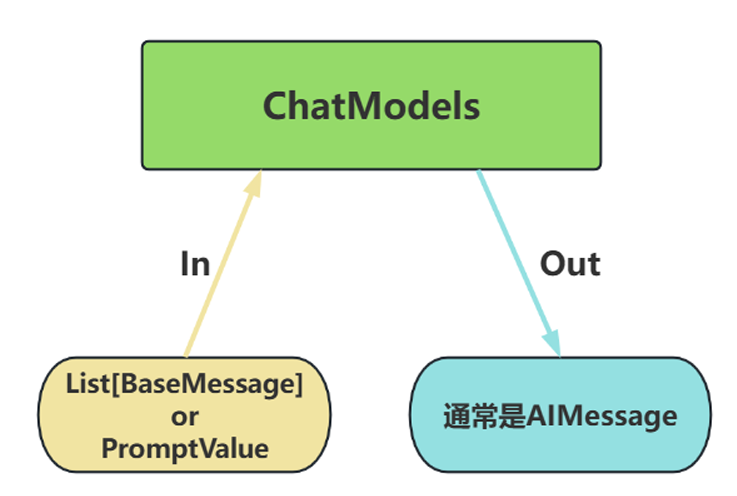
嵌入模型（Embedding Models）

### 1.1 模型 API

**LLMs，**也叫Text Model、非对话模型，是许多语言模型应用程序的支柱。

**ChatModels，**也叫聊天模型、对话模型，底层使用LLMs。

**大语言模型调用，以 ChatModel 为主！**



主要特点如下：

**输入**：接收消息列表 **List[BaseMessage]** 或 **PromptValue** ，每条消息需指定角色（如

SystemMessage、HumanMessage、AIMessage）

**输出**：总是返回带角色的 **消息对象** （ **BaseMessage** 子类），通常是 **AIMessage**

# !pip install --upgrade langchain  
# !pip install --upgrade langchain-openai  
# !pip install --upgrade langchain-community

#### 1.1.1 OpenAI 模型封装

from openai import OpenAI  
from dotenv import load\_dotenv  
import os  
load\_dotenv()  
key = os.getenv('DASHSCOPE\_API\_KEY')

from openai import OpenAI  
from dotenv import load\_dotenv  
import os  
load\_dotenv()  
api\_key = os.getenv('API\_KEY')  
​  
# 初始化客户端，将 base\_url 指向 DeepSeek 的 API 端点  
client = OpenAI(  
    api\_key=api\_key,  # 请替换为你在 DeepSeek 平台获取的密钥  
    base\_url="https://api.deepseek.com/v1"  # DeepSeek 的 API 端点[citation:1]  
)  
​  
# 创建聊天补全  
res = client.chat.completions.create(  
    model='deepseek-chat',  # 指定使用的 DeepSeek 模型[citation:6]  
    messages=[  
       {"role": "system", "content": "你是乐于助人的小助手，请根据用户的问题给出回答"},  
       {"role": "user", "content": "你是谁?"},  
   ]  
)  
​  
print(res.choices[0].message.content)  
print(res.model\_dump\_json())

# 通过Langchain调用DeepSeek对话模型  
from langchain\_openai import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
​  
# 加载环境变量（确保.env文件中配置了DeepSeek的密钥和地址）  
load\_dotenv()  
​  
# 获取DeepSeek的API密钥和基础URL  
deepseek\_api\_key = os.getenv("API\_KEY")  # 从.env文件读取，也可直接替换为你的密钥  
deepseek\_base\_url = os.getenv("BASE\_URL")  # DeepSeek兼容OpenAI的API地址  
​  
​  
# 1. 获取DeepSeek对话模型实例  
chat = ChatOpenAI(  
    model="deepseek-chat",  # DeepSeek对话模型名称（固定为deepseek-chat）  
    api\_key=deepseek\_api\_key,  # DeepSeek的API密钥  
    base\_url=deepseek\_base\_url or "https://api.deepseek.com/v1",  # DeepSeek兼容OpenAI的API基础地址  
    temperature=0.7  # 可根据需要调整随机性（0-1之间）  
)  
​  
# 2. 调用模型（对话格式与OpenAI兼容，支持system/user/assistant角色）  
res = chat.invoke([  
   {"role": "system", "content": "你是乐于助人的小助手，请根据用户问题进行回答"},  
   {"role": "user", "content": "你是谁?"},  
])  
​  
# 3. 处理返回结果（两种常用方式）  
print("原始JSON结果：")  
print(res.model\_dump\_json(indent=2))  # 格式化输出完整结果  
​  
print("\n模型回答内容：")  
print(res.content)  # 直接提取回答文本

#### **1.1.2 **模型调用的主要方法及参数****

**相关方法及属性：**

\*\*OpenAI(...) / ChatOpenAI(...)\*\* ：创建一个模型对象（非对话类/对话类）  
​  
\*\*model.invoke(xxx)\*\* ：执行调用，将用户输入发送给模型  
​  
\*\*.content\*\* ：提取模型返回的实际文本内容  
​  
模型调用函数使用时需初始化模型，并设置必要的参数。

**1）必须设置的参数：**

\*\*base\_url\*\* ：大模型 API 服务的根地址  
​  
\*\*api\_key\*\* ：用于身份验证的密钥，由大模型服务商（如 OpenAI、百度千帆）提供  
​  
\*\*model/model\_name\*\* ：指定要调用的具体大模型名称（如 \*\*gpt-4-turbo\*\* 、 \*\*ERNIE-3.5-8K\*\* 等）

**2）其它参数：**

\*\*temperature\*\* ：温度，控制生成文本的“随机性”，取值范围为0～1。  
​  
\*\*值越低\*\* → 输出越确定、保守（适合事实回答）  
​  
\*\*值越高\*\* → 输出越多样、有创意（适合创意写作）  
​  
通常，根据需要设置如下：  
​  
精确模式（0.5或更低）：生成的文本更加安全可靠，但可能缺乏创意和多样性。  
​  
平衡模式（通常是0.8）：生成的文本通常既有一定的多样性，又能保持较好的连贯性和准确性。  
​  
创意模式（通常是1）：生成的文本更有创意，但也更容易出现语法错误或不合逻辑的内容。

**max\_tokens** ：限制生成文本的最大长度，防止输出过长。

\*\*Token是什么？\*\*  
​  
\*\*基本单位\*\* : 大模型处理文本的最小单位是token（相当于自然语言中的词或字），输出时逐个token依次生成。  
​  
\*\*收费依据\*\* ：大语言模型(LLM)通常也是以token的数量作为其计量(或收费)的依据:  
​  
1个Token≈1-1.8个汉字，1个Token≈3-4个英文字母  
​  
Token与字符转化的可视化工具：  
​  
OpenAI提供：https:*//platform.openai.com/tokenizer*  
​  
*百度智能云提供：https://*console.bce.baidu.com/support/#/tokenizer

**max\_tokens设置建议：**

客服短回复：128-256。比如：生成一句客服回复（如“订单已发货，预计明天送达”）

常规对话、多轮对话：512-1024

长内容生成：1024-4096。比如：生成一篇产品说明书（包含功能、使用方法等结构）

**各平台API的调用**

OpenAI 官方API:  
https://www.closeai-asia.com  
​  
百度千帆平台:  
https://cloud.baidu.com/doc/qianfan-docs/s/Mm8r1mejk  
​  
阿里云百炼平台  
注册与key的获取：  
提前开通百炼平台账号并申请API KEY：https://bailian.console.aliyun.com/#/home  
​  
智谱的GLM  
注册智谱模型并获取API Key：  
https://www.bigmodel.cn/usercenter/proj-mgmt/apikeys  
​  
硅基流动平台  
官网：https://www.siliconflow.cn/

### 1.2 模型的调用

#### **1.**2.1 模型基本调用

##### ****关于对话模型的Message(消息)****

聊天模型，出了将字符串作为输入外，还可以使用 **聊天消息** 作为输入，并返回 **聊天消息** 作为输出。

**LangChain有一些内置的消息类型：**

🔥 **SystemMessage** ：设定AI行为规则或背景信息。比如设定AI的初始状态、行为模式或对话的总体目标。比如“作为一个代码专家”，或者“返回json格式”。通常作为输入消息序列中的第一个传递。

🔥 **HumanMessage** ：表示来自用户输入。比如“实现 一个快速排序方法”

🔥 **AIMessage** ：存储AI回复的内容。这可以是文本，也可以是调用工具的请求

**ChatMessage** ：可以自定义角色的通用消息类型

**FunctionMessage/ToolMessage** ：函数调用/工具消息，用于函数调用结果的消息类型

注意：  
FunctionMessage和ToolMessage分别是在函数调⽤和⼯具调⽤场景下才会使⽤的特殊消息类型，HumanMessage、AIMessage和SystemMessage才是最常⽤的消息类型。

**Messages 案例**

# 创建OpenAI的模型实例，使用代理  
from langchain\_openai import ChatOpenAI  
​  
os.environ["OPENAI\_API\_KEY"] = os.getenv("OPENAI\_KEY")  
os.environ["OPENAI\_BASE\_URL"] = os.getenv("OPENAI\_URL")  
​  
openai = ChatOpenAI(  
    model\_name = "gpt-4o-mini"  
)

# messages demo1  
# 实例化DeepSeek模型  
client= ChatOpenAI(  
    model\_name="deepseek-chat",  # DeepSeek对话模型固定名称  
    temperature=0.7  # 可调整随机性  
)  
​  
# 构造消息列表（与原格式一致，DeepSeek兼容SystemMessage/HumanMessage）  
sys\_message = SystemMessage(  
    content="我是一个人工智能的助手，我的名字叫小智",  
)  
human\_message = HumanMessage(content="猫王是一只猫吗？")  
human\_message1 = HumanMessage(content="你叫什么名字？")  
messages = [sys\_message, human\_message, human\_message1]  
​  
# 调用DeepSeek模型（传入消息列表，注意：单次调用不具备记忆能力，需手动维护对话历史）  
response = client.invoke(messages)  
print(response.content)

# messages demo2  
​  
from langchain\_core.messages import SystemMessage, HumanMessage  
from langchain\_openai import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 配置 DeepSeek 模型  
deepseek = ChatOpenAI(  
    model="deepseek-chat",  
    temperature=0.7,  
    openai\_api\_key=os.getenv("API\_KEY"),  # 建议使用专门的环境变量名  
    openai\_api\_base="https://api.deepseek.com/v1"  
)  
​  
# 构造消息列表  
sys\_message = SystemMessage(  
    content="我是一个人工智能的助手，我的名字叫小智",  
)  
human\_message = HumanMessage(content="猫王是一只猫吗？")  
sys\_message1 = SystemMessage(  
    content="我可以做很多事情，有需要就找我吧",  
)  
human\_message1 = HumanMessage(content="你叫什么名字？")  
messages = [sys\_message, human\_message, sys\_message1, human\_message1]  
​  
# 调用 DeepSeek 模型  
response = deepseek.invoke(messages)  
print(response.content)

# demo3 分成两组messages  
​  
from langchain\_core.messages import SystemMessage, HumanMessage  
from langchain\_openai import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 配置 DeepSeek 模型  
chat = ChatOpenAI(  
   model="deepseek-chat",  
   temperature=0.7,  
   openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
   openai\_api\_base="https://api.deepseek.com/v1"  
)  
​  
# 第一组消息  
sys\_message = SystemMessage(  
   content="我是一个人工智能的助手，我的名字叫小智",  
)  
human\_message = HumanMessage(content="猫王是一只猫吗？")  
message1 = [sys\_message, human\_message]  
​  
# 第二组消息  
sys\_message1 = SystemMessage(  
   content="我可以做很多事情，有需要就找我吧",  
)  
human\_message1 = HumanMessage(content="你叫什么名字？")  
message2 = [sys\_message1, human\_message1]  
​  
# 调用 DeepSeek 模型，传入两组消息  
response1 = chat.invoke(message1)  
print("第一组消息回复:", response1.content)  
​  
response2 = chat.invoke(message2)  
print("第二组消息回复:", response2.content)

# 多轮对话的封装 - DeepSeek版本  
from langchain\_core.messages import SystemMessage, HumanMessage, AIMessage  
from langchain\_openai import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 配置 DeepSeek 模型  
openai = ChatOpenAI(  
    model="deepseek-chat",  
    temperature=0.7,  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base="https://api.deepseek.com/v1"  
)  
​  
# 第一组对话  
messages1 = [  
    SystemMessage(content="你是沃林AI研究院的助手，请根据用户的问题给出回答"),  
    HumanMessage(content="我是学员，我叫卢小联"),  
    AIMessage(content="欢迎你的到来"),  
    HumanMessage(content="我是谁？"),  
]  
​  
# 第二组对话  
messages2 = [  
    SystemMessage(content="我是一个人工智能助手，我的名字叫小智"),  
    HumanMessage(content="很高兴认识你"),  
    AIMessage(content="欢迎你的到来"),  
    HumanMessage(content="我是谁？"),  
]  
​  
# 调用 DeepSeek 模型  
print("=== 第一组对话 ===")  
res1 = openai.invoke(messages1)  
print(res1.content)  
​  
print("\n=== 第二组对话 ===")  
res2 = openai.invoke(messages2)  
print(res2.content)

##### 关于模型调用的方法

为了尽可能简化自定义链的创建，我们实现了一个"Runnable"协议。许多LangChain组件实现了 Runnable 协议，包括聊天模型、提示词模板、输出解析器、检索器、代理(智能体)等。

Runnable 定义的公共的调用方法如下：

* invoke: 处理单条输入，等待LLM完全推理完成后再返回调用结果
* stream: 流式响应，逐字输出LLM的响应结果
* batch: 处理批量输入

在Langchain中，语言模型的输出分为了两种主要的模式：流式输出与非流式输出。

默认是非流式输出，当用户发出请求后，系统在后台等待模型生成完整响应，然后一次性将全部结果返回。

流式输出：一种更具交互感的模型输出方式，用户不再需要等待完整答案，而是能看到模型逐个 token地实时返回内容。

from langchain\_core.messages import HumanMessage  
from langchain\_openai import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
import time  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 配置 DeepSeek 模型  
chat = ChatOpenAI(  
    model='deepseek-chat',  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base="https://api.deepseek.com/v1",  
    streaming=True,  
    temperature=0.7  
)  
​  
messages = [HumanMessage(content="你好，请介绍一下自己")]  
​  
print("开始流式输出 DeepSeek 模型的响应：")  
print("=" \* 50)  
​  
try:  
    chunk\_count = 0  
    for chunk in chat.stream(messages):  
        chunk\_count += 1  
        if hasattr(chunk, 'content') and chunk.content:  
            # 模拟更自然的输出速度  
            time.sleep(0.05)  
            print(chunk.content, end="", flush=True)  
​  
    print(f"\n" + "=" \* 50)  
    print(f"流式输出完成，共接收 {chunk\_count} 个数据块")  
​  
except Exception as e:  
    print(f"\n流式调用过程中出错: {e}")

#### **1.**2.2 Model I/O之Prompt Template

Prompt Template，通过模板管理大模型的输入。

Prompt Template 是LangChain中的一个概念，接收用户输入，返回一个传递给LLM的信息（即提示 词prompt）。

在应用开发中，固定的提示词限制了模型的灵活性和适用范围。所以，prompt template 是一个模板化 的字符串，你可以将 模板化 变量插入到模板中，从而创建出不同的提示。

调用时： **以字典作为输入**，其中每个键代表要填充的提示模板中的变量。

输出一个 PromptValue 。这个 PromptValue 可以传递给 LLM 或 ChatModel，并且还可以转换 为字符串或消息列表。

**有几种不同类型的提示模板：**

PromptTemplate ：LLM提示模板，用于生成字符串提示。它使用 Python 的字符串来模板提示。

* ChatPromptTemplate ：聊天提示模板，用于组合各种角色的消息模板，传入聊天模型。
* XxxMessagePromptTemplate ：消息模板词模板，包括：SystemMessagePromptTemplate、 HumanMessagePromptTemplate、AIMessagePromptTemplate、 ChatMessagePromptTemplate等
* FewShotPromptTemplate ：样本提示词模板，通过示例来教模型如何回答
* PipelinePrompt ：管道提示词模板，用于把几个提示词组合在一起使用。
* 自定义模板：允许基于其它模板类来定制自己的提示词模板

##### ****使用str.format()****

Python的 **str.format()** 方法是一种字符串格式化的手段，允许在 **字符串中插入变量** 。使用这种方法，可以创建包含 **占位符** 的字符串模板，占位符由花括号 **{}** 标识。

调用format()方法时，可以传入一个或多个参数，这些参数将被顺序替换进占位符中。

str.format()提供了灵活的方式来构造字符串，支持多种格式化选项。

在LangChain的默认设置下， **PromptTemplate** 使用 Python 的 **str.format()** 方法进行模板化。这样在模型接收输入前，可以根据需要对数据进行预处理和结构化。

# 使用位置参数  
info = "Name: {0}, Age: {1}".format("Jerry", 25)  
print(info)

# 使用关键字参数  
info = "Name: {name}, Age: {age}".format(name="Tom", age=25)  
print(info)

# 使用字典解包  
person = {"name": "David", "age": 40}  
info = "Name: {name}, Age: {age}".format(\*\*person)  
print(info)

##### ****PromptTemplate 可以在模板中自定义变量****

PromptTemplate类，用于快速构建 \*\*包含变量\*\* 的提示词模板，并通过 \*\*传入不同的参数值\*\* 生成自定义的提示词。  
​  
\*\*主要参数介绍：\*\*  
​  
\*\*template：\*\*定义提示词模板的字符串，其中包含 \*\*文本\*\* 和 \*\*变量占位符（如{name}）\*\* ；  
​  
\*\*input\_variables：\*\* 列表，指定了模板中使用的变量名称，在调用模板时被替换；  
​  
\*\*partial\_variables：\*\*字典，用于定义模板中一些固定的变量名。这些值不需要再每次调用时被替换。  
​  
\*\*函数介绍：\*\*  
​  
\*\*format()\*\*：给input\_variables变量赋值，并返回提示词。利用format() 进行格式化时就一定要赋值，否则会报错。

##### **实例化PromptTemplate的方法**

**方式1：使用构造方法**

from langchain\_core.prompts import PromptTemplate  
​  
# 定义模板：描述主题的应用  
template = PromptTemplate(  
    template="请简要描述{topic}的应用。",  
    input\_variables=["topic"]  
)  
print(template)  
​  
# 使用模板生成提示词  
prompt\_1 = template.format(topic="机器学习")  
prompt\_2 = template.format(topic="自然语言处理")  
​  
print("提示词1:", prompt\_1)  
print("提示词2:", prompt\_2)  
​

# 定义多个变量的模版  
template = PromptTemplate(  
    template="请评价{product}的优缺点，包括{aspect1}和{aspect2}。",  
    input\_variables=["product", "aspect1", "aspect2"]  
)  
#使用模板生成提示词  
prompt\_1 = template.format(  
    product="智能手机",   
    aspect1="电池续航",   
    aspect2="拍照质量"  
)  
prompt\_2 = template.format(  
    product="笔记本电脑",   
    aspect1="处理速度",   
    aspect2="便携性"  
)  
​  
print("提示词1:",prompt\_1)  
print("提示词2:",prompt\_2)

**方式2：调用from\_template()**

prompt\_template = PromptTemplate.from\_template(  
 "请给我一个关于{topic}的{type}解释。"  
)  
#传入模板中的变量名  
prompt = prompt\_template.format(  
    type="详细",   
    topic="量子力学"  
)  
print(prompt)

##### **部分提示词模版**

在生成prompt前就已经提前初始化部分的提示词，实际进一步导入模版的时候只导入除已初始化的变量即可。

# 部分提示词模版  
template2 = PromptTemplate(  
    template = "{foo}{bar}",  
    input\_variables=["foo", "bar"],  
    partial\_variables={"foo": "Hello"}  
)  
​  
print(template2.format(bar="world"))

**使用 PromptTemplate.partial() 方法创建部分提示模板**

template1 = PromptTemplate(  
    template="{foo}{bar}",  
    input\_variables=["foo", "bar"]  
)  
#方式1：  
partial\_template1 = template1.partial(foo="hello")  
prompt1 = partial\_template1.format(bar="world")  
print(prompt1)

# 案例1  
# 完整的模版  
full\_template ="""  
   你是一个{role}, 请用{style}风格回答：  
   问题：{question}  
   答案：  
"""   
# 预填充角色和风格  
partial\_template = PromptTemplate.from\_template(full\_template).partial(  
    role="资深出示",   
    style="专业但幽默"  
)  
​  
# 只需提供剩余变量  
prompt = partial\_template.format(question="如何煎牛排?")  
print(prompt)

##### **format() 与 invoke() 对比**

只要对象是RunnableSerializable接口类型，都可以使用invoke()，替换前面使用format()的调用方式。

* format()，返回值为字符串类型；
* invoke()，返回值为PromptValue类型，接着调用to\_string()返回字 符串

from langchain\_core.prompts import PromptTemplate  
​  
prompt\_template = PromptTemplate.from\_template(  
 "请给我一个关于{topic}的{type}解释。"  
)  
#传入模板中的变量名  
# prompt = prompt\_template.format(  
#     type="详细",   
#     topic="量子力学"  
# )  
prompt = prompt\_template.invoke({"type":"详细", "topic":"量子力学"})  
print(prompt)  
print(type(prompt))

