2. Langchain 入门指南

## 2.1 初识Langchain

2023年注定是人工智能领域不平凡的一年，随着人工智能领域的飞速发展，开发者们都在寻找能够轻松、高效地构建应用的工具。尤其对于那些不熟悉大语言模型领域，或者初入此领域的开发者们来说，选择一个合适的工具尤为重要。在众多的选择中，有一个名字越来越受到大家的关注：LangChain。但为什么我们需要 LangChain 呢？

### 2.1.1 为什么需要 Langchain

让我们首先想象一个开发者在构建一个LLM应用时的常见场景。当你开始一个新项目，你可能会遇到一堆的API接口、数据格式和工具。对于一个非AI领域的开发者，要去研究每一个工具、每一个接口都是一个巨大的负担。现在，假设你要构建一个涉及到语言处理的应用，比如一个智能聊天机器人。你可能会想：我难道要一步步去学习如何训练一个语言模型、如何处理各种数据、还要解决所有的兼容性问题吗？

这就是 LangChain 的价值所在。LangChain 是一个集成框架，它为开发者提供了一系列的工具和组件，使得与各种数据（如 Google Analytics、Stripe、SQL、PDF、CSV等）的连接、语言模型的应用和优化变得简单直接。其实，LangChain 就好比是一个“瑞士军刀”，你不再需要为每一个任务找一个新工具，它提供了一站式的解决方案。这正如你要修理一个小小的家用电器，而拥有一个完整的工具箱。不管你遇到什么问题，打钉子、拧螺丝、剪线，工具箱里总有一个合适的工具等着你。LangChain 就为你提供了这样的工具箱，不仅涵盖了基础工具，还为个性化需求提供了自定义组件的解决方案。

现在，随着LangChain在开发者社区中的受欢迎程度逐渐上升，我们可以明显地看到使用LangChain的开发者数量呈现直线激增的趋势。在 2023 年 8 月，LangChain 这个开源框架已经收获了惊人的数字：其中包括5.82万的星标、557位的专注开发者以及7800位积极的分支开发者。这些数字深层次地代表了众多开发者对LangChain实用性和未来潜力的坚定认可。

正是因为LangChain连接了开发者和复杂的LLM应用，使得开发变得更为简单、高效。也因为这种受欢迎程度和媒体报道的广泛传播，使得越来越多的开发者，不论是LLM领域的还是非LLM领域的，都选择使用 LangChain。

但是，尽管LangChain获得了广泛的欢迎和认可，也并非没有争议。在开发者的热议中，也悄悄夹杂着一丝疑虑和担忧。硅谷的评论家甚至毫不客气地将其称为“玩具”。这一标签立刻在开发者社区内引起了广泛关注和讨论。影响之深，以至于部分开发者甚至开始公然表示，他们决定放弃继续使用LangChain。更使人困惑的是，当我们深入探究是否真的需要LangChain来开发大型语言模型，答案似乎并不明确。事实上，即使不使用LangChain，开发者们同样可以构建LLM应用程序。

这种反差强烈的声音，给开发者社区带来了很大的困扰，我们为什么需要LangChain。一方面，LangChain的数字和受欢迎程度无疑证明了其潜在价值和优越性；另一方面，那些质疑和冷嘲热讽使得人们不得不重新考虑自己的技术选择。然而，无论未来如何，目前LangChain已经为开发者提供了便利和效率。对于是否将其纳入长期的开发工具，每个开发者都需要基于自己的需求和经验来做出决策。但如何形成这样的经验和认知呢？答案很简单：真正的理解始于亲自的尝试和体验。只有当开发者亲自探索LangChain框架，才能真正感受到其背后的设计哲学和可能的局限性。通过这种深入的交互和实践，开发者可以从实际的使用中发现LangChain的真正价值，而不仅仅是从表面的描述或他人的评论中获得印象。

接下来，为了帮助开发者们更全面地了解LangChain，我们将详细介绍其三大使用场景和核心模块。这不仅是为了证明LangChain不是所谓的“玩具”，更是希望开发者能够从中发现其对自己项目的实际价值，从而作出更加明智的技术选择。对于那些关于使用LangChain出现的问题和质疑，我们在后面会列出这些具体问题进行分类分析且罗列一些有经验的开发者对新人开发者的使用LangChain的建议。

### 2.1.2 LLM应用开发的最后1公里

想象一下，一个对编程完全陌生的初学者，面临着如何与模型进行交互的诸多问题，哪怕是简单的GET或POST请求，都可能成为其开发路上的第一道门槛。而Langchain的存在，恰恰是为了打破这道门槛，使得LLM应用开发变得触手可及。

首先，Langchain的简洁性让它脱颖而出。开发者只需几行代码，就能运行一个大型LLM程序，甚至快速构建一个响应式的机器人。这种简便性意味着，无论是有经验的开发者还是初入此领域的新手，Langchain都为他们进入LLM应用开发世界，铺设了道路。

Langchain 还为开发者集成了丰富的内置链组件，为开发者解决了重复编写代码的问题。面对特定的任务，如摘要或问答，Langchain提供了专门的摘要链和问答链，简化了开发流程。Agent的引入，将工具和数据库的整合提升到一个新的层次，使得开发者可以全心投入任务，而无需为如何与各种工具或数据库互动而感到困扰。

除了通过Langchain，开发者能实现LLM与真实世界的在线数据增强。还允许开发者在私有环境中部署模型，或是针对特定任务选择更精确的模型平台及型号。

而对于那些未选择利用Langchain的开发者，他们很可能会被各模型平台的接口、提示词的编写以及输出格式处理等问题所困扰，这些复杂性会成为开发过程中的巨大障碍，甚至导致“从入门到放弃”。

在LLM应用开发中，一个经常被遗漏但至关重要的环节是如何为LLM编写合适的提示词，确保LLM能够准确执行开发者的意图。对于许多开发者，特别是初学者，这可能是一个具有挑战性的任务。然而，Langchain为这一问题提供了有力的解决方案。

对于那些在模型提示词编写上感到困惑的开发者，Langchain提供了多种模板供选择。这并不仅仅是一些随意整合的模板，而是与各种应用、工具紧密集成的组件，其中包含了大量已经经过实际验证的提示词模板。这意味着开发者无需从零开始编写，只需在Langchain提供的模板中找到与其任务相匹配的部分，并进行相应的调整。

以SQL为例，这是一个对许多开发者来说相对熟悉，但在与LLM结合时可能仍然有困惑的领域。如果一个开发者刚开始接触如何为SQL编写提示词，他可以轻松地在Langchain中找到SQL组件的提示词模板。这些模板包括了如何编写语法正确的PostgreSQL查询、如何查看查询结果以及如何返回针对输入问题的答案。更进一步，Langchain提供的模板也包括了各种查询的最佳实践，例如按照PostgreSQL查询最多的结果限制、正确使用列名、注意使用当前日期的函数等等。

例如，Langchain提供了以下格式化SQL提示词模板（翻译）：

1 你是一个PostgreSQL专家。给定一个输入问题，首先创建一个语法正确的PostgreSQL查询来运行，然后查看查询的结果，并返回对输入问题的答案。

2 除非用户在问题中明确指定要获得的特定数量的示例，否则使用LIMIT子句按照PostgreSQL查询最多{top\_k}的结果。你可以对结果进行排序，以返回数据库中的最有信息的数据。

3 绝不要查询表中的所有列。你只能查询回答问题所需的列。用双引号(")将每个列名包起来，表示它们是界定的标识符。

4 注意只使用你在下面的表中可以看到的列名。小心不要查询不存在的列。此外，注意哪一列在哪个表中。

5 如果问题涉及“今天”，请注意使用CURRENT\_DATE函数获取当前日期。

6

7 使用以下格式：

8

9 问题：这里的问题

10 SQL查询：要运行的SQL查询

11 SQL结果：SQL查询的结果

12 答案：这里的最终答案

13

14 只使用以下表格：

15

16 {table\_info}

17

18 问题：{input}

想象一下，如果没有Langchain提供的这个提示词模板，当你要开始进行一个SQL查询业务逻辑的代码编写，会走多少弯路？

### 2.1.3 Langchain 的2个关键词

在现代软件工程中，如何将庞大、复杂的系统划分为更小、更易于管理和使用的部分，已经成为设计和开发的核心考量。在这背景下，LangChain以“组件”和“链”作为其关键概念，为LLM应用的开发者带来了便利。LangChain的2个关键词是“组件”和“链”。

首先，来谈谈“组件”。在LangChain中，组件不仅仅是代码的拼凑，而是代表了一个具有明确功能和用途的单元。这些组件包括LLM模型包装器、聊天模型包装器以及与数据增强相关的一系列工具和接口。这些组件就是LangChain中的核心，你可以把它们看作数据处理的流水线上的各个工作站。每个组件都有其特定的职责，如处理数据的输入输出，或转化数据格式。

然而，单纯的组件还不足以满足复杂应用的需求，这时“链”的概念便显得尤为关键。在LangChain的体系中，“链”是将各种组件连接在一起的纽带，它能够确保组件之间的无缝集成和在程序运行环境中的高效调用。无论是针对LLM还是其他工具，链都扮演着至关重要的角色。举个例子，LLMChain，这是LangChain中最常用的链组件，它可以整合LLM模型包装器和记忆组件，就能让聊天机器人拥有“记忆”。

值得一提的是，LangChain并没有止步于提供基础的组件和链。反之，它进一步为这些核心概念提供了标准的接口，并与数据处理平台及实际应用工具紧密集成。这样的设计不仅强化了LangChain与其他数据平台和实际工具的连接，也确保开发者在一个开放且友好的环境中轻松地进行LLM应用开发。

以最常见的聊天机器人场景为例，为了在各种场景中为用户提供自然、流畅的对话体验，我们的聊天机器人需具备多种功能，包括与用户进行日常聊天、回答关于天气的询问以及实时搜索。这一设计目标意味着我们要处理的任务范围从简单的日常对话到复杂的信息查询都有，因此，一个结构化、模块化的设计方法是必要的。

在此背景下，LangChain的“组件”和“链”的设计概念为我们提供了极大的帮助。利用LangChain的组件，我们可以为聊天机器人设计不同的模块，如与用户日常交流的模块、获取天气信息的模块以及进行实时搜索的模块。每个组件都具备特定的功能，并专门处理与之相关的任务。例如，当需要回答关于天气的问题时，机器人可以调用“天气信息组件”来获取所需数据。

但是，单纯的组件无法满足整个机器人的运作。为了确保这些组件之间可以协同工作并为用户提供连贯的体验，我们需要LangChain的“链”来整合这些组件。例如，当用户询问一个涉及多个组件的问题时（如：“今天天气怎么样，同时告诉我近期的新闻头条”），LangChain的链就可以确保“天气信息组件”和“新闻搜索组件”协同工作，为用户提供完整的回答。

具体来说，当用户提出问题时，LangChain提供的API允许机器人执行以下操作：(1) 请求LLM解释用户的输入，并根据输入内容生成对应的查询请求，这可能涉及一个或多个组件；(2) 根据生成的查询，激活对应的组件以获取必要的数据或信息；(3) 利用LLM生成自然语言的回答，将各组件的结果整合成为用户可以理解的回复。

通过这种方式，开发者无需深入每个复杂的处理细节，只需利用LangChain的API将用户的问题输入，并将得到的答案呈现给用户。这不仅使聊天机器人能够提供丰富的信息服务，而且确保AI自然而然地融入人们的日常生活，达到我们的设计初衷。

### 2.1.4 Langchain 的3个场景

LangChain正在重新定义LLM应用的开发方法，尤其是在问答系统、数据处理与管理以及自动问答与客服机器人三个场景。以下是对LangChain在这三个场景中作用的进行分析。

第一个场景是问答系统。问答系统已经成为许多LLM应用的重要组成部分，从简单的搜索工具到复杂的知识库查询工具。LangChain在这方面展现了其出色的能力。当开发者面临需要从长篇文本或特定数据源中提取信息的挑战时，LangChain可以轻松地与这些外部数据源交互，迅速提取关键信息，然后进行生成步骤，以产生准确的答案。

第二个场景是数据增强与生成。在数据驱动的时代，如何有效地处理和管理数据是关键。LangChain不仅为数据转换提供了方便的工具，还使得从一个格式到另一个格式的数据转换变得更为直观。开发者可以轻松地利用LangChain与外部数据源进行交互，满足各种数据处理需求，包括文本摘要、数据分类和数据结构的调整。比如输出解析器和提示词模板都是LangChain 提供的数据组件。提示词模板让开发者构建提示词，输出解析器可以使得开发者轻松管理LLM输出的数据格式。

第三个场景是客服机器人。在许多在线平台上，客服机器人已经成为用户与公司之间的首要交互点。利用LangChain，开发者成功构建了能够实时响应用户查询的客服机器人。这种实时响应得益于LangChain的“Agent”功能，其中涉及到LLM进行决策、采取行动，并根据反馈不断优化交互过程。这样的设计使客服机器人不仅能够及时响应，还能提供更加精确的信息或解决方案。

LangChain已经在这三个关键场景中展现了其强大的潜力，为开发者提供了实用而又强大的工具，使其可以更加高效地满足各种开发需求。

### 2.1.5 Langchain 的6大模块

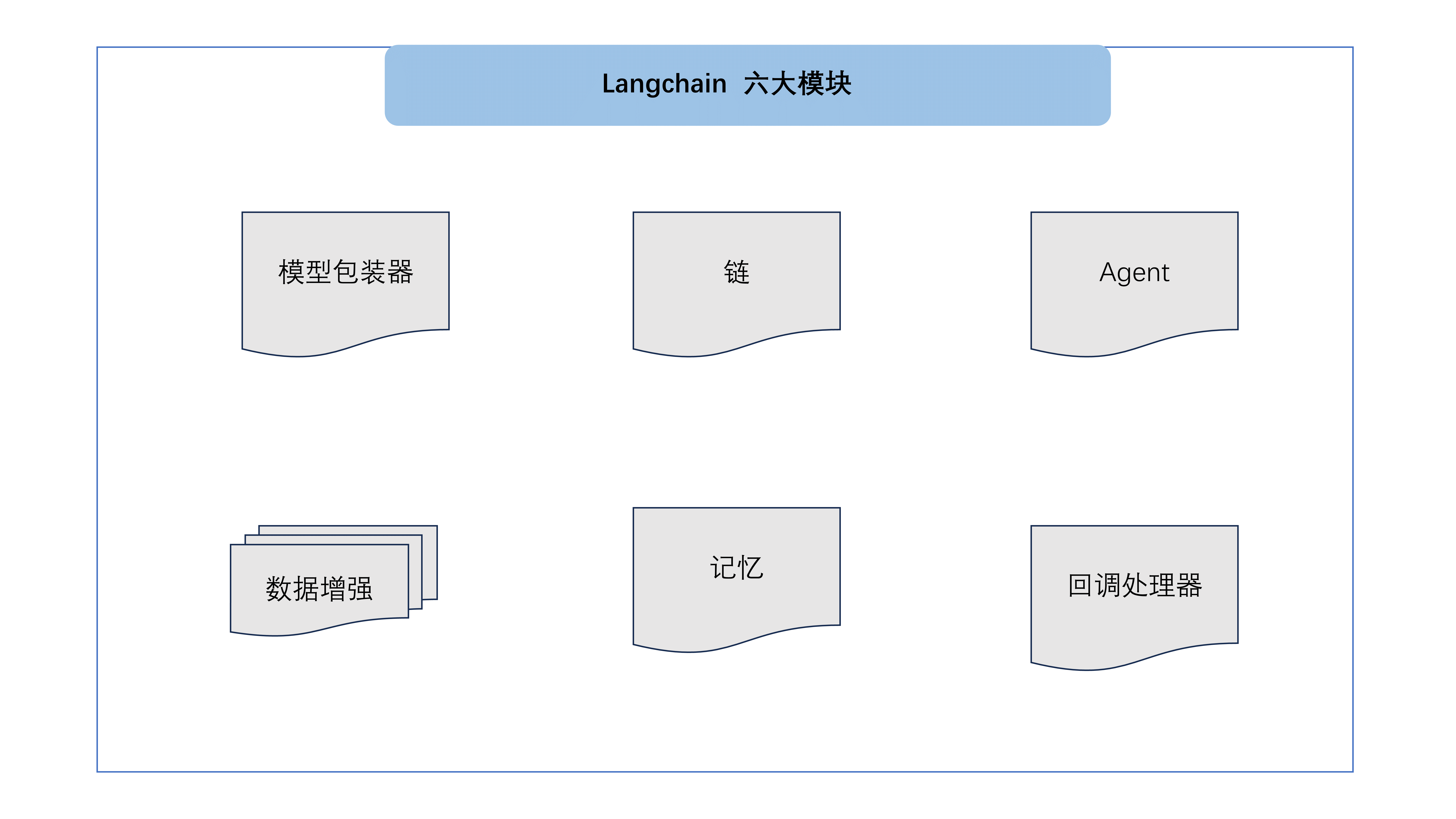
针对LLM应用开发者的需求，Langchain推出了六大核心模块。这些模块覆盖了从模型I/O到数据增强，从链到记忆，以及从Agent模块的推理决策到回调处理器的实时监控等全方位功能。借助这些模块中的包装器和组件，开发者能够更为简便地搭建和推出高级的LLM应用。

图 2‑1

1. 模型包装器（Model IO）：对于任何语言模型应用，其核心无疑是模型自身。LangChain提供了与任何语言模型接口的模型包装器，分为LLM和聊天模型包装器（Chat Model）。它的提示词模板功能使得开发者可以模板化、动态选择和管理模型输入。通过提供统一的语言模型接口，模型包装器简化了大语言模型的调用。此外，其输出解析器也帮助开发者从模型输出中提取所需的信息。

2. 数据增强（Data Connection）：许多LLM应用需要的用户特定数据并不在模型的训练集中。LangChain提供了加载、转换、存储和查询数据的构建块。开发者可以利用文档加载器从多个来源加载文档，通过文档转换器进行文档分割、转换等操作。矢量存储和数据检索工具则提供了对嵌入数据的存储和查询功能。

3. 链（Chain）：单独使用LLM对于简单应用可能是足够的，但对于更复杂的应用，往往需要将多个LLM或其他组件进行链式连接。LangChain为此类“链式”应用提供了Chain接口。

4. 记忆（Memory）：大部分的LLM应用都有一个对话式的界面，能够引用之前对话中的信息是至关重要的。LangChain提供了多种工具，帮助开发者为系统添加记忆功能。它们可以独立使用，也可以无缝集成到链中。记忆系统需要支持两个基本操作：读取和写入。在每次运行中，链首先从记忆系统读取，然后在执行核心逻辑后将当前运行的输入和输出写入记忆，以供未来引用。

5. Agent：代理的核心思想是利用LLM选择要采取的操作序列。在链中，操作序列是硬编码的，而在代理中，语言模型被用作推理引擎，确定哪些操作以及它们的执行顺序。

6. 回调处理器（Callback）：LangChain提供了一个回调系统，允许开发者在LLM应用的各个阶段中进行干预。这对于日志记录、监视、流处理等任务非常有用。通过整个API中提供的callbacks参数，开发者可以订阅这些事件。

## 2.2 LangChain的开发流程

为了更深入地理解LangChain的开发流程，我们以聊天机器人为例，展示如何将其实现。

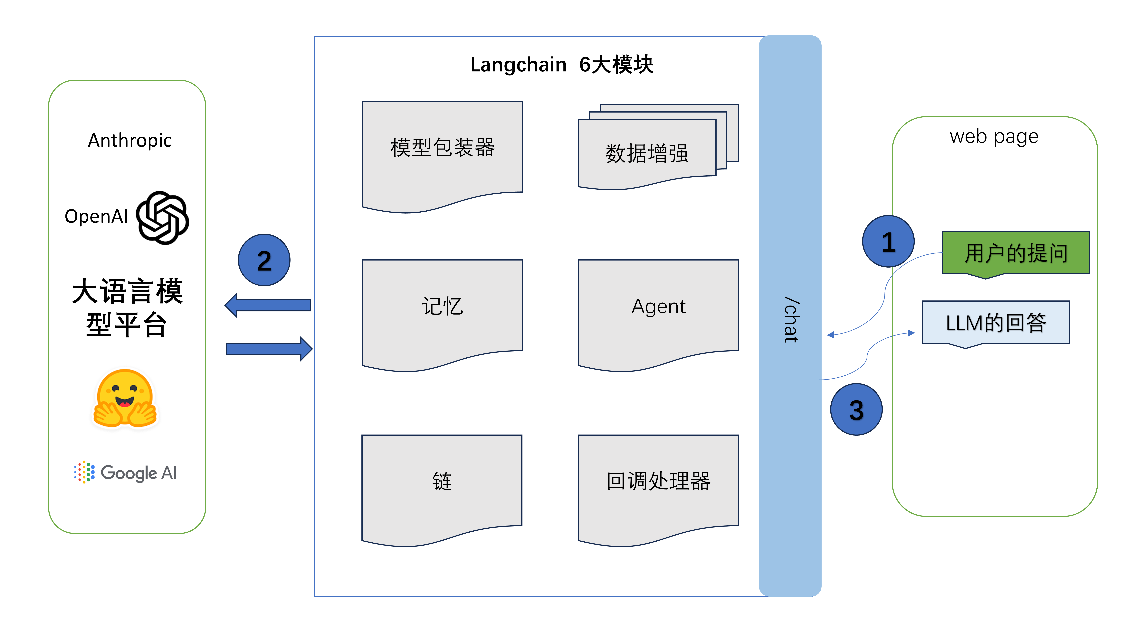


图 2‑2

作为一个例子，图2-2说明了一个设计为聊天机器人的LLM应用程序。它提供了一个聊天机器人的网页应用程序。除了包括客户端浏览器、Web服务器应用逻辑等传统组件外，这个应用程序的架构还引入了两个额外的组件：一个LLM集成中间件（图2-2中间端），如Langchain，和一个大语言模型（图2-2的左侧端）。中间件提供一个API，业务逻辑控制器调用它以启用聊天机器人功能。要使用的具体LLM是基于配置决定的。当用户提交一个问题时(步骤1)，聊天机器人控制器代码调用Langchain API（由Langchain 6大模块设置的接口），它在内部与LLM(步骤2)交互，由LLM来理解解释问题并生成一个用户的回答(步骤3)，显示在终端用户的聊天界面上（图2-2右侧端）。

清单1展示了如何使用Langchain和大语言模型平台OpenAI的gpt-3.5-turbo-0613语言模型实现聊天机器人业务逻辑。这段Python代码首先创建了ChatOpenAI类的实例(代表GPT-3.5 聊天模型包装器)。4-9行在路径 ‘/chat’上建立了一个POST端点，可以利用FastAPI库。当用户向聊天机器人助手提交一个问题时，chat函数就会被触发，请求对象在其输入属性中封装了用户的问题。为了处理请求，代码 (第7行)，实例化了一个LLMChain对象，它接收了一个聊天模型包装器 llm 和一个提示词模板 prompt，实现了一个Langchain的内置预配置聊天机器人，可以与终端用户交互(第7行)。在第8行处理用户的问题：运行LLMChain链组件，接收提出的问题作为输入，并返回gpt-3.5-turbo-0613语言模型生成的响应。这个响应持有对用户问题的答案，并在第9行发回给用户。

1 llm = ChatOpenAI( # LLM initialization parameters

2 model\_name="gpt-3.5-turbo-0613", openai\_api\_key="你的密钥" ↩→ ↩→↩→↩→→→→→→, temperature=0.9)

3 \_prompt = """ 你是一个发言友好的AI助理。请现在回答用户的提问：{question}。 """

4 @app.post("/chat") # Chatbot controller URL endpoint

5 async def chat (request):

6 prompt = PromptTemplate.from\_template(\_prompt)

7 chat\_chain = LLMChain(llm=llm,prompt=prompt)

8 response = chat\_chain(request.input) # 终端用户的提问字符串

9 return {"response": response["text"]}

清单1

### 2.2.1 Hello World：取名大师

在LLM应用的开发领域，LangChain为开发者带来了前所未有的可能性。这里深入探讨如何利用LangChain和OpenAI的组合，开发出一个独特的程序——“取名大师”。

安装和基础配置

首先，为了能够顺利进行开发工作，需要确保机器上安装了相应的Python包。开发者可以通过以下命令轻松完成安装：

pip install openai langchain

每一个与API交互的应用都需要一个API密钥。开发者可以创建一个账户并获取密钥，然后为了保障API密钥的安全性，最佳实践是将其设置为环境变量：

export OPENAI\_API\_KEY="你的API密钥"

但是，如果开发者不熟悉如何设置环境变量，也可以直接在初始化OpenAI LLM类时传入密钥：

from langchain.llms import OpenAI

llm = OpenAI(openai\_api\_key="你的API密钥")

取名大师的行动

有了这些基础设置，接下来就可以利用LLM进行实际的编程工作了。想象一下，有一个程序可以基于用户的描述来为公司、产品或项目提供创意的命名建议。比如，当问及“为一家制作多彩袜子的公司起个名字怎么样？”：

llm.predict("What would be a good company name for a company that makes colorful socks?")

Feetful of Fun 这个名字听起来不错。如此，一个简洁的、能提供有创意名称建议的程序就这样诞生了。

# 输出：Feetful of Fun

### 2.2.2 创建你的第一个聊天机器人

为了在各种场景中为用户提供自然、流畅的对话体验，我们的聊天机器人旨在满足多种需求。考虑一个典型的场景：用户早上打开机器人界面，首先是打招呼“早上好”，随后询问“今天天气怎么样？”并在结束对话前问：“最近有什么热门新闻吗？”这样的场景要求我们的机器人不仅具备与用户日常聊天的能力，还能即时回应关于天气的询问并进行实时的新闻搜索。

面对从简单的日常对话到复杂的信息查询的多重任务，我们需要一个强大而灵活的工具来支持。因此，我们选择依赖于像LangChain的组件和链来实现这些功能。

以刚才的场景为例，当用户询问天气或需要搜索新闻时，LangChain提供的API允许机器人轻松处理这些任务：

(i) 它首先请求LLM解释用户的输入（例如“今天天气怎么样？”），并根据这些输入为其生成一个辅助的查询请求。这里我们可以用到的组件是聊天模型包装器，LLMChain 链组件，或者是设置一个Agent；

(ii) 根据这个查询请求，它会从天气服务中获取相关的数据或从新闻数据库中搜寻相关内容。通过LangChain 的内置搜索工具，可以查询天气和新闻；

(iii) 最后，它请求LLM基于获得的数据为用户生成一个自然语言的回答，例如：“今天是晴天，温度约为35°C。关于热门新闻，最近国际上主要关注的是...”。

这意味着，开发者无需为每一个步骤编写复杂的后台代码。通过LangChain的组件和链，只需简单地将用户的问题输入，再将LangChain返回的答案直接传递给用户。这种方式不仅大大简化了开发流程，还确保了机器人为用户提供自然、丰富的信息服务。

环境配置和设置密钥

首先，我们需要安装他们的 Python 包：

pip install openai langchain

访问 API 需要一个 API 密钥，你可以通过创建一个账户并访问此处获得。一旦我们得到密钥，我们会想要将其设置为环境变量，通过运行：

export openai\_api\_key=""

LangChain 的 schema 定义了 AIMessage、HumanMessage 和 SystemMessage 这三种角色类型的数据模式。这些都是我们设计的数据模型，通过这些模型，我们可以像使用函数一样将参数传递给它们。

例如，如果我们想要与聊天机器人进行对话，我们只需要把想要说的话用 HumanMessage 函数封装起来，像这样：HumanMessage(content="你好!")。然后我们将这个消息放入一个列表中，传递给聊天模型包装器 ChatOpenAI。这样，我们就可以开始与聊天机器人进行交流了。如果我们想要使用这个聊天机器人来翻译一段英文为法文，我们可以这样编写代码：

from langchain.chat\_models import ChatOpenAI

from langchain.schema import (

    AIMessage,

    HumanMessage,

    SystemMessage

)

chat = ChatOpenAI(temperature=0)

chat.predict\_messages([HumanMessage(content="Translate this sentence from English to French. I love programming. ")])

在这段代码中，我们首先导入了需要的模块和函数。然后，我们创建了一个 ChatOpenAI 对象，并且设置了温度参数为 0，这意味着模型的输出将会具有更低的随机性。之后，我们调用 chat.predict\_messages 方法，向它传递了一个包含 HumanMessage 的消息对象的列表。这个 HumanMessage 对象包含了我们想要翻译的英文句子。最后，我们的模型将返回一个 AIMessage 对象，它包含了这句英文的法文翻译。I love programming 翻译为法文：J'aime programmer.

AIMessage(content="J'aime programmer.", additional\_kwargs={})

提示词模板

提示词模板是一种特殊的文本，它可以为特定任务提供额外的上下文信息。在LLM应用中，通常并不直接将用户的输入传递给LLM，而是将用户输入添加到一个更大的文本中，即提示词模板。提示词模板为当前的具体任务提供了额外的上下文信息，这能够更好地引导模型生成预期的输出。

在LangChain中，我们可以使用MessagePromptTemplate来创建提示词模板。我们可以从一个或多个MessagePromptTemplates创建一个ChatPromptTemplate。示例代码如下：

from langchain.prompts.chat import (

    ChatPromptTemplate,

    SystemMessagePromptTemplate,

    HumanMessagePromptTemplate,

)

template = "You are a helpful assistant that translates {input\_language} to {output\_language}."

system\_message\_prompt = SystemMessagePromptTemplate.from\_template(template)

human\_template = "{text}"

human\_message\_prompt = HumanMessagePromptTemplate.from\_template(human\_template)

chat\_prompt = ChatPromptTemplate.from\_messages([system\_message\_prompt, human\_message\_prompt])

chat\_prompt.format\_messages(input\_language = "English", output\_language = "French", text = "I love programming.")

在上述代码中，我们首先定义了两种模板：一个是系统消息模板，描述了任务的上下文（翻译助手的角色和翻译任务）；另一个是人类消息模板，这将会是用户的输入。

然后，我们使用ChatPromptTemplate。from\_messages方法，将这两个模板结合起来，生成了一个聊天提示词模板。

当我们需要生成预期的输出时，我们可以调用ChatPromptTemplate的format\_messages方法，获得的数据结果：

[

    SystemMessage(content = "You are a helpful assistant that translates English to French.", additional\_kwargs ={}),

    HumanMessage(content = "I love programming.")

]

通过这种方式，我们不仅可以生成预期的输出，还能让用户无需担心自己是否擅长写提示词获得有意义的LLM回答，他们只需要提供具体的任务即可。

创建第一个链

现在，让我们将上述步骤整合为一条链，以此创建我们的第一个链。我们将使用LangChain的LLMChain（大语言模型链）对模型进行包装，实现与提示词模板类似的功能。这种方式更为直观易懂，你会发现我们导入了一个包装链 LLMChain， 将提示词模板和模型传递进去后，我们就造好了链。链的运行可以是函数式调用，也可以run 一下。以下是相关代码：

from langchain import LLMChain

from langchain.chat\_models import ChatOpenAI

from langchain.prompts.chat import (

    ChatPromptTemplate,

    SystemMessagePromptTemplate,

    HumanMessagePromptTemplate,

)

# 初始化 ChatOpenAI 聊天模型，温度设置为 0

chat = ChatOpenAI(temperature = 0)

# 定义系统消息的模板

template = "You are a helpful assistant that translates {input\_language} to {output\_language}."

system\_message\_prompt = SystemMessagePromptTemplate.from\_template(template)

# 定义人类消息的模板

human\_template = "{text}"

human\_message\_prompt = HumanMessagePromptTemplate.from\_template(human\_template)

# 将这两种模板组合到聊天提示词模板中

chat\_prompt = ChatPromptTemplate.from\_messages([system\_message\_prompt, human\_message\_prompt])

# 使用 LLMChain 包装模型和提示词模板

chain = LLMChain(llm = chat, prompt = chat\_prompt)

# 运行模型链，传入参数

chain.run(input\_language = "English", output\_language = "French", text = "I love programming.")