##### ****ChatPromptTemplate**对象**

ChatPromptTemplate是创建 **聊天消息列表** 的提示模板。它比普通 PromptTemplate 更适合处理多角色、多轮次的对话场景。

**特点**：

支持 **System** / **Human** / **AI** 等不同角色的消息模板

对话历史维护

**参数类型：**列表参数格式是tuple类型（ **role** :str **content** :str 组合最常用）

元组的格式为：

(role: str | type, content: str | list[dict] | list[object])

其中 role 是：字符串（如 "system"、"human"、"ai"）

# ChatPromptTemplate  
from langchain\_core.prompts import ChatPromptTemplate  
​  
#参数类型这里使用的是tuple构成的list  
prompt\_template = ChatPromptTemplate(  
   [  
        # 字符串 role + 字符串 content  
       ("system", "你是一个AI开发工程师，你的名字是{name}"),  
       ("human", "你能开发哪些应用?"),  
       ("ai", "我能开发很多AI应用, 比如聊天机器人, 图像识别, 自然语言处理等."),  
       ("human", "{user\_input}")  
   ]  
)  
#调用方法，返回字符串  
prompt = prompt\_template.invoke(  
    input={"name": "沃林AI机器人", "user\_input": "你能帮我做些什么?"}  
)  
​  
print(prompt)

**也可以使用from\_messages()方法**

# 定义聊天提示词模版  
prompt\_template = ChatPromptTemplate.from\_messages(  
   [  
       ("system", "你是一个有帮助的AI机器人，你的名字是{name}。"),  
       ("human", "你好，最近怎么样？"),  
       ("ai", "我很好，谢谢！"),  
       ("human", "{user\_input}"),  
   ]  
)  
​  
# 格式化聊天提示词模版中的变量  
messages = prompt\_template.invoke(  
    input={"name":"小明", "user\_input":"你叫什么名字？"}  
)  
​  
print(messages)

##### ****模板调用的几种方式****

对比： **invoke()** 、 **format()** 、 **format\_messages()** 、 **format\_prompt()**

prompt\_template = ChatPromptTemplate(  
   [  
       ("system", "你是一个AI开发工程师，你的名字是{name}"),  
       ("human", "你能开发哪些应用?"),  
       ("ai", "我能开发很多AI应用, 比如聊天机器人, 图像识别, 自然语言处理等."),  
       ("human", "{user\_input}")  
   ]  
)  
# invoke()，返回值为PromptValue类型，接着调用to\_string()返回字符串。  
prompt = prompt\_template.invoke({"name": "沃林AI机器人", "user\_input": "你能帮我做些什么?"})  
​  
# format()，返回值为字符串类型  
# prompt = prompt\_template.format(name="沃林AI机器人", user\_input="你能帮我做些什么?")  
# format\_messages()，返回值为List[BaseMessage]类型  
# prompt = prompt\_template.format\_messages(name="沃林AI机器人", user\_input="你能帮我做些什么?")  
# format\_prompt()，返回值为PromptValue类型  
# prompt = prompt\_template.format\_prompt(name="沃林AI机器人", user\_input="你能帮我做些什么?")  
​  
print(type(prompt))  
print(prompt)

##### ****更丰富的实例化参数类型**(了解)**

前面讲了ChatPromptTemplate的两种创建方式。我们看到不管使用构造方法，还是使用

from\_messages()，参数类型都是 **列表类型** 。列表中的元素可以是多种类型

参数是列表类型，列表的元素可以是字符串、字典、字符串构成的元组、消息类型、提示词模板类型、消息提示词模板类型等

**类型1：dict类型**

列表参数格式是dict类型

from langchain\_core.prompts import ChatPromptTemplate  
​  
# 示例: 字典形式的消息  
​  
prompt = ChatPromptTemplate.from\_messages([  
   {"role": "system", "content": "你是一个{role}."},  
   {"role": "human", "content": "我的问题是{question}"},  
])  
​  
print(prompt.format\_messages(role="人工智能助手", question="你为什么那么帅"))

**类型2：Message类型**

# Message类型  
from langchain\_core.messages import SystemMessage,HumanMessage  
​  
chat\_prompt\_template = ChatPromptTemplate.from\_messages([  
    SystemMessage(content="我是一个贴心的智能助手"),  
    HumanMessage(content="我的问题是:人工智能英文怎么说？")  
])  
​  
messages = chat\_prompt\_template.format\_messages()  
print(messages)  
print(type(messages))

**类型3：BaseMessagePromptTemplate类型**

LangChain提供不同类型的MessagePromptTemplate。最常用的是

**SystemMessagePromptTemplate** 、 **HumanMessagePromptTemplate** 和

**AIMessagePromptTemplate** ，分别创建系统消息、人工消息和AI消息，它们是ChatMessagePromptTemplate的特定角色子类。

from langchain.prompts import (  
    ChatPromptTemplate,  
    HumanMessagePromptTemplate,  
    SystemMessagePromptTemplate  
)  
​  
template = ChatPromptTemplate.from\_messages(  
   [  
        SystemMessagePromptTemplate.from\_template("你是一个{product}的客服助手, 你的名字叫做{name}"),  
        HumanMessagePromptTemplate.from\_template("{query}")  
   ]  
)  
​  
prompt = template.format(  
    product="阿里AI研究院",   
    name="马云",   
    query="你是谁？"  
)  
print(prompt)  
res = chat.invoke(prompt)  
print(res.content)

##### ****插入消息列表：MessagesPlaceholder****

当你不确定消息提示模板使用什么角色，或者希望在格式化过程中 **插入消息列表** 时，该怎么办？ 这就需要使用 MessagesPlaceholder，负责在特定位置添加消息列表。

**使用场景：**多轮对话系统存储历史消息以及Agent的中间步骤处理此功能非常有用。

# 存储对话历史内容  
from langchain\_core.prompts import ChatPromptTemplate,MessagesPlaceholder  
from langchain\_core.messages import HumanMessage,AIMessage  
from langchain\_openai import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
# 加载环境变量  
load\_dotenv()  
​  
# 配置 DeepSeek 模型  
chat = ChatOpenAI(  
    model='deepseek-chat',  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base=os.getenv("BASE\_URL"),  
    streaming=True,  
    temperature=0.7  
)  
​  
# 存储对话历史内容  
prompt = ChatPromptTemplate.from\_messages([  
   ("system", "你是一个人工智能助手"),  
    MessagesPlaceholder(variable\_name="history"),  
   ("human", "{query}")  
])  
​  
messages = prompt.format\_messages(  
    history=[HumanMessage(content="1+2\*3等于几?"), AIMessage(content="1+2\*3=7")],  
    query="我刚才的问题是什么?"  
)  
response = chat.invoke(messages)  
print(response.content)

##### ****少量样本示例的提示词模板****

在构建prompt时，可以通过构建一个 **少量示例列表** 去进一步格式化prompt，这是一种简单但强大的指导生成的方式，在某些情况下可以 **显著提高模型性能** 。

每个示例的结构都是一个 字典，其中键是输入变量，值是输入变量的值。

# 少量示例模版  
# 使用新的导入路径  
from langchain\_core.prompts import PromptTemplate, FewShotPromptTemplate  
​  
# 1、创建示例集合  
examples = [  
   {"input": "北京天气怎么样", "output": "北京市"},  
   {"input": "南京下雨吗", "output": "南京市"},  
   {"input": "武汉热吗", "output": "武汉市"}  
]  
​  
# 2、创建PromptTemplate实例  
example\_prompt = PromptTemplate.from\_template(  
    template="Input: {input}\nOutput: {output}"  
)  
​  
# 3、创建FewShotPromptTemplate实例  
prompt = FewShotPromptTemplate(  
    examples=examples,  
    example\_prompt=example\_prompt,  
    suffix="Input: {input}\nOutput:",  # 要放在示例后面的提示模板字符串。  
    input\_variables=["input"]  # 传入的变量  
)  
​  
# 4、调用  
result = prompt.invoke({"input": "深圳有多大的面积?"})  
print("===Prompt===")  
print(result)

#1、创建提示模板  
from langchain\_core.prompts import PromptTemplate, FewShotPromptTemplate  
from langchain\_openai import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 初始化 DeepSeek 模型  
chat = ChatOpenAI(  
    model="deepseek-chat",  
    temperature=0.7,  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base="https://api.deepseek.com/v1"  
)  
​  
# 创建提示模板，配置一个提示模板，将一个示例格式化为字符串  
prompt\_template = "你是一个数学专家,算式： {input} 值： {output} 使用： {description} "  
# 这是一个提示模板，用于设置每个示例的格式  
prompt\_sample = PromptTemplate.from\_template(prompt\_template)  
# 2、提供示例  
examples = [  
   {"input": "2+2", "output": "4", "description": "加法运算"},  
   {"input": "5-2", "output": "3", "description": "减法运算"},  
]  
​  
# 3、创建一个FewShotPromptTemplate对象  
prompt = FewShotPromptTemplate(  
    examples=examples,  
    example\_prompt=prompt\_sample,  
    suffix="你是一个数学专家,算式: {input} 值: {output}",  
    input\_variables=["input", "output"]  
)  
print(prompt.invoke({"input": "2\*5", "output": "10"}))  
​  
# 4、大模型调用  
result = chat.invoke(prompt.invoke({"input": "2\*5", "output": "10"}))  
print(result.content)  # 使用: 乘法运算

##### **从文件加载 Prompt 模板**

# 从文件加载模版  
template = PromptTemplate.from\_file("./example\_prompt\_template.txt", encoding="utf-8")  
template.format(topic="黑色幽默")  
print(template.format(topic="黑色幽默"))

创建yaml文件：prompt.yaml

\_type: prompt  
input\_variables: ["name","what"]  
template: "请给{name}讲一个关于{what}的故事"

from langchain\_core.prompts import load\_prompt  
from dotenv import load\_dotenv  
load\_dotenv()  
​  
prompt = load\_prompt("./prompt.yaml", encoding="utf-8")  
# print(prompt)  
print(prompt.format(name="年轻人", what="滑稽"))

创建json文件

{  
    "\_type": "prompt",  
    "input\_variables": ["name", "what"],  
    "template": "请{name}讲一个{what}的故事。"  
}

prompt = load\_prompt("./prompt.json",encoding="utf-8")  
print(prompt.format(name="张三",what="搞笑的"))

### 1.3 结构化输出

语言模型返回的内容通常都是字符串的格式（文本格式），但在实际AI应用开发过程中，往往希望model可以返回**更直观、更格式化的内容**，以确保应用能够顺利进行后续的逻辑处理。此时，LangChain提供的 **输出解析器** 就派上用场了。

**输出解析器的分类**

LangChain有许多不同类型的输出解析器

* **StrOutputParser** ：字符串解析器
* **JsonOutputParser** ：JSON解析器，确保输出符合特定JSON对象格式
* **XMLOutputParser** ：XML解析器，允许以流行的XML格式从LLM获取结果
* **CommaSeparatedListOutputParser** ：CSV解析器，模型的输出以逗号分隔，以列表形式返回输出
* **DatetimeOutputParser** ：日期时间解析器，可用于将 LLM 输出解析为日期时间格式

#### **1.3.1 **字符串解析器 StrOutputParser****

StrOutputParser 简单地将 **任何输入** 转换为 **字符串** 。它是一个简单的解析器，从结果中提取content字段

#### **1.3.2 **JSON解析器 JsonOutputParser****

JsonOutputParser，即JSON输出解析器，是一种用于将大模型的 **自由文本输出** 转换为 **结构化JSON数据** 的工具。

**适合场景：**特别适用于需要严格结构化输出的场景，比如 API 调用、数据存储或下游任务处理。

**实现方式**

* 方式1：用户自己通过提示词指明返回Json格式
* 方式2：借助JsonOutputParser的 **get\_format\_instructions()** ，生成格式说明，指导模型输出

JSON 结构

from langchain\_core.messages import SystemMessage, HumanMessage  
from langchain\_core.output\_parsers import JsonOutputParser  
from langchain\_core.prompts import ChatPromptTemplate  
from langchain\_community.chat\_models import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
# 加载环境变量  
load\_dotenv()  
​  
os.environ["OPENAI\_API\_KEY"] = os.getenv("API\_KEY")  # DeepSeek的API密钥  
os.environ["OPENAI\_BASE\_URL"] = os.getenv("BASE\_URL")  # DeepSeek基础URL  
​  
# 实例化DeepSeek模型（替代原OpenAI实例）  
chat = ChatOpenAI(  
    model\_name="deepseek-chat",  # DeepSeek对话模型固定名称  
    temperature=0.7  # 可调整随机性  
)  
​  
chat\_prompt\_template = ChatPromptTemplate.from\_messages([  
   ("system","你是一个靠谱的{role}"),  
   ("human","{question}")  
])  
parser = JsonOutputParser()  
​  
# 方式1：  
# result = chat.invoke(chat\_prompt\_template.format\_messages(role="人工智能专家",question="人工智能用英文怎么说？问题用q表示，答案用a表示，返回一个JSON格式"))  
# print(result.content)  
# print(type(result))  
​  
# 方式2： 使用LCEL语法组合一个简单的链  
chain = chat\_prompt\_template | chat | parser  
response = chain.invoke({"role":"人工智能专家","question" : "人工智能用英文怎么说？问题用q表示，答案用a表示，返回一个JSON格式"})  
​  
print(response)

# 定义Json解析器  
from langchain\_core.messages import SystemMessage, HumanMessage  
from langchain\_core.output\_parsers import JsonOutputParser  
from langchain\_core.prompts import PromptTemplate  
from langchain\_community.chat\_models import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
# 加载环境变量  
load\_dotenv()  
​  
os.environ["OPENAI\_API\_KEY"] = os.getenv("API\_KEY")  # DeepSeek的API密钥  
os.environ["OPENAI\_BASE\_URL"] = os.getenv("BASE\_URL")  # DeepSeek基础URL  
​  
# 实例化DeepSeek模型（替代原OpenAI实例）  
chat = ChatOpenAI(  
    model\_name="deepseek-chat",  # DeepSeek对话模型固定名称  
    temperature=0.7  # 可调整随机性  
)  
# 定义Json解析器  
parser = JsonOutputParser()  
# 定义提示词模版  
# 注意，提示词模板中需要部分格式化解析器的格式要求format\_instructions  
prompt\_template = PromptTemplate(  
    template="回答用户的查询.\n{format\_instructions}\n{query}\n",  
    input\_variables=["query"],  
    partial\_variables={"format\_instructions": parser.get\_format\_instructions()},  
)  
# 5.使用LCEL语法组合一个简单的链  
chain = prompt\_template | chat | parser  
# 6.执行链  
output = chain.invoke({"query": "给我讲一个笑话"})  
​  
print(output)

​  
# 输出指定的Json格式  
from langchain\_core.messages import SystemMessage, HumanMessage  
from langchain\_core.output\_parsers import JsonOutputParser  
from langchain\_core.prompts import PromptTemplate  
from langchain\_community.chat\_models import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
# 加载环境变量  
load\_dotenv()  
​  
os.environ["OPENAI\_API\_KEY"] = os.getenv("API\_KEY")  # DeepSeek的API密钥  
os.environ["OPENAI\_BASE\_URL"] = os.getenv("BASE\_URL")  # DeepSeek基础URL  
​  
# 实例化DeepSeek模型（替代原OpenAI实例）  
chat = ChatOpenAI(  
    model\_name="deepseek-chat",  # DeepSeek对话模型固定名称  
    temperature=0.7  # 可调整随机性  
)  
# 定义Json解析器  
template = """  
你是一个日期提取助手。请从用户输入中提取日期信息，并以JSON格式返回。  
​  
JSON格式要求：  
{{  
   "year": 年份,  
   "month": 月份,   
   "day": 日期  
}}  
​  
用户输入:  
{query}  
​  
请只返回JSON格式的日期信息，不要包含其他内容。  
"""  
prompt = PromptTemplate(  
    template=template  
)  
query = "2024年十二月23日天气晴..."  
​  
input\_format = prompt.format(query=query)  
output = chat.invoke(input\_format)  
print("Json格式的输出：", output.content)

#### **1.3.3 **列表解析器 CommaSeparatedListOutputParser****

列表解析器：利用此解析器可以将模型的文本响应转换为一个用 **逗号分隔的列表（List[str]）** 。

from langchain\_core.prompts import PromptTemplate  
from langchain\_community.chat\_models import ChatOpenAI  
from langchain.output\_parsers import CommaSeparatedListOutputParser  
from dotenv import load\_dotenv  
import os  
# 加载环境变量  
load\_dotenv()  
​  
os.environ["OPENAI\_API\_KEY"] = os.getenv("API\_KEY")  # DeepSeek的API密钥  
os.environ["OPENAI\_BASE\_URL"] = os.getenv("BASE\_URL")  # DeepSeek基础URL  
​  
# 实例化DeepSeek模型（替代原OpenAI实例）  
chat = ChatOpenAI(  
    model\_name="deepseek-chat",  # DeepSeek对话模型固定名称  
    temperature=0.7  # 可调整随机性  
)  
​  
​  
# 创建解析器  
output\_parser = CommaSeparatedListOutputParser()  
# 创建LangChain提示模板  
chat\_prompt = PromptTemplate.from\_template(  
    "生成5个关于{text}的列表.\n\n{format\_instructions}",  
    partial\_variables={  
        "format\_instructions": output\_parser.get\_format\_instructions()  
})  
# 提示模板与输出解析器传递输出  
# chat\_prompt =  
chat\_prompt.partial(format\_instructions=output\_parser.get\_format\_instructions())  
# 将提示和模型合并以进行调用  
chain = chat\_prompt | chat| output\_parser  
res = chain.invoke({"text": "电影"})  
print(res)  
print(type(res))

#### **1.3.4 **日期解析器 DatetimeOutputParser (了解)****

​  
from langchain\_core.prompts import PromptTemplate,ChatPromptTemplate  
from langchain\_community.chat\_models import ChatOpenAI  
from langchain.output\_parsers import DatetimeOutputParser  
from dotenv import load\_dotenv  
import os  
# 加载环境变量  
load\_dotenv()  
​  
os.environ["OPENAI\_API\_KEY"] = os.getenv("API\_KEY")  # DeepSeek的API密钥  
os.environ["OPENAI\_BASE\_URL"] = os.getenv("BASE\_URL")  # DeepSeek基础URL  
​  
# 实例化DeepSeek模型（替代原OpenAI实例）  
chat = ChatOpenAI(  
    model\_name="deepseek-chat",  # DeepSeek对话模型固定名称  
    temperature=0.7  # 可调整随机性  
)  
​  
​  
​  
chat\_prompt = ChatPromptTemplate.from\_messages([  
   ("system","{format\_instructions}"),  
   ("human", "{request}")  
])  
output\_parser = DatetimeOutputParser()  
​  
# 方式1：  
# model\_request = chat\_prompt.format\_messages(  
# request="中华人民共和国是什么时候成立的",  
# format\_instructions=output\_parser.get\_format\_instructions()  
# )  
# response = chat\_model.invoke(model\_request)  
# result = output\_parser.invoke(response)  
# print(result)  
# print(type(result))  
​  
# 方式2：  
chain = chat\_prompt | chat | output\_parser  
resp = chain.invoke({"request":"中华人民共和国是什么时候成立的",  
    "format\_instructions":output\_parser.get\_format\_instructions()})  
print(resp)  
print(type(resp))

#### **1.3.5 直接输出 Pydantic 对象**

# Langchain的结构化输出  
# 定义你的Pydantic类  
from langchain\_core.prompts import PromptTemplate  
from langchain\_community.chat\_models import ChatOpenAI  
from datetime import date  
from pydantic import BaseModel, Field  
from langchain\_core.output\_parsers import PydanticOutputParser  
from dotenv import load\_dotenv  
import os  
# 加载环境变量  
load\_dotenv()  
​  
os.environ["OPENAI\_API\_KEY"] = os.getenv("API\_KEY")  # DeepSeek的API密钥  
os.environ["OPENAI\_BASE\_URL"] = os.getenv("BASE\_URL")  # DeepSeek基础URL  
​  
# 实例化DeepSeek模型（替代原OpenAI实例）  
chat = ChatOpenAI(  
    model\_name="deepseek-chat",  # DeepSeek对话模型固定名称  
    temperature=0.7  # 可调整随机性  
)  
​  
class MyDate(BaseModel):  
    year: int = Field(description="年")  
    month: int = Field(description="月")  
    day: int = Field(description="日")  
​  
​  
parser = PydanticOutputParser(pydantic\_object=MyDate)  
​  
template = """  
   提取用户输入中的日期  
   用户输入:  
   {query}.  
   Extract the desired information into JSON format from the following passage.  
   {format\_instructions}  
"""  
prompt = PromptTemplate(  
    template=template,  
    partial\_variables={"format\_instructions": parser.get\_format\_instructions()}  
)  
query = "2024年十二月23日天气晴..."  
​  
input\_format = prompt.format(query=query)  
output = chat.invoke(input\_format)  
print("原始输出：", output.content)  
​  
output2 = parser.invoke(output)  
print("结构化输出：", output2)  
​  
# 转换成Python中的日期类型  
res\_date = date(output2.year, output2.month, output2.day)  
print("转换成Python中的日期类型：", res\_date)