在这段代码中，我们首先初始化了一个ChatOpenAI聊天模型，然后定义了系统消息模板和人类消息模板，并将它们组合在一起创建了一个聊天提示词模板。接着，我们使用LLMChain来组合我们的聊天模型组件和聊天提示词模板组件。最后，我们运行了这个LLMChain链，并传入了用户的输入。这样，我们就可以方便地与LLM模型交互，并且不需要每次都为提示词模板提供所有的参数。

Agent

当我们的生活越来越依赖于各种信息，比如我们可能想要去郊游，需要查询当天的天气状况，路况信息等，这时候，我们的聊天机器人就可以发挥巨大的作用。不仅如此，他甚至可以帮我们制定计划。那么，如何让聊天机器人完成这样的任务呢？这就需要借助 Langchain 的高级组件：Agent。

Agent在 LangChain 中，是目前最先进的模块，它的主要职责是基于输入的信息，动态地选择执行哪些动作，以及确定这些动作的执行顺序。一个Agent会被赋予一些工具，这些工具可以执行特定的任务。Agent会反复选择一个工具，运行这个工具，观察输出结果，直到得出最终的答案。换句话说，Agent就像一个决策者，它决定使用什么工具来获取天气信息，而我们只需要关注它给我们的最终答案。

要创建并加载一个Agent，你需要选择以下几个要素：（1）聊天模型包装器：这是驱动代理的LLM。（2） 工具：执行特定任务的函数。例如，谷歌搜索、数据库查询、Python REPL，甚至其它LLM链。（3）代理名称：引用受支持的代理类的字符串。代理类主要由语言模型用于决定执行哪个动作的提示词模板参数化。在以下的代码示例中，我们将使用 SerpAPI 查询搜索引擎来创建一个代理：

首先安装必要的Python库。

pip -q install  openai

pip install git+https://github.com/hwchase17/langchain

设置密钥。

# 设置OpenAI的API密钥

os.environ["OPENAI\_API\_KEY"] = ""

# 设置谷歌搜索的API密钥

os.environ["SERPAPI\_API\_KEY"] = ""

from langchain.agents import load\_tools

from langchain.agents import initialize\_agent

from langchain.agents import AgentType

from langchain.chat\_models import ChatOpenAI

首先，加载控制代理的语言模型

chat = ChatOpenAI(temperature=0)

加载一些工具，注意这里的`llm-math`工具使用了一个LLM，因此需要将其传入

tools = load\_tools(["serpapi", "llm-math"], llm=llm)

最后，用工具、语言模型以及我们想要使用的代理类型初始化一个代理

agent =initialize\_agent(tools, chat, agent=AgentType.CHAT\_ZERO\_SHOT\_REACT\_DESCRIPTION, verbose=True)

现在我们测试一下代理

agent.run("What will be the weather in Shanghai three days from now?")

通过以上步骤，我们成功创建并运行了一个代理，这个代理能够帮助我们从网络上获取信息，并进行一些数学计算。这样，无论我们想要查询天气、路况，还是计划郊游，我们都可以轻松地通过这个聊天机器人得到所需的信息。

记忆组件

在此之前，我们制造的机器人虽然已经能使用工具进行搜索，进行数学运算，但它仍然是无状态的。这意味着它无法引用过去的交互，也就无法根据过去的交互理解新的消息。这显然对于聊天机器人来说是不足的，因为我们希望机器人能够理解新消息，并在此基础上理解过去的消息。

为了解决这个问题，langchain 提供了一个记忆模块。记忆模块提供了一种维持应用状态的方式。这个基础的记忆界面非常简单：它允许我们根据最新的运行输入和输出更新状态，并允许我们利用存储的状态修改或上下文的下一个输入。

在内置的记忆系统中，最简单的就是缓冲记忆。缓冲记忆只是将最近的一些输入/输出预置到当前的输入中。我们可以用代码来看这个过程：

首先，我们需要从 langchain.prompts 导入一些类和函数。然后，我们创建一个 ChatOpenAI 对象，这是我们的语言模型。

from langchain.prompts import (

    ChatPromptTemplate,

    MessagesPlaceholder,

    SystemMessagePromptTemplate,

    HumanMessagePromptTemplate

)

from langchain.chains import ConversationChain

from langchain.chat\_models import ChatOpenAI

from langchain.memory import ConversationBufferMemory

prompt = ChatPromptTemplate.from\_messages([

    SystemMessagePromptTemplate.from\_template(

        "The following is a friendly conversation between a human and an AI. The AI is talkative and "

        "provides lots of specific details from its context. If the AI does not know the answer to a "

        "question, it truthfully says it does not know."

    ),

    MessagesPlaceholder(variable\_name="history"),

    HumanMessagePromptTemplate.from\_template("{input}")

])

llm = ChatOpenAI(temperature=0)

接着，我们创建一个 ConversationBufferMemory 对象，这是内置的记忆组件。

memory = ConversationBufferMemory(return\_messages=True)

最后，我们创建一个 ConversationChain 对象，它是我们的会话链，会话链会用到之前创建的记忆和语言模型。会话链也是内置的链组件，传入参数后即可实例化运行。

conversation = ConversationChain(memory=memory, prompt=prompt, llm=llm)

创建了会话链之后，我们就可以用它来预测输入了。

conversation.predict(input="你好，我是美丽!")

例如，我们可以向会话链输入 “你好，我是美丽!”，然后会话链就会根据存储的状态和输入，生成一个响应。由于我们的记忆模型是缓冲记忆，所以会话链的响应会考虑到最近的一些输入/输出。

在后面的对话中，机器人会记住我们的名字。相反我们可以给机器人取一个特别的名字，因为有记忆的存在，机器人会记住他自己的名字。

总的来说，通过使用记忆组件，我们的聊天机器人不仅可以进行搜索和数学运算，还能引用过去的交互，理解新的消息。这大大提高了聊天机器人的实用性和智能水平。

这里祝贺大家，我们的第一个聊天机器人现已完成

## 2.3 GPT-4 等LLM的问题与挑战

在探讨LangChain框架问题之前，我们需要先理解LLM应用开发所面临的通用挑战。这些挑战往往会影响到使用此类工具进行开发时的体验。例如，虽然 GPT-4 是现代人工智能领域的一个巨大进步，但这并不意味着它是完美的。就像其他技术产品，GPT-4 也有其固有的限制，这些限制可能会在某些应用中造成不理想的用户体验。接下来，我们将使用GPT-4作为例子，深入探讨LLM当前所面临的一些普遍问题。

尽管 GPT-4 被广泛认为是人工智能领域的里程碑之一，但仍存在一些明显的限制。让我们详细地分析一下这些挑战：

1. 知识库依赖性：GPT-4 并未包含实时更新的真实世界知识库，因此其处理和生成响应的能力主要依赖于训练数据。这意味着 GPT-4 可能无法理解或处理一些特定的、需要依赖现有知识库的问题。

2. 上下文理解： GPT-4 的上下文理解能力有限，它主要依靠在训练数据中学习到的上下文信息来生成响应。这可能导致 GPT-4 在理解隐含的上下文或进行复杂的推理时出现困难。

3. 对话一致性：GPT-4 可能在保持对话一致性方面存在挑战，可能会在对话的不同部分之间出现语法、词汇和主题的不一致性。

### 2.3.1 LangChain 的解决方案

针对 GPT-4 的这些挑战，LangChain 提出了一系列有效的解决方案：

数据增强组件：为解决 GPT-4 的知识库依赖性问题，LangChain 的数据增强组件提供了一系列功能，例如文档上传、文本嵌入，向量库存储、文档提取等，这样可以实时更新知识库，提供更准确和多样的信息源。例如，如果用户在处理法律相关问题时需要引用最新的法规，LangChain 可以通过数据增强组件，将最新的法规上传并嵌入到向量库中，从而模型能够输出最新、最准确的法规信息。

实时信息获取：LangChain 整合了 API 工具链，允许模型搜索新闻、获取天气信息、地理位置等实时信息，大大提升了模型的实用性和准确性。比如，如果用户需要获取纽约的实时天气信息，LangChain 可以通过调用天气 API，获取并提供准确的天气信息。

数学计算能力：GPT-4 对数学计算不擅长。为弥补这一缺点，LangChain 封装了终端工具，使得模型可以调用 Python 算数，还有继承了 Program-aided Language Model (PAL) 数据模型。例如，如果用户问 “两个千亿相加等于多少”，LangChain 可以利用其终端工具调用 Python 的数学库进行计算，然后将计算结果返回给用户。

对话一致性：GPT-4 在保持对话一致性方面可能存在困难。LangChain 的记忆组件可以保存聊天记忆或者提取聊天关键信息，让模型的对话记住我们聊天的时候，人称代词缩写，人物对应，保持对话的一致性，获取像真人一样的聊天体验。例如，如果在聊天过程中引入了新的人物或主题，LangChain 可以提取和存储这些关键信息，以便在后续的对话中使用，从而保持对话的连贯性和一致性。

通过以上的改进，LangChain 在实用性、准确性以及问题处理能力等方面对 GPT-4 进行了有效的补充和优化，使其成为了一个更加强大、灵活且实用的LLM开发框架。

## 2.4 LangChain 的问题与挑战

LangChain作为LLM应用开发中的一大工具，在LLM应用开发领域，LangChain确实已经建立了自己的影响力。但和所有的工具或框架一样，它也有其不足。然而，正如我们之前所讨论的，开发者对 LangChain 的实用性提出了很多质疑。

比如有开发者认为从实用性和研究性的对比来看，评论指出LangChain更偏向于一个面向研究的工具，而不太适合生产环境。这意味着，尽管其为研究者提供了方便，但在实际应用中可能会遇到一些挑战。

文档的不清晰是另一个经常被开发者提及的问题。不清晰或混乱的文档可能会对开发者造成困扰，使得使用该工具变得更加困难。

此外，LangChain的抽象化设计也受到了批评。一些开发者认为LangChain过度抽象化，这使得在处理某些简单任务时，开发者反而需要付出更多的努力。

### 2.4.1 LangChain 的局限性

想要全面地了解这一开发框架的局限性，开发者必须深入研究其内在的缺点。为此，我们在《Re-implementing LangChain in 100 lines of code》这篇帖子中追踪了开发者们对于LangChain在LLM应用开发中的表现的评论，从而梳理出LangChain当前面临的一些问题。

帖子网址是：https://news.ycombinator.com/item?id=35820931，开发者均来自该帖子下的发言者。

1. 太过复杂问题:对于开发者而言，简单和直观的设计往往更受欢迎。与Deepset Haystack的Preprocessor、Reader/Retriever和PromptNode相比，LangChain在某些方面显得较为复杂。开发者kozikow为他的聊天机器人尝试使用LangChain，但最终选择了自定义Python。他认为，自定义开发可以更好地满足特定的需求，而LangChain在某些方面可能会带来过多的限制。

2. 文档清晰度问题:文档的清晰度对开发者的效率有直接的影响。LangChain的文档存在一些关键参数和细节的遗漏，可能使得开发者在LLM应用开发中遭遇障碍。

3. 过度抽象化问题:抽象化应该简化操作，但LangChain被指责为有过度抽象化的倾向。对于简单的任务来说，过多的冗余抽象层次可能适得其反，增加开发者的负担。fbrncci，一名经常使用LangChain的开发者，经常在思考是否真的需要一个完整的LLM框架来实现他的助手程序。尽管LangChain提供了某些有趣的功能，如异步代理执行，但该开发者发现，对于某些功能和应用，使用纯Python可能会更稳定、更可维护。

4. 模板固定性与可修改性问题:为了组装不完全符合LangChain模板的应用，开发者可能需要跨越多个文档页面，这会增加开发的复杂性。同时，当开发者需要进行小的修改以适应特定用例时，他们可能发现自己深陷于复杂的Python代码中，这也影响了开发的灵活性。

5. 开发者体验与错误率:一个高的错误率会对开发者的工作流产生直接影响。与直接使用OpenAI GPT相比，使用LangChain时有些开发者遇到了更多的问题。

6. 与其他工具的比较:开发者常常会与他们熟知的工具进行对比，例如Deepset Haystack和Hugging Face Transformers。在某些应用场景下，这些工具可能在性能和易用性上表现得更出色。开发者chaxor认为，LangChain的最重要的方面是它不使用OpenAI作为其语言模型。使用LangChain配合Galpaca等工具可以为用户提供一个高效、私密且低成本的助手来整理和管理大量文件。

7. 实用性与研究性的界限: LangChain被批评更偏向于研究工具，而不是生产工具。此外，它的简化可能导致它更适合玩具项目，而不是真实的LLM应用，这也引发了对它实际应用价值的质疑。开发者theturtletalks认为，ChatGPT将来可能会覆盖绝大多数的用例，并质疑一旦ChatGPT允许上传文档等功能，其他应用程序的价值是什么。

对于开发者，选择适当的工具是关键。当考虑使用LangChain作为LLM应用的开发工具时，应当仔细权衡其优缺点，确保选择对他们最有利的解决方案。

## 2.5 LangChain的设计哲学及其实践

LangChain的核心设计哲学受到了经典的Unix哲学的深刻启发，即“做一件事并做好”。这种哲学鼓励工具和组件专注于完成一个特定的任务，并确保它能够与其他工具和组件无缝地合作。具体到LangChain，这意味着它的各个组件都被设计成独立、组件化的，且可以通过管道操作符 | 进行链式组合，最终形成“管道化组合链”的序列形式，这是一种更接近自然语言编程方式。

考虑到LangChain构建的目标是LLM应用，开发者可以轻松地利用其提供的组件，如PromptTemplate、ChatOpenAI和OutputParser，为LLM应用创建自定义的处理链。例如，通过StrOutputParser，开发者可以轻松地将原始的LLM/ChatModel输出转换为更易于后续处理的字符串格式。以下代码示例展示了这种设计哲学的实际应用：

from langchain.prompts import ChatPromptTemplate

from langchain.chat\_models import ChatOpenAI

from langchain.schema.output\_parser import StrOutputParser

# 实例化提示词模板和聊天模型包装器

prompt = ChatPromptTemplate.from\_template("tell me a joke about {topic}")

model = ChatOpenAI(openai\_api\_key="你的API密钥")

# 定义处理链

chain = prompt | model | StrOutputParser()

# 调用处理链

response = chain.invoke({"foo": "bears"})

print(response)

# 输出："Why don't bears wear shoes?\n\nBecause they have bear feet!"

此外，LangChain的另一个关键效果是流水线处理。在软件开发中，流水线处理是一种将多个处理步骤组合在一起的方法，其中每个步骤的输出都是下一个步骤的输入。这种设计不仅简化了LLM应用开发流程，还确保了高效和可靠的输出。

开发者们在使用LangChain构建LLM应用时，不仅可以利用其组件化的设计优势，还可以确保他们的应用具有高度的灵活性和可扩展性，这些都是现代LLM应用开发中的关键要素。

需要注意的是，上述使用管道符进行链式调用(即prompt | model | StrOutputParser() )的功能需要新版本的LangChain仓库支持。要使用管道操作符 | 进行链式组合，开发者们请务必升级到LangChain的最新版本。

为了帮助开发者更好地理解和使用这种设计哲学，我们将在接下来的部分详细介绍LangChain中的一些常见管道化组合链。

### 2.5.1 常见管道化组合链

提示词模板+模型包装器

提示词模板与模型包装器的组合构成了最基础的链组件。通常是用在在大多数其他更复杂的链中。复杂的链组件通常都包含了提示词模板和模型包装器，这是我们跟LLM交互的基础组件，可以说缺一不可。

from langchain.prompts import ChatPromptTemplate

from langchain.chat\_models import ChatOpenAI

# 实例化提示词模板和聊天模型包装器

prompt = ChatPromptTemplate.from\_template("tell me a joke about {topic}")

model = ChatOpenAI(openai\_api\_key="你的API密钥")

# 定义处理链

chain = prompt | model

# 调用处理链

response = chain.invoke({"foo": "bears"})

print(response)

# 输出： AIMessage(content='Why don\'t bears use cell phones? \n\nBecause they always get terrible "grizzly" reception!', additional\_kwargs={}, example=False)

为了获得更加控制和有针对性的输出，确保输出的文本符合我们的期望和需求。我们经常希望将附加kwargs到传入的模型包装器上。在给出的代码示例中，chain = prompt | model.bind(stop=["\n"]) 这行代码表示，当LLM生成文本并遇到换行符 \n时，它应该停止进一步的生成。

chain = prompt | model.bind(stop=["\n"])

response = chain.invoke({"foo": "bears"})

# 输出response：AIMessage(content="Why don't bears use cell phones?", additional\_kwargs={}, example=False)

同样地支持OpenAI的函数回调功能，可以将函数描述列表传入的模型包装器上。

functions = [

{

"name": "joke",

"description": "A joke",

"parameters": {

"type": "object",

"properties": {

"setup": {

"type": "string",

"description": "The setup for the joke"

},

"punchline": {

"type": "string",

"description": "The punchline for the joke"

}

},

"required": ["setup", "punchline"]

}

}

]

chain = prompt | model.bind(function\_call= {"name": "joke"}, functions= functions)

response = chain.invoke({"foo": "bears"}, config={})

# 输出response：AIMessage(content='', additional\_kwargs={'function\_call': {'name': 'joke', 'arguments': '{\n "setup": "Why don\'t bears wear shoes?",\n "punchline": "Because they have bear feet!"\n}'}}, example=False)

提示词模板+模型包装器+输出解析器

我们可以在提示词模板与模型包装器的组合基础上，再增加一个输出解析器。

from langchain.schema.output\_parser import StrOutputParser

chain = prompt | model | StrOutputParser()

response = chain.invoke({"foo": "bears"}, config={})

# 输出response："Why don't bears wear shoes?\n\nBecause they have bear feet!"

当你定义一个要返回的函数，你可能不希望进行额外的处理，而只是希望直接对其进行解析。为了满足这种需求，LangChain为OpenAI提供了一个专门的函数回调解析器，名为JsonOutputFunctionsParser。这意味着在langchain.output\_parsers下的所有内置输出解析器类型都是可用的。此外，你还可以根据自己的需要使用自定义的输出解析器。

from langchain.output\_parsers.openai\_functions import JsonOutputFunctionsParser

chain = (

prompt

| model.bind(function\_call= {"name": "joke"}, functions= functions)

| JsonOutputFunctionsParser()

)

response = chain.invoke({"foo": "bears"})

# 输出response： {'setup': "Why don't bears wear shoes?",

'punchline': 'Because they have bear feet!'}

多功能组合链

首先定义两个提示词模板：prompt1 和 prompt2，它们分别询问某人来自哪个城市以及某个城市位于哪个国家。

chain1 是由 prompt1、模型和StrOutputParser组成的链，目的是根据给定的人名返回该人来自的城市。

chain2 是更复杂的链。它首先使用chain1的结果（某人的城市），然后结合itemgetter提取的“language”键值，生成prompt2的完整问题。这个问题随后传递给模型，并通过StrOutputParser解析。

from operator import itemgetter

prompt1 = ChatPromptTemplate.from\_template("what is the city {person} is from?")

prompt2 = ChatPromptTemplate.from\_template("what country is the city {city} in? respond in {language}")

chain1 = prompt1 | model | StrOutputParser()

chain2 = {"city": chain1, "language": itemgetter("language")} | prompt2 | model | StrOutputParser()

chain2.invoke({"person": "obama", "language": "spanish"})

当调用chain2并传递{"person": "obama", "language": "spanish"}作为输入时，整个流程将按顺序执行，并最终返回结果。

# 'El país en el que nació la ciudad de Honolulu, Hawái, donde nació Barack Obama, el 44º presidente de los Estados Unidos, es Estados Unidos.'

我们加大难度，创建一个更复杂的组合链。先定义了四个提示词模板，涉及颜色、水果、国家的旗帜颜色以及水果和国家的颜色关系。

chain1 是一个简单的链，根据prompt1生成一个随机颜色。

chain2 是一个复杂的链，它首先使用RunnableMap和chain1来获取一个随机颜色。

接下来，这个颜色被用作两个并行链的输入，分别询问此颜色的水果是什么，以及哪个国家的旗帜有这种颜色。

from langchain.schema.runnable import RunnableMap

prompt1 = ChatPromptTemplate.from\_template("generate a random color")

prompt2 = ChatPromptTemplate.from\_template("what is a fruit of color: {color}")

prompt3 = ChatPromptTemplate.from\_template("what is countries flag that has the color: {color}")

prompt4 = ChatPromptTemplate.from\_template("What is the color of {fruit} and {country}")

chain1 = prompt1 | model | StrOutputParser()

chain2 = RunnableMap(steps={"color": chain1}) | {

"fruit": prompt2 | model | StrOutputParser(),

"country": prompt3 | model | StrOutputParser(),

} | prompt4

最后，这两个并行链的结果（一个水果和一个国家）被用作prompt4的输入，询问这个水果和国家的颜色是什么。

chain2.invoke({})

# ChatPromptValue(messages=[HumanMessage(content="What is the color of A fruit that has a color similar to #7E7DE6 is the Peruvian Apple Cactus (Cereus repandus). It is a tropical fruit with a vibrant purple or violet exterior. and The country's flag that has the color #7E7DE6 is North Macedonia.", additional\_kwargs={}, example=False)])

## 2.6 本章小结

在本章中，我们深入探讨了LangChain，一个为了简化LLM应用开发而诞生的强大工具。

我们首先探讨了为什么需要LangChain，它的存在为了填补了LLM应用开发中的“最后一公里”。其核心思想体现在两个关键词上：“组件” 和 “链”，这两个元素不仅简化了复杂的数据处理流程，还提供了一个模块化的解决方案，使开发者能够轻松地进行LLM应用开发。我们还详细介绍了LangChain的三大应用场景，以及其由六大模块组成的构架。

在描述了LangChain的开发流程之后，我们展示了如何利用LangChain构建一个简单的应用——取名大师，并进一步引导读者创建了一个基于LangChain的聊天机器人，这种机器人不仅可以与用户进行日常对话，还可以即时回应用户的特定查询，如天气信息。

然后，我们转向了一些更为宏观的议题，探讨了GPT-4及其他LLM的普遍问题与挑战，如知识库依赖性、上下文理解困难和对话不一致性等。此外，我们也强调了LangChain如何尝试解决这些问题，以及LangChain本身的一些局限性和挑战。

最后，我们讨论了LangChain的设计哲学及其实践，强调了其对于管道化组合链的使用和重要性。

总体而言，本章为读者提供了一个全面的LangChain概览，从其基本理念到实际应用的各个方面，都进行了详细的讨论和示例介绍。这为后续章节中的深入学习和实践打下了坚实的基础。

3 模型 I/O

LangChain 打通了大语言模型应用开发的“最后一公里”。

2023年后，大语言模型（LLM）的模型平台如同春雨后的竹笋，一根接一根地冒出来。其中，知名度较高的几个包括OpenAI的GPT系列、Anthropic 的Claude 系列、谷歌的PaLM系列，以及最近Meta公司发布的LLaMA系列。这些模型都由各自的模型平台发布，并配备了API接口供开发者使用。

对于开发者来说，要想充分利用这些模型的能力，首先需要了解并掌握每个模型平台的API调用接口。有了这些知识，开发者就可以发起调用，向模型输入数据，并获取模型的输出结果。

问题是，对于初学者来说，面对众多的大语言模型平台和各自不同的API调用协议，确实可能会感到困惑甚至望而却步。毕竟，每个模型平台都有其特定的调用方式和规范，初学者需要投入大量的时间和精力去学习和理解。例如，OpenAI就发布了十几种不同的大语言模型型号。比如最新的GPT-4模型，它就需要使用"Chat"类型的API进行调用。这意味着，每当我们想要使用一个新的模型，就需要重新学习和理解这个模型特定的API调用方式，这对开发者来说无疑增加了额外的工作负担。这就像是每当我们遇到一个新的语言环境，就需要重新学习一种新的语言一样，既费时又费力。

对于那些想要利用大语言模型构建应用的开发者来说，同样如此。以应用程序为例，一个复杂的应用可能包含各种不同的功能需求，这就意味着可能需要调用不同类型的模型来满足这些需求。比如说，在处理文本分类的任务时，我们可能只需要一个参数较少、模型规模较小的模型就能够胜任。但在处理聊天场景时，我们就需要一个能够理解用户输入、并能让对话具有“说人话”的感觉，比如GPT-4。这样一来，我们就需要掌握和管理更多的模型调用方式，这无疑增加了开发的复杂度。

为了解决这些问题，LangChain推出了模型 I/O ，这是一种与大语言模型交互的基础组件。模型 I/O的设计目标是使开发者无需深入理解各个模型平台的API调用协议，就可以方便地与各种大语言模型进行交互。本质上来说，模型 I/O 组件是对各个模型平台API 的封装，这个组件下封装了50多个模型接口。

这就好比LangChain提供了这种“通用包装器”— 模型 I/O，无论你要和哪种模型进行交互，都可以通过这个“包装器”来实现。开发者可以很方便地与最新、最强大的模型（如2023年7月的GPT-4）进行交互，也可以与本地私有化部署的语言模型，甚至在HuggingFace上找到的开源模型进行交互。只需要几行代码，就可以实现与这些模型的对话，而无需关心模型平台的底层API调用方式。

那如何使用 LangChain 的基础组件模型 I/O来访问各个平台的大型语言模型？

模型 I/O 组件提供了三个核心功能，如图 3-1 所示。

[模型](file:///D:\docs\modules\model_io\models\)包装器：通过接口调用语言模型，即模型Model 的包装（图3-1中Predict部分）。

[提示](file:///D:\docs\modules\model_io\prompts\)管理：将模型输入模板化、动态选择和管理，即模型输入Model I（图3-1中Format部分）。

[输出解析器](file:///D:\docs\modules\model_io\output_parsers\)：从模型输出中提取信息 ，即模型输出Model O（图3-1中Parse部分）。

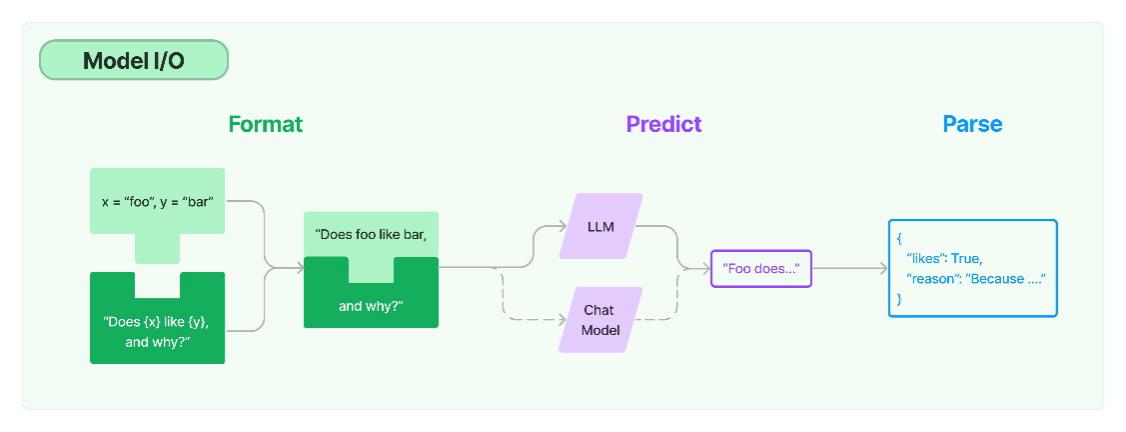


图 3-1

## 3.1 模型包装器

截至2023年7月，LangChain支持的大语言模型已经超过了50种，这其中包括了来自OpenAI、Facebook、Google等顶尖科技公司的开发大语言模型，以及各类优秀的开源大语言模型。对这些大语言模型，LangChain都提供了模型包装器进行交互。

随着大语言模型的发展，LangChain的模型包装器组件也在不断升级，以适应各大模型平台的API变化。到2023年，OpenAI发布了GPT-3.5-turbo模型，并且在他们的平台上增加了一个新的API类型，即"CHAT"类型API。这种API类型更适合处理聊天场景和复杂的应用场景。截至2023年7月，最新的GPT-4模型和Anthropic的Claude 2模型都采用了"CHAT"类型API。这种API类型正在成为模型平台API的发展趋势。如果不使用这种API，我们将无法利用最强大的GPT-4模型，也无法生成达到"人类质量"的自然语言对话文本。因此，选择适合自己应用需求的API类型，以及配套的LangChain模型包装器组件，是我们在使用大语言模型进行开发时，必须考虑的重要因素。

LangChain的模型包装器组件是基于各个模型平台的API协议进行开发的，它主要提供了两种类型的包装器。一种是通用的LLM类型 —— 文本补全“Completion”，另一种是专门针对"CHAT"类型API的ChatModel类型。

LLM类型的包装器适用于大部分的大语言模型，它们可以处理各种常见的自然语言处理任务，如文本分类、命名实体识别、情感分析等。在大部分情况下，LLM类型的包装器就能满足我们的需求。

然而，当我们要处理的是聊天场景或者更复杂的应用场景时，我们就需要使用ChatModel类型的包装器。ChatModel类型的包装器是专门针对"CHAT"类型API设计的，它能更好地适应聊天场景的特性，例如，它可以处理一系列的消息作为输入，返回一个消息作为输出，这使得它非常适合用于构建聊天机器人或者对话系统。

LangChain的模型包装器组件为我们提供了一种方便的方式来使用各种类型的大语言模型，无论是通用的LLM类型，还是专门针对聊天场景的ChatModel类型，都能让我们更高效地利用大语言模型的能力。

### 3.1.1[模型](file:///D:\docs\modules\model_io\models\)包装器分类

为了适应各种 模型平台的不同类型的API接口，LangChain设计了[模型](file:///D:\docs\modules\model_io\models\)包装器，主要支持两种类型的模型交互方式，它们分别是LLM类型、Chat Model类型。

LLM 包装器组件，是一种专门处理与大语言模型的文本补全类型API交互的组件。这种类型的大语言模型主要用于接收一个字符串作为输入，然后返回一个补全的字符串作为输出。比如，你可以输入一个英文句子的一部分，然后让模型生成句子的剩余部分。这种类型的模型非常适合用于自动写作、编写代码、生成创意内容等任务。

例如你想使用OpenAI 的“text-davinci-003”模型，你可以选择使用 OpenAI 包装器，如下所示：

from langchain.llms import OpenAI

openai = OpenAI(model\_name="text-davinci-003")

在这里， openai 是 OpenAI 类的一个实例，它继承了OpenAI类的所有属性和方法，你可以使用这个 openai 对象来调用 OpenAI 的 “text-davinci-003”模型， 导入的OpenAI 类即为 LangChain 的模型包装器，专门用于处理OpenAI 公司的“Completion” 类型API。

2023年以来，LangChain 已经实现了50种不同大语言模型 “Completion” 类型API的包装器，包括 OpenAI，LlamaCpp，Cohere，Anthropic等。也就是说开发者无需关注这50个模型平台的底层 API 是如何被调用的，LangChain 已经包装好了，让开发者可以“即插即用”。

通过 langchain.llms 获取的所有对象都是LLM类型包装器，这些对象我们称为LLM 类包装器。所有的 LLM 类包装器都是 BaseLLM 的子类，它们继承了 BaseLLM 的所有属性和方法，并根据需要添加或覆盖了一些自己的方法。这些包装器封装了各自平台上的大型语言模型的功能，使得开发者可以以面向对象的方式使用这些大型语言模型的功能，而无需直接与各个模型平台的底层 API 进行交互。

需要注意的是，OpenAI 的 “Text Completion” 类型API在2023年7月进行了最后一次更新，并且该API现在只能用于访问较旧的历史遗留模型，如“2020-2022年”的模型text-davinci-003、text-davinci-002、davinci、curie、babbage和ada等。

“Text Completion” 类型API与新的“Chat Completion” 类型API不同。 “Text Completion” 类型API使得开发者可以直接提供一段具有特定上下文的文本，然后让模型在这个上下文的基础上产生相应的输出。尽管这种方式在某些场景下可能会更方便，比如翻译和写文案场景，但在需要模拟对话或者需要更复杂交互的情况下，OpenAI 平台建议使用“Chat” 类型API。

相比之下，如果要使用OpenAI的最新模型，如2023年以后的模型gpt-4和gpt-3.5-turbo，那么你需要通过 Chat 类型API进行访问。这意味着，如果你想充分利用OpenAI最新的技术进展，你需要将你的应用程序或服务从使用 “Text Completion” 类型API迁移到使用Chat 类型API。

LangChain创建了Chat Model 类型的组件适配了模型平台的Chat 类型API。

Chat Model是一种专用于大语言模型 “Chat” 类型API 交互的包装器组件。设计这类包装器，主要是为了适配GPT4 等先进聊天模型，这类模型主要用于构建对话系统或聊天机器人。它接收一系列的消息作为输入，并返回一个消息作为输出。这种类型的模型非常适合用于构建能与人进行自然语言交流的应用，比如客服机器人、语音助手等。

2023年7月，LangChain 已经实现了6个针对不同模型平台的Chat Model类型的包装器，包括：

- ChatOpenAI ：用于包装OpenAI Chat 大型语言模型（如 gpt-4和gpt-3.5-turbo）

- AzureChatOpenAI ：用于包装 Azure 平台上的 OpenAI 模型

- PromptLayerChatOpenAI ：用于包装 PromptLayer 平台上的 OpenAI 模型

- ChatAnthropic ：用于包装 Anthropic 平台的大语言模型

- ChatGooglePalm ：用于包装 Google Palm 平台的大语言模型

- ChatVertexAI：用于包装 Vertex AI 平台的大语言模型

不同于LLM 类包装器是 BaseLLM的子类，Chat Model类包装器都是 BaseChatModel 的子类，继承了 BaseChatModel 的所有属性和方法，并根据需要添加或覆盖了一些自己的方法。

例如，如果你想要使用最先进的 GPT-4 模型，你可以选择使用 ChatOpenAI 包装器，如下所示：

from langchain.chat\_models import ChatOpenAI

llm = ChatOpenAI(temperature=0, model\_name="gpt-4")

所有通过 langchain.chat\_models 获取的所有对象都是Chat Model类型，这些对象我们称为Chat Model 类包装器。在这里， llm 是 ChatOpenAI 类的一个实例，你可以使用这个 llm 对象来调用 OpenAI Chat 的 GPT-4 模型的功能。

LLM 类和Chat Model类包装器，都是LangChain对各个大语言模型底层API 的封装，开发者无需关注各个模型平台的底层 API的实现方式，开发者只需关注像模型输入什么，以及模型输出什么。

LangChain 的官网文档中，凡是涉及到模型输入、输出相关的链 （Chain）和代理（Agent）的示例代码，都会提供两份。一份是使用LLM类型，一份是使用Chat Model类型。如图3-1中Predict部分，同样在区分为了LLM 和Chat Model。这是因为两者之间存在着细微但是很重要的区别。

LLM与Chat Model的区别

学习大语言模型应用开发的捷径是关注模型的输入和输出。

从输入和输出的角度来看，LLM 类型和 Chat Model 类型的包装器组件主要有以下区别：

1. 输入：

对于LLM 类型，其输入通常是一个单一的字符串提示（prompt）。例如，你可以输入"Translate the following English text to French: '{text}'"，然后模型会生成对应的法文翻译。主要用于文本完成任务，例如给定一个提示（例如："今天的天气如何？"），模型会生成一个相应的完成（例如："今天的天气很好。"）。

Chat Model 类型，其输入则是一系列的聊天消息。通常这些消息都带有发言人的标签（通常是系统、AI和人类）。每条消息都有一个role（角色）和 content（内容）。例如，你可以输入[{"role": "user", "content": 'Translate the following English text to French: "{text}"'}]，模型会返回对应的法文翻译。

2. 输出：

对于LLM 类型，其输出是一个字符串，这个字符串是模型对输入提示的补全。

Chat Model 类型，其输出是一个聊天消息，这个消息是模型对输入消息的响应。

虽然LLM 类型和Chat Model 类型在处理输入和输出的方式上有所不同，但是为了使它们可以互换使用，它们都实现了基础语言模型接口。这个接口公开了两个常见的方法：predict（接受一个字符串并返回一个字符串）和 predict messages （接受消息并返回一个消息）。这样，无论你是使用特定的模型，还是创建一个应该与不同类型的模型一起工作的应用，都可以通过这个共享的接口来进行操作。

之所以要区分这两种类型，主要是因为它们处理输入和输出的方式不同，且各自适用于不同的使用场景。通过这种方式，可以更好地利用不同类型的语言模型，提高模型的适用性和灵活性。

模型I/O的学习路径

在Langchain的发展迭代过程中，每个模块调用模型I/O功能，都精细地提供了LLM 和Chat Model两种代码方式。因为OpenAI平台的底层API 发生了迭代，Langchain为了不增加开发者修改代码量，更好地适配新的大语言模型发展要求，也做了类型的划分。这种划分已经形成了技术的趋势，同时也为我们学习LangChain提供了线索。当我们看到一个类的名字内包含 “Chat” ，比如ChatAgent ，那么我们要给模型输入的是消息类型的信息，而我们也可以预期 ChatAgent 输出的是消息列表类型。

相比之下，如果你使用的是 LangChain 的 llms 模块导出的对象，这些对象是 LLM 类包装器，主要用于处理自由形式的文本。输入的是一段或多段自由形式的文本，输出的则是模型生成的新文本。这些输出文本可能是对输入文本的回答、延续或其他形式的响应。

最后，如果你使用的是 LangChain 的 chat\_models 模块导出的对象，这些对象是专门设计用来处理对话消息的。输入的是一个对话消息列表，每条消息都由角色（如 user）和内容组成。这样的输入给了大语言模型一定的上下文环境，可以提高输出的质量。输出的也是一个消息类型，这些消息是连续对话内容的响应。

### 3.1.2 LLM 包装器

LLM 包装器是 LangChain 的核心组件之一。LangChain 不提供自己的大语言模型，而是提供了与许多不同的模型平台进行交互的标准接口。

下面我们通过示例代码，演示如何使用LLM 包装器。

示例代码我们使用的LLM 包装器是OpenAI包装器，它封装了OpenAI 平台的接口，导入方式和实例化方法对于所有 LLM 类型都是通用的。以下是示例代码：

安装和设置密钥

首先，我们需要安装 OpenAI Python 包：

pip install openai langchain

然后导入OpenAI包装器并设置好密钥：

from langchain.llms import OpenAI  
OpenAI.openai\_api\_key = "YOUR\_OPENAI\_API\_TOKEN"

使用 LLM 包装器的最简单方法： 输入一个字符串，输出一个字符串

# 运行一个最基本的LLM 包装器，由模型平台 OpenAI 提供的文本生成能力。 b  
llm = OpenAI()  
llm("Tell me a joke")

运行后的结果：

'Why did the chicken cross the road?\n\nTo get to the other side.'