#### **1.3.6 OutputFixingParser 利用大模型做格式自动纠错**

from langchain\_community.chat\_models import ChatOpenAI  
from langchain\_core.prompts import ChatPromptTemplate  
from pydantic import BaseModel, Field  
from langchain\_core.output\_parsers import PydanticOutputParser  
from langchain.output\_parsers import OutputFixingParser  
from dotenv import load\_dotenv  
from typing import List, Optional  
import os  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 设置环境变量  
os.environ["OPENAI\_API\_KEY"] = os.getenv("DEEPSEEK\_API\_KEY")  
os.environ["OPENAI\_BASE\_URL"] = os.getenv("BASE\_URL")  
​  
# 实例化DeepSeek模型  
chat = ChatOpenAI(  
    model\_name="deepseek-chat",  
    temperature=0.7  
)  
​  
​  
# 定义简历数据模型  
class Resume(BaseModel):  
    name: str = Field(description="姓名")  
    email: str = Field(description="邮箱")  
    experience\_years: int = Field(description="工作年限")  
    skills: List[str] = Field(description="技能列表")  
    last\_company: Optional[str] = Field(description="上一家公司")  
​  
​  
# 创建解析器  
resume\_parser = PydanticOutputParser(pydantic\_object=Resume)  
fixing\_parser = OutputFixingParser.from\_llm(parser=resume\_parser, llm=chat)  
​  
# 创建提示模板  
prompt\_template = ChatPromptTemplate.from\_messages([  
   ("system", "你是一个简历信息提取助手。请从文本中提取人员信息。\n{format\_instructions}"),  
   ("human", "请从以下文本中提取人员信息：\n{text}")  
])  
​  
​  
def extract\_resume\_info(text):  
    """提取简历信息的完整流程"""  
    try:  
        # 构建提示  
        prompt = prompt\_template.format\_prompt(  
            text=text,  
            format\_instructions=resume\_parser.get\_format\_instructions()  
       )  
​  
        # 调用模型  
        response = chat.invoke(prompt.to\_messages())  
​  
        print(f"模型原始输出: {response.content}")  
​  
        # 使用修复解析器处理输出  
        result = fixing\_parser.parse(response.content)  
        return result  
​  
    except Exception as e:  
        print(f"处理过程中出错: {e}")  
        return None  
​  
​  
# 测试用例  
test\_texts = [  
    "张三，邮箱zhangsan@email.com，有5年工作经验，擅长Python和Java，之前在ABC公司工作",  
    "李四的联系方式是lisi@example.com，工作经验3年，技能包括JavaScript、React，上一家公司是XYZ科技",  
    "王五，工作8年，会项目管理和团队协作，邮箱wangwu@test.com，之前在DEF集团"  
]  
​  
print("简历信息提取测试:\n")  
​  
for i, text in enumerate(test\_texts, 1):  
    print(f"测试 {i}: {text}")  
    result = extract\_resume\_info(text)  
​  
    if result:  
        print("✅ 提取成功:")  
        print(f"   姓名: {result.name}")  
        print(f"   邮箱: {result.email}")  
        print(f"   工作年限: {result.experience\_years}")  
        print(f"   技能: {', '.join(result.skills)}")  
        print(f"   上一家公司: {result.last\_company}")  
    else:  
        print("❌ 提取失败")  
​  
    print("-" \* 60)  
​

**练习**

'''  
   练习：  
   给出一段书籍的概述，从书籍的概述中，提取书名、作者，以及书籍的体裁  
'''  
from langchain\_community.chat\_models import ChatOpenAI  
from langchain\_core.prompts import ChatPromptTemplate  
from pydantic import BaseModel, Field  
from langchain\_core.output\_parsers import PydanticOutputParser  
from langchain.output\_parsers import OutputFixingParser  
from dotenv import load\_dotenv  
from typing import List, Optional  
import os  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 设置环境变量  
os.environ["OPENAI\_API\_KEY"] = os.getenv("DEEPSEEK\_API\_KEY")  
os.environ["OPENAI\_BASE\_URL"] = os.getenv("BASE\_URL")  
​  
# 实例化DeepSeek模型  
chat = ChatOpenAI(  
    model\_name="deepseek-chat",  
    temperature=0.7  
)  
# 定义数据结构  
class BookInfo(BaseModel):  
    book\_name: str = Field(description="书籍的名字")  
    author\_name: str = Field(description="书籍的作者")  
    genres: list[str] = Field(description="书籍的体裁")  
​  
# 初始化解析器  
output\_parser = PydanticOutputParser(pydantic\_object=BookInfo)  
​  
# 定义提示模板  
prompt = ChatPromptTemplate.from\_messages([  
   ("system", "{parser\_instructions} 你输出的结果请使用中文。"),  
   ("human", "请你帮我从书籍的概述中，提取书名、作者，以及书籍的体裁。书籍概述会被三个#符号包围。\n###{book\_introduction}###")  
])  
​  
# 示例书籍介绍  
book\_introduction = """  
《朝花夕拾》原名《旧事重提》，是现代文学家鲁迅的散文集，收录鲁迅于1926年创作的10篇回忆性散文。  
"""  
​  
input\_format = prompt.format\_prompt(  
    book\_introduction=book\_introduction,  
    parser\_instructions=output\_parser.get\_format\_instructions()  
)  
​  
​  
response = chat.invoke(input\_format)  
​  
# 解析输出  
result = output\_parser.invoke(response)  
print(result)

## Section02 **LangChain之Chain**

### ****2.1 Chain的基本概念****

Chain：链，用于将多个组件（提示模板、LLM模型、记忆、工具等）连接起来，形成可复用的 **工作流** ，完成复杂的任务。

**Chain 的核心思想**是通过组合不同的模块化单元，实现比单一组件更强大的功能。比如：

* 将 **LLM** 与 **Prompt Template** （提示模板）结合
* 将 **LLM** 与 **输出解析器** 结合
* 将 **LLM** 与 **外部数据** 结合，例如用于问答
* 将 **LLM** 与 **长期记忆** 结合，例如用于聊天历史记录
* 通过将 **第一个LLM** 的输出作为 **第二个LLM** 的输入，...，将多个LLM按顺序结合在一起

### ****2.2 LCEL 及其基本构成****

使用LCEL，可以构造出结构最简单的Chain。

LangChain表达式语言（LCEL，LangChain Expression Language）是一种声明式方法，可以轻松地将多个组件链接成 AI 工作流。它通过Python原生操作符（如管道符 **|** ）将组件连接成可执行流程，显著简化了AI应用的开发。

**LCEL的基本构成：**提示（Prompt）+ 模型（Model）+ 输出解析器（OutputParser）

# 在这个链条中，用户输入被传递给提示模板，然后提示模板的输出被传递给模型，然后模型的输出被传  
递给输出解析器。  
chain = prompt | model | output\_parser  
chain.invoke({"input":"What's your name?"})

**|** 符号类似于 shell ⾥⾯管道操作符，它将不同的组件链接在⼀起，将前⼀个组件的输出作为下⼀个组件的输⼊，这就形成了⼀个 AI ⼯作流。

**invoke**：所有LCEL对象都实现了 **Runnable** 协议，保证一致的调用方式

### ****2.3 Runnable****

Runnable是LangChain定义的一个抽象接口（Protocol），它 **强制要求** 所有LCEL组件实现一组标准方

法：

class Runnable(Protocol):  
def invoke(self, input: Any) -> Any: ... # 单输入单输出  
def batch(self, inputs: List[Any]) -> List[Any]: ... # 批量处理  
def stream(self, input: Any) -> Iterator[Any]: ... # 流式输出  
# 还有其他方法如 ainvoke（异步）等...

任何实现了这些方法的对象都被视为LCEL兼容组件。比如：聊天模型、提示词模板、输出解析器、检索器、代理(智能体)等。

每个 LCEL 对象都实现了 Runnable 接口，该接口定义了一组公共的调用方法。这使得 LCEL 对象链也自动支持这些调用成为可能。

from langchain\_core.prompts import PromptTemplate  
from langchain\_core.output\_parsers import StrOutputParser  
from langchain\_community.chat\_models import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
# 加载环境变量  
load\_dotenv()  
​  
# 配置 DeepSeek 模型  
chat = ChatOpenAI(  
    model='deepseek-chat',  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base=os.getenv("BASE\_URL"),  
    streaming=True,  
    temperature=0.7  
)  
​  
prompt\_template = PromptTemplate.from\_template(  
    template="请给我讲一个关于{topic}的笑话"  
)  
​  
parser = StrOutputParser()  
​  
# 情况1：没有使用chain  
# prompt\_value = prompt\_template.format(topic="黑色幽默")  
# result = chat.invoke(prompt\_value)  
# out\_put = parser.invoke(result)  
# print(out\_put)  
​  
​  
# 情况2：使用chain  
chain = prompt\_template | chat | parser  
output = chain.invoke({"topic": "黑色幽默"})  
print(output)

**传统Chain的使用 (了解)**

**基础链：LLMChain**

使用说明 LCEL之前，最基础也最常见的链类型是LLMChain。 这个链至少包括一个提示词模板（PromptTemplate），一个语言模型（LLM 或聊天模型）。

注意：LLMChain was deprecated in LangChain 0.1.17 and will be removed in 1.0. Use  
prompt | llm instead。

import warnings  
from langchain\_core.prompts import ChatPromptTemplate  
from langchain\_core.output\_parsers import StrOutputParser  
from langchain\_community.chat\_models import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
​  
# 过滤警告  
# warnings.filterwarnings("ignore", category=UserWarning, module=".\*ChatOpenAI.\*")  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 配置 DeepSeek 模型  
chat = ChatOpenAI(  
    model='deepseek-chat',  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base=os.getenv("BASE\_URL"),  
    streaming=True,  
    temperature=0.7  
)  
​  
# 创建提示词模板  
chat\_prompt\_template = ChatPromptTemplate.from\_messages([  
   ("system","你是一位{area}领域具备丰富经验的高端技术人才"),  
   ("human","给我讲一个 {adjective} 笑话")  
])  
parser = StrOutputParser()  
​  
# 创建链  
chain = chat\_prompt\_template | chat | parser  
​  
# 调用链  
result = chain.invoke({"area": "机器学习", "adjective": "工作中的"})  
print(result)

### ****2.4 顺序链之 SimpleSequentialChain****

顺序链（SequentialChain）允许将多个链顺序连接起来，每个Chain的输出作为下一个Chain的输入，形成特定场景的流水线（Pipeline）。

**顺序链有两种类型：**

单个输入/输出：对应着 SimpleSequentialChain

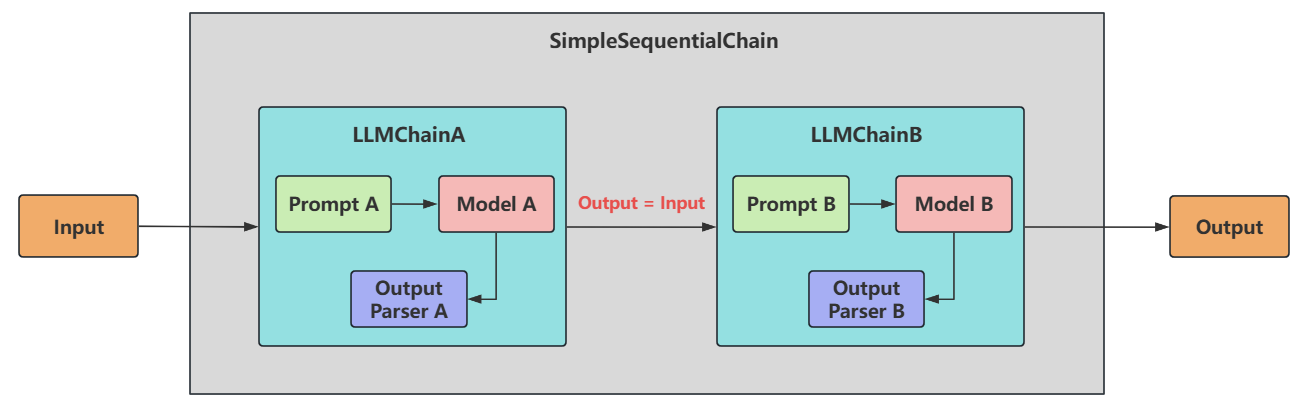
多个输入/输出：对应着：SequentialChain

**2.4.1 说明**

SimpleSequentialChain：最简单的顺序链，多个链 **串联执行** ，每个步骤都有 **单一** 的输入和输出，一个步骤的输出就是下一个步骤的输入，无需手动映射。

创建了两条chain，并且让第一条chain给剧名写大纲，输出该剧名大纲，作为第二条chain的输入，然后生成一个剧本的大纲评论。最后利用SimpleSequentialChain即可将两个chain直接串联起来。

# 1. 导入相关包  
from langchain.chains import LLMChain  
from langchain\_core.prompts import PromptTemplate  
from langchain.chains import SimpleSequentialChain  
from langchain\_community.chat\_models import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
​  
# 过滤警告  
# warnings.filterwarnings("ignore", category=UserWarning, module=".\*ChatOpenAI.\*")  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 配置 DeepSeek 模型  
chat = ChatOpenAI(  
    model='deepseek-chat',  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base=os.getenv("BASE\_URL"),  
    streaming=True,  
    temperature=0.7  
)  
​  
# 2. 创建大模型实例  
​  
# 3. 定义一个给剧名写大纲的LLMChain  
template1 = """你是个剧作家。给定剧本的标题，你的工作就是为这个标题写一个大纲。  
Title:{title}  
"""  
prompt\_template1 = PromptTemplate(  
    template=template1,  
    input\_variables=["title"]  
)  
synopsis\_chain = LLMChain(llm=chat, prompt=prompt\_template1)  
​  
# 4. 定义一个给剧本大纲写一篇评论的LLMChain  
template2 ="""你是<<纽约时报>>的剧作家。有了剧本的大纲，你的工作是为剧本写一篇关于这个评论。  
剧情大纲：  
{synopsis}  
"""  
prompt\_template2 = PromptTemplate(  
    template=template2,  
    input\_variables=["synopsis"]  
)  
review = LLMChain(llm=chat, prompt=prompt\_template2)  
​  
# 5. 定义一个完整的链按顺序运行这条链  
# verbose=True 打印链的执行过程  
overall\_chain = SimpleSequentialChain(chains=[synopsis\_chain, review], verbose=True)  
​  
# 6. 调用完整链顺序执行这两个链  
review = overall\_chain.run("日落海滩上的悲剧")  
​  
print(review)  
​  
​  
​



### **2.5 基于LCEL构建**create\_sql\_query\_chain****

create\_sql\_query\_chain，SQL查询链，是创建生成SQL查询的链，用于将 **自然语言** 转换成 **数据库的**

**SQL查询**

# 连接MySQL数据库  
from langchain\_community.utilities import SQLDatabase  
​  
# 连接 MySQL 数据库  
db\_user = "remote"  
db\_password = "zh&\*DB2021" #根据自己的密码填写  
db\_host = "127.0.0.1"  
db\_port = "3306"  
db\_name = "test\_db"  
# mysql+pymysql://用户名:密码@ip地址:端口号/数据库名  
db = SQLDatabase.from\_uri(f"mysql+pymysql://{db\_user}:{db\_password}@{db\_host}:{db\_port}/{db\_name}")  
print("哪种数据库：", db.dialect)  
print("获取数据表：", db.get\_usable\_table\_names())  
# 执行查询  
res = db.run("SELECT count(\*) FROM conversation\_history;")  
print("查询结果：", res)

# 连接MySQL数据库  
from langchain\_community.utilities import SQLDatabase  
from langchain.chains.sql\_database.query import create\_sql\_query\_chain  
from langchain\_community.chat\_models import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
​  
# 过滤警告  
# warnings.filterwarnings("ignore", category=UserWarning, module=".\*ChatOpenAI.\*")  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 配置 DeepSeek 模型  
chat = ChatOpenAI(  
    model='deepseek-chat',  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base=os.getenv("BASE\_URL"),  
    streaming=True,  
    temperature=0.7  
)  
# 连接 MySQL 数据库  
db\_user = "remote"  
db\_password = "zh&\*DB2021" #根据自己的密码填写  
db\_host = "127.0.0.1"  
db\_port = "3306"  
db\_name = "test\_db"  
# mysql+pymysql://用户名:密码@ip地址:端口号/数据库名  
db = SQLDatabase.from\_uri(f"mysql+pymysql://{db\_user}:{db\_password}@{db\_host}:{db\_port}/{db\_name}")  
print("哪种数据库：", db.dialect)  
print("获取数据表：", db.get\_usable\_table\_names())  
# 执行查询  
res = db.run("SELECT count(\*) FROM conversation\_history;")  
print("查询结果：", res)  
​  
​  
​  
# 调用Chain  
chain = create\_sql\_query\_chain(llm=chat, db=db)  
​  
# response = chain.invoke({"question": "数据表conversation\_history中时间最晚？"})  
# print(response)  
​  
response = chain.invoke({"question": "数据表processing\_logs中处理明细中包含失败的信息有多少条"})  
print(response)

### **2.6 create\_stuff\_documents\_chain(了解)**

create\_stuff\_documents\_chain用于将 多个文档内容 合并成 单个长文本 的链式工具，并一次性传递给 LLM处理（而不是分多次处理）。

适合场景： 保持上下文完整，适合需要全局理解所有文档内容的任务（如总结、问答） 适合处理 少量/中等长度文档 的场景。

from langchain.chains.combine\_documents import create\_stuff\_documents\_chain  
from langchain\_core.prompts import PromptTemplate  
from langchain\_core.documents import Document  
from langchain\_community.chat\_models import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
​  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 配置 DeepSeek 模型  
chat = ChatOpenAI(  
    model='deepseek-chat',  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base=os.getenv("BASE\_URL"),  
    streaming=True,  
    temperature=0.7  
)  
# 定义提示词模板  
prompt = PromptTemplate.from\_template("""  
如下文档{docs}中说，香蕉是什么颜色的？  
""")  
# 创建链  
chain = create\_stuff\_documents\_chain(chat, prompt, document\_variable\_name="docs")  
# 文档输入  
docs = [  
    Document(  
        page\_content="苹果，学名Malus pumila Mill.，别称西洋苹果、柰，属于蔷薇科苹果属的植物。苹果是全球最广泛种植和销售的水果之一，具有悠久的栽培历史和广泛的分布范围。苹果的原始种群主要起源于中亚的天山山脉附近，尤其是现代哈萨克斯坦的阿拉木图地区，提供了所有现代苹果品种的基因库。苹果通过早期的贸易路线，如丝绸之路，从中亚向外扩散到全球各地。"  
   ),  
    Document(  
        page\_content="香蕉是白色的水果，主要产自热带地区。"  
   ),  
    Document(  
        page\_content="蓝莓是蓝色的浆果，含有抗氧化物质。"  
   )  
]  
# 执行摘要  
result = chain.invoke({"docs": docs})  
print(result)

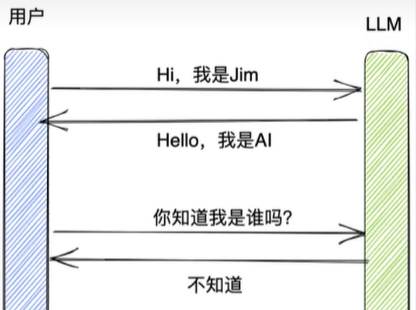
## **Section03 Memory**

### **3.1 **Memory概述****

大多数的大模型应用程序都会有一个会话接口，允许我们进行多轮的对话，并有一定的上下文记忆能

力。

但实际上，模型本身是 **不会记忆** 任何上下文的，只能依靠用户本身的输入去产生输出。



**如何实现记忆功能呢？**

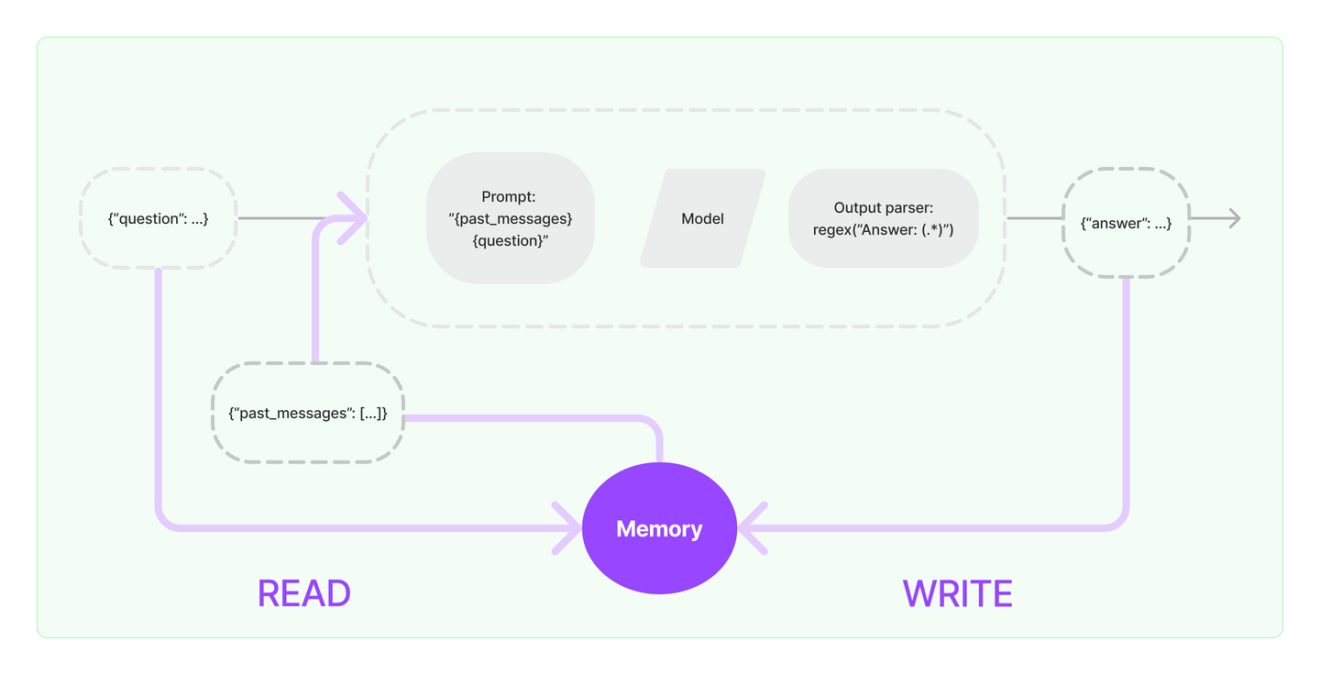
实现这个记忆功能，就需要 **额外的模块** 去保存我们和模型对话的上下文信息，然后在下一次请求时，把所有的历史信息都输入给模型，让模型输出最终结果。

而在 LangChain 中，提供这个功能的模块就称为 **Memory(记忆)** ，用于存储用户和模型交互的历史信息。

### **3.2 **什么是Memory****

**Memory，是LangChain中用于多轮对话中保存和管理上下文信息（比如文本、图像、音频等）的组件**。它让应用能够记住用户之前说了什么，从而实现对话的 **上下文感知能力** ，为构建真正智能和上下文感知的链式对话系统提供了基础。

**Memory的设计理念**



1. 输入问题：({"question": ...})  
2. 读取历史消息：从Memory中READ历史消息（{"past\_messages": [...]}）  
3. 构建提示（Prompt)：读取到的历史消息和当前问题会被合并，构建一个新的Prompt  
4. 模型处理：构建好的提示会被传递给语言模型进行处理。语言模型根据提示生成一个输出。  
5. 解析输出：输出解析器通过正则表达式 regex("Answer: (.\*)")来解析，返回一个回答（{"answer":  
...}）给用户  
6. 得到回复并写入Memory：新生成的回答会与当前的问题一起写入Memory，更新对话历史。  
Memory会存储最新的对话内容，为后续的对话提供上下文支持。

**问题：**一个链如果接入了 **Memory** 模块，其会与Memory模块交互几次呢？

链内部会与 **Memory** 模块进行两次交互：读取和写入：

1、收到用户输入时，从记忆组件中查询相关历史信息，拼接历史信息和用户的输入到提示词中传给LLM。

2、返回响应之前，自动把LLM返回的内容写入到记忆组件，用于下次查询。

**不使用Memory模块，如何拥有记忆？**

不借助LangChain情况下，我们如何实现大模型的记忆能力？

思考：通过 **messages** 变量，不断地将历史的对话信息追加到对话列表中，以此让大模型具备上下文记忆能力。

from langchain.prompts import ChatPromptTemplate  
from langchain\_core.messages import AIMessage, HumanMessage  
from langchain\_community.chat\_models import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 配置 DeepSeek 模型  
chat = ChatOpenAI(  
    model='deepseek-chat',  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base=os.getenv("BASE\_URL"),  
    streaming=True,  
    temperature=0.7  
)  
​  
​  
def chat\_with\_model(question):  
    # 初始化消息列表  
    messages = [  
       ("system", "你是一个助手，请根据用户问题给出回答"),  
       ("human", question)  
   ]  
​  
    # 创建初始提示模板  
    chat\_prompt\_template = ChatPromptTemplate.from\_messages(messages)  
​  
    while True:  
        # 调用模型  
        chain = chat\_prompt\_template | chat  
        response = chain.invoke({"question": question})  
        print(f'模型的回答：{response}')  
​  
        # 询问用户是否还有其它问题  
        user\_input = input("是否还有其它问题？(输入'退出'结束对话): ")  
​  
        # 设置退出条件  
        if user\_input == '退出':  
            break  
​  
        # 更新问题并重新创建提示模板  
        question = user\_input  
        messages.append(("ai", response))  
        messages.append(("human", user\_input))  
        chat\_prompt\_template = ChatPromptTemplate.from\_messages(messages)  
​  
​  
chat\_with\_model("你好！")

###

### **3.3基础Memory模块**

#### 3.3.1 Memory模块的设计思路

**如何设计Memory模块？**

层次1(最直接的方式)：保留一个聊天消息列表

层次2(简单的新思路)：只返回最近交互的k条消息

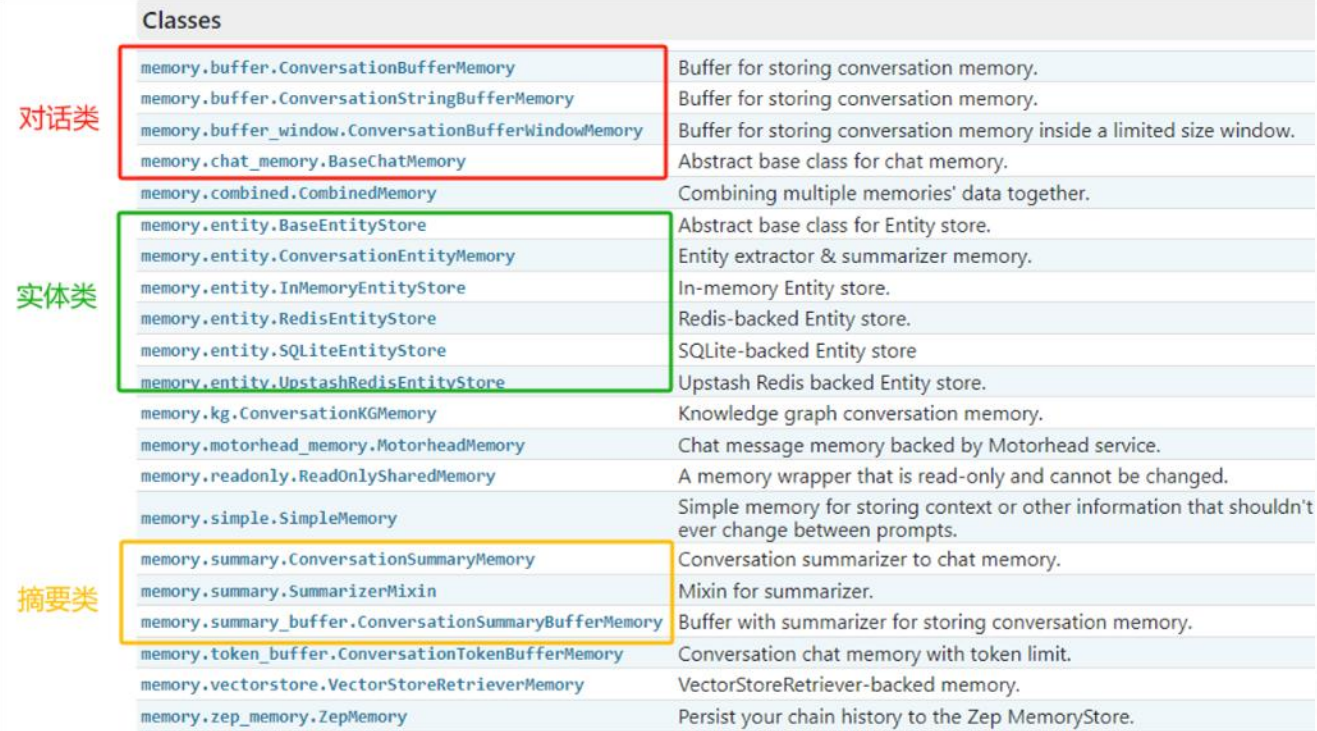
层次3(稍微复杂一点)：返回过去k条消息的简洁摘要

层次4(更复杂)：从存储的消息中提取实体，并且仅返回有关当前运行中引用的实体的信息

**LangChain的设计：**

针对上述情况，LangChain构建了一些可以直接使用的 **Memory** 工具，用于存储聊天消息的一系列集

成。



#### 3.3.2 ChatMessageHistory(基础)

ChatMessageHistory是一个用于 **存储和管理对话消息** 的基础类，它直接操作消息对象（如HumanMessage, AIMessage 等），是其它记忆组件的底层存储工具。

在API文档中，ChatMessageHistory 还有一个别名类：InMemoryChatMessageHistory；导包时，需

使用：from langchain.memory import ChatMessageHistory

**特点**：

纯粹是消息对象的“ **存储器** ”，与记忆策略（如缓冲、窗口、摘要等）无关。

不涉及消息的格式化（如转成文本字符串）

ChatMessageHistory是用于管理和存储对话历史的具体实现。

#1.导入相关包  
from langchain.memory import ChatMessageHistory  
#2.实例化ChatMessageHistory对象  
history = ChatMessageHistory()  
# 3.添加UserMessage  
history.add\_user\_message("hi!")  
# 4.添加AIMessage  
history.add\_ai\_message("whats up?")  
# 5.返回存储的所有消息列表  
print(history.messages)

from langchain.memory import ChatMessageHistory  
from langchain\_community.chat\_models import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 配置 DeepSeek 模型  
llm = ChatOpenAI(  
    model='deepseek-chat',  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base=os.getenv("BASE\_URL"),  
    streaming=True,  
    temperature=0.7  
)  
​  
​  
def interactive\_chat():  
    # 创建对话历史  
    history = ChatMessageHistory()  
​  
    # 系统初始消息  
    history.add\_ai\_message("我是一个无所不能的小智，很高兴为您服务！")  
​  
    print("开始对话（输入'退出'结束对话）")  
    print("=" \* 50)  
​  
    while True:  
        # 显示当前对话历史  
        print("\n当前对话历史:")  
        for i, message in enumerate(history.messages):  
            role = "AI" if message.type == "ai" else "Human"  
            print(f"{i + 1}. {role}: {message.content}")  
​  
        # 获取用户输入  
        user\_input = input("\n请输入您的问题: ")  
​  
        if user\_input.lower() in ['退出', 'exit', 'quit']:  
            print("对话结束！")  
            break  
​  
        # 添加用户消息到历史  
        history.add\_user\_message(user\_input)  
​  
        # 调用大模型  
        print("\nAI 正在思考...")  
        response = llm.invoke(history.messages)  
​  
        # 添加AI回复到历史  
        history.add\_ai\_message(response.content)  
​  
        print(f"\nAI 回复: {response.content}")  
​  
​  
# 启动交互式对话  
interactive\_chat()

#### 3.3.3 ConversationBufferMemory

ConversationBufferMemory是一个基础的 **对话记忆（Memory）组件** ，专门用于按 **原始顺序存储** 完整的对话历史。

**适用场景**：对话轮次较少、依赖完整上下文的场景（如简单的聊天机器）

**特点**：

* 完整存储对话历史
* 简单 、 无裁剪 、 无压缩
* 与 Chains/Models 无缝集成
* 支持两种返回格式（通过 **return\_messages** 参数控制输出格式）

return\_messages=True 返回消息对象列表（ **List[BaseMessage]**

return\_messages=False （默认）返回拼接的 **纯文本字符串**

# 1.导入相关包  
from langchain.memory import ConversationBufferMemory  
# 2.实例化ConversationBufferMemory对象  
memory = ConversationBufferMemory()  
# 3.保存消息到内存中  
memory.save\_context(inputs = {"input": "你好，我是人类"}, outputs = {"output": "你好，我是AI助手"})  
memory.save\_context(inputs = {"input": "很开心认识你"}, outputs = {"output": "我也是"})  
​  
# 4.读取内存中消息（返回消息内容的纯文本）  
print(memory.load\_memory\_variables({}))

注意：

不管inputs、outputs的key用什么名字，都认为inputs的key是human，outputs的key是AI。

打印的结果的json数据的key，默认是“history”。可以通过ConversationBufferMemory的**memory\_key** 属性修改。

# 1.导入相关包  
from langchain.memory import ConversationBufferMemory  
# 2.实例化ConversationBufferMemory对象  
memory = ConversationBufferMemory(return\_messages=True)  
# 3.保存消息到内存中  
memory.save\_context(inputs = {"input": "你好，我是人类"}, outputs = {"output": "你好，我是AI助手"})  
memory.save\_context(inputs = {"input": "很开心认识你"}, outputs = {"output": "我也是"})  
​  
# 4.读取内存中消息（返回消息）  
print(memory.load\_memory\_variables({}))  
# 5.读取内存中消息( 访问原始消息列表)  
print(memory.chat\_memory.messages)

**举例1，PromptTemplate结合Chain**

​  
from langchain.chains.llm import LLMChain  
from langchain.memory import ConversationBufferMemory  
from langchain\_core.prompts import PromptTemplate  
from langchain\_community.chat\_models import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 配置 DeepSeek 模型  
chat = ChatOpenAI(  
    model='deepseek-chat',  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base=os.getenv("BASE\_URL"),  
    streaming=True,  
    temperature=0.7  
)  
# 结合大模型调用  
​  
# 创建提示  
# 有两个输入键：实际输入与来自记忆类的输入 需确保PromptTemplate和ConversationBufferMemory中的键匹配  
template = """你可以与人类对话  
当前对话:{history}  
人类问题:{question}  
回答：  
"""  
​  
memory = ConversationBufferMemory() # 创建ConversationBufferMemory  
prompt = PromptTemplate.from\_template(template)  
​  
# 初始化链  
chain = LLMChain(  
    llm = chat,  
    prompt = prompt,  
    memory = memory  
)  
​  
# 提问  
res = chain.invoke({"question": "你好, 我的名字叫wolin"})  
# 第一次回答  
print(res)  
​  
# 继续提问  
res = chain.invoke({"question": "我的名字叫什么?"})  
print(res)  
​

**举例2：可以通过memory\_key修改memory数据的变量名**

​  
from langchain.chains.llm import LLMChain  
from langchain.memory import ConversationBufferMemory  
from langchain\_core.prompts import PromptTemplate  
from langchain\_community.chat\_models import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 配置 DeepSeek 模型  
chat = ChatOpenAI(  
    model='deepseek-chat',  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base=os.getenv("BASE\_URL"),  
    streaming=True,  
    temperature=0.7  
)  
# 结合大模型调用  
​  
# 创建提示  
# 有两个输入键：实际输入与来自记忆类的输入 需确保PromptTemplate和ConversationBufferMemory中的键匹配  
template = """你可以与人类对话  
当前对话:{chat\_history}  
人类问题:{question}  
回答：  
"""  
​  
memory = ConversationBufferMemory(memory\_key="chat\_history") # 创建ConversationBufferMemory  
prompt = PromptTemplate.from\_template(template)  
​  
# 初始化链  
chain = LLMChain(  
    llm = chat,  
    prompt = prompt,  
    memory = memory  
)  
​  
# 提问  
res = chain.invoke({"question": "你好, 我的名字叫Luke"})  
# 第一次回答  
print(res)  
​  
# 继续提问  
res = chain.invoke({"question": "我的名字叫什么?"})  
print(res)  
​  
​

说明：创建带Memory功能的Chain，并不能使用统一的LCEL语法。同样地， **LLMChain** 也不能使用管道运算符接 **StrOutputParser** 。这些设计上的问题，个人推测也是目前Memory模块还是Beta版本的原因之一吧。

**举例3：使用ChatPromptTemplate 和 return\_messages**

from langchain.chains.llm import LLMChain  
from langchain.memory import ConversationBufferMemory  
from langchain\_core.prompts import MessagesPlaceholder  
from langchain.prompts import ChatPromptTemplate  
from langchain\_community.chat\_models import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 配置 DeepSeek 模型  
chat = ChatOpenAI(  
    model='deepseek-chat',  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base=os.getenv("BASE\_URL"),  
    streaming=True,  
    temperature=0.7  
)  
# 创建提示词  
prompt = ChatPromptTemplate.from\_messages([  
   ("system", "你是一个与人类对话的机器人"),  
    MessagesPlaceholder(variable\_name="history"),  
   ("human", "问题：{question}")  
])  
​  
# 创建memory  
memory = ConversationBufferMemory(return\_messages=True)  
​  
# 创建调用链  
chain = LLMChain(  
    llm = chat,  
    prompt = prompt,  
    memory = memory  
)  
​  
# 调用链: 第一次调用，发现history中已经有了历史记录  
response1 = chain.invoke({"question": "中国的首都在哪里?"})  
print(response1)  
​  
# 再次调用  
response2 = chain.invoke({  
    "question": "我刚才的问题是什么?"  
})  
print(response2)  
​  
​

**二者对比**



**注意**：

我们观察到的现象不是 bug，而是 LangChain 为 **保障对话一致性** 所做的刻意设计：

1. 用户提问后，系统应立即"记住"该问题  
2. AI回答后，该响应应即刻加入对话上下文  
3. 返回给客户端的结果应反映最新状态

#### 3.4.4 ConversationChain

ConversationChain实际上是就是对 ConversationBufferMemory 和 LLMChain 进行了封装，并且提供 一个默认格式的提示词模版（我们也可以不用），从而简化了初始化ConversationBufferMemory的步骤。

# 举例1：使用PromptTemplate  
​  
from langchain.chains.conversation.base import ConversationChain  
from langchain.prompts import PromptTemplate  
from langchain\_community.chat\_models import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 配置 DeepSeek 模型  
chat = ChatOpenAI(  
    model='deepseek-chat',  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base=os.getenv("BASE\_URL"),  
    streaming=True,  
    temperature=0.7  
)  
# 结合大模型调用  
​  
# 创建提示  
# 有两个输入键：实际输入与来自记忆类的输入 需确保PromptTemplate和ConversationBufferMemory中的键匹配  
template = """你可以与人类对话  
当前对话:{history}  
人类问题:{input}  
回答：  
"""  
prompt = PromptTemplate.from\_template(template)  
chain = ConversationChain(llm=chat, prompt=prompt)  
​  
​  
# 提问  
res = chain.invoke({"input": "你好, 我的名字叫Luke"}) #注意，chain中的key必须是input，否则会报错  
# 第一次回答  
print(res)  
​  
​  
# 继续提问  
res = chain.invoke({"input": "我的名字叫什么?"})  
print(res)

#### 3.4.5 ConversationBufferWindowMemory

在了解了ConversationBufferMemory记忆类后，我们知道了它能够无限的将历史对话信息填充到 History中，从而给大模型提供上下文的背景。但这会 导致内存量十分大 ，并且消耗的token是非常多 的，此外，每个大模型都存在最大输入的Token限制。

我们发现，过久远的对话数据往往并不能对当前轮次的问答提供有效的信息，LangChain 给出的解决方 式是： ConversationBufferWindowMemory 模块。该记忆类会 保存一段时间内对话交互 的列表， 仅使用 最近 K 个交互 。这样就使缓存区不会变得太大。

特点：

* 适合长对话场景。
* 与 Chains/Models 无缝集成
* 支持两种返回格式（通过 return\_messages 参数控制输出格式）

return\_messages=True 返回消息对象列表（ List[BaseMessage]

return\_messages=False （默认） 返回拼接的 纯文本字符串

场景1：入门使用 通过内置在LangChain中的缓存窗口(BufferWindow)可以将meomory"记忆"下来。

# 1.导入相关包  
from langchain.memory import ConversationBufferWindowMemory  
​  
# 2.实例化ConversationBufferWindowMemory对象，设定窗口阈值  
memory = ConversationBufferWindowMemory(k=1)  
# 3.保存消息  
memory.save\_context({"input": "你好"}, {"output": "怎么了"})  
memory.save\_context({"input": "你是谁"}, {"output": "我是AI助手"})  
memory.save\_context({"input": "你的生日是哪天？"}, {"output": "我不清楚"})  
​  
# 4.读取内存中消息（返回消息内容的纯文本）  
print(memory.load\_memory\_variables({}))  
​

举例2： ConversationBufferWindowMemory 也支持使用聊天模型（Chat Model）的情况，同样可以通过 return\_messages=True 参数，将对话转化为消息列表形式。

# 1.导入相关包  
from langchain.memory import ConversationBufferWindowMemory  
​  
# 2.实例化ConversationBufferWindowMemory对象，设定窗口阈值  
memory = ConversationBufferWindowMemory(k=2, return\_messages=True)  
​  
# 3.保存消息  
memory.save\_context({"input": "你好"}, {"output": "怎么了"})  
memory.save\_context({"input": "你是谁"}, {"output": "我是AI助手小智"})  
memory.save\_context({"input": "初次对话，你能介绍一下你自己吗？"}, {"output": "当然可以了。我是一个无所不能的小智。"})  
​  
# 4.读取内存中消息（返回消息内容的纯文本）  
print(memory.load\_memory\_variables({}))