说明： 这里的运行结果是随机的，而不是唯一固定的回答。

### 3.1.3 Chat Model 包装器

当前最大的应用场景便是 “聊天 Chat” ，就像模型平台OpenAI的热门应用ChatGPT一样。为了紧跟用户需求，LangChain推出了专门用于聊天场景的Chat Model 包装器，以便我们能与各种模型平台的 “ Chat” 类型 API 进行交互。

在3.1.1节我们提到Chat Model 包装器的不同之处在于，提供以 “聊天消息” 作为输入和输出的接口。它们的输入不是单个字符串，而是聊天消息的列表。

在本节，我们将通过示例代码，演示输入聊天消息列表的Chat Model 包装器, 与上一节的 LLM包装器在使用上有什么区别？

安装和设置密钥

首先，我们需要安装 OpenAI Python 包。

pip install openai langchain

然后设置好密钥：

import os  
  
os.environ['OPENAI\_API\_KEY'] = ''

使用 Chat Model 包装器的方法： 输入一个消息列表，输出一个字符串

为了使用 Chat Model 包装器，我们将导入三个数据模式 （schema）：一个由 AI 生成的消息的数据模式（AIMessage），一个是人类用户输入的消息的数据模式（HumanMessage）和一个是系统消息的数据模式 （SystemMessage），通常用于设置聊天环境或提供上下文信息。它们定义了应用程序与大语言模型进行聊天时，不同类型消息的数据格式。然后导入 Chat Model 包装器—— ChatOpenAI，这个包装器封装了 OpenAI 平台 Chat 类型的API, 我们无需关注OpenAI 平台的接口如何调用，我们只需关注我们向这个 ChatOpenAI 包装器输入的内容。

from langchain.schema import (  
 AIMessage,  
 HumanMessage,  
 SystemMessage  
)  
from langchain.chat\_models import ChatOpenAI

我们向ChatOpenAI 包装器输入的内容要是一个消息列表。消息列表的数据格式需要符合 AIMessage, HumanMessage 和 SystemMessage 这三种类型的数据模式。这种设计的目的是为了提供一种标准和一致的方式来表示和序列化这些消息。序列化是将数据结构转换为可以存储或传输的格式的过程。在ChatOpenAI 包装器中，序列化是指将这些消息对象转换为可以通过 API 发送的数据格式。这样，接收消息的一方（OpenAI 平台的服务器）就能知道如何正确地解析和处理每个消息。

此时，SystemMessage是指在使用模型时用于配置系统的消息，而HumanMessage是指用户消息。下面将SystemMessage和HumanMessage组合成一个聊天消息列表，输入聊天模型。这里笔者使用的模型名称是“gpt-3.5-turbo”。如果你有“gpt4”，那么也可以使用“gpt4”。

chat = ChatOpenAI(model\_name="gpt-3.5-turbo",temperature=0.3)  
messages = [  
 SystemMessage(content="你是个取名大师，你擅长为创业公司取名字。"),  
 HumanMessage(content="帮我给新公司取个名字，要包含AI的")  
]  
response=chat(messages)  
  
print(response.content,end='\n')

我们创建一个消息列表 messages，这个列表包含了一系列 SystemMessage 和 HumanMessage 对象。每个消息对象都有一个 content 属性，用于存储实际的消息内容。例如，在这个示例代码中，系统消息的内容是“你是个取名大师，你擅长为创业公司取名字。” 人类消息的内容是“帮我给新公司取个名字，要包含AI的”。

当你调用 chat(messages) 时，ChatOpenAI 对象会接收这个消息列表，然后按照AIMessage, HumanMessage 和 SystemMessage 这三种类型的数据模式，将其序列化并发送到 OpenAI 平台的服务器。服务器会处理这些消息，生成 AI 的回应，然后将这个回应发送回 ChatOpenAI 包装器。这个包装器接收回应，回应是一个 AIMessage 对象，你可以通过 response.content 获取它的内容。

包含“ AI ” 的创业公司名称的建议：  
1. AIgenius  
2. AItech  
3. AIvision  
4. AIpros  
5. AIlink  
6. AIsense  
7. AIsolutions  
8. AIwave  
9. AInova  
10. AIboost  
  
希望这些名称能够给你一些启发！

正如你所提到的，Chat Model包装器在使用过程中，相比于LLM包装器，确实更显复杂一些。这主要是因为，Chat Model包装器需要我们先导入三种数据模式，并且需要我们组合一个消息列表messages，最后从响应对象中解析出我们需要的结果。而LLM包装器则简单许多，我们只需要输入一个字符串，然后就能直接得到一个字符串结果。

那么，为什么Chat Model包装器会设计得如此复杂呢？这其实是因为各个模型平台的"chat"类型API接收的数据格式并不统一。为了能够正确地与这些API进行交互，我们必须定义各种消息的数据格式，以满足各个模型平台的"chat"类型API所需的数据格式，以便获取到我们期望的聊天消息结果。

否则，如果我们不使用统一的数据模式，每次向不同的模型平台提交输入时，都需要对输入进行单独的格式处理，这无疑会增加我们的开发工作量。而且，如果我们不进行类型检查，那么一旦出现类型错误，可能会在代码运行时才被发现，这可能会导致程序崩溃，或者产生不符合预期的结果。

尽管Chat Model包装器在使用过程中的复杂性超过了LLM包装器，但这种复杂性是有价值的。通过前置数据处理为标准化格式的工作，我们可以简化和统一数据处理流程，减少出错的可能性。这样，开发者在使用大语言模型时，可以将更多的注意力放在业务逻辑的开发上，而不是被各种复杂的数据处理和错误处理所困扰。

这种设计理念，可以让开发者无需为不同模型平台的API调用方式而烦恼，可以更快速地集成和使用这些强大的大语言模型。

这个演示代码也体现了Chat Model包装器和LLM包装器的最大差异，在于它们所接受的输入数据格式的不同。

Chat Model包装器的设计目标是处理复杂的对话场景，它需要处理的输入是一个聊天消息列表。这个列表中的每一条消息都包含了消息的角色（AI或用户）和消息内容，它们合在一起构成了一个完整的对话上下文。这种输入方式非常适合处理那些需要引入历史对话内容，以便生成更加连贯、有上下文的聊天响应的任务。

LLM包装器的设计目标是处理那些只需要单一输入就可以完成的任务，如文本翻译、文本分类等。因此，它只需要一个字符串作为输入，不需要复杂的消息列表和对话上下文。这种简洁的输入方式，使得LLM包装器在处理一些简单任务时，使用起来更加简单。

## 3.2 提示词

一切都始于“提示词”（Prompt）。事实上，LangChain 的一切都是围绕“提示词”构建的。下面我们介绍一下“什么是‘提示词’”，以及在LangChain中的“提示词模板”的概念。

在大语言模型的世界里，“提示词”是一个关键的概念。提示词是引导大语言模型做出人类预期响应的文本。它由应用程序发送到模型平台，这个文本描述了我们希望解决的问题，或者表达了我们期望得到的结果。例如，最简单的提示词可能就是 “你好! ”。当我们输入“你好! ” 这个提示词时，我们期望的是模型能够以一个友好的方式回应，也可能是想确认AI是否在线。

如果大语言模型是个魔法世界，提示词的角色就像哈利波特手中的魔杖，它是我们引导模型产生预期响应的关键。每一个提示词，就像一条魔法咒语，我们挥舞手中的魔杖，将提示词“咒语”输入给模型，希望模型像魔法世界一样，呈现出我们期待的结果。我们对于提示词的探索就如同哈利波特拿到人生的第一支魔杖一样，充满了热情和好奇。人人都想拥有高级的魔杖，实施更强大的魔法，以至于2023年出现了一个全新的大语言模型开发领域，人们称之为“提示词工程”。

诞生“提示词工程”的原因在于实际应用场景的复杂性和多样性，简单的提示词不能满足实际场景的需求，而高级的“提示词”输入给模型，可以提高模型输出结果的质量。例如，当我们希望构建更复杂的应用，如生成专业报告、编写长篇小说或者提供专业咨询等，我们需要“提示词工程”构建出来的提示词，它可能是包含示例，引入外部知识库数据，清晰表达需求，且数据结构化的文本，而不仅仅是一句话。

“提示词工程”，通过对提示词的精心设计和优化，我们可以引导大语言模型生成更符合我们需求的输出。“提示词工程”是一门技术，需要我们对大语言模型的理解，对目标任务的理解，还需要我们的创新和实验。

在LangChain框架中，提示词不是简单的字符串，而是一个更复杂的结构，一个“提示词工程”。这个结构由一个或多个提示词模板（PromptTemplate 类的实例）组成，每个模板可以接受一些输入变量，并根据这些变量生成对应的提示词。这样，我们就可以根据具体的需求和情境，动态地创建出各种各样的提示词。

例如，我们可能有一个提示词模板用于生成写邮件的提示词，这个模板可能需要接受如 "收件人"、"主题"、"邮件内容" 等输入变量。然后，根据这些变量，模板字符串可以生成出如 "写一封给 {收件人} 的邮件，主题是 {主题}，内容是 {邮件内容}" 这样的提示词。

这种工程化的提示词构建方式，使得我们可以构造出适应各种应用程序的结果，而不仅仅是简单的聊天界面。这些复杂的提示词可以用于生成文章、写邮件、回答问题、执行任务等各种场景，大大提高了大语言模型的实用性和可用性。

在LangChain框架中，这种工程化的方式来构建提示词，是通过各种“PromptTemplate”类来生成提示词。在LangChain中，每一个“PromptTemplate” 类的实例都定义了一种特定类型的提示词的格式和生成规则。

“PromptTemplate” 类实际上提供了一个模板，可以在这个模板的基础上，方便地生成复杂的提示词。这就是提示词模板的核心思想和工作方式。

### 3.2.1 提示词模板

提示词可以被视为我们向大语言模型提出的请求，它们决定了我们希望模型进行何种反应。提示词的质量，直接影响了模型的回答质量和我们能否成功完成更复杂的任务。

在LangChain框架中，提示词是由“提示词模板”（PromptTemplate）这个包装器对象生成的。也就是说你在LangChain中，你要构造提示词，你必须学会使用这个包装器。

你可以将提示词模板视为一个幕后工作者，他在幕后默默地工作，用户只看到他们自己输入的简单关键词，却得到了模型出色的响应。这是因为开发者将用户的输入嵌入到预先设计好的提示词模板中，这个模板就是一个包装器，用户的输入经过包装器的包装后，最终输出一个高效的提示词。这个包装器还能组合不同的提示词，提供各种格式化、检验参数的工具，帮助开发者构造复杂的提示词。

提示词模板背后的魔力是，它让用户的简单输入转化为了高效的提示词，从而使得大语言模型能够提供高质量的回答和服务。

提示词模板的定义

在LangChain框架中，提示模板（PromptTemplate）是一个重要的概念。它指的是一种可复制、可重用的生成提示词的方式。提示模板是用于生成提示词的模板字符串，它是包含占位符的字符串，其中这些占位符可以在运行时被动态替换成实际终端用户输入的值。这样的字符串允许我们在其中插入变量、表达式或函数的结果。

提示模板可能包含以下三个元素，并不是必须包含：

1. 对语言模型的指令：这些指令可以指导语言模型理解用户的需求，并按照特定的方式进行回应。

2. 一组少量的示例（Few-shot examples）：这些示例可以帮助语言模型更好地理解任务，并生成更准确的响应。

3. 对语言模型的问题：这些问题可以直接引导语言模型生成特定的答案。

通过灵活使用这些元素，创建新的提示模板，我们可以更有效地利用语言模型的能力，提高其在各种应用场景下的表现。

提示词模板就是可批量生成提示词的包装器，这个包装器由 PromptTemplate 类生成的。它可以接收开发者对任务的描述文本，也可以接受用户输入的一系列参数。比如说，我们要给公司的产品取一个好听的名字，用户输入的就是产品的品类名字，例如“袜子”或“毛巾”。然而，我们并不需要为每一个品类都编写一个提示词，而是使用这个模板，根据用户输入的不同品类词，生成对应的提示词：“请给公司的 {品类名}，取一个简单容易传播的产品名字”。

提示词模板的职责就是根据大语言模型平台的API类型，包装并生成适合的提示词模板。为了满足不同类型的模型平台底层API需求，PromptTemplate 提供了format方法 和 format\_prompt方法，输出可以是字符串、消息列表，以及 ChatPromptValue 的形式提供。比如对于需要字符串输入的LLM类型的模型包装器，我们会使用 to\_string 方法将提示词转化为一个字符串。而对于需要消息列表输入的Chat Model类型的API，我们会使用 to\_messages 方法将提示词转化为一个消息列表。

这就是说，提示词模板可以将用户的简单输入，根据需求转化为复杂的、结构化的提示词。这个过程中，开发者可以向模板中添加实例来影响模型的生成效果，描述任务的特性，甚至包含聊天记录，让模型记住用户的特定信息，如用户的名字。所有这些都是为了让模型能够产生更符合用户需求的回答。

总的来说，提示词模板就是一种能够产生动态提示词的包装器，开发者组织数据输入给包装器，经过包装后，输出的是适配各个模型平台的提示词。

提示词模板的输入

提示词模板是一个输入数据和输出提示词的包装器，那开发者可以向它输入什么样的数据呢？

开发者向提示词模板输入的数据可以来自很多来源，我们可以将其分类为内部数据和外部数据。

内部数据是指那些已经被LangChain框架封装好的数据以及开发者写的示例和需求描述文本。比如，LangChain的许多Agent和Chain实例对象都内置了自己的提示词。这些提示词都被预先定义在源码 prompt.py 文件中，例如我们可以导入预制的API\_RESPONSE\_PROMPT ，它是引导模型根据API 响应回答用户问题的提示词，导入方式是 from langchain.chains.api.prompt import API\_RESPONSE\_PROMPT。

API\_RESPONSE\_PROMPT 在源码中的定义是这样：

API\_RESPONSE\_PROMPT\_TEMPLATE = (

API\_URL\_PROMPT\_TEMPLATE

+ """ {api\_url}

Here is the response from the API:

{api\_response}

Summarize this response to answer the original question.

Summary:"""

)

API\_RESPONSE\_PROMPT = PromptTemplate(

input\_variables=["api\_docs", "question", "api\_url", "api\_response"],

template=API\_RESPONSE\_PROMPT\_TEMPLATE,

)

我们可以直接导入 API\_RESPONSE\_PROMPT 这个内置模板后，格式化外部的输入变量，将提示词提交给模型平台的 API。

from langchain.chains.api.prompt import API\_RESPONSE\_PROMPT

prompt= API\_RESPONSE\_PROMPT.format(api\_docs=“”,question=“”, api\_url=“”, api\_response=“”)

构建复杂的提示词，像是盖房子，LangChain的提示词模板做了“盖房子”的基建工程，比如内置的提示词模板可以解决大多数的业务需求，还有检查数据格式，规划提示词的结构，格式化提示词等。这些基建工程通常用于描述模型的任务，或者用于指示模型的行为。

外部数据则是开发者可以自由添加的数据，这些数据可以来自各种来源。最主要的数据是用户的输入，用户和模型的聊天记录，以及开发者为模型增加的外部知识库数据、程序运行的上下文管理信息。例如，开发者可以收集并使用聊天历史记录，这些历史记录可以帮助模型理解之前的对话上下文，从而生成更加连贯和有用的回答。开发者也可以使用用户的输入，这些输入被填充到模板占位字符串中，可以帮助模型理解用户的需求，从而生成更加符合用户期望的回答。此外，开发者还可以编写自己的示例文本，或者导入外部的文档片段，这些示例文本和文档片段可以帮助大语言模型理解任务的需求，增加大语言模型的“脑容量”和“记忆时长”，从而生成更加高质量的回答。

总的来说，无论是内部数据还是外部数据，都是利用提示词模板这个包装器，让包装器生成更好的提示词。再结合内部指令数据和外部数据，我们可以构建出更加复杂和强大的提示词，从而开发出更加强大的大语言模型应用。构建提示词的过程，就是一项“提示词工程”。

提示词模板的输出

在 LangChain 中，我们使用提示词模板构造器来构造适应各种模型平台 API 类型的提示词。具体来说，我们可以根据模型平台的 API 类型，生成出两种类型的提示词。例如，OpenAI 平台的 API 就有两种类型，一种是 LLM 类，一种是 Chat 类。这两种 API 类型对应了两种不同类型的交互方式。LLM 类的 API 接收的是一个字符串作为输入，而 Chat 类的 API 接收的是一个消息列表。因此，如果你要使用的是 OpenAI 的 GPT-4 模型，那么你就需要准备一个消息类型的提示词。

对于 LLM 类的 API，我们有 BasePromptTemplate 类和它的子类 PromptTemplate，这些类实例化为一个包装器对象后，可以生成一个模板字符串的提示词。在这种情况下，包装器可以构造出一个字符串的提示词。这个字符串可能包含了一些特定的任务描述，用户的问题，或者其他的上下文信息，它们都被整合在一起，构成了一个完整的、用于引导模型生成预期输出的提示词。字符串的提示词如下：

' You are a helpful assistant that translates English to French '

而对于 Chat 类的 API，我们有 BaseChatPromptTemplate 类和它的子类 ChatPromptTemplate，实例化为一个包装器对象后，可以生成一个消息列表的提示词。因为 chat 类型的 API，我们通常需要一个消息列表作为输入。在这种情况下，包装器构造出一个包含多个消息对象的提示词。每个消息对象都代表了一条消息，它可能是一个用户的问题，一个 AI 的回答，或者一个系统的指令。这些消息被组织在一起，形成了一个清晰的对话流程，用于引导模型进行复杂的对话任务。消息列表的提示词如下：

[SystemMessage(content='You are a helpful assistant that translates English to French.', additional\_kwargs={}),

HumanMessage(content='I love programming.', additional\_kwargs={})]

无论你使用哪种类型的 API，只要选择对应的提示词模板类，就可以轻松地生成出符合 API 要求的提示词。这极大地简化了构建提示词的过程，使得开发者可以将更多的精力放在解决业务逻辑上，而无需手动处理复杂的数据转换和格式化工作。

模板的三个组成部分

提示词模板主要由三个部分组成：语言模型的说明，一组少数示例，以及一个问题或任务。Langchain将这三个部分组合起来，并为输出结果进行格式化的处理，以生成一个完整的提示。

提示词模板有许多实际应用场景。比如，它在做摘要或分类时非常实用。以新闻摘要为例，我们可以构建一个模板，模型的任务就是生成新闻的摘要。用户的输入可能是复制黏贴了一篇完整的新闻文章，给语言模型的说明可能是“生成这篇新闻的摘要”，示例则可以是一些已经生成过的新闻摘要。然后，Langchain将这三个部分组合起来，生成一个完整的提示，如：“请为这篇新闻生成一个摘要。举例” 根据这个提示，模型就能生成一个符合要求的新闻摘要。这样，我们就可以利用提示词模板，轻松地为任意一篇新闻生成摘要。

值得庆幸的是Langchain提示词模板构造的提示词，在任何大语言模型中是可用的，但是输出的回答质量取决于大语言模型自身的能力。

构造提示词的步骤

构造提示词是通过特定的格式和规则，将用户的输入数据包装为能够引导模型生成期望输出的提示词。LangChain 提供了一套内置提示词模板，这些模板可以用来生成各种任务的提示词。在一些基础和通用的场景中，使用内置模板可能就足够了，但在一些特定和复杂的场景中，可能需要创建自定义模板。这里我们介绍最常见的使用 LangChain 提供的内置模板来构造提示词。

PromptTemplate是LangChain提示词组件中最核心的一个类，构造提示词的步骤本质上是实例化这个类的过程。这个类被实例化为对象，也就是我们所说的提示词包装器，在LangChain各个链组件中被调用。在实例化PromptTemplate时，两个关键参数是template和input\_variables。我们只需要准备好这两个参数，就可以实例化一个基础的PromptTemplate ，结果就是一个PromptTemplate对象, 即一个提示词模板的包装器。

from langchain import PromptTemplate  
template = """  
You are an expert data scientist with an expertise in building deep learning models.   
Explain the concept of {concept} in a couple of lines  
"""

#实例化PromptTemplate：

prompt = PromptTemplate(template=template, input\_variables=["concept"])

在LangChain中，这个包装器会像快递箱一样，被传递给链组件进行调用。链组件的调用方式非常直观，例如上面代码的实例化后的对象 prompt，将这个包装器传递给LLMChain：chain = LLMChain(llm, prompt=prompt)。最后，我们可以使用 chain({"concept ": "NLP"}) 来运行链。这种使用方式是LangChain中最常见的模式。

值得注意的是，PromptTemplate包装器接受内部数据（即在实例化时定义的template和input\_variables）和外部数据（即在运行链时传递的数据）。在我们使用链组件调用时，外部数据（即用户的输入）是通过链组件传递的，而不是直接传递给提示词模板包装器。

如果我们不需要链组件进行调用，PromptTemplate包装器还提供了一些其他方法。例如，format方法可以将PromptTemplate包装器的内部和外部数据合并，形成一个完整的提示词，方便我们查看完整的提示词内容。

from langchain import PromptTemplate  
template = """  
You are an expert data scientist with an expertise in building deep learning models.   
Explain the concept of {concept} in a couple of lines  
"""

#实例化模板的第一种方式：

prompt = PromptTemplate(template=template, input\_variables=["concept"])

# 实例化模板的第二种方式：  
# prompt = PromptTemplate.from\_template(template)

# 将用户的输入通过format 方法嵌入到提示词模板，并且做格式化处理

final\_prompt = prompt.format(concept="NLP")

我们打印 final\_prompt 提示词的结果是：

'\nYou are an expert data scientist with an expertise in building deep learning models. \nExplain the concept of nlp in a couple of lines\n'

创建提示词模板主要涉及以下两个要求：

它需要有一个 input\_variables 属性，这个属性指定了提示词模板期望的输入变量。这些输入变量是在生成提示词时需要的数据，比如在我们的示例中，输入变量就是 {concept}。

如果不想显示指定输入变量，还可以使用 from\_template 方法，代码中的第二种方式。这个方法接收与预期的 input\_variables 对应的关键字参数，并返回格式化的提示词。在我们的示例中，from\_template 方法实例化模板，format 方法接收 concept="NLP" 作为输入，并返回格式化完成的提示词。

PromptTemplate类实例化的结果是包装器，包装器可以是自己被链组件调用，也可以调用方法（连结外部的用户输入和内部定义的关键参数），最终的结果是实现内部数据和外部数据的整合，形成一个完整的提示词。

同样地，Langchain 为Chat Model 类型的API 定义了ChatPromptTemplate 类，用于构造消息列表的提示词。构造的步骤基本与上面的LLM 类型一致，但是仍然有区别。

构造一个Chat Model类型的提示词

Chat Model类型的提示词模板的不同之处，构造的提示词是消息列表, 支持输出 Message 对象。Langchain 提供了聊天内置模板 （ChatPromptTemplate）和角色消息内置模板。角色消息提示词模板包含AIMessagePromptTemplate, SystemMessagePromptTemplate 和 HumanMessagePromptTemplate 三种角色消息提示词模板。

无论看起来多么复杂，构造都遵循构造提示词的步骤，实例化内置的模板类为包装器对象，包装器来格式化外部的用户输入，包装器调用类方法输出提示词。

我们将上一个示例代码改造为Chat Model类型的提示词模板对象，下面是代码示例：

先导入聊天内置模板 （ChatPromptTemplate）和角色消息内置模板：

from langchain.prompts import (  
 ChatPromptTemplate,  
 PromptTemplate,  
 SystemMessagePromptTemplate,  
 AIMessagePromptTemplate,  
 HumanMessagePromptTemplate,  
)

改造思路是生成人类和系统的消息提示类型的提示词对象，实例化SystemMessagePromptTemplate 类和HumanMessagePromptTemplate 类为包装器，再实例化 ChatPromptTemplate类，将前面两个对象作为参数传递给ChatPromptTemplate类实例化后的包装器，调用其from\_messages 方法，生成为消息列表的提示词包装器：

先使用 from\_template 方法实例化消息类型的内置模板， SystemMessagePromptTemplate 类和 HumanMessagePromptTemplate 类。传入定义的template模板字符串，实例化人类和系统两种模板对象。

from langchain import PromptTemplate  
  
template = """  
You are an expert data scientist with an expertise in building deep learning models.   
"""  
  
system\_message\_prompt = SystemMessagePromptTemplate.from\_template(template)  
human\_template="Explain the concept of {concept} in a couple of lines"  
human\_message\_prompt = HumanMessagePromptTemplate.from\_template(human\_template)

然后将上面两个模板对象作为参数传入ChatPromptTemplate 类的实例化方法 from\_messages，转化为 ChatPromptTemplate 的提示词包装器：

chat\_prompt=ChatPromptTemplate.from\_messages([system\_message\_prompt,human\_message\_prompt])

打印这个结果：

ChatPromptTemplate(input\_variables=['concept'], output\_parser=None, partial\_variables={}, messages=[SystemMessagePromptTemplate(prompt=PromptTemplate(input\_variables=[], output\_parser=None, partial\_variables={}, template='\nYou are an expert data scientist with an expertise in building deep learning models. \n', template\_format='f-string', validate\_template=True), additional\_kwargs={}), HumanMessagePromptTemplate(prompt=PromptTemplate(input\_variables=['concept'], output\_parser=None, partial\_variables={}, template='Explain the concept of {concept} in a couple of lines', template\_format='f-string', validate\_template=True), additional\_kwargs={})])

最后使用包装器对象的format 方法，将用户的输入传递进包装器，组合为完整的提示词对象：

chat\_prompt.format\_prompt(concept="NLP")

调用format\_prompt 方法，获得的是 ChatPromptValue 对象：

ChatPromptValue(messages=[SystemMessage(content='\nYou are an expert data scientist with an expertise in building deep learning models. \n', additional\_kwargs={}), HumanMessage(content='Explain the concept of NLP in a couple of lines', additional\_kwargs={}, example=False)])

ChatPromptValue 有to\_string 和 to\_messages 方法。

chat\_prompt.format\_prompt(concept="NLP").to\_messages()

我们调用to\_messages 方法获得的结果：

[SystemMessage(content='\nYou are an expert data scientist with an expertise in building deep learning models. \n', additional\_kwargs={}),

HumanMessage(content='Explain the concept of NLP in a couple of lines', additional\_kwargs={}, example=False)]

我们调用to\_string 方法获得的结果：

'System: \nYou are an expert data scientist with an expertise in building deep learning models. \n\nHuman: Explain the concept of NLP in a couple of lines'

值得一提的是，LLM和Chat Model类型的提示词包装器，在实现方式上存在差异，包括他们所使用的内置模板以及实例化的方法。

对于LLM类型，其内置模板是PromptTemplate，而Chat Model类型则使用的是ChatPromptTemplate。PromptTemplate的实例化方法相对简单，只需传递input\_variables和template参数后，直接进行函数式调用或者使用from\_template的类方法进行调用。比如：

PROMPT = PromptTemplate.from\_template (template=template)

相比之下，ChatPromptTemplate 的实例化就复杂得多。它接受的参数是已经实例化的多个对象列表（例如，代码中的system\_message\_prompt和human\_message\_prompt）。如果我们把ChatPromptTemplate实例化的对象视为“大包”，那么传入的包装器就是“小包”，形成了一种“大包装小包”的情况。此外，这个“大包”的实例化类方法也与PromptTemplate不同，它使用的是from\_messages方法，而这个方法只接受消息列表的数组，比如如下代码中的变量 messages ：

messages = [

SystemMessagePromptTemplate.from\_template(system\_template),

HumanMessagePromptTemplate.from\_template("{question}"),

]

CHAT\_PROMPT = ChatPromptTemplate.from\_messages(messages)

尽管存在这些差异，但LLM和Chat Model类型的实例化仍有一定的规律，方便我们记忆和使用。以format为前缀的类方法，主要用于在实例化模板对象后，将外部用户的输入格式化到对象内。如果是实例化LLM类型的内置模板对象，我们使用的是format方法，而对于Chat Model类型，则使用format\_prompt方法。

类似地，以from\_为前缀的类型方法，主要用于实例化内置模板对象。PromptTemplate类只能使用的是from\_template，而ChatPromptTemplate 类则使用from\_messages 方法。