场景2：结合chain 借助提示词模版去构建LangChain

# 1.导入相关包  
from langchain.memory import ConversationBufferWindowMemory  
from langchain\_core.prompts.prompt import PromptTemplate  
from langchain.chains.llm import LLMChain  
from langchain\_community.chat\_models import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 配置 DeepSeek 模型  
llm = ChatOpenAI(  
    model='deepseek-chat',  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base=os.getenv("BASE\_URL"),  
    streaming=True,  
    temperature=0.7  
)  
# 2.定义模版  
template = """以下是人类与AI之间的友好对话描述。AI表现得很健谈，并提供了大量来自其上下文的  
具体细节。如果AI不知道问题的答案，它会表示不知道。  
当前对话：  
{history}  
Human: {question}  
AI:"""  
# 3.定义提示词模版  
prompt\_template = PromptTemplate.from\_template(template)  
​  
# 5.实例化ConversationBufferWindowMemory对象，设定窗口阈值  
memory = ConversationBufferWindowMemory(k=3)  
# 6.定义LLMChain  
conversation\_with\_summary = LLMChain(  
    llm=llm,  
    prompt=prompt\_template,  
    memory=memory  
)  
# 7.执行链（第一次提问）  
respon1 = conversation\_with\_summary.invoke({"question":"你好，我是孙小空"})  
print(respon1)  
 # 8.执行链（第二次提问）  
respon2 =conversation\_with\_summary.invoke({"question":"我还有两个师弟，一个是猪小戒，一个是沙小僧"})  
print(respon2)  
# 9.执行链（第三次提问）  
respon3 =conversation\_with\_summary.invoke({"question":"我今年高考，竟然考上了1本"})  
print(respon3)  
# 10.执行链（第四次提问）  
respon4 =conversation\_with\_summary.invoke({"question":"我叫什么？"})  
print(respon4)

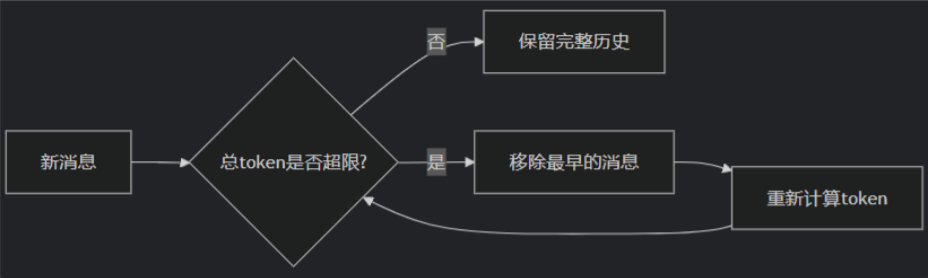
### **3.4 其他Memory模块**

#### **3.4.1 ConversationTokenBufferMemory**

ConversationTokenBufferMemory 是 LangChain 中一种基于 Token 数量控制 的对话记忆机制。如果 字符数量超出指定数目，它会切掉这个对话的早期部分，以保留与最近的交流相对应的字符数量。

特点：

* Token 精准控制
* 原始对话保留



# 1.导入相关包  
from langchain.memory import ConversationTokenBufferMemory  
​  
# 3.定义ConversationTokenBufferMemory对象  
memory = ConversationTokenBufferMemory(  
    llm=openai,  
    max\_token\_limit=30 # 设置token上限  
)  
# 添加对话  
memory.save\_context({"input": "你好吗？"}, {"output": "我很好，谢谢！"})  
memory.save\_context({"input": "今天天气如何？"}, {"output": "晴天，25度"})  
# 查看当前记忆  
print(memory.load\_memory\_variables({}))

#### **3.4.2 ConversationSummaryMemory**

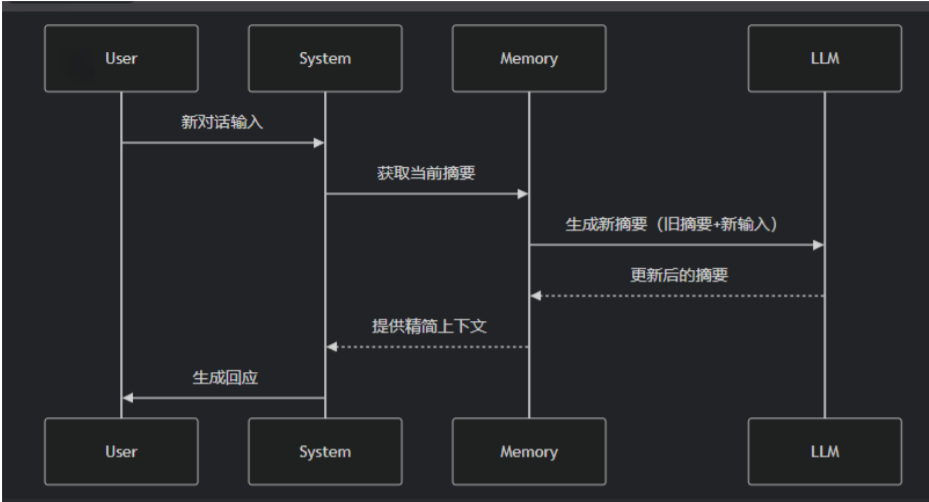
前⾯的⽅式发现，如果全部保存下来太过浪费，截断时⽆论是按照对话条数 还是 token 都是⽆法保证既节省内存⼜保证对话质量的，所以推出ConversationSummaryMemory、  
ConversationSummaryBufferMemory

ConversationSummaryMemory是 LangChain 中一种 智能压缩对话历史 的记忆机制，它通过大语言模 型(LLM)自动生成对话内容的 精简摘要 ，而不是存储原始对话文本。

这种记忆方式特别适合长对话和需要保留核心信息的场景。

特点：

* 摘要生成
* 动态更新 上下文优化



场景1： 如果实例化ConversationSummaryMemory前，没有历史消息，可以使用构造方法实例化

# 1.导入相关包  
from langchain.memory import ConversationSummaryMemory, ChatMessageHistory  
from langchain\_community.chat\_models import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 配置 DeepSeek 模型  
llm = ChatOpenAI(  
    model='deepseek-chat',  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base=os.getenv("BASE\_URL"),  
    streaming=True,  
    temperature=0.7  
)  
# 2.定义ConversationSummaryMemory对象  
memory = ConversationSummaryMemory(llm=llm)  
​  
# 3.存储消息  
memory.save\_context({"input": "你好"}, {"output": "怎么了"})  
memory.save\_context({"input": "你是谁"}, {"output": "我是AI助手小智"})  
memory.save\_context({"input": "初次对话，你能介绍一下你自己吗？"}, {"output": "当然可以了。我是一个无所不能的小智。"})  
​  
# 4.读取消息（总结后的）  
print(memory.load\_memory\_variables({}))  
​

场景2： 如果实例化ConversationSummaryMemory前，已经有历史消息，可以调用from\_messages()实例化

​  
# 1.导入相关包  
from langchain.memory import ConversationSummaryMemory, ChatMessageHistory  
from langchain\_community.chat\_models import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 配置 DeepSeek 模型  
llm = ChatOpenAI(  
    model='deepseek-chat',  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base=os.getenv("BASE\_URL"),  
    streaming=True,  
    temperature=0.7  
)  
# 2.假设原始消息  
history = ChatMessageHistory()  
history.add\_user\_message("你好，你是谁？")  
history.add\_ai\_message("我是AI助手小智")  
​  
# 3.初始化ConversationSummaryMemory实例  
memory = ConversationSummaryMemory.from\_messages(  
    llm=llm,  
    # 是生成摘要的原材料 保留完整对话供必要时回溯。当新增对话时，LLM需要结合原始历史生成新摘要  
    chat\_memory=history  
)  
​  
# 4.查看当前记忆中包括了对历史的摘要  
print(memory.load\_memory\_variables({}))  
​  
# 5. 增加新的信息  
memory.save\_context({"input": "我的名字叫做小明"}, {"output": "很高兴认识你"})  
print(memory.load\_memory\_variables({}))  
​  
print(memory.chat\_memory.messages)  
​

#### **3.4.3 ConversationSummaryBufferMemory**

ConversationSummaryBufferMemory 是 LangChain 中一种混合型记忆机制，它结合了 ConversationBufferMemory（完整对话记录）和 ConversationSummaryMemory（摘要记忆）的优点，在保留最近对话原始记录的同时，对较早的对话内容进行智能摘要 。

特点：

* 保留最近N条原始对话：确保最新交互的完整上下文
* 摘要较早历史：对超出缓冲区的旧对话进行压缩，避免信息过载
* 平衡细节与效率：既不会丢失关键细节，又能处理长对话

场景1：入门使用 情况1：构造方法实例化，并设置max\_token\_limit

# 1.导入相关的包  
from langchain.memory import ConversationSummaryBufferMemory  
​  
# 2.定义模型  
​  
​  
# 3.定义ConversationSummaryBufferMemory对象  
memory = ConversationSummaryBufferMemory(  
    llm=openai,   
    max\_token\_limit=40, # 控制缓冲区的大小  
    return\_messages=True  
)  
​  
# 4.保存消息  
memory.save\_context({"input": "你好，我的名字叫小明"}, {"output": "很高兴认识你，小明"})  
memory.save\_context({"input": "李白是哪个朝代的诗人"}, {"output": "李白是唐朝诗人"})  
memory.save\_context({"input": "唐宋八大家里有苏轼吗？"}, {"output": "有"})  
​  
# 5.读取内容  
print(memory.load\_memory\_variables({}))  
​  
# 6. 把缓冲区的大小修改成100 ，对比结果  
# print(memory.chat\_memory.messages)

场景2：客服

from langchain.memory import ConversationSummaryBufferMemory  
from langchain\_openai import ChatOpenAI  
from langchain\_core.prompts import ChatPromptTemplate, MessagesPlaceholder  
from langchain.chains.llm import LLMChain  
# 1、初始化大语言模型  
llm = ChatOpenAI(  
    model="gpt-4o-mini",  
    temperature=0.5,  
    max\_tokens=500  
)  
​  
# 2、定义提示模板  
prompt = ChatPromptTemplate.from\_messages([  
   ("system", "你是电商客服助手，用中文友好回复用户问题。保持专业但亲切的语气。"),  
    MessagesPlaceholder(variable\_name="chat\_history"),  
   ("human", "{input}")  
])  
# 3、创建带摘要缓冲的记忆系统  
memory = ConversationSummaryBufferMemory(  
    llm=llm,  
    max\_token\_limit=400,  
    memory\_key="chat\_history",  
    return\_messages=True  
)  
# 4、创建对话链  
chain = LLMChain(  
    llm=llm,  
    prompt=prompt,  
    memory=memory,  
)  
# 5、模拟多轮对话  
dialogue = [  
   ("你好，我想查询订单12345的状态", None),  
   ("这个订单是上周五下的", None),  
   ("我现在急着用，能加急处理吗", None),  
   ("等等，我可能记错订单号了，应该是12346", None),  
   ("对了，你们退货政策是怎样的", None)  
]  
# 6、执行对话  
for user\_input, \_ in dialogue:  
    response = chain.invoke({"input": user\_input})  
    print(f"用户: {user\_input}")  
    print(f"客服: {response['text']}\n")  
      
# 7、查看当前记忆状态  
print("\n=== 当前记忆内容 ===")  
print(memory.load\_memory\_variables({}))

#### **3.4.4 ConversationEntityMemory**

ConversationEntityMemory 是一种基于实体的对话记忆机制，它能够智能地识别、存储和利用对话中出现的实体信息（如人名、地点、产品等）及其属性/关系，并结构化存储，使 AI 具备更强的上下文理 解和记忆能力。

好处：解决信息过载问题

* 长对话中大量冗余信息会干扰关键事实记忆
* 通过对实体摘要，可以压缩非重要细节（如删除寒暄等，保留价格/时间等硬性事实）

应用场景：在医疗等高风险领域，必须用实体记忆确保关键信息（如过敏史）被100%准确识别和拦 截。

{"input": "我头痛，血压140/90，在吃阿司匹林。"},  
{"output": "建议监测血压，阿司匹林可继续服用。"}  
{"input": "我对青霉素过敏。"},  
{"output": "已记录您的青霉素过敏史。"}  
{"input": "阿司匹林吃了三天，头痛没缓解。"},  
{"output": "建议停用阿司匹林，换布洛芬试试。"}

使用ConversationSummaryMemory

"患者主诉头痛和⾼⾎压（140/90），正在服⽤阿司匹林。患者对⻘霉素过敏。三天后头痛未缓解，建议更换⽌痛药。"

使用ConversationEntityMemory

{  
"症状": "头痛",  
"⾎压": "140/90",  
"当前⽤药": "阿司匹林（⽆效）",  
"过敏药物": "⻘霉素"  
}

对比：ConversationSummaryMemory 和 ConversationEntityMemory



​  
from langchain.chains import LLMChain  
from langchain.memory import ConversationEntityMemory  
from langchain.memory.prompt import ENTITY\_MEMORY\_CONVERSATION\_TEMPLATE  
# 初始化大语言模型  
from langchain\_community.chat\_models import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 配置 DeepSeek 模型  
llm = ChatOpenAI(  
    model='deepseek-chat',  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base=os.getenv("BASE\_URL"),  
    streaming=True,  
    temperature=0.7  
)  
# 2.定义模型  
​  
# 使用LangChain为实体记忆设计的预定义模板  
prompt = ENTITY\_MEMORY\_CONVERSATION\_TEMPLATE  
# 初始化实体记忆  
memory = ConversationEntityMemory(llm=llm)  
# 提供对话链  
chain = LLMChain(  
    llm=llm,  
    prompt=ENTITY\_MEMORY\_CONVERSATION\_TEMPLATE,  
    memory=ConversationEntityMemory(llm=llm),  
    #verbose=True, # 设置为True可以看到链的详细推理过程  
)  
# 进行几轮对话，记忆组件会在后台自动提取和存储实体信息  
chain.invoke(input="你好，我叫蜘蛛侠。我的好朋友包括钢铁侠、美国队长和绿巨人。")  
chain.invoke(input="我住在纽约。")  
chain.invoke(input="我使用的装备是由斯塔克工业提供的。")  
# 查询记忆体中存储的实体信息  
print("\n当前存储的实体信息:")  
print(chain.memory.entity\_store.store)  
# 基于记忆进行提问  
answer = chain.invoke(input="你能告诉我蜘蛛侠住在哪里以及他的好朋友有哪些吗？")  
print("\nAI的回答:")  
print(answer)  
​

#### **3.4.5 ConversationKGMemory**

ConversationKGMemory是一种基于知识图谱（Knowledge Graph）的对话记忆模块，它比 ConversationEntityMemory 更进一步，不仅能识别和存储实体，还能捕捉实体之间的复杂关系，形成 结构化的知识网络。

特点：

* 知识图谱结构 将对话内容转化为 (头实体, 关系, 尾实体) 的三元组形式
* 动态关系推理

​  
#1.导入相关包  
from langchain.memory import ConversationKGMemory  
from langchain\_community.chat\_models import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
​  
# 加载环境变量  
load\_dotenv()  
​  
#2 配置 DeepSeek 模型  
llm = ChatOpenAI(  
    model='deepseek-chat',  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base=os.getenv("BASE\_URL"),  
    streaming=True,  
    temperature=0.7  
)  
​  
# 3.定义ConversationKGMemory对象  
memory = ConversationKGMemory(llm=llm)  
​  
# 4.保存会话  
memory.save\_context({"input": "向山姆问好"}, {"output": "山姆是谁"})  
memory.save\_context({"input": "山姆是我的朋友"}, {"output": "好的"})  
​  
# 5.查询会话  
memory.load\_memory\_variables({"input": "山姆是谁"})  
​  
# 6. 增加新的 会话  
result = memory.get\_knowledge\_triplets("她最喜欢的颜色是红色")  
print(result)

#### **3.4.6 VectorStoreRetrieverMemory**

VectorStoreRetrieverMemory是一种基于向量检索的先进记忆机制，它将对话历史存储在向量数据库中，通过 语义相似度检索 相关信息，而非传统的线性记忆方式。每次调用时，就会查找与该记忆关联最高的k个文档。

适用场景：这种记忆特别适合需要长期记忆和语义理解的复杂对话系统。

import os  
from dotenv import load\_dotenv  
from langchain.memory import VectorStoreRetrieverMemory, ConversationBufferMemory  
from langchain\_community.vectorstores import FAISS  
from langchain\_community.chat\_models import ChatOpenAI  
from langchain\_community.embeddings import HuggingFaceEmbeddings  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 1. 配置 DeepSeek 模型  
llm = ChatOpenAI(  
    model='deepseek-chat',  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base=os.getenv("BASE\_URL", "https://api.deepseek.com/v1"),  
    temperature=0.7  
)  
​  
# 2. 初始化对话记忆  
conversation\_memory = ConversationBufferMemory()  
​  
# 3. 添加对话到记忆  
conversation\_memory.save\_context({"input": "我最喜欢的食物是披萨"}, {"output": "很高兴知道"})  
conversation\_memory.save\_context({"input": "我喜欢的运动是跑步"}, {"output": "好的,我知道了"})  
conversation\_memory.save\_context({"input": "我最喜欢的运动是足球"}, {"output": "好的,我知道了"})  
​  
# 4. 使用更轻量的本地嵌入模型  
try:  
    embeddings\_model = HuggingFaceEmbeddings(  
        model\_name="sentence-transformers/all-MiniLM-L6-v2"  
   )

except:  
    # 如果上面的模型下载失败，尝试使用更基础的模型  
    embeddings\_model = HuggingFaceEmbeddings(  
        model\_name="sentence-transformers/paraphrase-albert-small-v2"  
   )  
​  
# 5. 直接从对话记忆构建文本  
texts = []  
for message in conversation\_memory.chat\_memory.messages:  
    texts.append(message.content)

​  
# 6. 初始化向量数据库  
vectorstore = FAISS.from\_texts(texts, embeddings\_model)  
​  
# 7. 定义检索对象  
retriever = vectorstore.as\_retriever(search\_kwargs=dict(k=1))  
​  
# 8. 初始化VectorStoreRetrieverMemory  
memory = VectorStoreRetrieverMemory(retriever=retriever)  
​  
print("记忆检索测试:")  
print("查询: 食物")  
result1 = memory.load\_memory\_variables({"prompt": "食物"})  
print(f"结果: {result1}")  
​  
print("\n查询: 运动")  
result2 = memory.load\_memory\_variables({"prompt": "运动"})  
print(f"结果: {result2}")  
​  
print("\n查询: 跑步")  
result3 = memory.load\_memory\_variables({"prompt": "跑步"})  
print(f"结果: {result3}")

Mysql数据库表：知识库、 文件名、拓展名、分块总数、分块内容、向量值 ID

法律合同 劳动法 1>>>> [0.9888,0.2999....]

法律合同 劳动法 2>>>> [0.9828,0.2999....]

法律合同 劳动法 3>>>> [0.9820,0.2999....]

向量数据库：ID、 向量值、 分块内容

0786 [0.9888,0.2999....] 1>>>>

0787 [0.9828,0.2999....] 2>>>>

0788 [0.9820,0.2999....] 3>>>>

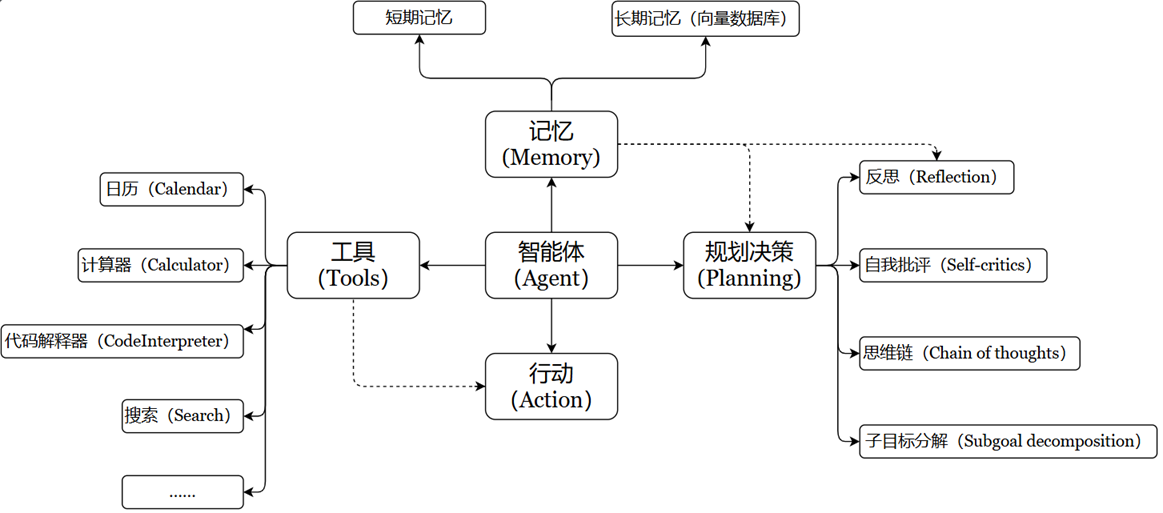
## **Section04 Tools**

### **4.1 Tools概述**

#### **4.1.1 介绍**

要构建更强大的AI工程应用，只有生成文本这样的“ 纸上谈兵 ”能力自然是不够的。工具Tools不仅仅 是“肢体”的延伸，更是为“大脑”插上了想象力的“翅膀”。借助工具，才能让AI应用的能力真正具 备无限的可能，才能从“ 认识世界 ”走向“ 改变世界 ”。

Tools 用于扩展大语言模型（LLM）的能力，使其能够与外部系统、API 或自定义函数交互，从而完成仅靠文本生成无法实现的任务（如搜索、计算、数据库查询等）。



特点：

* 增强 LLM 的功能 ：让 LLM 突破纯文本生成的限制，执行实际操作（如调用搜索引擎、查询数据 库、运行代码等）
* 支持智能决策 ：在Agent 工作流中，LLM 根据用户输入动态选择最合适的 Tool 完成任务。
* 模块化设计 ：每个 Tool 专注一个功能，便于复用和组合（例如：搜索工具 + 计算工具 + 天气查 询工具）

LangChain 拥有大量第三方工具。请访问工具集成查看可用工具列表。

<https://python.langchain.com/v0.2/docs/integrations/tools/>

#### **4.1.2 Tool 的要素**

Tools 本质上是封装了特定功能的可调用模块，是Agent、Chain或LLM可以用来与世界互动的接口。

* Tool 通常包含如下几个要素：
* name ：工具的名称
* description ：工具的功能描述
* 该工具输入的 JSON模式
* 要调用的函数
* return\_direct ：是否应将工具结果直接返回给用户（仅对Agent相关）

实操步骤：

步骤1：将name、description 和 JSON模式作为上下文提供给LLM

步骤2：LLM会根据提示词推断出 需要调用哪些工具 ，并提供具体的调用参数信息

步骤3：用户需要根据返回的工具调用信息，自行触发相关工具的回调



###

### **4.2 自定义工具**

#### **4.2.1 两种自定义方式**

第1种：使用@tool装饰器（自定义工具的最简单方式）

装饰器默认使用函数名称作为工具名称，但可以通过参数 name\_or\_callable 来覆盖此设置。 同时，装饰器将使用函数的 文档字符串 作为 工具的描述 ，因此函数必须提供文档字符串。

第2种：使用StructuredTool.from\_function类方法

这类似于 @tool 装饰器，但允许更多配置和同步/异步实现的规范

#### **4.2.2 几个常用属性**

Tool由几个常用属性组成：

| **属性** | **类型** | **描述** |
| --- | --- | --- |
| name | str | 必选的 ，在提供给LLM或Agent的工具集中必须是唯一的。 |
| description | str | 可选但建议 ，描述工具的功能。LLM或Agent将使用此描述作 为上下文，使用它确定工具的使用 |
| args\_schema | Pydantic BaseModel | 可选但建议 ，可用于提供更多信息（例如，few-shot示例） 或验证预期参数。 |
| return\_direct | boolean | 仅对Agent相关。当为True时，在调用给定工具后，Agent将 停止并将结果直接返回给用户。 |

#### **4.2.3 具体实现**

方式1：@tool 装饰器

from langchain\_core.tools import tool  
​  
@tool  
def add\_numbers(a: int, b: int) -> int:  
    """Add two numbers together."""  
    return a + b  
​  
print(f"name={add\_numbers.name}")  
print(f"description={add\_numbers.description}")  
print(f"args={add\_numbers.args}")  
print(f"return\_direct={add\_numbers.return\_direct}")

说明： return\_direct参数 的默认值是False。当return\_direct=False时，工具执行结果会返回给 Agent，让Agent决定下一步操作；而return\_direct=True则会中断这个循环，直接结束流程，返回结 果给用户。

举例2：通过@tool的参数设置进行重置

from langchain\_core.tools import tool  
​  
@tool(name\_or\_callable="add\_two\_numbers", description="计算两个数字的和", return\_direct= True)  
def add\_numbers(a: int, b: int) -> int:  
    """Add two numbers together."""  
    return a + b  
​  
print(f"name={add\_numbers.name}")  
print(f"description={add\_numbers.description}")  
print(f"args={add\_numbers.args}")  
print(f"return\_direct={add\_numbers.return\_direct}")

补充：还可以修改参数的说明

from langchain\_core.tools import tool  
from pydantic import BaseModel  
from pydantic import Field  
​  
class FieldInfo(BaseModel):  
    a: int = Field(description="第一个整型参数")  
    b: int = Field(description="第二个整型参数")  
      
@tool(name\_or\_callable="add\_two\_numbers", description="计算两个数字的和", return\_direct= True, args\_schema=FieldInfo)  
def add\_numbers(a: int, b: int) -> int:  
    """Add two numbers together."""  
    return a + b  
​  
print(f"name={add\_numbers.name}")  
print(f"description={add\_numbers.description}")  
print(f"args={add\_numbers.args}")  
print(f"return\_direct={add\_numbers.return\_direct}")  
​  
res = add\_numbers.invoke({"a": 1, "b": 2})  
print(res)

方式2：StructuredTool的from\_function()

StructuredTool.from\_function 类方法提供了比 @tool 装饰器更多的可配置性，而无需太多额外的代码。

from langchain\_core.tools import StructuredTool  
​  
def search\_fun(query: str):  
    return "最后的查询结果"  
​  
search\_tool = StructuredTool.from\_function(  
    func=search\_fun,  
    name="Search",  
    description="从Google搜索引擎进行查询"  
)  
​  
print(f"name={search\_tool.name}")  
print(f"args={search\_tool.args}")  
print(f"desciption={search\_tool.description}")  
​  
search\_tool.invoke("查询")

设置更丰富的参数

from langchain\_core.tools import StructuredTool  
from pydantic import Field, BaseModel  
​  
class FieldInfo(BaseModel):  
    query: str = Field(description="要检索的关键词")  
      
def search\_function(query: str):  
    return "LangChain"  
​  
search1 = StructuredTool.from\_function(  
    func=search\_function,  
    name="Search",  
    description="useful for when you need to answer questions about current events",  
    args\_schema=FieldInfo,  
    return\_direct=True,  
)  
​  
print(f"name = {search1.name}")  
print(f"description = {search1.description}")  
print(f"args = {search1.args}")  
print(f"return\_direct = {search1.return\_direct}")  
​  
search1.invoke("hello")

#### **4.2.4 工具调用举例**

我们通过大模型分析用户需求，判断是否需要调用指定工具。

举例1：大模型分析调用工具

#1.导入相关依赖  
from langchain\_community.tools import MoveFileTool  
from langchain\_core.messages import HumanMessage  
from langchain\_core.utils.function\_calling import convert\_to\_openai\_function  
from langchain\_community.chat\_models import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
​  
# 加载环境变量  
load\_dotenv()  
​  
#2 配置 DeepSeek 模型  
chat = ChatOpenAI(  
    model='deepseek-chat',  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base=os.getenv("BASE\_URL"),  
    streaming=True,  
    temperature=0.7  
)  
​  
​  
​  
# 3.定义工具  
tools = [MoveFileTool()]  
​  
# 4.这里需要将工具转换为openai函数，后续再将函数传入模型调用  
functions = [convert\_to\_openai\_function(t) for t in tools]  
# print(functions[0])  
​  
# 5. 提供大模型调用的消息列表  
messages = [HumanMessage(content="将文件a移动到桌面")]  
​  
# 6.模型使用函数  
response = chat.invoke(  
    input = messages,  
    functions=functions  
)  
​  
print(response)  
​

模型绑定工具，调用模型，传入Message对象。 作为对照，修改代码：

response = chat.invoke(  
   input = [HumanMessage(content="今天天气怎么样?")],  
   functions=functions  
)  
print(response)

content='我无法提供实时天气信息。建议您查看当地天气预报应用或网站获取准确的天气情况。' additional\_kwargs={'refusal': None} response\_metadata={'token\_usage': {'completion\_tokens': 22, 'prompt\_tokens': 192, 'total\_tokens': 214, 'completion\_tokens\_details': None, 'prompt\_tokens\_details': {'audio\_tokens': None, 'cached\_tokens': 0}}, 'model\_name': 'qwen-plus', 'system\_fingerprint': None, 'id': 'chatcmpl-4e3d0b9d-a234-491f-8ddb-0745858eeb14', 'service\_tier': None, 'finish\_reason': 'stop', 'logprobs': None} id='run--18163067-b2a3-44d7-a3e1-0ab1da355405-0' usage\_metadata={'input\_tokens': 192, 'output\_tokens': 22, 'total\_tokens': 214, 'input\_token\_details': {'cache\_read': 0}, 'output\_token\_details': {}}

调用工具说明 两种情况：

情况1：大模型决定调用工具

如果模型认为需要调用工具（如 MoveFileTool ），返回的 message 会包含：

* content : 通常为空（因为模型选择调用工具，而非生成自然语言回复）。
* additional\_kwargs : 包含工具调用的详细信息：

AIMessage(  
    content='', # 无自然语言回复  
    additional\_kwargs={  
        'function\_call': {  
        'name': 'move\_file', # 工具名称  
        'arguments':  
        '{"source\_path":"a","destination\_path":"/Users/YourUsername/Desktop/a"}' # 工具参数  
   }  
 }  
)

情况2：大模型不调用工具

如果模型认为无需调用工具（例如用户输入与工具无关），返回的 message 会是普通文本回复：

AIMessage(  
    content='我没有找到需要移动的文件。', # 自然语言回复  
    additional\_kwargs={'refusal': None} # 无工具调用  
)

举例2：如何调用大模型分析处理的工具

注意：

* 大模型跟Agent的核心区别：是否涉及到工具的调用
* 针对于大模型，仅能分析出要调用的工具，但是此工具(函数)不能真正的执行

对于Agent，除了分析出要调用的工具之外，还可以执行具体的工具(函数)

## **Section05 Agents**

### **5.1理解Agents**

通用人工智能（AGI）将是AI的终极形态，几乎已成为业界共识。同样，构建智能体（Agent）则是AI 工程应用当下的“终极形态”。 将 AI 和人类协作的程度类比自动驾驶的不同阶段：



#### **5.1.1 Agent与Chain的区别**

在Chain中行动序列是硬编码的、固定流程的 ，像是“线性流水线”，而Agent则采用语言模型作为推理引擎 ，具备一定的自主决策能力，来确定以什么样的顺序采取什么样的行动，像是“拥有大脑的机器工人”。

它可以根据任务 动态决定 ：

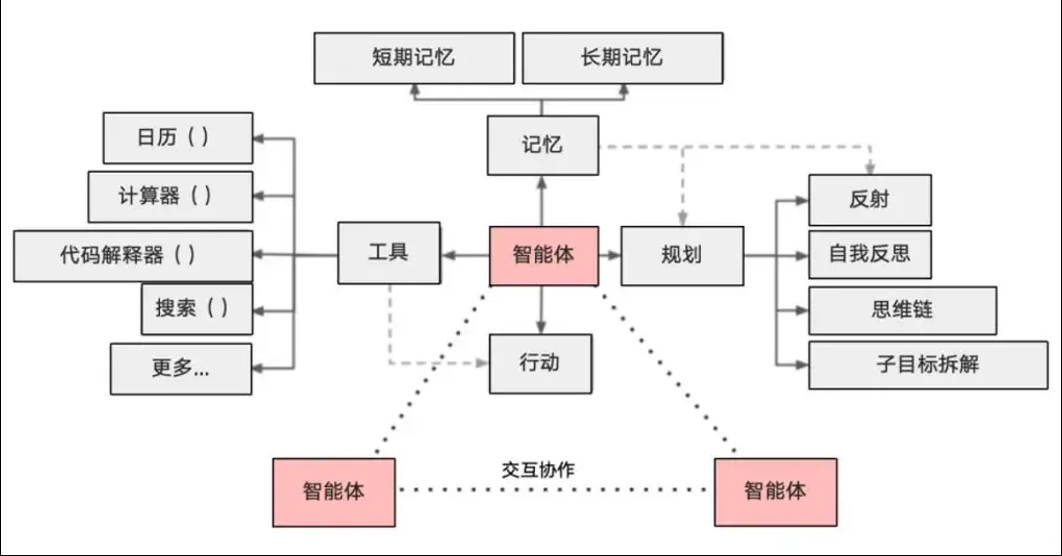
* 如何拆解任务
* 需要调用哪些工具
* 以什么顺序调用
* 如何利用好中间结果推进任务

#### **5.1.2 什么是Agent**

Agent（智能体） 是一个通过动态协调大语言模型（LLM） 和 工具（Tools） 来完成复杂任务的智能系 统。它让LLM充当"决策大脑"，根据用户输入自主选择和执行工具（如搜索、计算、数据库查询等）， 最终生成精准的响应。

#### **5.1.3 Agent的核心能力/组件**

作为一个智能体，需要具备以下核心能力：



1）大模型(LLM)：作为大脑，提供推理、规划和知识理解能力。

比如：OpenaAI()、ChatOpenAI()

2）记忆(Memory)：具备短期记忆（上下文）和长期记忆（向量存储），支持快速知识检索。

比如：ConversationBufferMemory、ConversationSummaryMemory、 ConversationBufferWindowMemory等

3）工具(Tools)：调用外部工具（如API、数据库）的执行单元

比如：SearchTool、CalculatorTool

4）规划(Planning)：任务分解、反思与自省框架实现复杂任务处理

5）行动(Action)：实际执行决策的能力

比如：检索、推理、编程

6）协作：通过与其他智能体交互合作，完成更复杂的任务目标。

问题：为什么要调用第三方工具（比如：搜索引擎或者 数据库）或借助第三方库呢？ 因为大模型虽然非常强大，但是也具备一定的局限性。比如不能回答 实时信息 、处理 复杂数学逻辑 问题 仍然非常初级等等。因此，可以借助第三方工具来辅助大模型的应用。 以MCP工具为例说明：[https://bailian.console.aliyun.com/?tab=mcp#/mcp-market](https://bailian.console.aliyun.com/?tab=mcp" \l "/mcp-market)

#### **5.1.4 关键组件**

Agents 模块有几个关键组件：

1、工具

Tool LangChain 提供了广泛的入门工具，但也支持 自定义工具 ，包括自定义描述。 在框架内，每个功能或函数被 封装成一个工具 （Tools），具有自己的输入、输出及处理方法。

具体使用步骤：

① Agent 接收任务后，通过大模型推理选择适合的工具处理任务。

② 一旦选定，LangChain将任务输入传递给该工具，工具处理输入生成输出。

③ 输出经过大模型推理，可用于其他工具的输入或作为最终结果返回给用户。

2、工具集 Toolkits

在构建Agent时，通常提供给LLM的工具不仅仅只有一两个，而是一组可供选择的工具集(Tool列表)， 这样可以让 LLM 在完成任务时有更多的选择。

3、智能体/代理 Agent

智能体/代理（agent）可以协助我们做出决策，调用相应的 API。底层的实现方式是通过 LLM 来决定 下一步执行什么动作。

4、代理执行器 AgentExecutor

AgentExecutor本质上是代理的运行时，负责协调智能体的决策和实际的工具执行。

AgentExecutor是⼀个很好的起点，但是当你开始拥有更多定制化的代理时，它就不够灵活了。 为了解决这个问题，我们构建了LangGraph，使其成为这种灵活、⾼度可控的运⾏时。

###

### **5.2 Agent 入门使用**

#### **5.2.1 Agent、AgentExecutor的创建**

|  | **环节1：创建Agent** | **环节2：创建 AgentExecutor** |
| --- | --- | --- |
| 方式1：传统 方式 | 使用 AgentType 指定 | initialize\_agent() |
| 方式2：通用 方式 | create\_xxx\_agent() 比如：create\_react\_agent()、 create\_tool\_calling\_agent() | 调用AgentExecutor() 构造方法 |

#### **5.2.2 Agent的类型**

顾名思义就是某件事可以由不同的⼈去完成，最终结果可能是⼀样的，但是做的过程可能各有千秋。⽐如⼀个公司需求， 普通开发 可以编写， 技术经理 也可以编写， CTO 也可以编写。虽然都能完成最后的需求，但是CTO做的过程可能更加直观，⾼效。

在LangChain中Agent的类型就是为你提供不同的"问题解决姿势"的。

API说明：<https://python.langchain.com/v0.1/docs/modules/agents/agent_types/>

Agents的核心类型有两种模式：

* 方式1：Funcation Call模式
* 方式2：ReAct 模式

2.2.1 FUNCATION\_CALL模式

* 基于 结构化函数调用 （如 OpenAI Function Calling）
* 直接生成工具调用参数（ JSON 格式 ）

典型 AgentType：

#第1种：  
AgentType.OPENAI\_FUNCTIONS  
#第2种：  
AgentType.OPENAI\_MULTI\_FUNCTIONS

工作流程示例：

第1步：找到Search工具：{"tool": "Search", "args": {"query": "LangChain最新版本"}}  
第2步：执行Search工具  
======================================  
第1步：找打scrape\_website工具：{"tool": "Search", "args": {"target": "LangChain最新版本","url":"要抓取的网站地址"}}  
第2步：执行scrape\_website工具  
​

2.2.2 ReAct 模式

* 基于文本推理的链式思考（Reasoning + Acting），具备反思和自我纠错能力。

推理（Reasoning）：分析当前状态，决定下一步行动

行动（Acting）：调用工具并返回结果

* 通过 自然语言描述决策过程
* 适合需要明确推理步骤的场景。例如智能客服、问答系统、任务执行等。

典型 AgentType：

#第1种：零样本推理(可以在没有预先训练的情况下尝试解决新的问题)  
AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION  
#第2种：无记忆对话  
AgentType.STRUCTURED\_CHAT\_ZERO\_SHOT\_REACT\_DESCRIPTION  
#第3种：带记忆对话  
AgentType.CONVERSATIONAL\_REACT\_DESCRIPTION

工作流程示例：

问题：我想要查询xxx  
思考：我需要先搜索最新信息 → 行动：调用Search工具 → 观察：获得3个结果 →  
思考：需要抓取第一个链接 → 行动：调用scrape\_website工具...→ 观察：获得工具结果  
最后：获取结果

Agent两种典型类型对比表

| **特性** | **Function Call模式** | **ReAct 模式** |
| --- | --- | --- |
| 底层机制 | 结构化函数调用 | 自然语言推理 |
| 输出格式 | JSON/结构化数据 | 自由文本 |
| 适合场景 | 需要高效工具调用 | 需要解释决策过程 |
| 典型延迟 | 较低 （直接参数化调用） | 较高 （需生成完整文本） |
| LLM要求 | 需支持函数调用（如gpt-4） | 通用模型即可 |

#### **5.2.3 AgentExecutor创建方式**

**传统方式：initialize\_agent()**

特点：

内置一些标准化模板（如 ZERO\_SHOT\_REACT\_DESCRIPTION ）

Agent的创建：使用AgentType

优点：快速上手（3行代码完成配置）

缺点：定制化能力较弱（如提示词固定）

from langchain.agents import initialize\_agent  
#第1步：创建AgentExecutor  
agent\_executor = initialize\_agent(  
    llm=llm,  
    agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION,  
    tools=[search\_tool],  
    verbose=True  
)  
#第2步：执行  
agent\_executor.invoke({"xxxx"})

**通用方式：AgentExecutor构造方法**

特点：

Agent的创建：使用create\_xxx\_agent

优点：

可自定义提示词（如从远程hub获取或本地自定义）

清晰分离Agent逻辑与执行逻辑

缺点：

需要更多代码 需理解底层组件关系

prompt = hub.pull("hwchase17/react")  
tools = [search\_tool]  
#第1步：创建Agent实例  
agent = create\_react\_agent(  
llm=llm,  
prompt=prompt,  
tools=tools  
)  
#第2步：创建AgentExecutor实例  
agent\_executor = AgentExecutor(  
agent=agent,  
tools=tools  
)  
#第3步：执行  
agent\_executor.invoke({"input":"xxxxx"})

#### **5.2.4 小结创建方式**



###

### **5.3 Agent中工具的使用**

#### **5.3.1 传统方式**

案例1：单工具使用

需求：今天北京的天气怎么样?