此外，为了实现这两种类型的相互转换，Chat Model类型使用format\_prompt方法实例化对象，生成的对象符合PromptValue数据模式（schema）。所有返回该数据模式的对象都包含以to\_为前缀的方法名，包括to\_string和to\_messages方法，分别用于导出字符串和包含角色的消息列表。

### 3.2.2 少样本提示词模板

少样本提示（FewShotPromptTemplate）是一种机器学习技术，它利用少量的样本（即为提示词）来引导模型进行特定任务的学习和执行。这些提示或示例提供了一种方式，让模型理解我们期望它完成的任务类型和风格。在给定的任务中，这些提示通常包含问题（或任务描述）以及相应的答案或解决方案。

例如，如果我们希望一个语言模型能够以某种特定的风格来回答用户的查询，那么我们可以给模型提供几个已经按照这种风格编写好的问题和答案对。这样，模型就能通过这些示例来理解我们期望的回答风格，并在处理新的用户查询时尽可能地模仿这种风格。

OpenAI的文档中也强调了这种技术的重要性。文档指出，尽管通常情况下，为所有示例提供适用的一般性指令比示例化所有任务的可能性更为高效，但在某些情况下，提供示例可能更为简单，尤其是当你想要模型复制一种难以明确描述的特定响应风格时。在这种情况下，"少样本提示"能够通过少量的示例，帮助模型理解并复制这种特定的响应风格，从而大大提高了模型的使用效率和效果。

FewShotPromptTemplate 是LangChain内置的一种提示词模板的类，其独特之处在于支持动态添加示例和选择示例。这样，示例在提示词中就不再是固定的，而是可以“动态变化”，能够适应不同的需求。这符合OpenAI文档中的建议，LangChain也认为这个内置模板是必须的，它可以为开发者节约大量时间。另外，LangChain还封装了示例选择器，以支持这种模板示例的动态化。

通过观察FewShotPromptTemplate类的源码，我们可以看到它如何实现这种动态化。这个类继承自StringPromptTemplate类，它的实例化方法和PromptTemplate类完全一样。然而，FewShotPromptTemplate类在参数上多了一些东西，例如examples（示例）和example\_selector（示例选择器）。这些参数使得我们可以在实例化对象时添加示例，或者在运行时动态选择示例。

即使这个类添加了一些新的特性，但它的使用方式仍然和PromptTemplate类一样。如果你想要在链组件上使用它，你只需要像使用PromptTemplate类一样来使用它。实际上，FewShotPromptTemplate类只是给实例化对象添加了更多的外部数据，即示例，但并没有改变它的使用方式。

如果你想引导模型得到更好的结果，那么你应该更多地使用FewShotPromptTemplate，因为它在PromptTemplate的基础上添加了示例功能。正如我们前面所说，添加示例的提示词会让模型得到更准确的回答。

FewShotPromptTemplate与PromptTemplate 的区别

我们现在讨论的是PromptTemplate 和 FewShotPromptTemplate 两个类的实例化过程以及它们之间的差异。这两个类都是LangChain的提示词模板，但它们有一些重要的区别。

PromptTemplate 类是一种基本的提示词模板，它接收一个包含变量的模板字符串和一个变量列表。例如，在以下的实例化过程中：

example\_prompt=PromptTemplate(input\_variables=["input","output"],

template="""

词语: {input}\n

反义词: {output}\n

""")

template是一个包含两个变量 {input} 和 {output} 的模板字符串，而 input\_variables 是一个包含这两个变量名称的列表。这个 PromptTemplate 对象可以用来生成提示词，例如通过调用 example\_prompt.format(input="好", output="坏") 可以生成提示词： “\n词语: 好\n\n反义词: 坏\n\n”。

然而， FewShotPromptTemplate 类提供了更高级的功能。它不仅继承了 PromptTemplate 的所有属性和方法，还添加了一些新的参数来支持少样本示例：

few\_shot\_prompt = FewShotPromptTemplate(

examples=examples,

example\_prompt=example\_prompt,

example\_separator="\n",

prefix="我们来玩个反义词游戏，我说词语，你说它的反义词\n",

suffix="词语: {input}\n反义词: ",

input\_variables=["input"],

)

在这个例子中， examples 是示例列表， example\_prompt 是用于格式化这些示例的 PromptTemplate 对象。而 prefix 和 suffix 则构成了用于生成最终提示词的模板，其中 suffix 还接收用户的输入。这种设计使得 FewShotPromptTemplate 可以在给出指导和接收用户输入的同时，还能够展示一系列的示例。

我们可以看到， FewShotPromptTemplate 的 prefix 和 suffix 参数的组合实际上等价于 PromptTemplate 的 template 参数，因此它们的目的和作用是一样的。然而， FewShotPromptTemplate 提供了更多的灵活性，因为它允许我们在提示词中添加示例。这些示例可以是硬编码在模板中的，也可以是动态选择的，这取决于我们是否提供了 ExampleSelector 对象。

总的来说， FewShotPromptTemplate 是 PromptTemplate 的扩展，它在保留了 PromptTemplate 的所有功能的同时，还提供了对少样本示例的支持。这使得我们可以更方便地使用少样本提示技术，而这种技术已被证明能够改善模型的性能。

### 3.2.3 少样本提示词模板的使用

这里，我们将通过示例探讨如何通过少样本提示词模板，如何加入一些具体的例子进入提示词模板，构造一个包含示例的提示词。

首先，我们需要认识新参数。在LangChain的FewShotPromptTemplate类中，参数example\_selector，example\_prompt，prefix 和 suffix具有以下含义和使用方式：

example\_selector: 这是一个ExampleSelector对象，用于选择要格式化进提示词的示例。如果你想让模型基于一组示例来生成响应，那么你可以提供一个ExampleSelector对象，该对象会根据某种策略（例如随机选择，基于某种标准选择等）从一组示例中选择一部分示例。如果你没有提供ExampleSelector对象，那么你应该直接提供一个示例列表（通过examples参数）。它是必填参数。

example\_prompt: 这是一个PromptTemplate对象，用于格式化单个示例。当你提供了一组示例（无论是直接提供示例列表，还是提供了ExampleSelector对象）后，FewShotPromptTemplate会用example\_prompt来格式化这些示例，生成最终的提示词。它是必填参数。

prefix 和 suffix 参数的组合实际上等价于 PromptTemplate 的 template 参数，因此它们的目的和作用是一样的。suffix 参数是必填的。

现在，让我们通过一个提供一个示例列表（通过examples参数）的例子来理解这个过程，这样很简单也很容易理解，少样本提示词模板的实例化过程。

examples = [

    {"input": "高", "output": "矮"},

    {"input": "胖", "output": "瘦"},

    {"input": "精力充沛", "output": "萎靡不振"},

    {"input": "快乐", "output": "伤心"},

    {"input": "黑", "output": "白"},

]

假设我们现在的任务是让模型进行反义词接龙。在这个任务中，我们会给模型一个词，然后期望模型返回这个词的反义词。因此，我们需要提供一些示例，例如 “高” 对应的反义词是 “矮”，“胖” 的反义词是 “瘦”，以此类推。

example\_prompt=PromptTemplate(input\_variables=["input","output"],

    template="""

词语:  {input}\n

反义词:  {output}\n

""")

我们像构造提示词模板对象一样，构造一个普通的 PromptTemplate 对象，用于格式化单个示例。

example\_prompt.format(\*\*examples[0])

# 打印的结果：'\n词语: 高\n\n反义词: 矮\n\n'

调用 format 方法，填入 input 和 output 参数。 当你写example\_prompt.format(\*\*examples[0])`时，\*\*examples[0] 会将第一个字典的键值对解开，然后作为关键字参数传递给format方法。这等价于example\_prompt.format(input="高", output="矮")。

然后，我们通过实例化 FewShotPromptTemplate类来设置我们的提示词模板。

few\_shot\_prompt = FewShotPromptTemplate(

  examples=examples,

  example\_prompt=example\_prompt,

  example\_separator="\n",

  prefix="我们来玩个反义词游戏，我说词语，你说它的反义词\n",

  suffix="现在轮到你了，词语: {input}\n反义词: ",

  input\_variables=["input"],

)

few\_shot\_prompt.format(input="好")

我们为模型设置一些标准的示例 （examples），以帮助模型理解任务需求。接下来，我们会实例化一个FewShotPromptTemplate类，然后在这里传入我们的示例。example\_prompt是一个PromptTemplate对象，用于格式化单个示例。我们还会为此设置一个前缀（prefix="我们来玩个反义词游戏，我说词语，你说它的反义词\n"）和一个后缀（suffix="词语: {input}\n反义词: "），这样可以帮助构造出一个结构清晰的提示词文本。

'我们来玩个反义词游戏，我说词语，你说它的反义词

词语: 高

反义词: 矮

词语: 胖

反义词: 瘦

词语: 精力充沛

反义词: 萎靡不振

词语: 快乐

反义词: 伤心

词语: 黑

反义词: 白

词语: 好

反义词: '

可以看到我们仍然是使用 FewShotPromptTemplate 类的函数式调用来实例化对象。然后，我们运行代码，看看模型能否正确地生成我们期望的结果。例如，如果我们输入“冷”，模型就应该返回“热”。这就是我们期望看到的反义词。

from langchain.llms import OpenAI

from langchain.chains import LLMChain

chain = LLMChain(llm=OpenAI(openai\_api\_key="这里填入OpenAI的密钥"),

prompt=few\_shot\_prompt)

chain.run("冷")

在这段代码中，我们首先实例化了一个LLMChain对象。这个对象是LangChain库中的一个核心组件，可以理解为一个执行链，它将各个步骤连接在一起，形成一个完整的运行流程。LLMChain对象在实例化时需要两个关键参数：一个是llm，这里我们使用了OpenAI提供的大型语言模型；另一个是prompt，这里我们传入的是我们刚刚创建的few\_shot\_prompt对象。

然后，我们通过调用LLMChain对象的run方法来运行这个执行链。在这个方法中，我们传入了一个字符串"冷"，这个字符串将作为输入传递给我们的few\_shot\_prompt对象。

' 热'

模型回应了我们“热”。这就是我们期望看到的反义词。

### 3.2.4 示例选择器

在实际应用开发中，我们面临的情况常常更为复杂。例如，我们可能需要将一篇新闻的摘要作为示例加入到提示词中。更具挑战性的是，我们还可能需要在提示词中加入大量的历史聊天信息或者从外部知识库获取的数据。然而，大型语言模型可以处理的字数是有限的容量。如果我们提供的每个示例都是一篇新闻的摘要，那么很可能就会超过模型能够处理的字数。

为了解决这个问题，LangChain在 FewShotPromptTemplate 类上，设计了示例选择器（Example Selector）的参数。示例选择器的作用是在传递给模型的示例中进行选择，以确保示例的数量和长度不会超过模型的处理能力。这样，即使我们有大量的示例，也能够确保模型能够有效地处理我们的提示词，而不会因为示例过多或者过长而导致模型无法处理。而且，尝试适应所有示例可能会很快变得非常昂贵，尤其是在计算资源和时间上。

这就是示例选择器发挥作用的地方，它帮助我们选择最适合的示例来提示模型。示例提示选择器（Example Selector）提供了一套工具，来解决这个问题。这些工具能基于策略选择合适的例子，如根据例子的长度、输入与例子之间的n-gram重叠分数来评估其相似度打分、找到与输入具有最大余弦相似度的例子, 以及多样性等因素来选择例子, 从而保持提示成本的相对稳定。

根据长度选择示例，是很普遍和现实的需求，以下是根据长度选择示例的代码，在上一节少样本提示词模板的实例代码里，我们没有提供示例选择器对象，而是直接提供一个示例列表（通过examples参数）。这一次我们提供 ExampleSelector 参数，使用示例选择器，选择的策略工具根据长度选择示例的LengthBasedExampleSelector 类，其他几种策略工具类，Langchain 都设计好了，开发者可以直接导入使用。

在这段代码中，首先导入了langchain库的LengthBasedExampleSelector类, 其他均重复上一节少样本提示词模板的实例代码。这个类是一个示例选择器，用于根据指定的长度选择示例。

from langchain.prompts.example\_selector import LengthBasedExampleSelector

然后，实例化了一个LengthBasedExampleSelector对象，传入了之前定义的示例（examples）和示例提示模板（example\_prompt），并设置了最大长度（max\_length）为25。这意味着选择器将选择那些长度不超过25的示例。

example\_selector = LengthBasedExampleSelector(

  examples=examples,

  example\_prompt=example\_prompt,

  max\_length=25,

)

接着，创建了一个FewShotPromptTemplate对象，这次传入了新创建的示例选择器（example\_selector）以及其他参数。这样，生成的提示词将根据选择器选择的示例来生成。

example\_selector\_prompt = FewShotPromptTemplate(

  example\_selector=example\_selector,

  example\_prompt=example\_prompt,

  example\_separator="\n",

  prefix="我们来玩个反义词游戏，我说词语，你说它的反义词\n",

  suffix="现在轮到你了，词语: {input}\n反义词: ",

  input\_variables=["input"],

)

example\_selector\_prompt.format(input="好")

当我们调用example\_selector\_prompt.format(input="好")后，程序将根据input值和示例选择器来生成一个提示词。

'我们来玩个反义词游戏，我说词语，你说它的反义词\n\n\n词语: 高\n\n反义词: 矮\n\n\n\n词语: 胖\n\n反义词: 瘦\n\n\n现在轮到你了，词语: 好\n反义词: '

在结果中，我们发现并没有所有的示例都出现在了生成的提示词中，这是因为我们设置的最大长度（max\_length）为25，因此一些过长的示例被选择器过滤掉了。

example\_selector = LengthBasedExampleSelector(

  examples=examples,

  example\_prompt=example\_prompt,

  max\_length=100, # 修改25为100

)

如果我们将最大长度参数改为100（max\_length=100），那么所有的示例都将被选择，因为所有示例的长度都不超过100。

[{'input': '高', 'output': '矮'}, {'input': '胖', 'output': '瘦'}, {'input': '精力充沛', 'output': '萎靡不振'}, {'input': '快乐', 'output': '伤心'}, {'input': '黑', 'output': '白'}]

这段代码展示了如何使用基于长度的示例选择器（LengthBasedExampleSelector）和少样本提示模板（FewShotPromptTemplate）来创建复杂的提示词。这种方法可以有效地管理复杂的示例集，确保生成的提示词不会因为过长而被截断。

实例选择器（Example Selector）是一种用于选择需要在提示词中包含的示例的工具。如果你有大量的示例，实例选择器可以帮助你确定哪些示例应该被包含在提示词中。

在LangChain中，我们提供了多种实例选择器，分别实现了不同的选择策略：

1. 基于长度的选择器（LengthBasedExampleSelector）：这个选择器根据示例的长度来选择示例。这在你担心提示词长度超过模型处理窗口长度时非常有用。对于较长的输入，它会选择较少的示例，而对于较短的输入，它会选择更多的示例。

2. 最大边际相关性选择器（MaxMarginalRelevanceExampleSelector，MMR）：这个选择器根据示例与输入的相似度以及示例之间的多样性来选择示例。它通过找到与输入最相似（即嵌入向量的余弦相似度最大）的示例，然后迭代地添加示例，同时对已选择的示例进行惩罚。

3. 基于n-gram重叠的选择器（NGramOverlapExampleSelector）：这个选择器根据示例与输入的n-gram重叠度来选择和排序示例。n-gram重叠度是一个介于0.0和1.0之间的浮点数。选择器还允许设置一个阈值，重叠度低于或等于阈值的示例将被排除。

4. 基于相似度的选择器（SemanticSimilarityExampleSelector）：这个选择器根据示例与输入的相似度来选择示例。它通过找到与输入最相似（即嵌入向量的余弦相似度最大）的示例来实现。

LangChain设计实例选择器的目的是为了帮助开发者在面对大量示例时，能够有效地选择最适合当前输入的示例，以提升模型的性能和效率。

在这些示例选择器中，实例化参数的确有所不同。它们都需要传入基础的参数，如examples和example\_prompt，但根据选择器的不同，还有一些额外的参数需要设置。

对于LengthBasedExampleSelector，除了examples和example\_prompt外，还需要传递max\_length参数来设置示例的最大长度。

example\_selector = LengthBasedExampleSelector(

examples=examples,

example\_prompt=example\_prompt,

max\_length=25,

)

对于MaxMarginalRelevanceExampleSelector，除了传入示例（examples）外，还需要传入一个用于生成语义相似性测量的嵌入类（OpenAIEmbeddings()），一个用于存储嵌入和执行相似性搜索的VectorStore类（FAISS），以及需要生成的示例数量（k=2）。

example\_selector = MaxMarginalRelevanceExampleSelector.from\_examples(

examples,

OpenAIEmbeddings(),

FAISS,

k=2,

)

对于NGramOverlapExampleSelector，除了examples和example\_prompt外，还有一个threshold参数用于设定选择器的停止阈值。

example\_selector = NGramOverlapExampleSelector(

examples=examples,

example\_prompt=example\_prompt,

threshold=-1.0,

)

对于SemanticSimilarityExampleSelector，除了传入示例（examples）外，还需要传入一个用于生成语义相似性测量的嵌入类（OpenAIEmbeddings()），一个用于存储嵌入和执行相似性搜索的VectorStore类（Chroma 或者其他的VectorStore类都可以），以及需要生成的示例数量（k=1）。

example\_selector = SemanticSimilarityExampleSelector.from\_examples(

examples,

OpenAIEmbeddings(),

Chroma,

k=1

)

每种选择器都有其独特的参数设置，以满足不同的示例选择需求。参数设置并不一样，但是使用的方式跟LengthBasedExampleSelector 类是一样。实例化自身后，通过 example\_selector 参数传递给 FewShotPromptTemplate 类。

我们应该注意的是，每一种示例选择器都可以通过函数的方式实例化，或者使用类方法from\_examples来实例化。比如 MaxMarginalRelevanceExampleSelector 类我们使用类方法from\_examples来实例化，而LengthBasedExampleSelector 则是函数的方式实例化。

### 3.2.5 扩展提示词模板

LangChain提供了极其灵活的提示词模板方法和组合提示的方式，满足各种开发需求。在所有的这些方法中，基础模板和少样本提示词模板是最基础的，其他所有的方法都在此基础上进行扩展。

LangChain提供了一套默认的提示词模板，可以生成适用于各种任务的提示。然而，可能会出现默认提示词模板无法满足你的需求的情况。例如，你可能需要创建一个带有特定动态指令的提示词模板。在这种情况下，LangChain 支持你可以创建一个自定义的提示词模板。

为了个性化大语言模型应用，你可能需要将大语言模型应用与特定用户的最新信息进行组合。特征库可以很好地保持这些数据的新鲜度，而LangChain提供了一种方便的方式，可以将这些数据与大语言模型应用进行组合。做法是从提示词模板内部调用特征库，检索值，然后将这些值格式化为提示。

针对聊天模型需求，LangChain提供了不同类型的消息提示词模板。最常用的是AIMessagePromptTemplate，SystemMessagePromptTemplate和HumanMessagePromptTemplate，它们分别创建一个AI消息、系统消息和人类消息。

此外，LangChain还支持 “Partial” 提示词模板，也就是说，传入一部分所需的值，以创建一个只期望剩余子集值的新提示词模板，适用于已经创建了提示模板对象，但是还没有明确的输入变量的场景。LangChain以两种方式支持这一点：实例化的时候指定属性值（partial\_variables={"foo": "foo"})）或者实例化为一个实例对象后调用partial 方法。

我们可以通过PipelinePrompt来组合多个提示。这在你希望重用部分提示时非常有用。

full\_template = """{introduction}

{example}

{start}"""

full\_prompt = PromptTemplate.from\_template(full\_template)

input\_prompts = [

("introduction", introduction\_prompt),

("example", example\_prompt),

("start", start\_prompt)

]

pipeline\_prompt=PipelinePromptTemplate(final\_prompt=full\_prompt,pipeline\_prompts=input\_prompts)

PipelinePromptTemplate实例化的时候，pipeline\_prompts 属性设置为一个包含三个模板对象的列表，并且设置了final\_prompt 属性的模板字符串。将这三个模板对象与模板字符串一起整合为一个完整的提示词对象。

另外，LangChain为了将提示存储为文件提供了支持的文件类型有JSON 和 YAML 的 load\_prompt 方法。这样可以方便地共享、存储和版本控制提示。

{

"\_type": "few\_shot",

"input\_variables": ["adjective"],

"prefix": "Write antonyms for the following words.",

"example\_prompt": {

"\_type": "prompt",

"input\_variables": ["input", "output"],

"template": "Input: {input}\nOutput: {output}"

},

"examples": "examples.json",

"suffix": "Input: {adjective}\nOutput:"

}

例如你有一个JSON 文件，里面定义了实例化提示词模板类的参数。

prompt = load\_prompt("few\_shot\_prompt.json")

print(prompt.format(adjective="funny"))

使用这个方法，我们很便利地利用外部文件，实现了自己的少样本提示词模板。

Write antonyms for the following words.

Input: happy

Output: sad

Input: tall

Output: short

Input: funny

Output:

最后，PromptTemplate会验证模板字符串，检查input\_variables是否与模板中定义的变量匹配。你可以通过将validate\_template设为False来禁用这种行为。这意味着，如果你确信你的模板字符串和输入变量是正确匹配的，你可以选择关闭这个验证功能，以节省一些额外的计算时间。PromptTemplate默认使用Python f-string作为其模板格式。然而，它也可以使用其他格式，如jinja2，这可以通过template\_format参数来指定。这意味着，除了Python的f-string格式，你还可以选择使用像jinja2这样的更强大、更灵活的模板引擎，以适应更复杂的模板格式需求。

## 3.3 输出解析器

在使用 GPT-4 或者类似的大型模型时，一个常见的挑战是如何将模型生成的输出转化为我们可以在代码中直接使用的格式。这里，我们会使用 LangChain 的 输出解析器（OutputParsers） 工具来解决这个问题。

虽然语言模型输出的文本信息可能非常有用，但开发的应用与真实的软件数据世界连接的时候，我们更希望得到的不仅仅是文本，而是更加结构化的数据。为了在应用程序中展示这些信息，我们需要将这些输出转换为某种常见的数据格式。我们可以编写一个函数来提取这个输出，但这并不理想。比如在提示词的模型指导加上“请输出答案为JSON格式”，模型会返回字符串形式的JSON，我们还需要通过函数将其转化为JSON对象。但是在实践中，我们常常会遇到异常问题，例如返回的字符串JSON无法被正确解析。

处理生产环境中的数据时，我们更可能会遇到千奇百怪的输入，导致模型的响应无法解析，增加额外的补丁来进行处理异常。这就使得整个处理流程变得更为复杂。

结构化数据,如数组或JSON对象, 在软件开发中起着至关重要的作用, 它提高了数据处理的效率，便利了数据的存储和检索，支持了数据分析，并且有助于提高数据质量。

还有,大语言模型目前确实存在一些问题，例如机器幻觉，这是指模型在理解或生成文本时产生的错误或误解。另一个问题是为了显得“聪明”，模型有时候会加入不必要的冗长和华丽的语句，这可能会导致模型过度详细，显得“话痨”了。比如你提示的结尾加上“你的答案是：”，模型就不会“话痨”了。

在真实的开发环境中，我们不仅希望获取模型的输出结果，而且还希望能够进行后处理，比如解析输出的结构化数据。

这就是为什么在大语言模型的开发中，结构化数据，如数组或JSON对象，显得尤为重要。他们可以帮助我们更好地理解和处理模型的输出结果，比如通过解析输出的JSON对象，我们可以得到模型的预测结果，而不仅仅是一个长长的文本字符串。我们也可以根据需要对这些结果进行进一步的处理，例如提取关键信息，进行数据分析等。这样，我们不仅可以得到模型的“直接回答”，而且可以根据自己的需求进行定制化的**后处理**， 比如传递给下一个任务函数，从而更好地利用大语言模型。

### 3.3.1 输出解析器的功能

这就是输出解析器的用武之地。输出解析器包含两大功能：构造提示词模板的指令部分和解析大语言模型包装器的输出。看到这里你也许很奇怪，这个输出解析器跟提示词模板有什么关系？

确实，从名字上看，输出解析器（OutputParser）似乎与提示词没有关系，因为它听起来像是用于处理和解析大语言模型输出的工具。然而，实际上，OutputParser在构建提示词模板，指导模型的输出行为。这主要表现在以下两个方面：

1. 构造提示词模板的输出指令部分。指令部分是使用一段指导语句，这个指导语句作为提示词的一部分，可以告诉大语言模型我们希望它以何种格式产出结果。这在处理复杂任务时非常有用，例如，我们需要返回一种特定格式的数据，或者我们希望大语言模型按照某种特定的方式回答问题，这些要求我们都以注入的方式加到提示词模板里。

2. 解析大语言模型输出：当大语言模型产出结果后，输出解析器还可以将大语言模型的输出从一段普通的文本转化为更为结构化的数据。比如字典列表格式。

因此，虽然输出解析器的名字听起来与提示词无关，但实际上，它在构建和使用提示词的过程中发挥着重要的作用。通过输出解析器，我们可以更好地控制大语言模型的行为，并且更方便地处理大语言模型的输出。

在这章的伊始，我们提到提示词通常包括三个部分：对大语言模型的指导，用户的输入，以及用户的问题。输出解析器其实是在构造提示词的一部分：对大语言模型的指导，下面我们称为提示词模板的指令部分，指导大语言模型做出反应和行为的文字。再次强调它仅仅是提示词的其中之一，并没有什么特别之处。我们使用输出解析器的原因是它预制了很多数据类型的提示词，我们不需要单独为此付出努力。

举例来说，假设我们要让大语言模型解释“深度学习”这个概念。在这种情况下，我们可能会构建一个如下的提示词：“假设你是人工智能的专家，请你解释深度学习的概念是什么？例如：Q : 请你解释强化学习是什么？A: 强化学习是....” 在这里，“假设你是人工智能的专家”是我们构造提示词模板的指令部分，它告诉大语言模型需要以一个AI专家的身份来回答问题。“深度学习”是用户的输入，“请你解释概念是什么？”则是用户输入的问题。而那个“QA问答对”示例则是我们提供的少样本示例提示（此处是一个固定的示例），它可以帮助大语言模型理解并更好地回答我们的问题。

这是我们通常如何构建提示词。然而，在实际的应用中，我们可能需要更加复杂和精细的提示词，例如我们可能需要指定大语言模型的输出格式，或者需要考虑到一些特定的需求和限制。为了解决这个问题，我们可以使用OutputParser帮助我们自动化地生成大语言模型的指令部分，例如我们可以通过Pydantic的数据模式定义来生成大语言模型的输出格式指导。这样，我们就可以构建出更加规范和高效的提示词，同时也能使大语言模型的输出更加符合我们的需求。

Langchain 提供了一系列预设的输出解析器，这些输出解析器能够针对不同的数据类型给出合适的大语言模型输出指导，并解析大语言模型的输出。这些输出解析器包括：

1. BooleanOutputParser ：用于解析布尔值类型的输出。

2. CommaSeparatedListOutputParser ：用于解析逗号分隔的列表类型的输出。

3. DatetimeOutputParser ：用于解析日期时间类型的输出。

4. EnumOutputParser ：用于解析枚举类型的输出。

5. ListOutputParser ：用于解析列表类型的输出。

6. PydanticOutputParser ：用于解析符合 Pydantic 大语言模型的输出。

7. StructuredOutputParser ：用于解析具有特定结构的输出。

举例来说，CommaSeparatedListOutputParser 这个解析器就提供了用于指导大语言模型生成逗号分隔的列表类型输出的提示词。同时，解析器还包含一个 parse方法，能够将大语言模型的文本输出解析为逗号分隔的列表。这意味着，当你使用 CommaSeparatedListOutputParser 时，你不需要自己编写大语言模型输出指导的提示词，因为 Langchain 已经为你准备好了。

class CommaSeparatedListOutputParser(ListOutputParser):

"""Parse out comma separated lists."""

def get\_format\_instructions(self) -> str:

return (

"Your response should be a list of comma separated values, "

"eg: `foo, bar, baz`"

)

def parse(self, text: str) -> List[str]:

"""Parse the output of an LLM call."""

return text.strip().split(", ")

从 CommaSeparatedListOutputParser 这个解析器的源码可以很直观看到提示词的模型指导词部分：

"Your response should be a list of comma separated values, "

"eg: `foo, bar, baz`"

输出解析器的主要目标是提供预设的提示词来指导模型（如GPT-4），并对模型的输出进行解析，使其更符合我们的需求。一旦我们理解了这一点，我们就可以很容易地使用这些输出解析器。

例如，我们可以使用列表输出解析器 CommaSeparatedListOutputParser。我们首先实例化一个 CommaSeparatedListOutputParser 对象，然后我们可以使用 get\_format\_instructions() 方法获取预设的提示词。这个方法返回的字符串包含了如何格式化语言模型输出的指导。

对于CommaSeparatedListOutputParser，get\_format\_instructions() 方法会返回一条指导模型生成逗号分隔列表的指令。之后，我们可以将这个指令作为变量 format\_instructions 传入 PromptTemplate。这样我们的模型就知道我们希望得到的输出是一个逗号分隔的列表。

输出解析器还提供了 parse方法，它接收一个字符串（假定是模型的响应），并将其解析成某种结构。例如，对于 CommaSeparatedListOutputParser，parse 方法将接收一个字符串，并将其解析为一个逗号分隔的列表，改变了数据的格式。

总的来说，输出解析器为我们提供了一种便捷的方式，使我们能够通过预设的提示词来指导模型的输出，并且能够将模型的输出解析为我们所需要的数据。这使得我们能够更加容易地使用模型来完成实际任务。

3.3.2 输出解析器的使用在Langchain框架中，我们可以利用 PromptTemplate 对象的 partial 方法或在实例化 PromptTemplate 对象时传递 partial\_variables 参数，来增加或选择已经设置好的变量。这样做可以提高代码的灵活性，使得提示词的占位符变量可以根据需要动态地增加或减少。我们使用这种方式来为提示词模板添加指导词部分。

具体操作如下：首先，我们使用 output\_parser.get\_format\_instructions() 获取预设的指导词部分 format\_instructions。然后，我们在实例化 PromptTemplate 类时，将 format\_instructions 作为 partial\_variables 的一部分传入。如此，我们就在提示词模板中追加了提示词模板字符串 format\_instructions 变量。

以下是相关的代码示例：

format\_instructions = output\_parser.get\_format\_instructions()

prompt = PromptTemplate(

template="List five {subject}.\n{format\_instructions}",

input\_variables=["subject"],

partial\_variables={"format\_instructions": format\_instructions}

)

在这段代码中，PromptTemplate 的模板字符串 template 包含两个占位符变量 {subject} 和 {format\_instructions}。我们在实例化 PromptTemplate 对象时，除了传入 input\_variables=["subject"] 参数外，还通过 partial\_variables={"format\_instructions": format\_instructions} 参数预先填充了 {format\_instructions} 变量。这样，我们就成功地为提示词模板添加了输出解析器所提供的指导词部分。

我们现在通过下面的示例代码，完成输出解析器的二大功能 — 添加指导词和解析数据格式，而且展示如何运用到链组件上。

首先，我们引入了 CommaSeparatedListOutputParser，这是一个特定的输出解析器，其主要功能是将模型输出的逗号分隔的文本解析为列表格式。

from langchain.output\_parsers import CommaSeparatedListOutputParser

from langchain.prompts import PromptTemplate

from langchain.llms import OpenAI

output\_parser = CommaSeparatedListOutputParser()

然后，我们使用 output\_parser.get\_format\_instructions() 方法来获取预设的格式化指令。这些指令会指导模型如何将其输出格式化为一个逗号分隔的列表。接下来，我们创建了一个 PromptTemplate 对象。

format\_instructions = output\_parser.get\_format\_instructions()

prompt = PromptTemplate(

template="List five {subject}.\n{format\_instructions}",

input\_variables=["subject"],

partial\_variables={"format\_instructions": format\_instructions}

)

在这个提示模板中，我们定义了一个字符串模板，其中包含两个占位符变量：subject 和 format\_instructions。subject 是我们希望模型产生的列表主题，例如 "ice cream flavors"，而 format\_instructions 是我们之前从输出解析器中获取的预设格式化指令。这里我们引入 OpenAI 模型。

我们可以打印format\_instructions的结果是“Your response should be a list of comma separated values, eg: `foo, bar, baz`”

from langchain.chains import LLMChain

chain = LLMChain(llm=OpenAI(openai\_api\_key="填入OpenAI 的密钥"),

 prompt=prompt )

我们将 subject 的值设为 "ice cream flavors"，然后调用 prompt.format(subject="ice cream flavors") 方法，这将返回一个完整的提示字符串，包含指导模型产生五种冰淇淋口味的指令。

我们导入 LLMChain 链组件，为OpenAI模型类设置密钥，将PromptTemplate 类实例化后的 prompt 对象传入LLMChain 链。

output = chain("ice cream flavors")

运行这个链得到的是一个 JSON 对象，output['text']是模型回答的字符串，然后我们调用输出解析器的 parse() 方法解析这个字符串为一个列表。由于我们的输出解析器是 CommaSeparatedListOutputParser，所以它会将模型输出的逗号分隔的文本解析为列表。

output\_parser.parse(output['text'])

所以，最后得到的结果是一个包含五种冰淇淋口味的列表：

['Vanilla',

'Chocolate',

'Strawberry',

'Mint Chocolate Chip',

'Cookies and Cream']

### 3.3.2 Pydantic JSON解析器

JSON对象，这种格式最大的特点是人和机器都看得懂。

你可以把JSON对象想象成一个大家都认识的“信息盒子”。在这个“信息盒子”里，我们可以存储各种各样的信息，比如你的名字、你的年龄、你最喜欢的食物，甚至是你所有玩具的列表等等。这些信息都被整齐地放在“信息盒子”里，每一种信息都有自己的标签，比如“名字”、“年龄”、“食物”、“玩具”。

在我们开发语言模型应用的时候，我们经常用到这个“信息盒子”。因为它可以帮我们更好地整理和使用机器人的答案。比如，机器人可能会给我们一个包含很多信息的答案，而我们可以用这个“信息盒子”来把这些信息整理得更清晰，更易于理解和使用。

所以，JSON对象就像一个非常有用的“信息盒子”，可以帮助我们更好地使用和理解语言模型的答案。

请记住，大语言模型是有“缺陷”的抽象！你需要使用一个具有足够能力的模型来生成格式良好的JSON。在OpenAI 模型家族中，DaVinci可以做到这一点，但Curie的能力已经大幅下降。Langchain 这种输出解析器可以指定一个任意的JSON结构，并向大语言模型查询，输出符合该架构的JSON。

你可以使用Pydantic来声明你的数据模型。Pydantic的BaseModel就像一个Python数据类，但它具有实际的类型检查和强制转换功能。

下面是最简单的Pydantic JSON解析器代码：

导入语言模型 OpenAI和 提示词模板类。

from langchain.prompts import (  
 PromptTemplate,  
 ChatPromptTemplate,  
 HumanMessagePromptTemplate,  
)  
from langchain.llms import OpenAI  
from langchain.chat\_models import ChatOpenAI

获取格式化指令 PydanticOutputParser:

from langchain.output\_parsers import PydanticOutputParser  
from pydantic import BaseModel, Field, validator  
from typing import List

这里我们使用LLM 类型的模型包装器，并且配置模型型号为： text-davinci-003。

model\_name = "text-davinci-003"  
temperature = 0  
model = OpenAI(model\_name=model\_name, temperature=temperature)

定义你想要的数据格式：

# Define your desired data structure.  
class Joke(BaseModel):  
 setup: str = Field(description="question to set up a joke")  
 punchline: str = Field(description="answer to resolve the joke")  
  
 # You can add custom validation logic easily with Pydantic.  
 @validator("setup")  
 def question\_ends\_with\_question\_mark(cls, field):  
 if field[-1] != "?":  
 raise ValueError("Badly formed question!")  
 return field  
  
  
# And a query intented to prompt a language model to populate the data structure.  
joke\_query = "Tell me a joke."  
  
# Set up a parser + inject instructions into the prompt template.  
parser = PydanticOutputParser(pydantic\_object=Joke)  
  
prompt = PromptTemplate(  
 template="Answer the user query.\n{format\_instructions}\n{query}\n",  
 input\_variables=["query"],  
 partial\_variables={"format\_instructions": parser.get\_format\_instructions()},  
)  
  
\_input = prompt.format\_prompt(query=joke\_query)  
  
output = model(\_input.to\_string())  
  
parser.parse(output)

将用户的输入 ice cream flavors 绑定到提示模板中，使用LLM 类型的模型包装器与模型平台进行交互:

\_input = prompt.format(subject="ice cream flavors")  
output = model(\_input)

调用解析器的parse方法,解析数据为Pydantic JSON 格式.

output\_parser.parse(output)

最终的结果是:

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

### 3.3.3 结构化输出解析器

OutputParsers是一组工具，其主要目标是处理和格式化模型的输出。它包含了多个部分，但对于我们实际的开发需求来说，其中最关键的部分是结构化输出解析器（StructuredOutputParser）。这个工具可以将模型原本返回的字符串形式的输出，转化为可以在代码中直接使用的数据结构。

使用结构化输出解析器时，我们首先需要定义我们所期望的输出格式。解析器将根据这个定义来生成模型的提示，从而引导模型产生我们所需的输出。例如，我们想要使用StructuredOutputParser来获取多个字段的返回值。尽管Pydantic/JSON解析器更强大，但在早期实验中，我们选择的数据结构只包含文本字段。以下是实例代码：

首先，我们从langchain库导入所需的类和方法：

from langchain.output\_parsers import StructuredOutputParser, ResponseSchema

from langchain.prompts import PromptTemplate, ChatPromptTemplate, HumanMessagePromptTemplate

from langchain.llms import OpenAI

from langchain.chat\_models import ChatOpenAI

然后，我们定义我们想要接收的响应模式：

response\_schemas = [ ResponseSchema(name="answer", description="answer to the user's question"), ResponseSchema(name="source", description="source used to answer the user's question, should be a website.") ]

output\_parser = StructuredOutputParser.from\_response\_schemas(response\_schemas)

接着，我们获取一个包含响应应该如何格式化的指令的字符串，然后将其插入我们的提示中：

format\_instructions = output\_parser.get\_format\_instructions()

prompt = PromptTemplate( template="answer the users question as best as possible.\n{format\_instructions}\n{question}",input\_variables=["question"],

partial\_variables={"format\_instructions": format\_instructions} )

现在，我们可以使用这个提示格式化一个要发送给LLM类型的模型包装器的提示，然后解析返回的结果：

model = OpenAI(temperature=0)

\_input = prompt.format\_prompt(question="what's the capital of france?")

output = model(\_input.to\_string())

output\_parser.parse(output)

返回的结果如下：

{'answer': 'Paris', 'source': 'https://www.worldatlas.com/articles/what-is-the-capital-of-france.html'}

接下来是一个在Chat Model的模型包装器中使用这个方法的例子：

chat\_model=ChatOpenAI(temperature=0)

prompt = ChatPromptTemplate( messages=[ HumanMessagePromptTemplate.from\_template("answer the users question as best as possible.\n{format\_instructions}\n{question}") ], input\_variables=["question"],partial\_variables={"format\_instructions":format\_instructions} )

\_input = prompt.format\_prompt(question="what's the capital of france?")

output = chat\_model(\_input.to\_messages())

output\_parser.parse(output.content)

返回的结果如下：

{'answer': 'Paris', 'source': 'https://en.wikipedia.org/wiki/Paris'}

这就是如何使用langchain库的StructuredOutputParser和PromptTemplate来格式化和解析语言模型的输入和输出。

## 3.4 本章小结

本章将详细介绍Langchain框架的核心模块 — 模型I/O。我们重点探讨了三个主题：

1. 模型 I/O 的模型: Langchain为OpenAI为主的大语言模型平台，设计了两种包装器用于简化与平台的底层API交互的流程，两种包装器为LLM类型和Chat Model类型。

2. 模型 I/O 的 Input: 这些包装器的输入的内容是什么？提示词模板。

3. 模型I/O 的 Output: 这些包装器的输出的结果是什么？输出解析器用于构造提示词模板的指令部分以指导大语言模型的输出和解析包装器的输出。

需要强调的是，这些包装器的输入和输出，都是由提示词模板来完成的。这是理解模型I/O的关键。这也是为什么在本章开始便强调Langchain框架的一切都围绕提示词模板而构建的原因。如果我们将这个讨论进一步简化，那么这一章实质上讲述的就是两件事情：首先，我们设计了适配主流大语言模型底层API交互的两种包装器；其次，我们如何在langchain框架中编写和应用提示词模板。

总的来说，这一章我们走过了从模型输入到输出的全过程，对大语言模型的运作机制有了更深入的理解。这是开发应用的最基础，也是打好基本功的关键。只有深入理解这些基础知识，我们才能在大语言模型的应用开发中乘风破浪，开创新的可能。

4数据增强

在这一章，我们将主要探讨如何在Langchain框架中连接外部的数据，即数据连接模块（Data Connection）。我们的生活周围充斥着各种各样的数据，例如本地的文档、网页上的知识、企业内部的知识库、各类研究报告、软件数据库以及聊天的历史记录等。这些数据，无论是广泛的互联网数据，还是具有特定价值的企业内部数据，都是我们构建和优化大语言模型的重要资源。

但你可能会问，既然我们已经有了强大的大语言模型，例如OpenAI的gpt-4，为什么我们还需要连接外部的数据呢？原因其实很简单，那就是大语言模型的 “知识 ”是有限的。以OpenAI的gpt-4为例，它的数据集只训练到2021年的9月份，也就是说，这个时间之后的数据并没有被模型学习和理解。所以，到2023年下旬，我们仍会看到ChatGPT在其界面上提示： ChatGPT可能会产生关于人、地点或事实的不准确信息。 这是因为模型在训练数据集之外的知识领域中，其预测能力是受限的。

除此之外，我们还需要个性化的知识，比如企业的内部知识。想象一下，如果你是一个企业，你可能会希望你的聊天机器人能够理解和回答一些关于你的产品或服务的具体问题，这些问题的答案往往需要依赖于你企业内部的专有知识。大语言模型无法直接访问这些知识，因此我们需要将这些知识以某种方式连接到我们的大语言模型中。

连接外部数据不仅可以修复大语言模型的“知识 ”缺失，而且还能让我们开发的应用程序更加“可靠”。当我们的模型需要回答一个问题时，它可以根据真实的外部数据进行回答，而不是仅仅依赖于它在训练时学习的知识。例如，当我们问模型"2023年的新冠病毒疫苗有哪些副作用？”时，模型可以根据最新的医学研究报告来提供答案，而不是依赖于它在两年前学习的可能已经过时的知识。

这些大语言模型不仅需要连接外部的数据，填补缺失的“知识”，同时还受到了提示词的限制。正如我们在模型I/O 的提示词模板数据来源提到的，构建好的提示词模板需要依靠外部数据。然而，这种提示词的字符数量是有限的，这就是我们所说的Max Tokens 概念。

Max Tokens，是指模型在执行任务时能够接收和处理的最大 Token 数量。这个数量包括了我们提交给模型的提示词以及模型返回给我们的答案。也就是说，我们同模型的交互，无论是请求还是响应，都会累积到最大Token数量的计数中。比如我们发出的提示词数量是1000 Tokens，返回的也是1000 Tokens，那么总共Tokens数量就是2000 Tokens，而不仅仅是1000 Tokens。

另外，在大语言模型中，一个Token并不是指一个字符，而是指一个词或者一个词的一部分。对于英文，一个Token可能是一个完整的单词，也可能是一个单词的一部分。对于中文，通常一个汉字就是一个Token。这是由语言模型的编码方式决定的。

让我们以英文为例，英文通常会被分割为子词或者字符。例如，“apple”可能被分割为一个Token，即["apple"]，而“apples”可能被分割为两个Token，即["apple", "s"]。这是因为模型在训练时学习到，“s” 常常用于表示复数。所以，“apples”被分割为两个Token。

对于中文，由于其语言特性，通常每个字符就是一个Token，即每个汉字都是一个Token。但是在某些特殊情况下，如一些复杂的或者不常见的汉字，可能会被编码为两个或者更多的Token。比如，例如，gpt-4模型的Max Tokens是8192个 Tokens 。如果我们将 8192 个 Tokens 全部用于中文，实际能处理的汉字数量可能会比8192少。

以OpenAI的产品为例，我们可以看到，不同的模型型号有着不同的Max Tokens设置。例如，gpt-4模型的Max Tokens是8192个，这意味着它一次能处理8192个Tokens，假设平均每个英文单词对应0.75个Token，那么这个模型大约可以处理约 10922 个英文单词（8192除以0.75）。而gpt-4-32k模型的Max Tokens则是32768个，是gpt-4的四倍，这使得它能处理更长的文本，理解更复杂的上下文。

因此，我们在使用这些大型语言模型时，需要了解其Max Tokens的限制，并根据实际需要选择合适的模型。同时，也需要注意，随着技术的不断更新和迭代，模型的性能和能力也会有所提高，我们需要密切关注这些变化，以确保我们能够最大限度地利用这些强大的工具。

为了解决这些大型语言模型的限制问题，LangChain设计了数据增强模块。这个模块的目的是检索到用户输入的问题相关的外部数据，包括筛选相关问题和相关的文档。然后，这些相关数据会形成提示词模板，提交给LLM或Chat Model类型的模型包装器。这些模型包装器封装了各个大语言模型平台的底层API，使得我们可以方便地与这些平台进行交互，获取大语言模型平台的输出。

然而，一旦我们加载了这些外部的文档数据，我们经常会希望对它们进行转换以更好地适应我们的应用。最简单的例子是我们可能希望将一个长文档分割成较小块，避免超过gpt-4模型的Max Tokens。为了实现这一目标，LangChain框架提供了一系列内置的文档转换器，使得分割、组合、过滤和以其他方式操作文档变得容易。例如，我们可以使用这些转换器将一篇长篇的研究报告分割成一系列的小段落，每个段落都可以作为一个独立的输入提交给我们的模型。

数据增强模块的LEDVR

数据增强模块是一个多功能的数据增强工具集成，可以方便记忆为LEDVR，分别代表加载器 (Loader), 嵌入模型包装器(Text Embedding Model), 文档转换器 (Document Transformers)，向量存储库 (VectorStore) 和检索器 (Retriever) 。

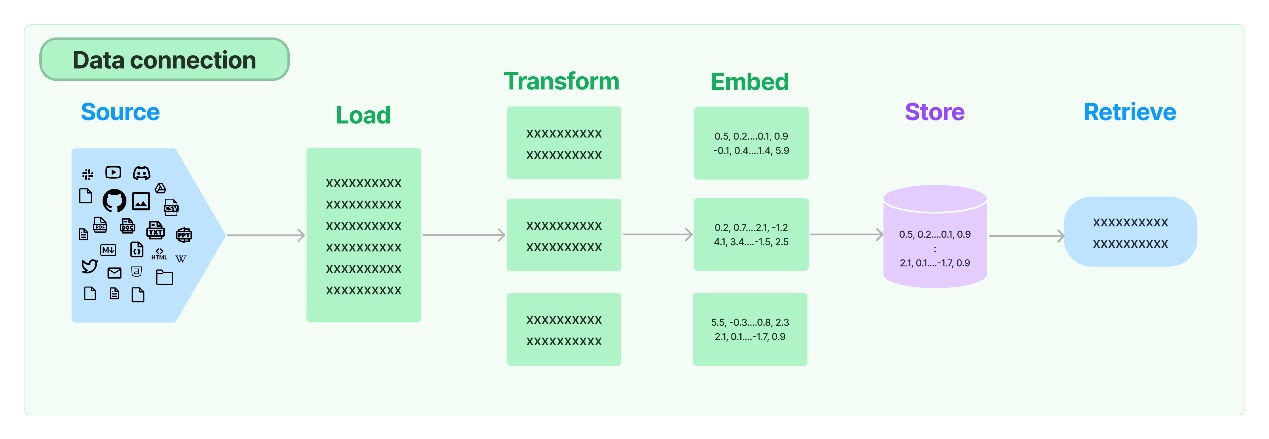
L代表加载器 (Loader),，负责从各种来源加载数据作为Document，其中Document是由文本和相关元数据组成的。无论你的数据是简单的.txt文件，还是任何网页的文本内容，Loader都可以将它们加载为Document。

E代表嵌入模型包装器(Text Embedding Model)，是Langchain 封装了各个模型平台的Text Embedding Model 类型的 API 接口。效果同模型I/O 模块的LLM 和 Chat Model 类型的模型包装器。

D代表文档转换器 (Document Transformers)。一旦你加载了文档，你通常会希望将它们转化以更好地适应你的应用。数据增强模块提供了一系列内置的文档转换器，使得分割、组合、过滤和其他方式的操作文档变得简单。比如OpenAI 平台的主要文本嵌入模型的型号是“text-embedding-ada-002”。

V代表向量存储库 (VectorStore)，是用于存储和检索嵌入向量的工具，处理的数据是模型平台的文本嵌入模型（Text Embedding Model）转换的向量数据，这是处理非结构化数据的一种常见方法。VectorStore负责存储嵌入数据并为你执行向量搜索。在查询时，可以将非结构化查询嵌入，并检索与嵌入查询“最相似”的嵌入向量。

R代表检索器 (Retriever)，它是一个接口，返回给定非结构化查询的文档。它比VectorStore更通用。检索器无需存储文档，只需要返回（或检索）它。



LEDVR 的示例代码

让我们以一个具体的代码示例来解析LangChain数据处理流程中的各个步骤。在这个例子中，我们将使用LangChain的各种工具，包括文档加载器L、文本嵌入模型E、文档转换器D、向量存储库V和检索器R，来完成一个从加载、转换、嵌入、存储到检索的完整流程。

首先，我们使用文档加载器L，创建一个WebBaseLoader实例，用于从网络加载数据。在这个例子中，我们加载的是一个博客文章。加载器读取该网址的内容，并将其转换为一份文档数据。

from langchain.document\_loaders import WebBaseLoader

loader = WebBaseLoader("http://developers.mini1.cn/wiki/luawh.html")

data = loader.load()

随后，我们使用文本嵌入模型E，将这些分割后的文本数据转换为向量数据。我们创建一个OpenAIEmbeddings实例，用于将文本转换为向量。

from langchain.embeddings.openai import OpenAIEmbeddings

embedding = OpenAIEmbeddings(openai\_api\_key="填入你的OpenAI密钥")

接下来，文档转换器D，我们需要将这份文档数据进行分割，转换为 Document 数据。这是为了让数据更好地适应数据增强模块的工作流程。我们创建一个RecursiveCharacterTextSplitter实例，并指定每个片段的大小为500个字符。然后，我们使用这个分割器将文档数据分割成多个片段。

from langchain.text\_splitter import RecursiveCharacterTextSplitter

text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=500, chunk\_overlap=0)

splits = text\_splitter.split\_documents(data)

然后，向量存储库V，我们创建一个向量存储库：FAISS实例，用于存储这些向量数据。

from langchain.vectorstores import FAISS

vectordb = FAISS.from\_documents(documents=splits, embedding=embedding)

最后，检索器R，我们需要在这些数据中进行检索。我们创建一个ChatOpenAI实例和一个MultiQueryRetriever实例，用于执行检索操作。在这个例子中，我们检索的问题是“Task Decomposition的方法有哪些？”。

from langchain.chat\_models import ChatOpenAI

from langchain.retrievers.multi\_query import MultiQueryRetriever

question = "LUA的宿主语言是什么?"

llm = ChatOpenAI(openai\_api\_key="填入你的OpenAI密钥")

retriever\_from\_llm = MultiQueryRetriever.from\_llm(

retriever=vectordb.as\_retriever(), llm=llm

)

docs = retriever\_from\_llm.get\_relevant\_documents(question)

通过这个例子，最后我们获得了四个与相关的Document数据。打印第一条我们找到了“LUA的宿主语言是什么?”这个问题的答案，“Lua提供了非常易于使用的扩展接口和机制：由宿主语言(通常是C或C++)提供这些功能”。

print(docs[0])

<langchain.vectorstores.faiss.FAISS at 0x228dfa4b050>

Document(page\_content='Lua提供了非常易于使用的扩展接口和机制：由宿主语言(通常是C或C++)提供这些功能，Lua可以使用它们，就像是本来就内置的功能一样。 其它特性:', metadata={'source': '<http://developers.mini1.cn/wiki/luawh.html>', 'title': '什么是Lua编程 | 开发者脚本帮助文档', 'description': '迷你世界开发者接口文档', 'language': 'zh-CN'})

我们可以看到LangChain如何将文档加载器L、文本嵌入模型E、文档转换器D、向量存储库V和检索器R有机地组合在一起，形成了一个从加载、转换、嵌入、存储到检索的完整流程。

两种数据类型

在数据增强模块中，操作的主要是两种类型的数据：文档数据和向量数据。这两种数据类型是LangChain数据增强模块中处理流程的类型，它们支持数据在这个模块中的自由流通和转换。

文档数据是数据增强模块的起点。这些数据主要通过加载器从各种不同的源加载进来。无论是简单的文本文件，还是网页内容，甚至是YouTube视频的转录，都可以被加载为文档数据。在这个过程中，每一份文档都被视为一个包含文本和相关元数据的单元。

一旦文档数据被加载进来，它们就可以被传递给文档转换器进行处理。这些处理最主要的功能是分割文档，其他功能还有压缩文档、过滤文档。这些转换过程使得文档数据更好地适应我们的应用需求。

这些经过处理的文档数据会被传递给文本嵌入模型，在这里，它们会被转化为向量数据。向量数据实际上是一种将非结构化的文本转化为一列浮点数的表示形式。这些向量数据会被存储在向量存储库中，并在需要时进行检索。检索过程由检索器完成，它根据用户的查询返回相应的数据。

总的来说，数据在LangChain的数据增强模块中，通过文档加载器L、文本嵌入模型E、文档转换器D、向量存储库V和检索器R，形成了一个从加载、转换、嵌入、存储到检索的完整流程。在这个流程中，数据被规范化为文档数据和向量数据两种类型，使得数据能够自由地在各个组件之间流通和转换。

## 4.1 L文档加载器

在LangChain的数据处理流程中，Document Loaders起着至关重要的作用。它们被用来从各种来源加载数据，并将其转换为“文档”（Document）的形式。

一个“文档”可以理解为一段文本及其相关元数据。例如，我们可以有专门用于加载简单.txt文件的文档加载器，也可以有加载任何网页的文本内容的文档加载器，甚至还可以有加载YouTube视频转录文本的文档加载器。不同类型的文档加载器，使得LangChain可以从各种各样的数据源中抽取并处理数据。

这些文档加载器都会暴露出一个名为”load”的方法，用于从配置的数据源加载数据作为文档。这个”load”方法可以从指定的数据源中读取数据，并将其转换成一份或多份文档。这使得LangChain能够处理各种形式的输入数据，不仅仅限于文本文件，还可以是网页、视频字幕等等。

值得注意的是，文档加载器还可以选择性地实现一个名为”lazy load”的方法，这个方法的作用是实现数据的懒加载，即在需要时才将数据加载到内存中。这种方式可以有效减少内存占用，并提高数据处理的效率。

总的来说，通过Document Loaders，LangChain可以将各种各样的数据源无缝地转换为标准的文档形式，为后续的数据处理和分析提供了坚实的基础。

下面是最简单的文档加载器的代码示例：

加载简单.txt文件的文档加载器。

from langchain.document\_loaders import TextLoader  
# 代码仓库中有这个文件，也可以加入自己的测试文件。如果是文件中包含中文，请指定encoding="utf-8"  
loader = TextLoader(file\_path="./index.md",encoding="utf-8")  
loader.load()

文件中的所有内容都被加载到了Document 数据中。

[Document(page\_content='在语言模型中，一个Token并不是指一个字符，而是指一个词或者一个词的一部分。对于英文，一个Token可能是一个完整的单词，也可能是一个单词的一部分。对于中文，通常一个汉字就是一个Token。这是由语言模型的编码方式决定的。\n\n让我们以英文为例。在许多NLP任务和一些语言模型中，英文通常会被分割为子词或者字符。例如，“apple”可能被分割为一个Token，即["apple"]，而“apples”可能被分割为两个Token，即["apple", "s"]。这是因为模型在训练时学习到，“s”常常用于表示复数。所以，“apples”被分割为两个Token。\n\n对于中文，由于其语言特性，通常每个字符就是一个Token，即每个汉字都是一个Token。但是在某些特殊情况下，如一些复杂的或者不常见的汉字，可能会被编码为两个或者更多的Token。这通常发生在使用子词编码方法的模型中，如Byte Pair Encoding（BPE）或Unigram Language Model（ULM）。\n\n至于每个英文单词对应0.75个Token的例子，这是一个假设的平均值，用于说明如果一个英文单词被分割为多个Token，那么模型能处理的单词数量可能会比Token数量多。实际情况下，这个比值可能会根据具体的文本和模型的编码方式有所不同。\n\n这里需要明确的是，无论英文还是中文，一个Token并不一定等同于一个字符或一个单词，而是取决于具体的编码方式。在理解和使用语言模型时，我们需要考虑到这一点。', metadata={'source': './index.md'})]

### 4.1.1 常见的加载器

LangChain的数据加载能力并不限于单一的数据源或格式，它可以处理各种常见的数据格式，例如CSV、文件目录、HTML、JSON、Markdown以及PDF等。下面，我们将分别解析一下这些不同格式数据的加载方法。

CSV 加载器

逗号分隔值（Comma-Separated Values，简称CSV）文件是一种使用逗号来分隔值的文本文件。每一行都是一条数据记录，每条记录包含一个或多个用逗号分隔的字段。LangChain可以加载CSV数据，其中每一行都被视为一个独立的文档。

这个加载器的典型用法是创建一个 CSVLoader 实例，然后调用其 load 方法来加载文件。

CSVLoader是BaseLoader的子类，主要用于从CSV文件加载数据，将其转换为一系列Document对象。每个Document代表CSV文件中的一行，其中每行被转换为键值对，并输出到Document的page\_content中。对于从CSV文件加载的每个文档，默认情况下源都被设置为file\_path参数的值。如果你设置了source\_column参数为CSV文件中的列名，那么每个文档的源将被设置为指定source\_column的列的值。

它的主要方法是：

- load: 从CSV文件加载数据，转化为Document对象的列表。

你可以设置的主要参数包括：

- file\_path：CSV文件的路径。

- source\_column：可选参数，用于指定作为文档源的列的名称。

- encoding：可选参数，用于指定打开文件的编码方式。

- csv\_args：可选参数，传递给csv.DictReader的参数。

这个类的典型用法是创建一个CSVLoader实例，然后调用其load方法来加载CSV文件的数据。例如：

loader = CSVLoader(file\_path='data.csv', encoding='utf-8')

documents = loader.load()

在这个例子中，documents就是从data.csv文件中加载的Document对象的列表。每个Document对象代表文件中的一行数据。

文件目录的加载

对于文件目录，LangChain提供了一种方法来加载目录中的所有文档。在底层，它默认使用UnstructuredLoader来实现这个功能。这意味着，只要将文档存放在同一目录下，无论数量多少，LangChain都能够将它们全部加载进来。

这个加载器的典型用法是创建一个DirectoryLoader 实例，然后调用其 load 方法来加载文件。

DirectoryLoader 是 BaseLoader 的子类，主要用于从一个指定的目录加载文档。每个从目录中加载的文件都将被处理为一个 Document 对象。

它的主要方法是：

- load ：加载目录中的所有文件，并将它们作为文档返回。这个方法不接受任何参数。

你可以设置的主要参数包括：

- loader\_cls ：用于加载文件的加载器类，它需要是 BaseLoader 的子类。

- loader\_kwargs ：传递给加载器类的参数。

- recursive ：是否递归加载子目录中的文件。

- show\_progress ：是否显示加载进度。

这个类的典型用法是创建一个 DirectoryLoader 实例，然后调用其 load 方法来加载目录中的文件。例如：

loader = DirectoryLoader(path='data\_directory')

documents = loader.load()

在这个例子中， documents 是从 data\_directory 目录中加载的 Document 对象的列表。每个 Document 对象代表目录中的一个文件。

HTML文件的加载

HTML是用于设计在Web浏览器中显示的文档的标准标记语言。LangChain可以将HTML文档加载为我们后续使用的文档格式。这就意味着，我们可以直接从网页上提取并处理数据。

这个加载器的典型用法是创建一个UnstructuredHTMLLoader或者BSHTMLLoader 实例，然后调用其 load 方法来加载文件。它们都可以将HTML文件加载为我们可以在后续过程中使用的文档格式。

我们可以利用 BSHTMLLoader 类来完成这项工作。这个加载器会从HTML文件中提取文本，并将其存储在 page\_content 中。同时，它还会提取网页标题，并将其作为 title 存储在元数据 metadata 中。

这种方法的优点在于它可以从HTML文档中提取出结构化的信息，比如段落、标题等，这些信息在后续的处理中可能会很有用。

使用 BSHTMLLoader 加载HTML文档的例子如下：

from langchain.document\_loaders import BSHTMLLoader

loader = BSHTMLLoader(file\_path='example.html')

documents = loader.load()

在这个例子中， documents 是从 example.html 文件中加载的 Document`对象的列表，其中每个`Document`对象都代表HTML文件中的一部分内容。

JSON加载器

JSON是一种使用人类可读文本来存储和传输数据对象的开放标准文件格式和数据交换格式，这些对象由属性-值对和数组（或其他可序列化值）组成。LangChain的JSONLoader使用指定的jq模式来解析JSON文件。jq是一种适用于Python的软件包。JSON文件的每一行都被视为一个独立的文档。

这个加载器的典型用法是创建一个JSONLoader 实例，然后调用其 load 方法来加载文件。

JSONLoader 类是用于加载JSON文件并将其转换为 Document 对象的加载器。这个类可以通过引用一个jq schema（一种用于处理JSON数据的查询语言）来提取文本并加载到文档中。

这个类的主要方法是 load ，它会加载并返回来自JSON文件的文档。在加载过程中，会使用到 jq\_schema 参数指定的jq模式来处理JSON数据。

JSONLoader 类的构造函数接受以下参数：

- file\_path ：JSON文件的路径。

- jq\_schema ：用于从JSON中提取数据或文本的jq模式。

- content\_key ：如果jq模式的结果是对象（字典）的列表，则使用此键来从JSON中提取内容。

- metadata\_func ：一个函数，接受由jq模式提取的JSON对象和默认的元数据，返回更新后的元数据的字典。

下面是一个使用 JSONLoader 加载JSON文件的例子：

from langchain.document\_loaders import JSONLoader

loader = JSONLoader(file\_path='example.json', jq\_schema='.[]')

documents = loader.load()

在这个例子中， documents 是从 example.json 文件中加载的 Document`对象的列表，其中每个 Document 对象都代表JSON文件中的一部分内容。

下面的列表提供了一些可能的 jq\_schema 参考值，用户可以根据 JSON 数据的结构使用这些值来提取内容。

JSON -> [{"text": ...}, {"text": ...}, {"text": ...}]

jq\_schema -> ".[].text"

JSON -> {"key": [{"text": ...}, {"text": ...}, {"text": ...}]}

jq\_schema -> ".key[].text"

JSON -> ["...", "...", "..."]

jq\_schema -> ".[]"

Markdown加载器

Markdown是一种使用纯文本编辑器创建格式化文本的轻量级标记语言。LangChain可以将Markdown文档加载为我们后续使用的文档格式。你可以通过指定 mode="elements" 来分离Unstructured 会为不同的文本块创建不同的元素。

这个加载器的典型用法是创建一个UnstructuredMarkdownLoader实例，然后调用其 load 方法来加载文件。

下面是一个使用 UnstructuredMarkdownLoader 加载JSON文件的例子：

markdown\_path = "../../../../../README.md"

loader = UnstructuredMarkdownLoader(markdown\_path, mode="elements")

documents = loader.load()

PDF文件加载器

PDF（Portable Document Format）是Adobe在1992年开发的一种文件格式，用于以独立于应用软件、硬件和操作系统的方式呈现文档，包括文本格式化和图像。LangChain可以将PDF文档加载为我们后续使用的文档格式。

PDF是目前应用最广泛的一种文档类型，包括但不限于金融研究报告、学术论文、市场研究报告等。LangChain 的数据模块设计了多种文档加载器以加载 PDF 文档。以下我们将介绍一些主要的 PDF 文档加载器及其用法。

1. PyPDF 文档加载器 ：它可以将 PDF 使用 PyPDF 加载为文档数组，每个文档包含页面内容和包含页码的元数据。示例如下：

from langchain.document\_loaders import MathpixPDFLoader

loader = MathpixPDFLoader("example\_data/layout-parser-paper.pdf")

或者使用 UnstructuredPDFLoader：

from langchain.document\_loaders import UnstructuredPDFLoader

loader = UnstructuredPDFLoader("example\_data/layout-parser-paper.pdf")

在底层，Unstructured 会为不同的文本块创建不同的元素。默认情况下，我们会将这些元素合并在一起，但你可以通过指定 mode="elements" 来轻松地保持这种分离。

2. 在线 PDF 加载器 ：我们也可以加载在线 PDF 文件，将其转化为我们可以在下游使用的文档格式。这对于各种在线 PDF 网站（例如 https://open.umn.edu/opentextbooks/textbooks/ 和 https://arxiv.org/archive/）都非常有用。注意，所有其他 PDF 加载器也可以用于获取远程 PDF，但 OnlinePDFLoader 是一个遗留功能，专门与 UnstructuredPDFLoader 一起使用。

from langchain.document\_loaders import OnlinePDFLoader

loader = OnlinePDFLoader("https://arxiv.org/pdf/2302.03803.pdf")