使用Tavily搜索工具 Tavily的搜索API是一个专门为人工智能Agent(或LLM)构建的搜索引擎，可以快速提供实时、 准确和真实的结果。 LangChain 中有一个内置工具，可以轻松使用 Tavily 搜索引擎 作为工具。

TAVILY\_API\_KEY申请：<https://tavily.com/>，注册账号并登录，创建 API 密钥。

方式1：ReAct模式

AgentType是 ZERO\_SHOT\_REACT\_DESCRIPTION

from langchain.agents import initialize\_agent, AgentType  
from langchain.tools import Tool  
from langchain\_community.tools.tavily\_search import TavilySearchResults  
from langchain\_community.chat\_models import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 配置 DeepSeek 模型  
chat = ChatOpenAI(  
    model='deepseek-chat',  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base=os.getenv("BASE\_URL"),  
    streaming=True,  
    temperature=0.7  
)  
# 1. 设置API密钥  
os.environ["TAVILY\_API\_KEY"] = "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"  
​  
# 2. 初始化搜索工具  
search = TavilySearchResults(max\_results=3)  
​  
# 3. 创建Tool的实例，这里的name是工具名称，description是工具的描述，func是工具的函数  
search\_tool = Tool(  
    name="Search",  
    description="用于从互联网搜索信息",  
    func=search.run,  
)  
​  
# 4. 初始化大模型  
# 使用前面初始化好的 chat  
​  
# 5. 创建AgentExecutor  
agent\_executor = initialize\_agent(  
    tools=[search\_tool],  
    llm=chat,  
    agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION,  
    verbose=True,  
)  
​  
# 6. 测试查询  
query = "今天深圳的天气怎么样？"  
result = agent\_executor.run(query)  
print(result)

方式2：FUNCATION\_CALL模式

AgentType是 OPENAI\_FUNCTIONS

提示：只需要修改前面代码中的initialize\_agent中的agent参数值。

from langchain.agents import initialize\_agent, AgentType  
from langchain.tools import Tool  
from langchain\_community.tools.tavily\_search import TavilySearchResults  
from langchain\_community.chat\_models import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 配置 DeepSeek 模型  
chat = ChatOpenAI(  
    model='deepseek-chat',  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base=os.getenv("BASE\_URL"),  
    streaming=True,  
    temperature=0.7  
)  
# 1. 设置API密钥  
os.environ["TAVILY\_API\_KEY"] = "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"  
​  
# 2. 初始化搜索工具  
search = TavilySearchResults(max\_results=3)  
​  
# 3. 创建Tool的实例，这里的name是工具名称，description是工具的描述，func是工具的函数  
search\_tool = Tool(  
    name="Search",  
    description="用于从互联网搜索信息",  
    func=search.run,  
)  
​  
# 4. 初始化大模型  
# 使用前面初始化好的 chat  
​  
# 5. 创建AgentExecutor  
agent\_executor = initialize\_agent(  
    tools=[search\_tool],  
    llm=chat,  
    agent=AgentType.OPENAI\_FUNCTIONS , # 唯一变化的位置  
    verbose=True,  
)  
​  
# 6. 测试查询  
query = "今天深圳的天气怎么样？"  
result = agent\_executor.run(query)  
print(result)

二者对比：ZERO\_SHOT\_REACT\_DESCRIPTION和OPENAI\_FUNCTIONS



案例2：多工具使用

需求： 计算特斯拉当前股价是多少？ 比去年上涨了百分之几？（

提示：调用PythonREPL实例的run方法） 多个（两个）工具的选择

方式1：ReAct 模式 AgentType是 ZERO\_SHOT\_REACT\_DESCRIPTION

pip install langchain\_experimental

from langchain.agents import initialize\_agent, AgentType  
from langchain.tools import Tool  
from langchain\_community.tools.tavily\_search import TavilySearchResults  
from langchain\_experimental.utilities import PythonREPL  
from langchain\_community.chat\_models import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 配置 DeepSeek 模型  
chat = ChatOpenAI(  
    model='deepseek-chat',  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base=os.getenv("BASE\_URL"),  
    streaming=True,  
    temperature=0.7  
)  
# 1. 设置API密钥  
os.environ["TAVILY\_API\_KEY"] = "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"  
​  
# 2. 初始化搜索工具  
search = TavilySearchResults(max\_results=3)  
​  
​  
# 4. 初始化大模型  
# 使用前面初始化好的 chat  
​  
# 4.定义计算工具  
python\_repl = PythonREPL() # LangChain封装的工具类可以进行数学计算  
calc\_tool = Tool(  
    name="Calculator",  
    func=python\_repl.run,  
    description="用于执行数学计算，例如计算百分比变化"  
)  
​  
​  
# 5. 创建AgentExecutor  
agent\_executor = initialize\_agent(  
    tools=[search, calc\_tool],  
    llm=chat,  
    agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION,  
    verbose=True,  
)  
​  
# 6. 测试查询  
query = "比亚迪现在的股价是多少?比去年上涨了百分之几?"  
result = agent\_executor.run(query)  
print(result)

方式2：FUNCATION\_CALL模式

AgentType是 FUNCATION\_CALL

from langchain.agents import initialize\_agent, AgentType  
from langchain.tools import Tool  
from langchain\_community.tools.tavily\_search import TavilySearchResults  
from langchain\_experimental.utilities import PythonREPL  
from langchain\_community.chat\_models import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 配置 DeepSeek 模型  
chat = ChatOpenAI(  
    model='deepseek-chat',  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base=os.getenv("BASE\_URL"),  
    streaming=True,  
    temperature=0.7  
)  
# 1. 设置API密钥  
os.environ["TAVILY\_API\_KEY"] = "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"  
​  
# 2. 初始化搜索工具  
search = TavilySearchResults(max\_results=3)  
​  
​  
# 4. 初始化大模型  
# 使用前面初始化好的 chat  
​  
# 4.定义计算工具  
python\_repl = PythonREPL() # LangChain封装的工具类可以进行数学计算  
calc\_tool = Tool(  
    name="Calculator",  
    func=python\_repl.run,  
    description="用于执行数学计算，例如计算百分比变化"  
)  
​  
​  
# 5. 创建AgentExecutor  
agent\_executor = initialize\_agent(  
    tools=[search, calc\_tool],  
    llm=chat,  
    agent=AgentType.OPENAI\_FUNCTIONS,  # 唯一变化的位置  
    verbose=True,  
)  
​  
# 6. 测试查询  
query = "比亚迪现在的股价是多少?比去年上涨了百分之几?"  
result = agent\_executor.run(query)  
print(result)

案例3：自定义函数与工具

需求：计算3的平方，Agent自动调用工具完成

from langchain.agents import initialize\_agent, AgentType, Tool  
from langchain\_community.chat\_models import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 配置 DeepSeek 模型  
chat = ChatOpenAI(  
    model='deepseek-chat',  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base=os.getenv("BASE\_URL"),  
    streaming=True,  
    temperature=0.7  
)  
​  
# 1. 定义工具 - 计算器（要求字符串输入）  
def simple\_calculator(expression: str) -> str:  
    """  
   基础数学计算工具，支持加减乘除和幂运算  
   参数:  
   expression: 数学表达式字符串，如 "3+5" 或 "2\*\*3"  
   返回:  
   计算结果字符串或错误信息  
   """  
    print(f"\n[工具调用] 计算表达式: {expression}")  
    print("只因为在人群中多看了你一眼，确认下你调用了我^\_^")  
    return str(eval(expression))  
# 2. 创建工具对象  
math\_calculator\_tool = Tool(  
    name="Math\_Calculator", # 工具名称（Agent将根据名称选择工具）  
    func=simple\_calculator, # 工具调用的函数  
    description="用于数学计算，输入必须是纯数学表达式（如'3+5'或'3\*\*2'表示平方）。不支持字母或特殊符号" # 关键：明确输入格式要求  
)  
# 3. 初始化大模型  
# 使用初始化好的chat  
​  
# 4. 初始化AgentExecutor（使用零样本React模式、增加超时设置）  
agent\_executor = initialize\_agent(  
    tools=[math\_calculator\_tool], # 可用的工具列表  
    llm=chat,  
    agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION, # 简单指令模式  
    verbose=True # 关键参数！在控制台显示详细的推理过程  
)  
# 5. 测试工具调用（添加异常捕获）  
print("=== 测试：正常工具调用 ===")  
response = agent\_executor.invoke("计算3的平方") # 向Agent提问  
print("最终答案:", response)  
​

#### **5.3.2 通用方式**

需求：今天北京的天气怎么样？？

**方式1：FUNCATION\_CALL模式**

from langchain\_core.prompts import ChatPromptTemplate  
from langchain.agents import AgentType, create\_tool\_calling\_agent, AgentExecutor  
from langchain.tools import Tool  
from langchain\_community.tools.tavily\_search import TavilySearchResults  
​  
# 1. 设置API密钥  
os.environ["TAVILY\_API\_KEY"] = "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"  
​  
# 2. 初始化搜索工具  
search = TavilySearchResults(max\_results=3)  
​  
# 3.自定义提示词模版  
prompt = ChatPromptTemplate.from\_messages([  
   ("system","您是一位乐于助人的助手，请务必使用 tavily\_search\_results\_json 工具来获取信息。"),  
   ("human", "{input}"),  
   ("system", "{agent\_scratchpad}"),  
])  
​  
​  
# 4. 初始化大模型  
# 使用前面初始化好的 chat  
​  
# 5.创建Agent对象  
agent = create\_tool\_calling\_agent(  
    llm = chat,  
    tools = [search],  
    prompt = prompt  
)  
​  
# 6.创建AgentExecutor执行器  
agent\_executor = AgentExecutor(agent=agent, tools=[search], verbose=True)  
​  
​  
# 7. 测试查询  
result = agent\_executor.invoke({"input": "今天北京的天气怎么样？?"})  
print(result)

注意：agent\_scratchpad必须声明，它用于存储和传递Agent的思考过程。比如，在调用链式工具时（如先搜索天气再推荐行程）， agent\_scratchpad 保留所有历史步骤，避免上下文丢失。format方法会将intermediate\_steps转换为特定格式的字符串，并赋值给agent\_scratchpad变量。如果不传递intermediate\_steps参数，会导致KeyError: 'intermediate\_steps'错误。

**方式2：ReAct模式**

**体会1：使用PromptTemplate**

提示词要体现可以使用的工具、用户输入和agent\_scratchpad。

远程的提示词模版通过<https://smith.langchain.com/hub/hwchase17>获取

举例：<https://smith.langchain.com/hub/hwchase17/react>，这个模板是专为ReAct模式设计的提示模板。这个模板中已经有聊天对话键 **tools** 、 **tool\_names** 、 **agent\_scratchpad**

from langchain\_core.prompts import PromptTemplate  
from langchain.agents import AgentType, create\_tool\_calling\_agent, AgentExecutor, create\_react\_agent  
from langchain.tools import Tool  
from langchain\_community.tools.tavily\_search import TavilySearchResults  
from langchain\_community.chat\_models import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 配置 DeepSeek 模型  
chat = ChatOpenAI(  
    model='deepseek-chat',  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base=os.getenv("BASE\_URL"),  
    streaming=True,  
    temperature=0.7  
)  
# 1. 设置API密钥  
os.environ["TAVILY\_API\_KEY"] = "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"  
​  
# 2. 初始化搜索工具  
search = TavilySearchResults(max\_results=3)  
​  
# 3. 定义template 来自于上面的网站  
template = """  
Answer the following questions as best you can. You have access to the following tools:  
​  
{tools}  
​  
Use the following format:  
​  
Question: the input question you must answer  
Thought: you should always think about what to do  
Action: the action to take, should be one of [{tool\_names}]  
Action Input: the input to the action  
Observation: the result of the action  
... (this Thought/Action/Action Input/Observation can repeat N times)  
Thought: I now know the final answer  
Final Answer: the final answer to the original input question  
​  
Begin!  
​  
Question: {input}  
Thought:{agent\_scratchpad}  
"""  
​  
# 3.自定义提示词模版  
prompt = PromptTemplate.from\_template( template)  
​  
​  
# 4. 初始化大模型  
# 使用前面初始化好的 chat  
​  
# 5.创建Agent对象  
agent = create\_react\_agent(llm=chat, tools=[search], prompt=prompt)  
​  
# 6.创建AgentExecutor执行器  
agent\_executor = AgentExecutor(agent=agent, tools=[search], verbose=True, handle\_parsing\_errors=True)  
​  
​  
# 7. 测试查询  
result = agent\_executor.invoke({"input": "今天北京的天气怎么样？?"})  
print(result)

**体会2：使用ChatPromptTemplate**

提示词中需要体现使用的工具、用户输入和agent\_scratchpad。

from langchain\_core.prompts import ChatPromptTemplate  
from langchain.agents import AgentType, create\_react\_agent, AgentExecutor  
from langchain.tools import Tool  
from langchain\_community.tools.tavily\_search import TavilySearchResults  
from langchain\_community.chat\_models import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 配置 DeepSeek 模型  
chat = ChatOpenAI(  
    model='deepseek-chat',  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base=os.getenv("BASE\_URL"),  
    streaming=True,  
    temperature=0.7  
)  
# 1. 设置API密钥  
os.environ["TAVILY\_API\_KEY"] = "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"  
​  
# 2. 初始化搜索工具  
search = TavilySearchResults(max\_results=3)  
​  
# 3. 创建模版  
template = """  
Answer the following questions as best you can. You have access to the following tools:  
​  
{tools}  
​  
Use the following format:  
​  
Question: the input question you must answer  
Thought: you should always think about what to do  
Action: the action to take, should be one of [{tool\_names}]  
Action Input: the input to the action  
Observation: the result of the action  
... (this Thought/Action/Action Input/Observation can repeat N times)  
Thought: I now know the final answer  
Final Answer: the final answer to the original input question  
​  
Begin!  
"""  
​  
# 3.自定义提示词模版  
prompt = ChatPromptTemplate.from\_messages([  
   ("system",template),  
   ("human", "{input}"),  
   ("system", "{agent\_scratchpad}"),  
])  
​  
​  
# 4. 初始化大模型  
# 使用前面初始化好的 chat  
​  
# 5.创建Agent对象  
agent = create\_react\_agent(llm=chat, tools=[search], prompt=prompt)  
​  
# 6.创建AgentExecutor执行器  
agent\_executor = AgentExecutor(  
    agent=agent,  
    tools=[search],  
    verbose=True  
    # handle\_parsing\_errors=True  
    # max\_iterations=6 # 可选：限制最大迭代次数，防止无限循环  
)  
​  
​  
# 7. 测试查询  
result = agent\_executor.invoke({"input": "今天北京的天气怎么样？?"})  
print(result)

上述执行可能会报错。  
错误原因：  
使用ReAct模式时，要求 LLM 的响应必须遵循严格的格式（如包含 Thought: 、 Action: 等标记）。  
但 LLM 直接返回了自由文本（非结构化），导致解析器无法识别。  
​  
修改：  
任务不变，添加 handle\_parsing\_errors=True 。用于控制 Agent 在解析工具调用或输出时发生错误的容错行为。  
handle\_parsing\_errors=True 的作用  
自动捕获错误并修复：当解析失败时，Agent 不会直接崩溃，而是将错误信息传递给 LLM，让LLM 自行修正并重试 。  
降级处理：如果重试后仍失败，Agent 会返回一个友好的错误消息（如 "I couldn't process that request." ），而不是抛出异常。

小结：

* *传统方式，相对于通用方式来讲，不用提供提示词模版*
* *对于通用方式来讲：*

*Function\_call模式，在创建Agent时，推荐使用ChatPromptTemplate*

*ReAct模式，在创建Agent时，可以使用ChatPromptTemplate, PromptTemplate. 相对来讲，推荐使用PromptTemplate*

### **5.**4 Agent嵌入记忆组件****

#### ****5.4.1 传统方式****

比如：北京明天的天气怎么样？上海呢？ （通过两次对话实现）

举例：以REACT模式为例

from langchain.memory import ConversationBufferMemory  
from langchain.agents import initialize\_agent, AgentType  
from langchain.tools import Tool  
from langchain\_community.tools.tavily\_search import TavilySearchResults  
from langchain\_community.chat\_models import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 配置 DeepSeek 模型  
chat = ChatOpenAI(  
    model='deepseek-chat',  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base=os.getenv("BASE\_URL"),  
    streaming=True,  
    temperature=0.7  
)  
​  
# 1. 设置API密钥  
os.environ["TAVILY\_API\_KEY"] = "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"  
​  
# 2. 初始化搜索工具  
search\_tool = TavilySearchResults(max\_results=3)  
​  
# 3. 定义记忆组件(ConversationBufferMemory为例)  
memery = ConversationBufferMemory(  
    memory\_key="chat\_history",  # 必须是chat\_history，否则会报错  
    return\_messages=True  
)  
​  
# 4. 创建AgentExecutor  
agent\_executor = initialize\_agent(  
    tools=[search\_tool],  
    llm=chat,  
    agent=AgentType.CONVERSATIONAL\_REACT\_DESCRIPTION,  
    verbose=True,  
    handle\_parsing\_errors=True,  
    memory=memery,  
)  
​  
# 5. 测试对话  
query = "深圳明天的天气怎么样？"  
result = agent\_executor.invoke(query)  
print(result)  
​  
# 6. 继续对话  
query2 = "上海呢?"  
result2 = agent\_executor.invoke(query2)  
print(result2)

例子2

from langchain.memory import ConversationBufferMemory  
from langchain.agents import initialize\_agent, AgentType  
from langchain.tools import Tool  
from langchain\_community.tools.tavily\_search import TavilySearchResults  
from langchain\_community.chat\_models import ChatOpenAI  
from dotenv import load\_dotenv  
import os  
​  
# 加载环境变量  
load\_dotenv()  
​  
# 配置 DeepSeek 模型  
chat = ChatOpenAI(  
    model='deepseek-chat',  
    openai\_api\_key=os.getenv("DEEPSEEK\_API\_KEY"),  
    openai\_api\_base=os.getenv("BASE\_URL"),  
    streaming=True,  
    temperature=0.7  
)  
​  
# 1. 设置API密钥  
os.environ["TAVILY\_API\_KEY"] = "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"  
​  
# 2. 初始化搜索工具  
search\_tool = TavilySearchResults(max\_results=3)  
​  
# 3. 定义记忆组件(ConversationBufferMemory为例)  
memery = ConversationBufferMemory(  
    memory\_key="chat\_history",  # 必须是chat\_history，否则会报错  
    return\_messages=True  
)  
​  
# 4. 创建AgentExecutor  
agent\_executor = initialize\_agent(  
    tools=[search\_tool],  
    llm=chat,  
    agent=AgentType.CONVERSATIONAL\_REACT\_DESCRIPTION,  
    verbose=True,  
    handle\_parsing\_errors=True,  
    memory=memery,  
)  
​  
print("------------查询昨天的新闻---------------")  
result = agent\_executor.invoke("昨天的新闻头条是什么?")  
print("查询结果1：", result)  
​  
# 第二个查询，基于上下文  
print("------------用中文总结---------------")  
result = agent\_executor.invoke("用中文总结下")  
print("总结结果2：", result)  
​  
# 可以继续对话  
result = agent\_executor.invoke("这些新闻对股市有什么影响?")  
print("查询结果3：", result)  
​

#### ****5.4.2 通用方式**（删除）**

通用方式，相较于传统方式，可以提供自定义的提示词模板

**举例1：FUNCATION\_CALL模式**

如果使用的是FUNCTION\_CALL方式，则创建Agent时，推荐使用ChatPromptTemplate

from langchain.memory import ConversationBufferMemory  
from langchain.agents import initialize\_agent, AgentType, create\_tool\_calling\_agent  
from langchain.tools import Tool  
from langchain\_community.tools.tavily\_search import TavilySearchResults  
​  
# 1. 设置API密钥  
os.environ["TAVILY\_API\_KEY"] = "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"  
​  
# 2. 初始化搜索工具  
search\_tool = TavilySearchResults(max\_results=3)  
​  
# 3. 定义提示词模板  
prompt = ChatPromptTemplate.from\_messages([  
   ("system", "你是一个有用的助手，可以回答问题并使用工具。"),  
   ("placeholder", "{chat\_history}"), # 存储多轮对话的历史记录 如果你没有显式传入 chat\_history，Agent 会默认将其视为空列表 []  
   ("human", "{input}"),  
   ("placeholder", "{agent\_scratchpad}")  
])  
​  
# 3. 定义记忆组件(ConversationBufferMemory为例)  
memery = ConversationBufferMemory(  
    memory\_key="chat\_history",  
    return\_messages=True  
)  
​  
# 5. 创建Agent  
agent = create\_tool\_calling\_agent(  
    llm=qw\_model,  
    tools=[search\_tool],  
    prompt=prompt  
)  
​  
# 4. 创建AgentExecutor执行器对象(通过源码可知，memory参数声明在AgentExecutor父类中)  
agent\_executor = AgentExecutor(  
    agent=agent,  
    tools=[search\_tool],  
    verbose=True,  
    memory=memery,  
)  
​  
# 5. 测试对话  
query = "北京的天气是多少？"  
result = agent\_executor.invoke({"input":query})  
print(result)  
​  
# 6. 继续对话  
query2 = "上海呢?"  
result2 = agent\_executor.invoke({"input":query2})  
print(result2)

拓展1：如果ChatPromptTemplate.from\_messages()中没有提供("placeholder", "{chat\_history}")这