3. PyPDFium2 文档加载器 ：使用 PyPDFium2 加载 PDF 文件。

from langchain.document\_loaders import PyPDFium2Loader

loader = PyPDFium2Loader("example\_data/layout-parser-paper.pdf")

data = loader.load()

4. PDFMiner 文档加载器 ：使用 PDFMiner 加载 PDF 文件。

from langchain.document\_loaders import PDFMinerLoader

loader = PDFMinerLoader("example\_data/layout-parser-paper.pdf")

5. 使用 PDFMiner 生成 HTML 文本 ：这对于将文本语义地划分为各个部分非常有帮助，因为输出的 HTML 内容可以通过 BeautifulSoup 进行解析，以获取关于字体大小、页码、PDF 头/页脚等的更多结构化和丰富的信息。

6. PyMuPDF 文档加载器 ：这是 PDF 解析选项中最快的一种，它包含关于 PDF 及其页面的详细元数据，且每页返回一个文档。

from langchain.document\_loaders import PyMuPDFLoader

7. 从目录加载 PDF ：

from langchain.document\_loaders import PyPDFDirectoryLoader

8. pdfplumber 文档加载器 ：与 PyMuPDF 类似，输出的文档包含关于 PDF 及其页面的详细元数据，且每页返回一个文档。

以上就是 LangChain 支持的 PDF 文档加载器及其使用示例。

### 4.1.2加载器的分类

LangChain通过实现各种文档加载器，实现了对多元化数据源的处理。这些加载器的设计使得LangChain可以从各种服务提供商加载数据，进一步扩大了其在不同业务场景的应用范围。下面，我们将对这些加载器进行分类并简要描述。

这类加载器文件和目录加载器主要处理存储在本地或云端的文件和目录，例如CSVLoader，DirectoryLoader，JSONLoader，S3DirectoryLoader，S3FileLoader等。

文本和标记语言文件加载器，这类加载器负责处理各种文本和标记语言格式的文件。

比如BSHTMLLoader，MarkdownLoader，TextLoader，UnstructuredHTMLLoader，UnstructuredMarkdownLoader等。

文档和电子书加载器，处理电子书和文档的加载器包括Docx2txtLoader，PDFMinerLoader，UnstructuredEPubLoader，UnstructuredPDFLoader，UnstructuredWordDocumentLoader等。

社交媒体和在线论坛加载器，这类加载器主要用于处理来自社交媒体和在线论坛的数据，如FacebookChatLoader，GitHubIssuesLoader，RedditPostsLoader，TwitterTweetLoader等。

云服务提供商加载器，对于云服务提供商的数据源，LangChain提供了相应的加载器，如AirbyteJSONLoader，AirtableLoader，AzureBlobStorageContainerLoader，AzureBlobStorageFileLoader，GoogleDriveLoader，OneDriveLoader等。

文本聊天和消息服务加载器，这类加载器主要处理各种聊天和消息服务的数据，

比如DiscordChatLoader，SlackDirectoryLoader，TelegramChatApiLoader，TelegramChatFileLoader，WhatsAppChatLoader等。

网页和网站数据加载器，对于网页和网站的数据，如SitemapLoader，UnstructuredURLLoader，WebBaseLoader等加载器提供了处理方法。

其他特殊类型的加载器，这类加载器主要处理一些特定的或者特殊的数据源，如HuggingFaceDatasetLoader，UnstructuredImageLoader，WeatherDataLoader，YoutubeAudioLoader等。

通过实现这些加载器，LangChain可以从各种各样的服务提供商处加载数据，进一步提升了它的多样化数据处理能力。这对于需要处理大量、多样性数据的机器学习、自然语言处理等领域具有非常重要的意义。

## 4.2 E嵌入模型包装器

在深度学习和自然语言处理领域，嵌入（Embedding）是一种将文本数据转化为浮点数值表示形式的技术，它能够衡量两段文本之间的相关性。嵌入的一个典型例子是词嵌入，它将每个词映射到一个多维空间中的点，使得语义上相似的词在空间中的距离更近。词嵌入是将词语映射到向量空间中的一种技术，它通过对大量文本数据的训练，为每个词语生成一个高维向量。这个向量能够捕获词语的语义信息，例如，相似的词语（如“男”和“国王”，“女”和“女王”）在向量空间中的位置会非常接近。这是因为嵌入模型在训练过程中学习到了词语之间的语义关系。

例如，我们可以使用预训练的Word2Vec或GloVe等模型得到每个词的向量表示。假设“国王”的向量表示为[1.2, 0.7, -0.3]，“男”的向量表示为[1.1, 0.6, -0.2]，“女王”的向量表示为[-0.9, -0.8, 0.2]，“女”的向量表示为[-0.8, -0.7, 0.3]。我们会发现，相同性别的词语（如“国王”和“男”）在向量空间中的距离更近，这就反映了他们之间的语义关系。这种关系可以通过计算向量之间的余弦相似度来量化。

词嵌入的一个重要应用就是在自然语言处理任务中，例如文本分类、命名实体识别、情感分析等。通过将词语转换为向量，我们可以利用深度学习模型来处理文本数据，实现对语言的理解。

Langchain框架提供了一个名为Embeddings的类，它为多种文本嵌入模型（如OpenAI、Cohere、Hugging Face等）提供了统一的接口。该类可以将一段文本转化为一个向量表示，这使得我们可以在向量空间中处理文本，执行如语义搜索这样的操作，寻找在向量空间中最相似的文本。我们把实例化的Embeddings类称为嵌入模型包装器。同 Model I/O 模块的LLM 类和 Chta Model 类型模型包装器并列为三大模型包装器。OpenAI 平台的嵌入模型，使用大量的文本数据进行训练，以尽可能地捕捉和理解人类语言的复杂性。这使得OpenAI的嵌入模型可以生成高质量的向量表示，有效地捕捉文本中的语义关系和模式。在Langchain框架中最常用的是OpenAI平台的嵌入模型，第二代嵌入模型，text-embedding-ada-002。这种嵌入模型对于搜索、聚类、推荐、异常检测和分类任务等都有很好的效果。

嵌入模型包装器同其他两个模型包装器一样使用，需要导入 Embedding 类，设置密钥。嵌入模型包装器提供了两个主要的方法，分别是 embed\_documents 和 embed\_query 。前者接受一组文本作为输入并返回他们的嵌入向量，而后者接受一个文本并返回其嵌入向量。这两个方法之所以分开，是因为模型平台的嵌入模型对于待搜索的文档和搜索查询本身有不同的嵌入方法。

例如，在使用OpenAI的嵌入模型时，我们可以通过以下代码示例来嵌入一组文档和一个查询：

通过简单的代码示例，展示这个嵌入模型包装器的使用过程。

from langchain.embeddings import OpenAIEmbeddings

embeddings\_model = OpenAIEmbeddings(openai\_api\_key="填入你的密钥")

接下来，我们可以使用 embed\_documents 方法将一系列文本嵌入为向量。例如，我们可以将以下五句话嵌入为向量：

embeddings = embeddings\_model.embed\_documents(

[

"Hi there!",

"Oh, hello!",

"What's your name?",

"My friends call me World",

"Hello World!"

]

)

len(embeddings), len(embeddings[0])

这将返回一个嵌入向量列表，其中每个嵌入向量的长度为1536。

最后，我们可以使用 embed\_query方法将单个查询嵌入为向量。这在你想要将一个查询和其他已嵌入的文本进行比较时非常有用：

embedded\_query = embeddings\_model.embed\_query("What was the name mentioned in the conversation?")

embedded\_query[:5]

这将返回该查询的嵌入向量，我们在这里只展示了向量的前5个元素。

[0.0053587136790156364,

-0.0004999046213924885,

0.038883671164512634,

-0.003001077566295862,

-0.00900818221271038]

### 4.2.1 嵌入模型包装器的类型

在LangChain中，我们为各种大语言模型平台提供了嵌入模型接口的封装。这其中，OpenAI平台提供的接口封装为“OpenAIEmbeddings”。这类嵌入方式的特点是能够充分利用大规模预训练模型的语义理解能力，其中包括OpenAI、Hugging Face等提供的自然语言处理模型。以下是一些具体的类型：

1. 自然语言模型嵌入：这类嵌入包括“OpenAIEmbeddings”、“HuggingFaceEmbeddings”、“HuggingFaceHubEmbeddings”、“HuggingFaceInstructEmbeddings”、“SelfHostedHuggingFaceEmbeddings”和“SelfHostedHuggingFaceInstructEmbeddings”。这些嵌入方法主要利用诸如OpenAI、Hugging Face等自然语言处理模型进行文本嵌入。

2. AI平台或云服务嵌入：这类嵌入主要依托AI平台或云服务的能力进行文本嵌入，典型的包括Elasticsearch、SagemakerEndpoint、DeepInfra等。这些嵌入方式的主要特点是能够利用云计算的优势，处理大规模的文本数据。

3. 专门的嵌入模型：这类嵌入主要是专门用于处理特定结构文本的嵌入模型，例如AlephAlpha的AsymmetricSemanticEmbedding和SymmetricSemanticEmbedding，适用于结构不同或相似的文本。

4. 自托管嵌入：这类嵌入一般适用于用户自行部署和管理的场景，如SelfHostedEmbeddings，给予用户更大的灵活性和控制权。

5. 仿真或测试用嵌入：FakeEmbeddings一般用于测试或模拟场景，不涉及实际的嵌入计算。

6. 其他类型：此外，LangChain还支持一些其他类型的嵌入方式，如Cohere、LlamaCpp、ModelScope、TensorflowHub、MosaicMLInstructor、MiniMax、Bedrock、DashScope和Embaas等。这些嵌入方式各有特点，能够满足不同的文本处理需求。

在以上这些类型中，用户可以根据自己的具体需求，选择最合适的文本嵌入类型。同时，LangChain也将持续引入更多的嵌入类型，以进一步提升其处理文本的能力。

## 4.3 D文档转换器

在大语言模型开发时代，处理海量文档成为了一个常见且重要的任务。LangChain 框架的数据增强模块为此提供了一系列强大的包装器，其中文档转换器（Document Transformers）就是我们解决这个问题的关键工具之一。

文档转换器的主要任务分为两个步骤：第一步是对文档进行分割，主要由切割器完成，第二步是将分割后的文档转换为Document数据格式。尽管从名称上看，文档转换器主要进行的是转换操作，但实际上，这是从结果出发来定义的。在数据增强模块中，数据以Document对象和向量形式在各个包装器中流通。向量形式的数据由向量存储库管理，而转为向量之前，数据以Document对象的形式存在。

文档转换器的主要任务是将文档数据进行切割和转换为 Document 对象的数据格式。然后，这些数据会被传递给文本嵌入模型包装器进行处理，转换为嵌入向量，最后存储在向量存储库中，检索器从中检索与用户输入的问题相关的文档内容。

你可能会问，为什么我们需要切割文档呢？我们先看看主要用于切割的转换器和加载器之间的区别。

### 4.3.1 加载器和转换器的区别

文档加载器的主要任务是从各种源加载数据，并将这些数据转换为 Document 对象。Document 是一个包含文本及其相关元数据的对象。例如，文档加载器可以加载简单的.txt文件，也可以加载网页的文本内容，甚至可以加载 YouTube 视频的转录内容。这是处理数据的第一步，将不同格式、不同来源的数据统一为 Document 对象，为后续的嵌入向量存储打下基础。

然而，加载后的文档可能非常长，可能包含几十页甚至几百页的内容。处理这样长的文档可能会带来一些问题。一方面，大语言模型平台处理长文本的能力是有限的，例如，某些模型平台有最大 Max Tokens 的限制。另一方面，将整个文档作为一个整体处理可能无法充分发挥模型的效果，因为文档中不同部分的内容可能在语义上存在较大的差异。因此，我们需要将长文档切割为较小的文本块，使得每个文本块在语义上尽可能的一致，这就是转换器的文本切割任务，由文本切割器完成。

文本切割器按照一定的策略将文档切割为多个小块。这些策略可能包括如何切割文本（例如，按照句子切割）、如何确定每个小块的大小（例如，按照一定的字符数切割）等。通过合理的切割，我们可以保证每个小块的内容在语义上尽可能的一致，并且可以被模型平台处理。

因此，文档加载器和文本转换器是处理文档数据的两个重要步骤。文档加载器负责将各种源的数据转换为 Document 对象，而转换器则负责将长文档切割为可以被模型平台处理的小块。通过这两步，我们可以将各种各样的数据转换为适合模型平台处理的形式。

切割的需求

从加载器和转换器的区别我们看到，切割的需求源于对大量文档数据的处理效率、适应模型平台的限制。如果数据太大，增删改查等操作的效率就会受到影响。特别是当我们处理几十页、几百页的文档时，我们需要考虑在何处进行切割，是在使用加载器加载完文档后进行切割，还是在加载之前就进行切割。如果我们在加载大型文件时遇到中断，那么我们就需要重新开始。为了避免这种情况，我们可以将大型文件切割为多个数据块。在 LangChain 框架中，提供了多种内置的转换器完成在加载之前进行切割的任务，满足开发者的切割文档的需求。

转换的需求

当然，文档转换器不仅仅提供了切割功能，它还能根据开发者的需要，提供转换的方法。比如，如果我们已经使用加载器将文档加载并转换为 Document 对象的格式，那么我们可以调用文档转换器的 split\_documents 方法进行切割。

from langchain.document\_loaders import WebBaseLoader

from langchain.text\_splitter import RecursiveCharacterTextSplitter

# 这里我们使用的是WebBaseLoader加载器加载了网络文档

loader = WebBaseLoader("https://lilianweng.github.io/posts/2023-06-23-agent/")

data = loader.load()

# 此时我们切割的Documents 对象的数据

text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=500, chunk\_overlap=0)

splits = text\_splitter.split\_documents(data)

如果我们并没有使用加载器，而是直接使用 Python 的内置方法打开和读取文件，那么我们可以将读取到的结果传递给文档转换器的 create\_documents 方法进行转换。

# 这里我们没有使用加载器.

with open('../../../state\_of\_the\_union.txt') as f:

state\_of\_the\_union = f.read()

from langchain.text\_splitter import CharacterTextSplitter

text\_splitter = CharacterTextSplitter(

separator = "\n\n",

chunk\_size = 1000,

chunk\_overlap = 200,

length\_function = len,

)

texts = text\_splitter.create\_documents([state\_of\_the\_union])

print(texts[0])

此外，如果我们只需要字符串，还可以使用 split\_text 方法进行转换。

with open('../../../state\_of\_the\_union.txt') as f:

state\_of\_the\_union = f.read()

text\_splitter = CharacterTextSplitter(

separator = "\n\n",

chunk\_size = 1000,

chunk\_overlap = 200,

length\_function = len,

)

text=text\_splitter.split\_text(state\_of\_the\_union)[0]

无论开发者使用什么方式读取文件，这些内置的文档转换器都可以帮助我们处理数据，进行切割和转换，满足开发者的转换文档的需求。加载器和转换器的目的都是适应大语言模型平台的合适大小的“块” 数据，这些“块”像快递包裹一样，通过数据增强模块的LEDVR工作流，或被链组件包装，或被代理组件包装，最后提交给模型平台，获取模型平台的响应回答。

### 4.3.2 文本切割器

当处理长文本时，往往需要将文本分割成块。尽管这看起来简单，但实际上可能涉及很多复杂性。理想情况下，您会希望将语义相关的文本部分保持在一起。而“语义相关”的含义可能取决于文本的类型。下面将介绍几种实现这一目标的方法。

文本分割器的工作原理是：将文本分割成小的、语义上有意义的块（通常是句子）。将这些小块开始组合成一个大的块，直到达到某个大小（通过某种函数进行测量）。一旦达到该大小，将该块作为自己的文本片段，然后开始创建新的文本块，新的文本块和前一个文本块会有一些重叠（以保持块与块之间的上下文）。这意味着，您可以沿着两个不同的轴来定制你的文本分割器：文本如何被分割块的大小以及如何被测量。

默认推荐的文本分割器是RecursiveCharacterTextSplitter。这个文本分割器接受一个字符列表，它尝试基于第一个字符进行分割，但如果任何块太大，它就会移动到下一个字符，依此类推。默认情况下，它尝试分割的字符是["\n\n", "\n", " ", ""] 。

除了可以控制分割的字符，您还可以控制以下几点：

length\_function：如何计算块的长度。默认只计算字符数量，但是在此通常会传入一个标记计数器。

chunk\_size：你的块的最大大小（由长度函数测量）。

chunk\_overlap：块之间的最大重叠。有一些重叠可以在块之间保持连续性（例如采用滑动窗口的方式）。

add\_start\_index：是否在元数据中包含每个块在原始文档中的起始位置。

4.3.3文本分割器的使用在处理大规模文本数据时，LangChain提供了多种文本分割方法，以满足各种类型的应用需求。下面我们通过示例代码，展示如何使用不同的文本分割器方法。

1. 按字符分割

这是最简单的方法。它基于字符（默认为"\n\n"）进行分割，并通过字符数量来衡量块的大小。 chunk\_size 属性值设置块大小， chunk\_overlap 属性值设置块之间的最大重叠。

# This is a long document we can split up.

with open('../../../state\_of\_the\_union.txt') as f:

state\_of\_the\_union = f.read()

from langchain.text\_splitter import CharacterTextSplitter

text\_splitter = CharacterTextSplitter(

chunk\_size = 1000,

chunk\_overlap = 200,

)

texts = text\_splitter.create\_documents([state\_of\_the\_union])

print(texts[0])

2. 代码分割

CodeTextSplitter允许你对多种语言的代码进行分割。导入枚举Language并指定语言即可。

我们通过代码主要展示了如何使用 RecursiveCharacterTextSplitter 切割器来处理编程语言的代码。在这个例子中，我们处理的是 JavaScript 代码。我们定义了一段 JavaScript 代码，然后使用 RecursiveCharacterTextSplitter 的 from\_language 类方法创建了一个适用于 JavaScript 语言的切割器。这个方法接受一个 language 参数，它的类型是枚举类型 Language，可以表示多种编程语言。除了 JavaScript，目前还支持 'cpp', 'go', 'java', 'js', 'php', 'proto', 'python', 'rst', 'ruby', 'rust', 'scala', 'swift', 'markdown', 'latex', 'html', 'sol' 等多种编程语言。

from langchain.text\_splitter import (

RecursiveCharacterTextSplitter,

Language,

)

JS\_CODE = """

function helloWorld() {

console.log("Hello, World!");

}

// Call the function

helloWorld();

"""

js\_splitter = RecursiveCharacterTextSplitter.from\_language(

language=Language.JS, chunk\_size=60, chunk\_overlap=0

)

js\_docs = js\_splitter.create\_documents([JS\_CODE])

js\_docs

3. Markdown标题文本分割器

在处理各种聊天或问答应用时，我们常常需要在嵌入和向量存储之前将输入文档切分。这是因为当我们将整个段落或文档进行嵌入时，嵌入过程会考虑到文本内部的整体上下文和句子、短语之间的关系。这样可以得到一个更全面的向量表示，捕捉到文本的广义主题和主旨。

这些场景中，我们切分的目标通常是将具有共同上下文的文本保持在一起。因此，我们可能希望特别尊重文档本身的结构。例如，一个 Markdown 文件是按照标题进行组织的。在特定的标题组内创建块是一种直观的想法。实现这样的切分想法，我们可以使用 MarkdownHeaderTextSplitter。这个工具可以根据指定的一组标题来分割一个 Markdown 文件。例如，如果我们希望分割以下 Markdown：

# Markdown的一级标题

## Markdown的二级标题

Markdown的段落。Markdown的段落Markdown的段落Markdown的段落。

Markdown的段落。

Markdown的段落。

Markdown的段落。

Markdown的段落。

我们可以设置切分的标题为：

headers\_to\_split\_on = [

("#", "Header 1"),

("##", "Header 2"),

]

接着，我们可以使用 MarkdownHeaderTextSplitter 来进行切分，实例化分割器后，我们调用实例的split\_text 方法，这个方法接收了Markdown 文件的内容，返回Document格式的数据，一旦转换为这种数据格式，就可以使用其他的切割器的 split\_documents 方法进行再切割：

# MD splits

markdown\_splitter = MarkdownHeaderTextSplitter(headers\_to\_split\_on=headers\_to\_split\_on)

md\_header\_splits = markdown\_splitter.split\_text(markdown\_document)

这样，我们就得到了按标题切分的文档。然而，这仍然可能不够。如果某个标题下的内容非常长，我们可能需要进一步切分。这时，我们可以使用 RecursiveCharacterTextSplitter 来进行字符级别的切分：

# Char-level splits

from langchain.text\_splitter import RecursiveCharacterTextSplitter

chunk\_size = 250

chunk\_overlap = 30

text\_splitter = RecursiveCharacterTextSplitter(

chunk\_size=chunk\_size, chunk\_overlap=chunk\_overlap

)

# Split

splits = text\_splitter.split\_documents(md\_header\_splits)

这样，我们就可以得到更小的、便于处理的文本块了。

4. 递归按字符分割

这是通用文本推荐的文本分割器。它由一系列字符参数化。直到块足够小，它会尝试按顺序分割它们。默认列表是["\n\n", "\n", " ", ""] 。这样做的效果是尽可能地将所有段落（然后是句子，然后是单词）保持在一起，因为这些通常看起来是最强的语义相关的文本部分。

5. 按标记（Token）分割

在处理自然语言处理任务时，我们经常需要将长文本分割成小块以便于模型处理。这就需要使用标记切割器。标记切割器的主要任务是按照一定的规则将文本切割成小块，这些小块的长度通常由模型的输入限制决定。以下就是一些常用的标记切割器：

Tiktoken 标记切割器：tiktoken 是由 OpenAI 创建的一种快速的字节对编码（BPE）标记器。我们可以使用它来估计使用的标记数量。对于 OpenAI 的模型来说，它的准确度可能会更高。tiktoken 的文本切割方式是按照传入的字符进行切割，块大小的计算方式是通过 tiktoken 标记器进行计算。这个标记切割器使用的方式更复杂一点，我们通过代码展示它的使用方式。先要安装tiktoken python包。然后导入 CharacterTextSplitter 类，使用类方法 from\_tiktoken\_encoder 实例化这个类。跟SpacyTextSplitter 等其他内置的切割器不一样的是，Tiktoken 标记切割器是由 CharacterTextSplitter 类的类方法实例而来。

#!pip install tiktoken

# This is a long document we can split up.

with open("../../../state\_of\_the\_union.txt") as f:

state\_of\_the\_union = f.read()

from langchain.text\_splitter import CharacterTextSplitter

text\_splitter = CharacterTextSplitter.from\_tiktoken\_encoder(

chunk\_size=100, chunk\_overlap=0

)

texts = text\_splitter.split\_text(state\_of\_the\_union)

SpacyTextSplitter 标记切割器：spaCy 是一种用于高级自然语言处理的开源软件库，用 Python 和 Cython 编写。作为 NLTK 的另一种替代方案，我们可以使用 spaCy 的标记器。spaCy 的文本切割方式是通过 spaCy 标记器进行切割，块大小的计算方式是通过字符数量进行计算。

SentenceTransformersTokenTextSplitter标记切割器：是专门用于处理句子转换模型的专用文本切割器。默认行为是将文本切割成适合所要使用的句子转换器模型的标记窗口的块。

NLTKTextSplitter 标记切割器：Natural Language Toolkit（NLTK）是一套用于符号和统计自然语言处理的库和程序，用 Python 编程语言编写。与仅仅在 "\n\n" 处切割不同，我们可以使用 NLTK 基于 NLTK 标记器进行切割。NLTK 的文本切割方式是通过 NLTK 标记器进行切割，块大小的计算方式是通过字符数量进行计算。

Hugging Face 标记切割器：Hugging Face 提供了许多标记器。我们使用 Hugging Face 的标记器 GPT2TokenizerFast 来计算文本长度（以标记为单位）。这个文本切割方式是按照传入的字符进行切割，块大小的计算方式是通过 Hugging Face 标记器计算出的标记数量进行计算。这个标记切割器使用的方式更复杂一点，我们通过代码展示它的使用方式。先要安装 transformers 库，导入 GPT2TokenizerFast 包，调用这个类的 from\_pretrained 方法，实例化一个 tokenizer 。然后 CharacterTextSplitter 类的tokenizer 属性为这个标志器实例，使用类方法 from\_huggingface\_tokenizer 实例化这个类。

from transformers import GPT2TokenizerFast

tokenizer = GPT2TokenizerFast.from\_pretrained("gpt2")

# This is a long document we can split up.

with open("../../../state\_of\_the\_union.txt") as f:

state\_of\_the\_union = f.read()

from langchain.text\_splitter import CharacterTextSplitter

text\_splitter = CharacterTextSplitter.from\_huggingface\_tokenizer(

tokenizer, chunk\_size=100, chunk\_overlap=0

)

texts = text\_splitter.split\_text(state\_of\_the\_union)

这些标记切割器是框架内置的切割器。它们的使用方式我们通过选择哪种标记切割器主要取决于我们的任务需求和所使用的模型。在选择标记切割器时，我们需要考虑到模型的输入限制、我们希望保留的上下文信息以及我们希望如何切割文本等因素。

## 4.4 V向量存储库

在嵌入模型包装器（LEDVR工作流的 E）这一部分，我们了解到嵌入模型包装器提供了两个主要的方法，分别是 embed\_documents 和 embed\_query 。前者接受一组文本作为输入并返回他们的嵌入向量，而后者接受一个文本并返回其嵌入向量。我们也展示了如何利用这个包装器将查询语句转换成浮点数列表，也就是向量。但是，当我们得到这个向量后，我们应该如何使用它呢？

这就是向量存储库（LEDVR工作流的 V）要解决的问题。

向量存储库可以看作是一个大的包装器，它负责处理数据增强模块中LEDVR工作流的LED三个环节的输出结果。对于开发者来说，使用向量存储库可以极大地简化工作。开发者不需要关心如何与各个模型平台进行交互，也不需要将数据处理成其他形式。比如，我们LEDVR工作流一直都在处理文档对象格式的数据，开发者只需要专注于这个格式，然后将数据交给向量存储库就可以了。向量存储库会在底层处理数据格式的转换，解析模型包装器的返回数据等各种复杂的工作。

举个简单例子，如果我们单独将查询语句转向量，做法是实例化嵌入模型包装器后，调用embed\_documents 方法，可是这个方法接收的是字符串列表。如果我们不使用向量存储库这个包装器，我们要先把文本分割器处理过的文档数据格式的结果，转换为字符串列表，最终得到嵌入模型包装器的字符串结果后，我们又要考虑如何转换为向量存储库的格式，否则使用不了向量存储库的查询功能。

相比之下，如果我们使用向量存储库这个包装器，只需要将原始的文档对象格式的数据交给向量存储库，向量存储库会负责将文档转换成字符串，然后将字符串转换成向量，最后将向量存储起来。当我们需要查询时，我们只需要提供查询语句，向量存储库会自动将查询语句转换成向量，然后进行查询。这样一来，我们就可以把所有复杂的数据处理工作都交给向量存储库。简单粗暴一点便是我们可以忘掉embed\_documents 方法了。因为向量存储库帮我们做好了，这也正是 Langchain 的设计理念，让Langchain 为开发者做更多。

### 4.4.1 向量存储库的使用

在数据增强模块中，数据以Document对象和向量形式在各个包装器中流通。向量形式的数据由向量存储库管理，我们为什么要使用向量这种数据格式？这是因为传统的数据库是结构化的，而如今很多数据都是非结构化数据。

非结构化数据是指在日常操作中并不遵循固定格式或者不容易被数据库系统识别的数据。例如，电子邮件、博客、社交媒体帖子、音频和视频等等。这些数据无法通过预定义的数据模型进行分类，或者不适合通过常规的关系数据库进行处理。

我们对非结构化数据的需求主要是存储和搜索。存储是为了保留这些数据以供日后分析和使用，而搜索则是为了从海量数据中找到我们所需要的信息。例如，当我们在互联网上搜索关键词时，搜索引擎会从非结构化的网页数据中找到与关键词相关的信息。而在大数据和人工智能领域，非结构化数据也被广泛用于情感分析、文本分类、语义理解等任务。

处理非结构化数据的一种常见方法是将其嵌入并存储生成的嵌入向量，然后在查询时将非结构化查询嵌入，并检索与嵌入查询“最相似”的嵌入向量。这种方法将复杂的非结构化数据转化为了结构化的向量，大大简化了数据的处理和分析。向量存储库 (VectorStore) 就是实现这个功能的工具，它负责存储嵌入的数据并执行向量搜索。

向量存储库的工作流程可以通过以下的代码示例来说明。首先，我们需要安装faiss-cpu这个Python包，这是一个用于高效相似性搜索和聚类的库。

pip install faiss-cpu

向量存储库的工作原理是由VectorStore类实例化而来，这个类主要提供了一些实例化的类方法。通过理解这些类方法的功能，我们能够更好地使用向量存储库，并能够根据自己的需求进行定制。其中，from\_documents是一个常用的方法，它接受一个文档列表和一个嵌入模型包装器作为输入，返回一个初始化后的向量存储库。这个方法首先从每个文档中提取文本和元数据，然后调用from\_texts方法，将文本、嵌入模型以及元数据作为输入，生成向量存储库。这个方法的异步版本，afrom\_documents，提供了同样的功能，但是它以异步的方式运行。除此之外，from\_texts是一个更基础的方法，它直接接受一组文本和一个嵌入模型包装器，以及可选的元数据，生成向量存储库。这个方法的异步版本，afrom\_texts，也提供了相同的功能。最后，as\_retriever方法返回一个VectorStoreRetriever对象，这个对象包装了向量存储库，并提供了一些用于查询的方法。例如，它可以执行相似性搜索，也可以执行最大边缘相关性搜索。

所以我们实例化一个FAISS向量存储库（Langchain 封装了几十个向量数据库平台的服务，这里我们选的是Facebook AI Similarity Search (FAISS) 库，你可以选择其他库），并将文档块 documents和OpenAI的嵌入模型包装器 OpenAIEmbeddings 一起传递给这个向量存储库，使用from\_documents 方法实例化向量存储库 db 。此时，向量存储库 db 会自动调用嵌入模型将每个文档块转换成一个向量，并将这些向量存储起来。我们已经完成了向量存储库的准备工作，接下来我们就可以通过这个向量存储库来对文档进行高效的相似性搜索了。

from langchain.document\_loaders import TextLoader

from langchain.embeddings.openai import OpenAIEmbeddings

from langchain.text\_splitter import CharacterTextSplitter

from langchain.vectorstores import FAISS

# LEDVR： raw\_documents是L, OpenAIEmbeddings()是 E, documents是D, db 是 V

raw\_documents = TextLoader('../../../state\_of\_the\_union.txt').load()

text\_splitter = CharacterTextSplitter(chunk\_size=1000, chunk\_overlap=0)

documents = text\_splitter.split\_documents(raw\_documents)

db = FAISS.from\_documents(documents, OpenAIEmbeddings())

我们使用similarity\_search方法来嵌入一个查询， “ "What did the president say about Ketanji Brown Jackson" ”。

query = "What did the president say about Ketanji Brown Jackson"

docs = db.similarity\_search(query)

print(docs[0].page\_content)

通过比较查询向量与存储库中向量的相似度，我们就可以找到与查询最相关的文本。

Tonight. I call on the Senate to: Pass the Freedom to Vote Act. Pass the John Lewis Voting Rights Act. And while you’re at it, pass the Disclose Act so Americans can know who is funding our elections.

Tonight, I’d like to honor someone who has dedicated his life to serve this country: Justice Stephen Breyer—an Army veteran, Constitutional scholar, and retiring Justice of the United States Supreme Court. Justice Breyer, thank you for your service.

One of the most serious constitutional responsibilities a President has is nominating someone to serve on the United States Supreme Court.

And I did that 4 days ago, when I nominated Circuit Court of Appeals Judge Ketanji Brown Jackson. One of our nation’s top legal minds, who will continue Justice Breyer’s legacy of excellence.

向量存储库是处理非结构化数据的一个强大工具。它可以将复杂的非结构化数据转化为结构化的向量，使得数据的存储和搜索变得更为简单和高效。通过嵌入和向量相似度的计算，帮助我们在海量的非结构化数据中快速找到我们需要的信息。

### 4.4.2 向量存储库的搜索方法

我们刚刚通过实例代码了解了如何使用向量存储库。现在我们还需要知道，向量存储库主要提供以下几种搜索方法：

1. similarity\_search(query: str, k: int = 4) -> List[Document]: 这个方法接受一个字符串查询和一个整数k作为参数，返回与查询最相似的k个文档的列表。查询是要搜索的字符串，k是要返回的文档数量，默认为4。

2. similarity\_search\_by\_vector(embedding: List[float], k: int = 4) -> List[Document]: 这个方法接受一个嵌入向量和一个整数k作为参数，返回与嵌入向量最相似的k个文档的列表。嵌入向量是由文本嵌入模型生成的查询的向量表示。

3. max\_marginal\_relevance\_search(query: str, k: int = 4, fetch\_k: int = 20, lambda\_mult: float = 0.5) -> List[Document]: 这个方法使用最大边际相关性算法返回选择的文档。最大边际相关性算法优化了查询的相似性和所选择文档之间的多样性。查询是要搜索的字符串，k是要返回的文档数量，默认为4。fetch\_k是要传递给MMR算法的文档数量。lambda\_mult是一个在0和1之间的数字，决定了结果之间的多样性程度，0对应最大的多样性，1对应最小的多样性，默认为0.5。

4. max\_marginal\_relevance\_search\_by\_vector(embedding: List[float], k: int = 4, fetch\_k: int = 20, lambda\_mult: float = 0.5) -> List[Document]: 这个方法与上面的max\_marginal\_relevance\_search方法类似，但是接受的是嵌入向量而不是查询字符串。

以上所有的方法都有对应的异步版本，方法名前加上'a'，比如asimilarity\_search、asimilarity\_search\_by\_vector等。这些异步方法可以在协程中使用，让你的程序在等待结果的同时可以执行其他任务，提高了程序的效率。

我们可以观察到这些方法，返回的结果都是 List[Document] 的数据格式。这也是我们在数据增强模块的最主要数据格式。无论是加载器加载的结果，还是实例化向量存储库的时候，我们都使用的是文档数据类型。而浮点列表的向量数据类型，通常都在包装器的内部流通使用，我们甚至可以不知道到底转成了什么浮点数字，对于我们大部分人来说，浮点数列表只是一堆数字。

### 4.5 R检索器

在我们的LEDVR数据处理流程中，有一个环节可能让你感到疑惑，那就是最后的“检索器”（Retriever）。你可能会问，既然我们已经通过LEDV流程把外部数据转化为了向量形式并保存在向量库中，而且我们还可以对这个库进行查询并获取相关文档，为什么还需要一个检索器呢？实际上，这正是我们想要讨论的重点。此外，我们之前还强调了向量存储库实例的 as\_retriever 方法，这个方法返回一个 VectorStoreRetriever 对象。这个对象甚至还“包装”了向量存储库。为什么要一个检索器？为什么一定是 LEDVR ？

在众多的向量库数据平台中，比如“Chroma”,“FAISS”,“Pinecone”, “Zilliz”等等，我们可以看到向量库的种类繁多。直接和这些数据库如“Qdrant”进行交互，可能需要我们具备深入的数据库操作知识，如了解查询语法，管理数据库连接，处理错误和异常等。这样的操作可能在使用上较为复杂和不便。如果我们有一种方法，能将各种向量库平台统一到一个接口上，那将会非常方便。这就是LangChain做的事情，它封装了 VectorStoreRetriever 类，为我们提供了一个统一的查询接口，无论向量存储库的底层是什么，无论是什么向量存储库，我们将向量存储库的实例调用as\_retriever 方法得到了一个基于向量存储库的检索器，即VectorStoreRetriever 类的实例。所以创建一个检索器很容易，如果你已经创建好了向量存储库，直接调用as\_retriever 方法，就得到了一个检索器实例，这个检索器可以查询，更重要的是可以被传递到 Langchain 框架的其他模块中，让检索器在整个框架中自由流通。

当我们从原理上来理解检索器后，我们可以简化记忆，从向量存储库到检索器，中间只需一个as\_retriever 方法。向量存储库调用它，便创建了一个检索器。这就组合了LEDVR 工作流，用 as\_retriever 方法粘合了V 和 R，整个LEDVR 工作流到此结束。

那么，检索器是什么呢？可以把检索器看作是一个向量存储库的包装器，包装了一套统一的接口，无论底层的向量存储库是什么，都可以使用同样的方式进行查询，而这个包装器的核心就是包装了向量存储库的实例。这使得我们可以轻松地切换不同的向量存储库，而无需修改查询代码。

还有问题没解决的是谁使用检索器？检索器作用是什么？。其实，在LangChain框架中，所有的基础模块都是为了链（Chain）模块的基建工作而设计的。这就像一座大厦的建设，每一块砖，每一捆钢筋，都在为整个建筑的最终成型做准备。从“谁使用了检索器”这个视角来看，我们可以更清晰地理解这个过程。

在处理用户查询时，我们首先需要通过检索器获取相关的文档，这些文档能够帮助我们回答用户的问题。然后，我们需要将这些文档提交给模型平台，利用大型语言模型的能力生成回答。这就好比我们在一个外部数据的大型知识库中寻找答案，然后通过强大的语言理解和生成能力，以一种人类易于理解的方式表达这个答案。

这时，你可能会问，我们是不是需要调用模型平台的能力了？答案是肯定的。然而，LangChain的设计目标是让这一切变得简单和直观。在内置的链组件中，LangChain将所有这些步骤都整合在一起。只需要指定模型包装器和检索器，链组件就能完成所有的功能。

比如，当我们在LangChain中配置好模型包装器和检索器之后，链组件会首先利用检索器找到相关的文档，然后将这些文档送入模型包装器，最后返回模型生成的答案。在这个过程中，我们只需要关心如何配置模型包装器和检索器，而不需要担心这些基础模块之间的交互细节，因为所有这些都由链组件自动完成了。

### 4.5.1 检索器的使用

我们用代码示例展示如何在LangChain中配置模型包装器和检索器，然后使用链组件来实现信息检索和问题回答的全过程。

首先，我们使用 TextLoader 加载文本文件"state\_of\_the\_union.txt"，该文件包含了一系列的文档：

loader = TextLoader("../../state\_of\_the\_union.txt")

documents = loader.load()

接着，我们使用 CharacterTextSplitter 对文档进行分割，将每篇文档分割成一系列的文本块。每个文本块的大小为1000个字符，相邻的文本块之间有重叠部分：

text\_splitter = CharacterTextSplitter(chunk\_size=1000, chunk\_overlap=0)

texts = text\_splitter.split\_documents(documents)

然后，我们使用 OpenAIEmbeddings 为文本块生成嵌入向量，并使用 Chroma 将这些文本块和对应的嵌入向量存储起来，创建了一个向量存储库的实例 docsearch：

embeddings = OpenAIEmbeddings()

docsearch = Chroma.from\_documents(texts, embeddings)

最后，我们使用 RetrievalQA 创建了一个检索式问答系统。这个系统使用了我们之前创建的 docsearch 作为检索器，和OpenAI大型语言模型作为回答生成器。这个系统可以根据用户的问题，找到相关的文本块，然后生成回答：

qa = RetrievalQA.from\_chain\_type(llm=OpenAI(),

chain\_type="stuff", retriever=docsearch.as\_retriever())

在这个过程中，我们只需要关注如何配置模型包装器和检索器，而不需要关心这些组件之间的交互细节，因为链组件已经帮我们自动处理了所有的事情。

### 4.5.2 检索器的类型

在实际的信息检索过程中，我们可能会遇到各种各样的问题和需求，比如需要精确匹配关键词、需要理解语义、需要根据时间排序、需要从网络上获取最新的数据等等。每种类型的检索器都是为了解决这些特定需求而设计的。分类可以让我们更清楚地理解每种检索器的功能和用途，从而能够更有效地选择和使用。

具体来说，以下是每种类型的检索器所解决的问题和适用的场景：自查询检索器：这种检索器适用于需要通过自然语言查询来检索具有一定结构或元数据的文档的场景。比如，在一个电子商务网站中，用户可能会输入“ "最新的iPhone手机" ”，检索器可以将这个查询转化为一个结构化的查询，比如{"category": "手机", "brand": "iPhone", "order": "newest"}，从而能够更精确地获取到用户想要的结果。时间加权向量存储检索器：这种检索器适用于信息的新旧程度对查询结果影响较大的场景。比如，在新闻检索中，用户通常更关心最新的新闻，因此检索器需要根据新闻的发布时间来对结果进行排序。向量存储支持的检索器：这种检索器适用于需要基于语义相似度进行检索的场景。比如，在问答系统中，用户的问题可能会有很多种表达方式，只有理解了问题的语义，才能找到正确的答案。网络研究检索器：这种检索器适用于需要从网络上获取最新数据的场景。比如，用户可能想要获取关于一个热点事件的最新信息，此时检索器可以直接从网络上进行搜索，获取到最新的数据。

目前主流的Retriever检索器主要有以下几种类型：向量存储库支持的检索器、自查询检索器、MultiQueryRetriever、Contextual Compression Retriever和Ensemble Retriever。

首先，Vector store-backed retriever 是最常见的一种检索器，我们的案例多采用这种方式实现检索器。这是一种使用向量存储来检索文档的检索器。它是一个围绕向量存储库包装，使其符合Retriever接口。它使用向量存储实现的搜索方法，如相似性搜索和最大边缘相关性（MMR），来查询向量存储中的文本。一旦你构建了一个向量存储，就很容易构建一个检索器。如果你已经创建好了向量存储库，直接调用as\_retriever 方法，就得到了一个检索器实例。

让我们来看看MultiQueryRetriever。在基于距离的向量数据库检索中，我们将查询语句嵌入到高维空间中，并通过计算"距离"来寻找相似的嵌入文档。但是，检索结果可能因查询语句的微小变化或嵌入并未完全捕获数据语义而有所不同。人们有时通过提示工程或调整来手动解决这些问题，但这是一项繁琐的工作。MultiQueryRetriever（多查询检索器）通过使用大型语言模型（LLM）从不同角度生成多个查询，自动化了提示调整的过程。对于每个查询，它都会检索一组相关文档，并在所有查询中取唯一的并集，从而获得一组可能相关的文档。通过对同一问题的多角度生成，MultiQueryRetriever可能能够克服基于距离的检索的某些限制，并获得更丰富的结果。

然后，我们来讲一下Contextual Compression Retriever。检索的一个挑战在于，当你将数据输入系统时，你通常不知道你的文档存储系统将面临什么样的特定查询。这意味着与查询最相关的信息可能被埋藏在一篇包含大量无关文本的文档中。将整篇文档通过你的应用程序可能会导致LLM调用更昂贵，并导致回应效果更差。为了解决这个问题，提出了Contextual Compression Retriever。这个方法很简单：在直接返回检索到的文档时，可以使用给定查询的上下文来压缩它们，这样只有相关的信息会被返回。“压缩”在这里指的是压缩单个文档的内容，并完全过滤掉一些文档。要使用Contextual Compression Retriever，你需要一个基本的检索器和一个文档压缩器。Contextual Compression Retriever将查询传给基本的检索器，获取初始文档，然后将它们传递给文档压缩器。文档压缩器接收一个文档列表，通过减少文档的内容或整个删除文档来缩短它。

再来看看Ensemble Retriever。EnsembleRetriever接受一个检索器列表作为输入，汇集他们的get\_relevant\_documents()方法的结果，并根据Reciprocal Rank Fusion算法对结果进行重新排序。通过利用不同算法的优势，EnsembleRetriever可以获得比任何单个算法更好的性能。最常见的模式是将稀疏检索器（如BM25）与密集检索器（如Embedding similarity）结合在一起，因为它们的优势是互补的。这也被称为"混合搜索"。稀疏检索器擅长根据关键词找到相关文档，而密集检索器擅长根据语义相似性找到相关文档。

我们再来看看其他一些类型的检索器：Self-querying retriever、Time-weighted vector store retriever、Vector store-backed retriever和WebResearchRetriever。

Self-querying retriever正如其名称所示，具有自我查询的能力。具体而言，对于任何自然语言查询，检索器使用一个查询构造的大型语言模型链来编写一个结构化的查询，然后将这个结构化的查询应用到其底层的向量存储。这使得检索器不仅可以使用用户输入的查询来与存储文档的内容进行语义相似度比较，还可以从用户查询中提取出存储文档元数据的过滤器，并执行这些过滤器。

Time-weighted vector store retriever使用语义相似度和时间衰减的组合。他们的得分算法为：semantic\_similarity + (1.0 - decay\_rate) ^ hours\_passed 。值得注意的是，`hours\_passed`指的是自检索器中的对象上次被访问以来经过的小时数，而不是它被创建以来的小时数。

最后，我们来看看WebResearchRetriever（网络研究检索器）。给定一个查询，这个检索器会：构建一组相关的Google搜索；对每一个进行搜索；加载所有结果的URL；然后将查询与合并的页面内容进行嵌入和相似性搜索。

在上述检索器中，你可以根据实际情况选择最适合你需求的一种，或者根据需要将它们结合使用，以达到更好的检索效果。

### 4.6 本章小结

在本章节中，我们深入探讨了LangChain的核心组成部分：LEDVR（加载器，嵌入模型包装器，文档转换器，向量存储库和检索器）工作流。这五个模块共同构成了一个流程，处理外部数据加入到LangChain框架中，让我们看到了数据是什么形式的，以什么方式流动的，它们又流向了哪里整个过程。

首先，加载器（Loader）是我们从各种来源（如本地文件、数据库、网络等）获取数据的入口。加载器将原始数据加载到系统中，为后续的处理步骤做好准备。

然后，我们有嵌入模型包装器（Text Embedding Model）。它将原始的文本数据转化为高维的向量表示，这使得我们可以利用向量空间中的距离来度量文本之间的语义相似性。

接着，文档转换器（Document Transformers）对嵌入向量进行处理，以适应特定的检索需求。例如，我们可能需要将长文档分割成多个小块。

在文档经过嵌入模型包装器和文档转换器的处理后，我们将得到的向量表示存储在向量存储库（VectorStore）中。向量存储库内置了很多搜索方法，这些搜索方法适用于不同的应用程序适用场景，向量存储库还内置了各大向量数据库平台的包装器，这些包装器可以快速检索出与给定查询最相关的文档。

最后，我们有检索器（Retriever）。检索器利用向量存储库提供的检索能力，对用户的查询进行响应。

总结来说，LEDVR工作流是一种强大的信息检索和处理工具，我们在本章节中详细介绍了每个模块的定义和使用方法、以及分类方法，希望能帮助读者更好地理解和使用LangChain框架中的数据增强模块的功能。

5链

## 5.1 为什么要链？

在许多人第一次接触LangChain的时候，可能会因为其名字误以为它是区块链相关的内容。然而实际上，LangChain的名字源自其框架的核心设计思路：用最简单的链（Chain），将大语言模型开发的各个组件链接起来，构建复杂的应用程序。

在了解了模型I/O模块后，我们可以使用模型包装器与大语言模型进行对话；在掌握了数据连接模块后，我们可以连接外部的数据和文档，使用LEDVR 工作流实现对与用户输入问题最相关的文档的检索。当我们知道如何为大语言模型增加记忆后，我们又可以提升其智能处理能力。然而，每一个模块在完成自身的功能并获得结果后，面临的都是同样的问题 — 下一步要做什么？如果仅仅一步，我们无法完全回答用户的问题，那么我们应如何安排下一步的行动？LEDVR 工作流的终点是什么？

这个问题的答案就在链这个模块和一系列的链组件中。链的主要功能是管理应用程序中的数据流动，它将不同的组件（或其他链组件）链接在一起，形成一个完整的数据处理流程。每个组件都是链中的一个环节，它们按照预设的顺序，接力完成各自的任务。在这个过程中，链自动管理各个环节之间的数据传递和格式转换，保证了整个流程的顺畅运行。

因此，链实质上是我们在处理复杂问题、需要多步骤配合解决时的“接力棒”。它将多个功能模块串联起来，使得我们可以将复杂问题分解为一系列的小问题，然后依次解决，最终实现对用户问题的全面解答。所以，无论我们走到哪一步，链都是帮助我们迈向下一步的关键工具。

人工智能的发展历程可以追溯到上个世纪，人们一直期待着人工智能能够像一个智能助理那样，帮助我们完成各种任务。从日常生活中的琐事，如预约餐厅、管理日程，到更复杂的工作任务，如数据分析、科研研究等，人们期待人工智能能够解放我们的双手，让我们能够将更多的精力投入到更具创新性和挑战性的工作中。随着大语言模型的发展，如今的人工智能已经能够理解人类的语言，对人类的问题进行理解并给出合理的回答，我们对于人工智能的预期正一步步地变为现实。

人们对人工智能的期望是像真实的人类助理一样，提供实质性的帮助。举例来说，如果你问你的智能助理“2023年是中国农历生肖是什么？”我们期望她能够犹如真人助理一般，迅速给出“兔年”的答案。然而，实际情况却是，尽管大语言模型在语言理解方面已经做得非常出色，但在获取实时信息、进行实质性的查询等方面，仍然存在着一定的局限。

为了解决这一问题，我们需要在后台实现一个查询的步骤，让应用程序能够“自己去查找”相关信息，然后再通过大语言模型的语言生成能力，将查询结果告诉用户。这个查询的过程，虽然对于用户来说是看不见的，但对于我们作为开发者来说，却是必须要去处理的。只有这样，我们开发的智能助理才能真正做到像人类助理一样，不仅能够“说话”，更能够“做事”。构建复杂应用程序需要将多个大语言模型或者其它模块的能力链接在一起。

不知道你有没有想过，开发者对于Langchain 有没有什么预期，接下来我们从开发者的角度来谈谈对大语言模型的期待。作为开发者，我们也渴望有一个像人类助理一样的框架，这个框架预设了大量的知识和模板，为我们的开发工作提供便利。

比如，我们可以将大语言模型的开发过程想象成一个复杂的机械装配任务。我们有很多不同的模块，就像一堆精密的零配件，我们需要把这些零配件组装在一起，构建出一个能够正常运行的机器。

这时候，如果有一个“助理”可以帮我们管理这些零配件，提供合适的模板，那么这将大大提升我们的工作效率。这个“助理”可以帮我们将零配件按照一定的顺序和规则排列组织，将相关的部件放在一起，使得我们可以更加高效地找到需要的部件。

再进一步想象，这个“助理”还可以为我们提供一个良好的工作环境，就像一个设施齐全的工厂。在这个工厂里，我们有足够的空间进行零件的拼接，还有各种必要的设施，比如电力供应、供水设施等，使得我们可以在一个稳定、舒适的环境下进行开发工作。

在Langchain框架中，链模块扮演了一种类似于这样的“助理”的角色，就像我们在生活中遇到各种琐碎的事情，需要一位贴心的助理帮我们理顺，提供有效的解决方案。同样，当我们在开发复杂的应用时，也需要这样一个助理，它可以帮我们有序地组织和管理数据流动，帮助我们处理数据连接、模型输入输出等环节中的各种细节。

我们可以想象，这位“助理”就像一位负责运营的经理，处理的不是公司的人力资源，而是数据和模型包装器。当我们将各个模型包装器和内置的链组件，也可以看作公司的员工，各自有各自的职责和专长时，这位助理将他们有序地组织在一起，链接在一起，让他们在各自的岗位上发挥最大的效能，形成一个井然有序的工作流程。

这就是链模块的作用，它以极其简单的方式，实现了强大的功能。它将复杂的任务简化，将庞大的应用模块化，让我们在开发复杂应用时，能更专注于解决问题，而非陷入琐碎的数据流动和格式转换中。

对于我们开发者来说，链模块的设计正好满足了我们对开发大语言模型的期望。链的强大功能及其简洁的设计让我们可以更容易地实现复杂应用的开发和维护。这也是链模块设计的核心原理。

链模块很好地体现了其解决问题的理念。每一个链都是由一系列链组件构成，这些链组件可以是大语言模型、数据查询模块或者是文档处理链，他们都是为了解决某一特定问题而设计的。这样的设计，让我们可以灵活地组合使用各种链组件，形成一个完整的数据处理流程，从而解决更复杂的问题。

当我们理解了链的概念，掌握了链的基本使用方法后，我们就可以开始了解各种各样的内置链组件。这些链组件都是为了解决一些常见问题而设计的，他们包括数据的查询、记忆的处理、模型的调用等等。通过了解这些链组件解决的问题，我们可以更好地选择和使用他们，进一步提升我们的开发效率。这就是我们选择Langchain框架，选择使用链模块的原因。

在本书中，整个链的内容是称为“链模块”，而具体的链，我们称为链组件，比如常见的模型链、会话链、QA链都是链组件。组件的称呼可以帮助我们理解链的内容。

### 5.1.1 链的定义

相信你也留意到 LangChain的Logo — 一只鹦鹉和链条。鹦鹉，象征着大预言模型的“学舌”能力，寓意着这类模型对人类文本的强大预测能力。而链条，则由无数链环组成，象征着链模块中各种链组件的有序连接。

由链模块组织和管理的的数据流动，正如哲学家赫拉克利特所言，“万物皆流”。链模块的设计理念，也契合了古印度哲学家龙树的观点：“没有任何本身就独立于其他事物的存在”，每个组件的存在和运行都依赖于其他组件，与其他组件息息相关。链模块通过“包装器”的形式，将这种相互依存和关联的链组件具象化，通过这个“包装器”把链组件都“包”在一起，将复杂的程序设计流程变得可视化。

通过链模块的组织和管理数据，我们看到了一个完整的数据处理流程，从输入数据的接收，到数据的处理，再到最终的模型预测。这一流程，就像链条上的每一个链环，都是为了解决特定问题而存在，相互依赖，相互关联。通过将这些组件连接起来，我们形成了一个有序、高效的数据处理链条，从而创造出强大的应用力量。这就是链模块的强大之处，它将分散的组件连接在一起，使得整个应用程序的流程更加清晰、有序，从而更易于理解和管理。

所以链到底是什么？链是连接组件，管理组件数据流的“包装器”。

那如果没有链，对于语言大模型开发会有什么影响？其实，对于简单的应用是可以没有链的存在。链也不是“万金油”，到哪里都好用，如果是简单的应用，并不需要链。但是对于更复杂的应用，我们需要将多个大语言模型或组件进行“链”连接和组合，这样才能创造出更强大、更具协同性的应用。

这种链的思想在创新应用中的价值已经得到了验证。2023年，Johei Nakajima在Twitter上分享了一篇名为《使用 GPT-4、Pinecone、LangChain 进行多样化应用的任务驱动自主代理》的论文，其中他介绍了最新的Baby AGI。虽然Baby AGI现在还只是概念代码阶段，但是通过这个概念我们可以看出，链式结构是实现创新应用的非常有价值的工具。

我们再通过代码示例，观看链模块的工作流程：从输入数据的接收，到数据的处理，再到最终的大语言模型预测。整个流程是通过使用链模块将各个链组件连接在一起实现的，如同一条有序、高效的数据处理链组件，链组件包含的数据即“万物皆流”。

首先，我们需要安装所需的库。安装 OpenAI 和 LangChain 这两个 Python 库。然后，我们设置环境变量 `OPENAI\_API\_KEY` 用于认证和调用 OpenAI 的 API。

pip -q install openai langchain

在 llm = OpenAI(temperature=0.9) 这行代码中，我们实例化了一个大语言模型对象 `llm`，并设置了模型的生成文本的多样性参数 temperature ，值应设置为0到1之间，越接近1，代表创意性越强。

from langchain.llms import OpenAI  
from langchain.prompts import PromptTemplate  
  
llm = OpenAI(temperature=0.9)

我们定义了一个输入模板 PromptTemplate 。这个模板接收一个名为 "product" 的输入变量，并使用这个变量来生成一个关于为制作该产品的公司取名的问题。

prompt = PromptTemplate(  
 input\_variables=["product"],  
 template="What is a good name for a company that makes {product}?",  
)

接下来的代码 chain = LLMChain(llm=llm, prompt=prompt) 就是我们实际使用的链。我们在这里构建了一个 LLMChain 包装器，这是我们使用的第一个内置链组件。该对象将 llm 和 prompt 这两个组件连接在一起。

from langchain.chains import LLMChain  
chain = LLMChain(llm=llm, prompt=prompt)

最后，我们调用 chain.run("colorful socks") 来运行这个链。该链首先将输入的文本 "colorful socks" 插入到我们之前定义的 PromptTemplate 中，生成一个完整的提示词。然后，它将这个问题传递给 LLM 模型包装器，并输出模型的预测结果。

chain.run("colorful socks")

这就是链模块的使用示例，它将输入数据的接收、处理、和模型预测等步骤连接在一起，形成一个完整的应用程序工作流程。这样做的好处是，使得整个应用程序的数据流程更加清晰、有序，从而更易于组织和管理。

### 5.1.2 链的使用

Chain 是一个Python类，表示一种操作流程。这是一种可以接受一些输入，通过特定的逻辑处理这些输入，然后产生一些输出的对象。在这里分三步，介绍链的使用 。

1. 准备输入

首先，你需要准备一些输入，输入是一个字典，其键是由 prompt 对象的 input\_variables 属性决定的。你需要根据实际的 prompt 对象来确定需要哪些输入。

2. 实例化Chain

接着，你需要实例化Chain。你需要提供一个 BasePromptTemplate 对象和一个 BaseLanguageModel 对象。

3. 运行Chain

使用函数式调用是最方便的方法，传递好参数便运行Chain。 还可以使用run() 、 arun() 或 apply() 方法来运行Chain。这些方法都接受输入以及一些可选的参数。

- inputs ：字典类型，包含了需要的输入变量。

- return\_only\_outputs （可选）：布尔值，表示是否只返回输出。如果为 True ，则只返回由这个Chain生成的新键。如果为 False ，则返回输入键和由这个Chain生成的新键。默认为 False 。

- callbacks （可选）：用于这个Chain运行的回调。如果未提供，将使用传递给Chain的回调。

- include\_run\_info （可选）：布尔值，表示是否在响应中包含运行信息。默认为 False 。

run() 和 arun() 方法都是直接运行Chain并获取字符串的方法。这两个方法区别在于是否异步, arun 支持异步调用。

apply() 方法是一个可以由子类自定义的方法。例如，在 LLMChain 中， apply() 方法接受一个字典列表，每个字典都包含一组输入。

这些都是使用`Chain`的基本步骤。根据具体的Chain类和你的需求，可以适当调整这些步骤。

异步支持

LangChain通过利用asyncio库为链（Chain）提供了异步支持。

目前在LLMChain（通过arun, apredict, acall）、LLMMathChain（通过arun和acall）、ChatVectorDBChain以及QA链中支持异步方法。其他链的异步支持正在规划中。

使用方法解析

所有的链都可以像函数一样被调用。当链对象只有一个输出键（也就是说，它的output\_keys中只有一个元素）的时候，我们预期的结果只需要一个字符串，可以使用run方法。

在LangChain中，所有继承自Chain类的对象，提供了一些用于执行链逻辑的方式。其中一种比较直接的方式就是使用\_\_call\_\_方法。\_\_call\_\_ 方法是 Chain 类的一个方法，它让 Chain 类的实例可以像函数一样被调用，比如 result = chain(inputs, return\_only\_outputs=True)就完成了调用链。

先看看\_\_call\_\_方法的定义：

def \_\_call\_\_(  
 self,  
 inputs: Union[Dict[str, Any], Any],  
 return\_only\_outputs: bool = False,  
 callbacks: Callbacks = None,  
 \*,  
 tags: Optional[List[str]] = None,  
 include\_run\_info: bool = False,  
 ) -> Dict[str, Any]:

这个 \_\_call\_\_ 方法接收的参数，最有用的是以下三个：

inputs：这个参数是要传递给链的输入。它的类型是 Any，这意味着可以接收任何类型的输入。

return\_only\_outputs：这个参数是一个布尔值，如果设为 True，则只返回输出结果。如果设为 False，则可能返回其他额外的信息。

callbacks：这个参数是回调函数的列表，它们将在链执行过程中的某些时刻被调用。

\_\_call\_\_ 方法返回一个字典，这个字典包含了链执行的结果和可能的其他信息。

在 Python 中，如果一个类定义了 \_\_call\_\_ 方法，那么这个类的实例就可以像函数一样被调用。例如，如果 chain 是 Chain 类的一个实例，那么你可以像调用函数一样调用 chain：

result = chain(inputs, return\_only\_outputs=True)

在这个调用中，inputs 是要传递给链的输入，return\_only\_outputs=True 表示只返回输出结果。返回的 result 是一个字典，包含了链执行的结果。

使用的时候，最重要的参数是 inputs:

chat = ChatOpenAI(temperature=0)  
prompt\_template = "Tell me a {adjective} joke"  
llm\_chain = LLMChain(llm=chat, prompt=PromptTemplate.from\_template(prompt\_template))  
  
llm\_chain(inputs={"adjective": "corny"})

返回的结果是：

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

你可以通过设置return\_only\_outputs为True来配置它只返回输出键值。

llm\_chain("corny", return\_only\_outputs=True)

返回的结果就不包含 "adjective": "corny"：

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

然而，当链对象只有一个输出键（也就是说，它的output\_keys中只有一个元素）的时候，我们可以使用run方法。

# llm\_chain only has one output key, so we can use run  
llm\_chain.output\_keys['text']

output\_keys中只有一个元素 ['text']，我们可以run方法：

llm\_chain.run({"adjective": "corny"})

如果输入的键值只有一个，预期的输出也是一个字符串，那么输入可以是字符串也可以是对象，可以使用run方法也可以使用\_\_call\_\_方法。

run方法将整个链的输入键值（input key values）进行处理，并返回处理后的结果。需要注意的是，与\_\_call\_\_方法可能返回字典形式的结果不同，run方法总是返回一个字符串。这也是为什么当链对象只有一个输出键的时候，我们倾向于使用run方法，因为这时候处理结果自然只有一个，返回字符串形式更直观也更便于处理。

例如，假设我们有一个链对象，它的任务是根据输入的文本生成摘要，那么在调用run方法的时候，我们可以直接将待摘要的文本作为参数输入，然后得到摘要后的文本。在这种情况下，你可以直接输入字符串，而无需指定输入映射。

另外，你可以很容易地将一个Chain对象作为一个工具，通过它的run方法集成到你的Agent中，这样可以将链的处理能力直接用于你的Agent逻辑中。

支持自定义链

你可以子类化Chain并实现你自己的自定义链。从其输出中仅仅调试链对象可能会比较困难，因为大多数链对象涉及到相当多的输入提示预处理和LLM输出后处理。

链的调试

将verbose设置为True将会在运行链对象时打印出一些链对象的内部状态。

conversation = ConversationChain(  
 llm=chat,  
 memory=ConversationBufferMemory(),  
 verbose=True  
)  
conversation.run("What is ChatGPT?")

加记忆的链

链可以使用Memory对象进行初始化，这将使得在调用链时数据持久化，使得链具有状态。

from langchain.chains import ConversationChain  
from langchain.memory import ConversationBufferMemory  
  
conversation = ConversationChain(  
 llm=chat,  
 memory=ConversationBufferMemory()  
)  
  
conversation.run("Answer briefly. What are the first 3 colors of a rainbow?")  
# -> The first three colors of a rainbow are red, orange, and yellow.  
conversation.run("And the next 4?")  
# -> The next four colors of a rainbow are green, blue, indigo, and violet.