样操作，会怎样呢？

拓展2：如果删除AgentExecutor构造方法中的memory参数，又会如何呢？

**举例2：ReAct 模式**

ReAct模式下，创建Agent时，可以使用ChatPromptTemplate、PromptTemplate

from langchain\_core.prompts import PromptTemplate  
from langchain.agents import AgentType, create\_tool\_calling\_agent, AgentExecutor, create\_react\_agent  
from langchain.tools import Tool  
from langchain\_community.tools.tavily\_search import TavilySearchResults  
import os  
​  
# 1. 设置API密钥  
os.environ["TAVILY\_API\_KEY"] = "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"  
​  
# 2. 初始化搜索工具  
search = TavilySearchResults(max\_results=1)  
​  
# 3. 定义template  
template = """  
Assistant is a large language model trained by OpenAI.  
​  
Assistant is designed to be able to assist with a wide range of tasks, from answering simple questions to providing in-depth explanations and discussions on a wide range of topics. As a language model, Assistant is able to generate human-like text based on the input it receives, allowing it to engage in natural-sounding conversations and provide responses that are coherent and relevant to the topic at hand.  
​  
Assistant is constantly learning and improving, and its capabilities are constantly evolving. It is able to process and understand large amounts of text, and can use this knowledge to provide accurate and informative responses to a wide range of questions. Additionally, Assistant is able to generate its own text based on the input it receives, allowing it to engage in discussions and provide explanations and descriptions on a wide range of topics.  
​  
Overall, Assistant is a powerful tool that can help with a wide range of tasks and provide valuable insights and information on a wide range of topics. Whether you need help with a specific question or just want to have a conversation about a particular topic, Assistant is here to assist.  
​  
TOOLS:  
------  
​  
Assistant has access to the following tools:  
​  
{tools}  
​  
To use a tool, please use the following format:  
​  
```  
Thought: Do I need to use a tool? Yes  
Action: the action to take, should be one of [{tool\_names}]  
Action Input: the input to the action  
Observation: the result of the action  
```  
​  
When you have a response to say to the Human, or if you do not need to use a tool, you MUST use the format:  
​  
```  
Thought: Do I need to use a tool? No  
Final Answer: [your response here]  
```  
​  
Begin!  
​  
Previous conversation history:  
{chat\_history}  
​  
New input: {input}  
{agent\_scratchpad}  
"""  
​  
# 3.自定义提示词模版  
prompt = PromptTemplate.from\_template( template)  
​  
​  
# 4. 初始化大模型  
# 使用前面初始化好的 chat  
​  
# 5. 定义记忆组件(以ConversationBufferMemory为例)  
memery = ConversationBufferMemory(  
    memory\_key="chat\_history",  
    return\_messages=True  
)  
​  
# 6.创建Agent对象  
agent = create\_react\_agent(llm=openai\_model, tools=[search], prompt=prompt)  
​  
# 6.创建AgentExecutor执行器  
agent\_executor = AgentExecutor(agent=agent, tools=[search], verbose=True, memory=memery)  
​  
​  
# 7. 测试查询  
result = agent\_executor.invoke({"input": "我的名字叫Bob?"})  
print(result)

继续提问

agent\_executor.invoke({"input": "我的名字叫什么?"})

**举例3：远程获取提示词模版**

以通用方式create\_xxx\_agent的ReAct模式为例，FUNCATION\_CALL一样

远程的提示词模版通过<https://smith.langchain.com/hub/hwchase17>获取

举例：<https://smith.langchain.com/hub/hwchase17/react-chat>，这个模板是专为聊天场

景设计的ReAct提示模板。这个模板中已经有聊天对话键 **chat\_history** 、 **agent\_scratchpad**

from langchain\_core.prompts import PromptTemplate  
from langchain import hub  
from langchain.agents import AgentType, create\_tool\_calling\_agent, AgentExecutor, create\_react\_agent  
from langchain.tools import Tool  
from langchain\_community.tools.tavily\_search import TavilySearchResults  
import os  
​  
# 1. 设置API密钥  
os.environ["TAVILY\_API\_KEY"] = "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"  
​  
# 2. 初始化搜索工具  
search = TavilySearchResults(max\_results=1)  
​  
# 3. 定义template  
prompt = hub.pull("hwchase17/react-chat")  
​  
# 4. 初始化大模型  
# 使用前面初始化好的 chat  
​  
# 5. 定义记忆组件(以ConversationBufferMemory为例)  
memery = ConversationBufferMemory(  
    memory\_key="chat\_history",  
    return\_messages=True  
)  
​  
# 6.创建Agent对象  
agent = create\_react\_agent(llm=qw\_model, tools=[search], prompt=prompt)  
​  
# 6.创建AgentExecutor执行器  
agent\_executor = AgentExecutor(agent=agent, tools=[search], verbose=True, memory=memery)  
​  
​  
# 7. 测试查询  
result = agent\_executor.invoke({"input": "北京明天的天气怎么样"})  
print(result)

## **Section06 Retrieval**

Retrieval直接翻译过来即“检索”，本章RetrievCXal模块包括与检索步骤相关的所有内容，例如数据的获取、切分、向量化、向量存储、向量等模块。常被应用于构建一个“ **企业/私人的知识库** ”，提升大模型的整体能力。

### ****6.1 Retrieval模块的设计意义****

#### ****6.1.1 大模型的幻觉问题****

拥有记忆后，确实扩展了AI工程的应用场景。

但是在专有领域，LLM无法学习到所有的专业知识细节，因此在 **面向专业领域知识** 的提问时，无法给出可靠准确的回答，甚至会“胡言乱语”，这种现象称之为 **LLM的“幻觉”** 。

大模型生成内容的不可控，尤其是在金融和医疗领域等领域，**一次金额评估的错误，一次医疗诊断的失误，哪怕只出现一次都是致命的**。但，对于非专业人士来说可能难以辨识。目前还没有能够百分之百解决这种情况的方案。



**当前大家普遍达成共识的一个方案：**

首先，为大模型提供一定的上下文信息，让其输出会变得更稳定。

其次，利用本章的RAG，将检索出来的 **文档和提示词** 输送给大模型，生成更可靠的答案。

#### ****6.1.2 RAG的解决方案-+JVCXASDF Q****

可以说，当应用需求集中在利用大模型去 **回答特定私有领域的知识** ，且知识库足够大，那么除了 **微调大模型** 外， **RAG** 就是非常有效-0的一种缓解大模型推理的“幻觉”问题的解决方案。

LangChain对这一流程提供了解决方案。

如果说LangChain相当于给LLM这个“⼤脑”安装了“四肢和躯⼲”，RAG则是为LLM提供了接⼊“⼈类知识图书馆”的能⼒。

目前，已经出现了非常多的产品几乎完全建立在 RAG 之上，包括客服系统、基于大模型的数据分析，以及成千上万的数据驱动聊天应用，应用场景五花八门。

#### ****6.1.3 RAG的优缺点****

**RAG的优点**

1）相比提示词工程，RAG有 **更丰富的上下文和数据样本** ，可以不需要用户提供过多的背景描述，就能生成比较符合用户预期的答案。

2）相比于模型微调，RAG可以提升问答内容的 **时效性** 和 **可靠性**

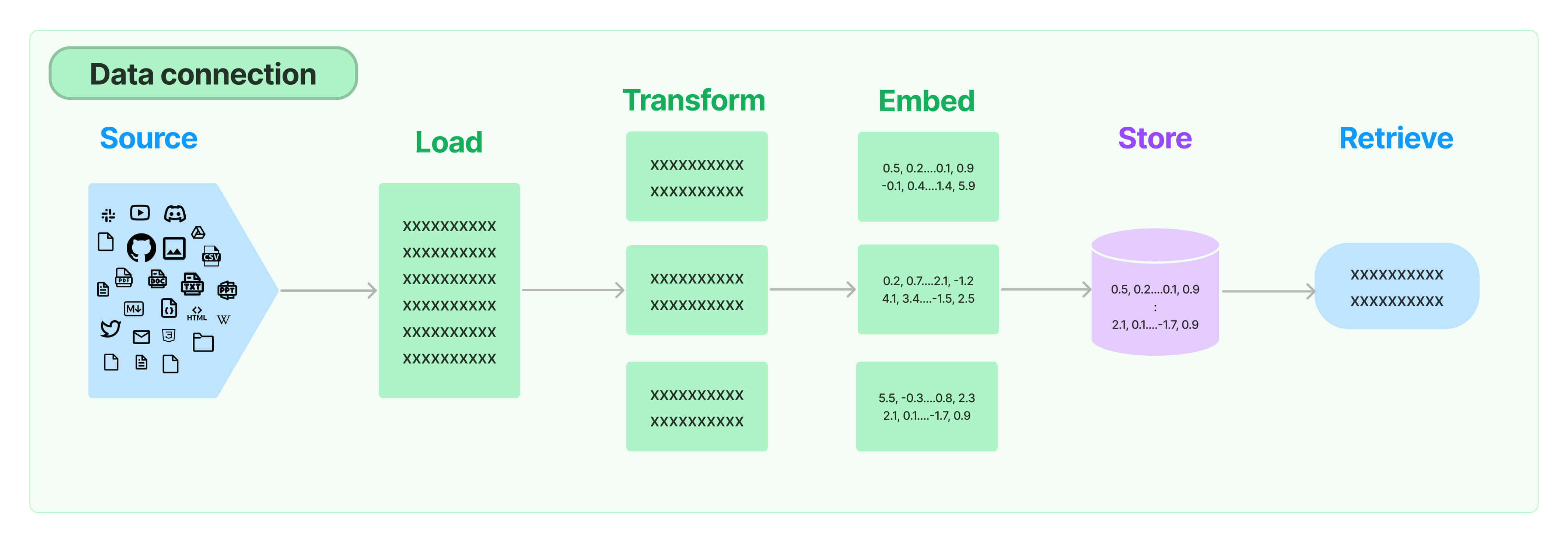
3）在一定程度上保护了业务数据的 **隐私性** 。

**RAG的缺点**

1）由于每次问答都涉及外部系统数据检索，因此RAG的 **响应时延** 相对较高。

2）引用的外部知识数据会 **消耗大量的模型Token** 资源。

#### ****6.1.4 Retrieval流程****



**环节1：Source（数据源）**

指的是RAG架构中所外挂的知识库。这里有三点说明：

1、原始数据源类型多样：如：视频、图片、文本、代码、文档等

2、形式的多样性：

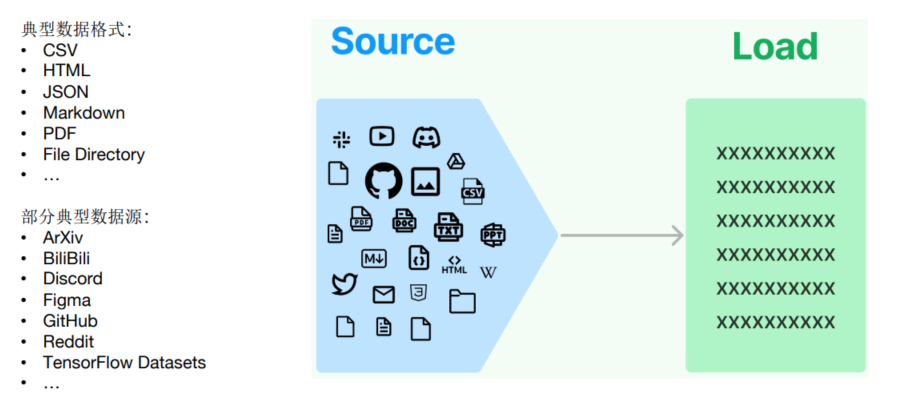
* 可以是上百个.csv文件，可以是上千个.json文件，也可以是上万个.pdf文件
* 可以是某一个业务流程外放的API，可以是某个网站的实时数据等

**环节2：Load（加载）**

文档加载器（Document Loaders）负责将来自不同数据源的非结构化文本，加载到 **内存** ，成为 **文档(Document)对象** 。

文档对象包含 **文档内容** 和相关 **元数据信息** ，例如TXT、CSV、HTML、JSON、Markdown、PDF，甚至

YouTube 视频转录等。



文档加载器还支持“ **延迟加载** ”模式，以缓解处理大文件时的内存压力。

文档加载器的编程接口使用起来非常简单，以下给出加载TXT格式文档的例子。

from langchain.document\_loaders import TextLoader  
​  
text\_loader = TextLoader("./test.txt")  
​  
docs = text\_loader.load()  
print(docs)

**环节3：Transform（转换）**

**文档转换器(Document Transformers)** 负责对加载的文档进行转换和处理，以便更好地适应下游任务的需求。

文档转换器提供了一致的接口（工具）来操作文档，主要包括以下几类：

**文本拆分器(Text Splitters)** ：将长文本拆分成语义上相关的小块，以适应语言模型的上下文窗口限制。

* **冗余过滤器(Redundancy Filters)** ：识别并过滤重复的文档。
* **元数据提取器(Metadata Extractors)** ：从文档中提取标题、语调等结构化元数据。
* **多语言转换器(Multi-lingual Transformers)** ：实现文档的机器翻译。
* **对话转换器(Conversational Transformers)** ：将非结构化对话转换为问答格式的文档。

总的来说，文档转换器是 LangChain 处理管道中非常重要的一个组件，它丰富了框架对文档的表示和操作能力。

在这些功能中，文档拆分器是必须的操作。下面单独说明。

**环节3.1：Text Splitting（文档拆分）**

* **拆分/分块的必要性** ：前一个环节加载后的文档对象可以直接传入文档拆分器进行拆分，而文档切块后才能 向量化 并存入数据库中。
* **文档拆分器的多样性** ：LangChain提供了丰富的文档拆分器，不仅能够切分普通文本，还能切分Markdown、 JSON、HTML、代码等特殊格式的文本。
* **拆分/分块的挑战性** ：实际拆分操作中需要处理许多细节问题， **不同类型的文本** 、 **不同的使用场景** 都需要采用不同的分块策略。

可以按照 **数据类型** 进行切片处理，比如针对 **文本类数据** ，可以直接按照字符、段落进行切片； **代码** **类数据** 则需要进一步细分以保证代码的功能性；

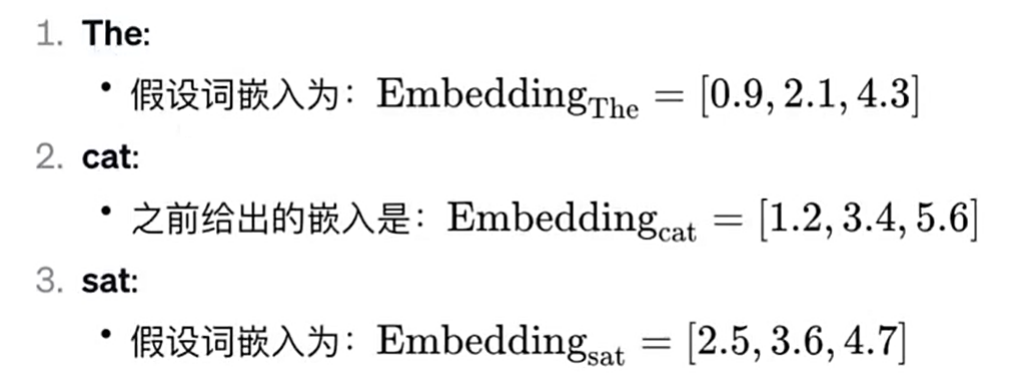
可以直接根据 **token** 进行切片处理

**在构建RAG应用程序的整个流程中，拆分/分块是最具挑战性的环节之一，它显著影响检索效果**。目前还没有通用的方法可以明确指出哪一种分块策略最为有效。不同的使用场景和数据类型都会影响分块策略的选择。

**环节4：Embed（嵌入）**

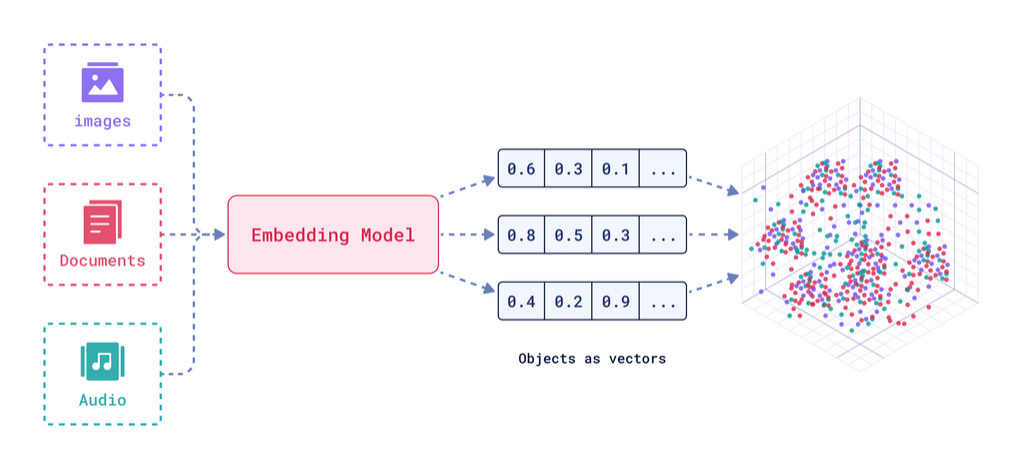
文档嵌入模型（Text Embedding Models）负责将 **文本** 转换为 **向量表示** ，即**模型赋予了文本计算机可\***\*理解的数值表示\*\*，使文本可用于向量空间中的各种运算，大大拓展了文本分析的可能性，是自然语言处理领域非常重要的技术。

举例：



实现原理：通过 **特定算法** （如Word2Vec）将语义信息编码为固定维度的向量，具体算法细节需后续深入。

关键特性：**相似的词在向量空间中距离相近**，例如"猫"和"犬"的向量夹角小于"猫"和"汽车"。



文本嵌入为 LangChain 中的问答、检索、推荐等功能提供了重要支持。具体为：

**语义匹配** ：通过计算两个文本的向量余弦相似度，判断它们在语义上的相似程度，实现语义匹配。

**文本检索** ：通过计算不同文本之间的向量相似度，可以实现语义搜索，找到向量空间中最相似的文本。

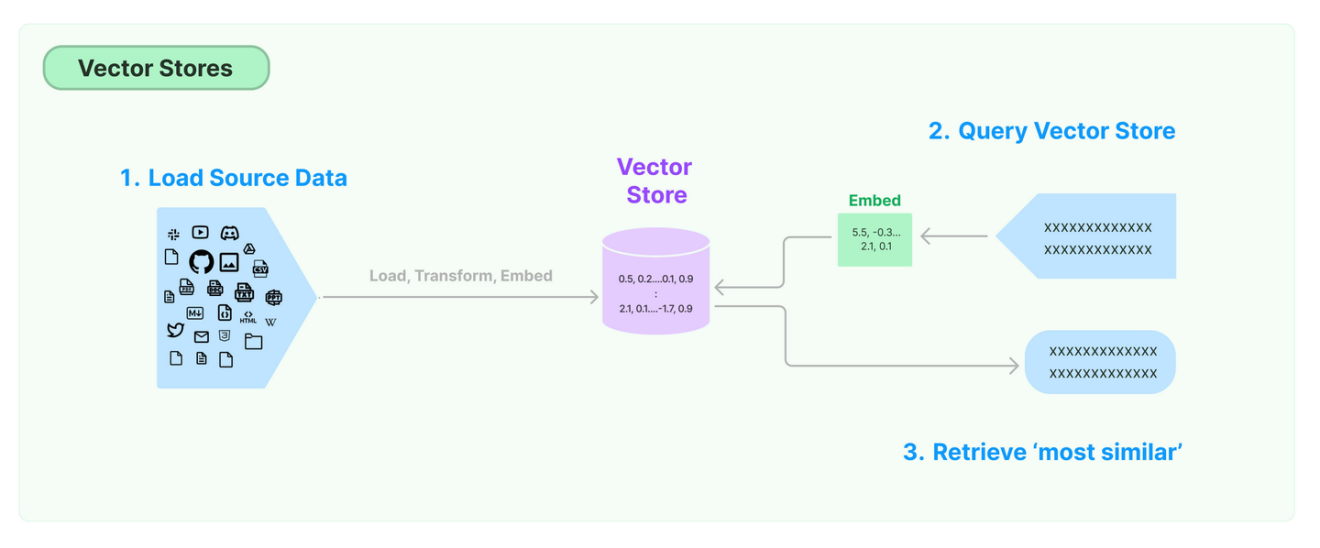
**信息推荐** ：根据用户的历史记录或兴趣嵌入生成用户向量，计算不同信息的向量与用户向量的相似度，推荐相似的信息。

**知识挖掘** ：可以通过聚类、降维等手段分析文本向量的分布，发现文本之间的潜在关联，挖掘知识。

**自然语言处理** ：将词语、句子等表示为稠密向量，为神经网络等下游任务提供输入。

**环节5：Store（存储）**

LangChain 还支持把文本嵌入存储到向量存储或临时缓存，以避免需要重新计算它们。这里就出现了数据库，支持这些嵌入的高效 **存储** 和 **搜索** 的需求。



**环节6：Retrieve（检索）**

检索器（Retrievers）是一种用于 **响应非结构化查询** 的接口，它可以返回符合查询要求的文档。

LangChain 提供了一些常用的检索器，如 **向量检索器** 、 **文档检索器** 、 **网站研究检索器** 等。

通过配置不同的检索器，LangChain 可以灵活地平衡检索的精度、召回率与效率。检索结果将为后续的问答生成提供信息支持，以产生更加准确和完整的回答。