链序列化

我们使用的序列化格式是json或yaml。目前，只有一些链支持这种类型的序列化。我们将随着时间的推移增加支持的链的数量。首先，让我们看看如何将链保存到磁盘。这可以通过.save方法完成，并指定一个带有json或yaml扩展名的文件路径。我们可以使用load\_chain方法从磁盘加载链。

### 5.1.3 基础链类型

基础链的类型分为四种，包括LLMChain、RouterChain、顺序链（Sequential Chains）和转换链（Transformation Chain）。

LLMChain是一种简单的链。它在LangChain中被广泛应用，包括在其他链和代理中。LLMChain由提示词模板和模型包装器（可以是LLM或 Chat Model模型包装器）组成。它使用提供的输入键值格式化提示词模板，将格式化的字符串传递给LLM 模型包装器，并返回LLM模型包装器的输出。上一节的示例代码便是使用了 LLMChain 这个链组件。

RouterChain是一种使用路由器链创建的链，它可以动态地选择给定输入的下一条链。路由器链由两部分组成：路由器链本身（负责选择要调用的下一条链）和目标链（路由器链可以路由到的链）。

顺序链（Sequential Chains）是在调用语言模型后的下一步，特别是当你希望将一次调用的输出作为另一次调用的输入时。顺序链允许你连接多个链并将它们组成执行特定场景的流水线。顺序链有两种类型：SimpleSequentialChain（最简单形式的顺序链，其中每一步都有一个单一的输入/输出，一个步骤的输出是下一个步骤的输入）和SequentialChain（一种更通用的顺序链，允许多个输入/输出）。

转换链（Transformation Chain）是一种使用通用转换链的方法。转换链是一个用于文本转换的链，它用来将输入文本进行转换，具体的转换操作是由 transform\_func 函数定义的。TransformChain 是 LangChain 中的一个组件，它允许用户自定义对输入数据的处理操作。

### 5.1.4 工具链类型

在Langchain 中，“链”的概念是最经常使用的。这些”链”其实就是由一系列工具链构成的，每一个工具都可以视为整个链中的一个环节。这些环节可能非常简单，例如将一个提示词模板和一个大型语言模型链接起来，形成一个大型语言模型链。然而，也可能更加复杂，例如在整个流程中，通过多个环节进行多个步骤的链接。这可能还包括多个大型语言模型以及各种不同的实用工具等。在工具链中，一个链的输出将成为下一个链的输入，这就形成了一个输入输出的链式流程。例如，你可能会从大型语言模型的输出中提取某些内容，将其作为Wolfram Alpha查询的输入，然后将查询结果带回，并再次通过大型模型生成将返回给用户的响应。这就是一个典型的工具链的示例。

常见工具链的功能与应用

在实际的应用中，一些常见的工具链如APIChain、ConversationalRetrievalQA等已经被封装好了。

APIChain使得大型语言模型可以与API进行交互，以获取相关的信息。构建该链时，需要提供一个与所提供的API文档相关的问题。

ConversationalRetrievalQA链在检索问答链的基础上提供了一个聊天历史组件。它首先将聊天历史（要么明确传入，要么从提供的内存中检索）和问题合并成一个独立的问题，然后从检索器中查找相关的文档，最后将这些文档和问题传递给一个问答链，以返回响应。

对于需要对多个文档进行文档合并的任务，我们可以使用文档合并链，如MapReduceDocumentsChain或StuffDocumentsChain等。

对于需要从同一段落中提取多个实体及其属性的任务，我们可以使用提取链。

还有一些专门设计用来满足特定需求的链，如ConstitutionalChain，这是一个保证大型语言模型输出遵循一定宪法原则的链，通过设定特定的规则和指导方针，使得生成的内容符合这些原则，从而提供更受控、符合伦理和上下文适当的回应。

工具链的使用方法

这些工具链的使用方法通常是先使用类方法实例化，然后通过run方法调用，输出结果是一个字符串，然后将这个字符串传递给下一个链。类方法通常以 “from” 和下划线开始，比较常见的有from\_llm() 和 from\_chain\_type()，他们都接受外部的数据来源作为参数。

from\_llm() 意味着实例化的时候，传递的 llm 模型包装器在内部已被包装为 LLMChain。而使用from\_chain\_type() 方法构造链的情况，只发生在要设置 combine\_documents\_chain 属性的子类上，目前只有文档问答的链才会使用这个类方法，比如 load\_qa\_with\_sources\_chain 和 load\_qa\_chain。也是因为只有这些文档回答链才会要对文档进行合并处理。

下面以SQLDatabaseChain为例子，看看如何使用工具链。SQLDatabaseChain就是一个通过from\_llm()方法实例化的链，它用于回答SQL数据库上的问题。

from langchain import OpenAI, SQLDatabase, SQLDatabaseChain  
  
db = SQLDatabase.from\_uri("sqlite:///../../../../notebooks/Chinook.db")  
llm = OpenAI(temperature=0, verbose=True)  
  
db\_chain = SQLDatabaseChain.from\_llm(llm, db, verbose=True)  
  
db\_chain.run("How many employees are there?")

运行的结果是：

> Entering new SQLDatabaseChain chain...  
 How many employees are there?  
 SQLQuery:  
  
 /workspace/langchain/langchain/sql\_database.py:191: SAWarning: Dialect sqlite+pysqlite does \*not\* support Decimal objects natively, and SQLAlchemy must convert from floating point - rounding errors and other issues may occur. Please consider storing Decimal numbers as strings or integers on this platform for lossless storage.  
 sample\_rows = connection.execute(command)  
  
  
 SELECT COUNT(\*) FROM "Employee";  
 SQLResult: [(8,)]  
 Answer:There are 8 employees.  
 > Finished chain.  
  
  
  
  
  
 'There are 8 employees.'

## 5.2 细说基础链

### 5.2.1 从 LLMChain 链说起

LLMChain 链将是一个非常简单的链组件 。这绝对是您最常见到的链组件。基本上只是将一个大型语言模型与提示（Prompt）链在一起。然后使用提示词模板来提供输入,并将一些内容输入到其中。

以下是文章的事实提取场景下，使用LLMChain 链的示例代码，这个比我们示例代码更加复杂一点：

首先我们安装库:

!pip -q install openai langchain huggingface\_hub

还需要设置密钥:

import os  
  
os.environ['OPENAI\_API\_KEY'] = ''  
os.environ['HUGGINGFACEHUB\_API\_TOKEN'] = ''

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

在这里，您可以看到我正在设置OpenAI text-davinci-003 模型，我们将温度设置为零。如果你知道这是一个默认的标准模型的话，你肯定知道其中许多将根据默认值进行设置。

llm = OpenAI(model\_name='text-davinci-003',   
 temperature=0,   
 max\_tokens = 256)

我在这里有一篇小文章，所以我将要做的实际上是事实提取。因此，在这里，我基本上提取了一篇关于Coinbase的文章。所以这是一篇相当长的文章，如果我们看一下，它有3500个字符。

article = '''Coinbase, the second-largest crypto exchange by trading volume, released its Q4 2022 earnings on Tuesday, giving shareholders and market players alike an updated look into its financials. In response to the report, the company's shares are down modestly in early after-hours trading.In the fourth quarter of 2022, Coinbase generated $605 million in total revenue, down sharply from $2.49 billion in the year-ago quarter. Coinbase's top line was not enough to cover its expenses: The company lost $557 million in the three-month period on a GAAP basis (net income) worth -$2.46 per share, and an adjusted EBITDA deficit of $124 million.Wall Street expected Coinbase to report $581.2 million in revenue and earnings per share of -$2.44 with adjusted EBITDA of -$201.8 million driven by 8.4 million monthly transaction users (MTUs), according to data provided by Yahoo Finance.Before its Q4 earnings were released, Coinbase's stock had risen 86% year-to-date. Even with that rally, the value of Coinbase when measured on a per-share basis is still down significantly from its 52-week high of $206.79.That Coinbase beat revenue expectations is notable in that it came with declines in trading volume; Coinbase historically generated the bulk of its revenues from trading fees, making Q4 2022 notable. Consumer trading volumes fell from $26 billion in the third quarter of last year to $20 billion in Q4, while institutional volumes across the same timeframe fell from $133 billion to $125 billion.The overall crypto market capitalization fell about 64%, or $1.5 trillion during 2022, which resulted in Coinbase's total trading volumes and transaction revenues to fall 50% and 66% year-over-year, respectively, the company reported.As you would expect with declines in trading volume, trading revenue at Coinbase fell in Q4 compared to the third quarter of last year, dipping from $365.9 million to $322.1 million. (TechCrunch is comparing Coinbase's Q4 2022 results to Q3 2022 instead of Q4 2021, as the latter comparison would be less useful given how much the crypto market has changed in the last year; we're all aware that overall crypto activity has fallen from the final months of 2021.)There were bits of good news in the Coinbase report. While Coinbase's trading revenues were less than exuberant, the company's other revenues posted gains. What Coinbase calls its "subscription and services revenue" rose from $210.5 million in Q3 2022 to $282.8 million in Q4 of the same year, a gain of just over 34% in a single quarter.And even as the crypto industry faced a number of catastrophic events, including the Terra/LUNA and FTX collapses to name a few, there was still growth in other areas. The monthly active developers in crypto have more than doubled since 2020 to over 20,000, while major brands like Starbucks, Nike and Adidas have dived into the space alongside social media platforms like Instagram and Reddit.With big players getting into crypto, industry players are hoping this move results in greater adoption both for product use cases and trading volumes. Although there was a lot of movement from traditional retail markets and Web 2.0 businesses, trading volume for both consumer and institutional users fell quarter-over-quarter for Coinbase.Looking forward, it'll be interesting to see if these pieces pick back up and trading interest reemerges in 2023, or if platforms like Coinbase will have to keep looking elsewhere for revenue (like its subscription service) if users continue to shy away from the market.  
'''

然后我们需要的提示词是从这段文本中提取关键事实，不包括观点，给每个事实编号，并保持它们的句子简短。

Fact\_extraction\_prompt = PromptTemplate(  
 input\_variables=['text\_input'],  
 template='Extract the key facts out of this text. Don’t include opinions. Give each fact a number and keep them short sentences. :\n\n {text\_input}'  
)

然后我将基本上将输入设置为这个文本输入。好吧，制作链实际上非常简单，我们只需说我们将使用 LLMChain ，我们传入 LLMChain。然后我们传入我们将要使用的提示词模板，所以这里我有事实提取提示。然后我们将其传入。然后我们可以运行它。

fact\_extraction\_chain = LLMChain(llm=llm, prompt=fact\_extraction\_prompt)  
  
facts = fact\_extraction\_chain.run(article)  
  
print(facts)

您可以看到，在运行它之后，确实发生了变化。

1. Coinbase released its Q4 2022 earnings on Tuesday.  
2. Coinbase generated $605 million in total revenue in Q4 2022.  
3. Coinbase lost $557 million in the three-month period on a GAAP basis.  
4. Coinbase's stock had risen 86% year-to-date before its Q4 earnings were released.  
5. Consumer trading volumes fell from $26 billion in Q3 2022 to $20 billion in Q4 2022.  
6. Institutional volumes across the same timeframe fell from $133 billion to $125 billion.  
7. The overall crypto market capitalization fell about 64%, or $1.5 trillion during 2022.  
8. Trading revenue at Coinbase fell from $365.9 million in Q3 2022 to $322.1 million in Q4 2022.  
9. Coinbase's "subscription and services revenue" rose from $210.5 million in Q3 2022 to $282.8 million in Q4 2022.  
10. Monthly active developers in crypto have more than doubled since 2020 to over 20,000.

它很好地从我们的文章中获取了事实。它做得相当不错，我们从这篇文章中得到了10个事实。

### 5.2.2 顺序链

另一种通用链是顺序链 (Sequential)，它基本上是多个链组件的组合。您会发现，虽然链由单个工具组成，但它们也可以由其他链组成，这些链会连接在一起。工具链是许多神奇事情发生的地方。有各种不同的工具链，这里只是其中一部分，而且随着时间的推移，预计还会添加更多链组件。

我们在上一节制作了一个提取新闻10个事实的LLMChain, 为了让你理解顺序链是如何工作的，我们先加一个新的链。

现在我们要做的是制作一个新的链组件。然后我们将把其中一些内容链在一起，所以下一个要做的也是一个LLMChain。

这将采用上一节的10个事实。但我们将把它们改写成投资者报告的形式，所以您可以在这里看到，我们的提示词是这样写：“你是高盛的分析师，接受以下事实列表，并用它们为投资者撰写一个简短的段落，不要遗漏关键信息。我们也可以放一些东西在这里，也不要杜撰信息，但这是我们要传入的事实。” {facts} 是模板字符串的占位符，也就是提示词模板的 input\_variables 的一个值： input\_variables=["facts"] 。

investor\_update\_prompt = PromptTemplate(  
 input\_variables=["facts"],  
 template="You are a Goldman Sachs analyst. Take the following list of facts and use them to write a short paragrah for investors. Don't leave out key info:\n\n {facts}"  
)

再次强调，这是一个LLMChain ，我们传入LLM，我们仍然使用上面定义的原始模型，区别在于我们传入提示词已经不同了。然后我们可以运行它。

investor\_update\_chain = LLMChain(llm=llm, prompt=investor\_update\_prompt)  
  
investor\_update = investor\_update\_chain.run(facts)  
  
print(investor\_update)  
len(investor\_update)

可以看到，文章内容和这篇文章的字符串长度值都打印出来了。

Coinbase released its Q4 2022 earnings on Tuesday, revealing total revenue of $605 million and a GAAP loss of $557 million. Despite the losses, Coinbase's stock had risen 86% year-to-date before its Q4 earnings were released. Consumer trading volumes fell from $26 billion in Q3 2022 to $20 billion in Q4 2022, while institutional volumes fell from $133 billion to $125 billion. The overall crypto market capitalization fell about 64%, or $1.5 trillion during 2022. Trading revenue at Coinbase fell from $365.9 million in Q3 2022 to $322.1 million in Q4 2022, while its "subscription and services revenue" rose from $210.5 million in Q3 2022 to $282.8 million in Q4 2022. Despite the market downturn, monthly active developers in crypto have more than doubled since 2020 to over 20,000.  
788

它写了一篇相当连贯的好文章。它比之前的要短得多。

加入顺序链

我们将使用简单的顺序链（SimpleSequentialChain）来完成刚刚的提炼10个事实和写摘要的2个任务。简单的顺序链就像PyTorch中的标准顺序模型一样，你只是从A到B到C，没有做任何复杂的操作。

我们还有这样的想法，一个链的输出将成为下一个链的输入。你可以看到，我们已经设置了完整的链，我们将拥有我们的提取事实链。而且我们将在这里拥有我们的投资摘要链。

现在当我取出原始文章并运行它时，它将完成这两个操作。你可以看到，现在它已经完成了事实提取。现在它已经完成了重写。然后它在这里完成了链的操作。这是我们将这些事物链在一起的一种方式，而不是必须重写代码来完成一件事，然后再去做另一件事等等。

from langchain.chains import SimpleSequentialChain, SequentialChain  
  
full\_chain = SimpleSequentialChain(chains=[fact\_extraction\_chain, investor\_update\_chain], verbose=True)

我们确实可以看到，如果我们取出我们得到的响应，我们得到了投资摘要链的结果，即使我们传入的是原始文章。

response = full\_chain.run(article)

通过使用顺序链，我们成功地将原本需要手动执行的二步操作简化为了一步。在这个顺序链组件内部，数据的流动和管理都被自动处理了，我们并不需要关心这些细节。我们所需要做的，只是指定任务以及任务的执行顺序，顺序链会按照我们的要求，将任务有序地组织起来并执行，最后返回我们想要的结果。

在本章开始时，我们提到，作为开发者，我们希望对大语言模型开发的框架有一个“助理”来协助我们的工作。在我们刚才使用顺序链的过程中，链组件就像一个“助理”，帮助我们处理了数据的组织和管理，与模型平台的API进行交互，甚至处理可能发生的错误。我们不需要过多地关心这个“助理”是如何工作的，只需知道，最后我们得到了我们想要的“答案”。我们的时间更多的放在如何让这个“答案”变得更准确。

### 5.2.3 工具链 APIChain 链

一个非常有用的工具链的例子是API工具链，所以在这里我只是向你展示了用于天气信息的一个例子。我们设置了要使用的API，这将根据这些文档编写API调用。这就是这个调用将输出的内容。然后这将使用该调用查询API并返回结果。

from langchain import OpenAI  
from langchain.chains.api.prompt import API\_RESPONSE\_PROMPT  
from langchain.chains import APIChain  
from langchain.prompts.prompt import PromptTemplate

openai\_api\_key="填入你的OpenAI密钥"

llm = OpenAI(temperature=0, openai\_api\_key=openai\_api\_key  
 max\_tokens=100)

显然，它只能回答API能够给你的内容，这里它基本上是在给我们返回这个并且它告诉我们，是的，这个JSON响应中的一些事情表明正在下雨，但是需要注意的是，通常文档加上URL加上JSON会超过大型语言模型可以处理的标记数，因此如果您在达芬奇模型上使用超过4000个标记，可能会出现错误。

from langchain.chains.api import open\_meteo\_docs  
chain\_new = APIChain.from\_llm\_and\_api\_docs(llm, open\_meteo\_docs.OPEN\_METEO\_DOCS, verbose=True)

chain\_new.run('What is the temperature like right now in Bedok, Singapore in degrees Celcius?')

这里你可以看到，我问它在新加坡 Bedok 的温度是多少摄氏度。它写下了这个用于查询的URL。

> Entering new APIChain chain...  
https://api.open-meteo.com/v1/forecast?latitude=1.3&longitude=103.9&hourly=temperature\_2m&current\_weather=true&temperature\_unit=celsius  
{"latitude":1.375,"longitude":103.875,"generationtime\_ms":0.38802623748779297,"utc\_offset\_seconds":0,"timezone":"GMT","timezone\_abbreviation":"GMT","elevation":6.0,"current\_weather":{"temperature":26.1,"windspeed":10.5,"winddirection":16.0,"weathercode":3,"time":"2023-02-22T14:00"},"hourly\_units":{"time":"iso8601","temperature\_2m":"°C"},"hourly":{"time":["2023-02-22T00:00","2023-02-22T01:00","2023-02-22T02:00","2023-02-22T03:00","2023-02-22T04:00",27.6,27.5,27.2,26.8,26.4,26.1,25.7,25.5,25.4,25.3,25.2,25.1,25.0,24.9,24.9,24.9,24.9,24.9]}}  
  
 > Finished chain.  
 The temperature right now in Bedok, Singapore is 26.1 degrees Celcius.

它给我们返回了当前的温度和位置。

另外要考虑的一件事是，这是相当昂贵的，如果我们每千个标记支付两美分我们刚刚输入了4000个标记，只是为了获取天气或其他东西，这并不总是最高效的方法，但它确实显示了 LangChain 可以做这些事情。您可以编写一些代码来调用您想要的API调用。

### 5.2.4 工具链 PALChain 链

为什么用 PALChain ？

我们基本上是使用提示词模板完成的，这样做的方式是将这个文字陈述转化为一个小型的Python函数，然后计算数学问题，而不是仅仅依靠语言模型猜测。

来看一个有趣的例子。这是非常简单的数学问题，食堂有23个苹果，如果他们用了20个来吃午饭，又买了6个，那么他们现在还剩下多少个苹果。

问题在于，如果你使用的是大型语言模型，它们可能会得到正确的答案，但如果你使用的是一些较小的模型，甚至只是像T5模型一样，大多数T5模型都会得到这样的错误答案，而不是依赖其中一个模型来做这个，我们可以使用这个方法，它基本上是获取这些数据并进行重写。

openai\_api\_key="填入你的OpenAI密钥"

llm = OpenAI(temperature=0, openai\_api\_key=openai\_api\_key  
 max\_tokens=100)

from langchain.chains import PALChain  
pal\_chain = PALChain.from\_math\_prompt(llm, verbose=True)  
  
question = "Jan has three times the number of pets as Marcia. Marcia has two more pets than Cindy. If Cindy has four pets, how many total pets do the three have?"

pal\_chain.run(question)

你可以看到它写了一个Python函数，它使用了文档字符串将我们之前的内容放在这里，我们从苹果的初始开始，所以它只是将这些变量赋值。然后苹果剩下的数量等于初始苹果减去使用的苹果加上购买的苹果，它确实给了我们准确的结果。

> Entering new PALChain chain…  
def solution():  
 “””"""The cafeteria had 23 apples. If they used 20 for lunch and bought 6 more, how many apples do they ha”””"""  
 apples\_initial = 23  
 apples\_used = 20  
 apples\_bought = 6  
 apples\_left = apples\_initi–l - apples\_used + apples\_bought  
 result = apples\_left  
 return result  
  
 > Finished chain.  
9

然后我们可以将输出带入另一个大型语言模型中，然后以对话的方式重新表达它，这样它可以告诉你苹果剩下的数量是多少，或者我们可以直接从这个模块中获取输出。

## 5.3 合并文档链

在许多应用场景中，我们需要与文档进行交互，如阅读说明书、浏览产品手册等等。近来，基于这些场景开发的应用，如ChatDOC和ChatPDF，都受到了广大用户的欢迎。为了满足对特定文档进行问题回答、提取摘要等需求，Langchain设定了几种合并文档链类型。

这些核心链都是为处理文档而设计的。它们在对文档进行概括、回答文档问题、从文档中提取信息等方面非常有用。

但是文档链的类型给初学者造成了很大的困扰。主要是因为我们通常不清楚在指定了这些类型后，中间的处理流程发生了什么。如果我们能从各个类型的具体步骤进行理解，就会发现，这些类型的主要区别在于它们处理输入文档的方式，以及在中间过程中与模型的交互次数和答案来源于哪些阶段。理解了这些，我们就可以更清楚地认识到各种类型的优缺点，从而在生产环境中做出更好的决策。

换句话说，一旦我们理解了每个类型的具体步骤提交了什么提示词模板，就可以明确知道使用哪种类型更符合我们的需求。我们会在后面对每个类型经历的具体步骤进行拆解。在这里我们先做个概述，没看懂可移步相应的文档类型的小节。

“Stuff链” 是处理文档链中最直接的一个。它接收一组文档，将它们全部插入到一个提示中，然后将该提示传递给LLM。这种链适合于文档较小且大部分调用只传入少量文档的应用。

“Refine链” 通过遍历输入文档并迭代更新其答案来构建响应。对于每个文档，它将所有非文档输入、当前文档和最新的中间答案传递给LLMChain链组件，以获得新的答案。

由于精化链一次只向LLM传递一个文档，因此它非常适合需要分析比模型上下文能容纳更多的文档的任务。但显然，这种链会比如Stuff链这样的链调用更多的LLMChain链组件。此外，还有一些任务很难通过迭代来完成。例如，当文档经常相互交叉引用或任务需要许多文档的详细信息时，精化链的表现可能较差。

“Map Reduce 链”首先将LLMChain链组件单独应用于每个文档（Map步骤），并将链输出视为新的文档。然后，它将所有新文档传递给一个单独的“Combine Documents Chain”，以获得单一的输出（Reduce步骤）。它可以选择首先压缩或合并映射的文档，以确保它们适合“Combine Documents Chain”（这将经常将它们传递给LLMChain链）。如果需要，这个压缩步骤将递归地执行。

“重排链（Map Re-rank）”对每个文档运行初始提示，不仅试图完成任务，还对其答案的确定程度给出评分。得分最高的响应将被返回。

### 5.3.1 Stuff链

在大语言模型应用开发时代，各种不同的处理链组件可以用来优化信息检索和生成答案。本文将解析 “Stuff链” 类型的处理链，并说明其如何通过改变输入的组织和输出的生成方式来提高文档搜索的质量。

整体流程是“Stuff链”文档处理链是一种直接的处理方式。它接收一组文档，将所有文档插入到一个提示中，然后将该提示传递给模型包装器。

在插入文档阶段，系统接收一组文档，将它们全部插入到一个提示中。提示(Prompt) 是全部文档内容。这种方式适用于文档较小且大部分调用只传入少量文档的应用。它可以简单地将所有文档拼接在一起，形成一个大的提示，然后将这个提示传递给模型包装器。

在生成答案阶段，系统将包含所有文档的提示传递给模型包装器。模型包装器根据这个提示生成答案。由于所有的文档都被包含在同一个提示中，所以模型包装器生成的答案会考虑到所有的文档。

最终实现效果是通过 “Stuff链” 文档处理链，系统可以对包含多个文档的问题生成一个全面的答案。这种处理方式可以提高文档搜索的质量，特别是在处理小文档和少量文档的情况下。

适用场景的话，由于 “Stuff链” 的处理链方式主要适用于处理小文档和少量文档的情况，所以它特别适用于那些大部分调用只传入少量文档的应用。然而，对于需要处理大量文档或者文档较大的情况，可能需要使用其他类型的处理链，如 Refine 或 MapReduce。

### 5.3.2 Refine链

本文将解析Refine链，并说明其如何通过改变输入的组织和输出的生成方式来提高文档搜索的质量。

整体流程是 “Refine链”文档处理链通过遍历输入文档并迭代更新其答案来构建响应。对于每个文档，它将所有非文档输入（例如用户的问题或其他与当前文档相关的信息）、当前文档和最新的中间答案传递给LLMChain，以获得新的答案。包含中间答案的提示词是这个类型的重要特征。

在遍历文档阶段，系统会遍历输入的所有文档。对于每个文档，一起作为提示词传递给 LLMChain 的内容有：一些上下文信息，例如用户的问题或其他与当前文档相关的信息。最新的中间答案。中间答案是系统在处理之前的文档时产生的。一开始，中间答案可能是空的，但随着系统处理更多的文档，中间答案会不断更新。当前文档，与Map reduce 链和重排链不同的是它不产生新文档，只不断更新的是提示词模板，迭代出更全面的答案。而且文档之间的影响是传递性的，上一个文档形成的答案会影响下一个文档的答案。

在更新答案阶段，系统将提示传递给LLMChain，然后将LLMChain生成的答案作为新的中间答案。这个过程会迭代进行，直到所有的文档都被处理。

Refine链最终实现效果是系统可以对包含多个文档的问题生成一个全面的答案，而且每个文档的处理结果都会影响后续文档的处理。这种处理方式可以提高文档搜索的质量，特别是在处理大量文档的情况下。

Refine链主要适用于处理大量文档的情况，特别是当这些文档不能全部放入模型的上下文中时。然而，这种处理方式可能会使用更多的计算资源，并且在处理某些复杂任务（如文档之间频繁地交叉引用，或者需要从许多文档中获取详细信息）时可能表现不佳。

通过使用Refine链，系统可以有效地处理大量文档的情况，从而提高文档搜索的质量。然而，这种处理方式可能需要更多的计算资源，并且可能在处理复杂任务时表现不佳。

### 5.3.3 MapReduce 链

本文将解析 MapReduce 链的数据流和工作中间环节，并说明其如何通过改变输入的组织和输出的生成方式来提高文档搜索的质量。

整体流程主要由两个部分组成：映射（Map）阶段和归约（Reduce）阶段。在映射阶段，系统对每个文档单独应用一个 LLMChain，并将链输出视为新的文档。在归约阶段，系统将所有新文档传递给一个单独的合并文档链，以获得单一的输出。如果需要，系统会首先压缩或合并映射的文档，以确保它们适合合并文档链。

在映射阶段，系统使用LLMChain，对每个输入的文档进行处理。处理的方式是，将当前文档作为输入传递给LLMChain，然后将LLMChain的输出视为新的文档。这样，每个文档都会被转化为一个新的文档，这个新文档包含了原始文档的处理结果。新文档是MapReduce 链的主要特征。

对于每个文档，作为提示词模板的一部分传递给 LLMChain 的内容是原始文档。比起“ Stuff ”类型多了预处理，也就说每个文档都产生了一个新的文档，这个新文档是运行 LLMChain 的结果。

每个原始文档都经过LLMChain 处理的结果写入一个新文档，这就是映射的过程。比如原文档有2000字，经过LLMChain 处理的结果是200字。200字的结果存储为一个新文档，但是跟2000字原文档存着映射关系。

在归约阶段，系统使用合并文档链将映射阶段得到的所有新文档合并成一个。如果新文档的总长度超过了合并文档链的容量，那么系统会使用一个压缩过程将新文档的数量减少到合适的数量。这个压缩过程会递归进行，直到新文档的总长度满足要求。

最终实现效果是系统可以对每个文档单独进行处理，然后将所有文档的处理结果合并在一起。这种处理方式可以提高文档搜索的质量，特别是在处理大量文档的情况下。

适用场景是需要处理大量文档的情况，特别是当这些文档不能全部放入模型的上下文中时。通过并行处理每个文档并合并处理结果，这种处理方式可以在有限的资源下处理大量的文档。然而，这种处理方式可能会使用更多的计算资源，并且可能在处理某些复杂任务（如文档之间频繁地交叉引用，或者需要从许多文档中获取详细信息）时可能表现不佳。

MapReduce 链，系统可以有效地处理大量文档的情况，从而提高文档搜索的质量。然而，这种处理方式可能需要更多的计算资源，并且可能在处理复杂任务时表现不佳。

### 5.3.4 重排链

本文将解析 重排链（Map Re-rank）类型的文档处理链，并说明其如何通过改变输入的组织和输出的生成方式来提高文档搜索的质量。

整体流程是对每个文档运行初始提示，这个提示不仅试图完成任务，还对其答案的确定程度给出评分。最后，得分最高的响应将被返回。

在映射和评分阶段，系统对每个文档运行初始提示。每个文档都会被独立地处理，处理的方式是，系统不仅试图完成任务，还对其答案的确定程度给出评分。这样，每个文档都会被转化为一个新的文档，这个新文档包含了原始文档的处理结果和评分。

对于每个文档，作为提示词的一部分传递给 LLMChain 的内容是原始文档, 但是提示词模板增加了评分规则。拿到 LLMChain 的答案后，存储为一个新文档，与原文档形成映射关系。

在重排阶段（Re-rank Stage）阶段，系统根据每个新文档的评分进行重排。具体来说，系统会选择得分最高的新文档，并将其作为最终的输出。只有这个类型有自动重排的机制，因为只有这个类型，对原始文档进行处理的时候，添加了评分规则的提示。

重排链的最终实现效果是系统可以对每个文档独立地进行处理和评分，然后选择得分最高的结果作为最终输出。这种处理方式可以提高文档搜索的质量，特别是在处理大量文档的情况下。

重排链的适用场景是处理大量文档的情况，特别是当需要从多个可能的答案中选择最优答案时。通过对每个文档的处理结果进行评分和重排，这种处理方式可以在有限的资源下找到最优的答案。然而，这种处理方式可能会使用更多的计算资源，并且可能在处理某些复杂任务（如文档之间频繁地交叉引用，或者需要从许多文档中获取详细信息）时可能表现不佳。

总的来说，通过使用重排链，系统可以有效地处理大量文档的情况，并从多个可能的答案中选择最优答案，从而提高文档搜索的质量。然而，这种处理方式可能需要更多的计算资源，并且可能在处理复杂任务时表现不佳。

## 5.4 揭秘链的复杂性

在前面的章节中，我们了解了基础链、工具链和基础合并文档链。这些链都是为了满足基础的业务场景而设计的。例如，如果我们只是希望简单地与大型语言模型进行交互，那么可以使用基础链，如LLMChain。工具链则是为了帮助我们完成应用程序中的特定任务，如API链就是专门用来解析API的。而文档处理链则承载了数据连接模块的LEDVR工作流，当我们通过检索获取了相关的文档后，需要考虑如何让这些文档能够正确地回答用户输入的问题。

然而，这些链都是为了满足比较简单的业务场景而设计的，使用这些链的步骤也并不复杂。而与此相比，Langchain的链模块却是最难以理解的。为什么会这样呢？

这主要是因为，链需要承担的责任越多，链的内部就会越复杂。如果说基础链和工具链只是把几个包装器包裹在链组件里，那么更复杂的链其实是“套娃”，即一个链套着另一个链。

以LLMChain作为基础链的代表，它之所以简单，是因为它把模型I/O的三个核心部分：模型包装器，提示词模板包装器，输出解析器都包裹在了LLMChain内部。而对于更复杂的链，比如A链是LLMChain，B链是一个文档合并的链，而B链又包含了A链，C链则可能包含了A链和B链。比如上一节我们了解了不同类型的合并文档链，可是好像并没有地方可以运用这些链，原因在于在实际业务中，我们用的最多的是QA问答链和摘要链，而这些内置的链组件包含了合并文档链，链包链，甚至包了三四个链，嵌套了几层链。

随着我们对链的使用需求变得越来越复杂，链的设计和组织也会变得越来越复杂。接下来的章节，我们将一起探索这些更复杂的链，了解它们的工作原理和使用方法。

我们先通过 BaseQAWithSourcesChain 的源码，来探秘这种“套娃”的设计。BaseQAWithSourcesChain 就是一个复杂的链，它的内部包含了多个其他的链。实际上，BaseQAWithSourcesChain 还仅仅是开始，如果要做QA问答链和摘要链，还要继承这个类，也就是继续“套娃”。

首先，我们有 combine\_documents\_chain 链 ，这是一个 BaseCombineDocumentsChain 链，它本身就可能是一个复杂的链，由多个子链组成。

然后，我们看到 from\_llm 方法中，首先创建了两个 LLMChain 链（即 A 链），接着创建了一个 StuffDocumentsChain 链 和一个 MapReduceDocumentsChain 链 （即 B 链）。 StuffDocumentsChain 链内部包含了一个 LLMChain 链， MapReduceDocumentsChain 链则包含了一个 LLMChain 链和一个 StuffDocumentsChain 链。就是我们提到的“套娃”设计。

最后，这些链被包裹在 BaseQAWithSourcesChain 链内部，形成了我们的 C 链。这样的设计使 BaseQAWithSourcesChain 链可以处理复杂的问题回答任务，同时还可以处理源文档。

class BaseQAWithSourcesChain(Chain, ABC):

    """Question answering with sources over documents."""

    combine\_documents\_chain: BaseCombineDocumentsChain

    """Chain to use to combine documents."""

    question\_key: str = "question"  #: :meta private:

    input\_docs\_key: str = "docs"  #: :meta private:

    answer\_key: str = "answer"  #: :meta private:

    sources\_answer\_key: str = "sources"  #: :meta private:

    return\_source\_documents: bool = False

    """Return the source documents."""

    @classmethod

    def from\_llm(

        cls,

        llm: BaseLanguageModel,

        document\_prompt: BasePromptTemplate = EXAMPLE\_PROMPT,

        question\_prompt: BasePromptTemplate = QUESTION\_PROMPT,

        combine\_prompt: BasePromptTemplate = COMBINE\_PROMPT,

        \*\*kwargs: Any,

    ) -> BaseQAWithSourcesChain:

        """Construct the chain from an LLM."""

        llm\_question\_chain = LLMChain(llm=llm, prompt=question\_prompt)

        llm\_combine\_chain = LLMChain(llm=llm, prompt=combine\_prompt)

        combine\_results\_chain = StuffDocumentsChain(

            llm\_chain=llm\_combine\_chain,

            document\_prompt=document\_prompt,

            document\_variable\_name="summaries",

        )

        combine\_document\_chain = MapReduceDocumentsChain(

            llm\_chain=llm\_question\_chain,

            combine\_document\_chain=combine\_results\_chain,

            document\_variable\_name="context",

        )

        return cls(

            combine\_documents\_chain=combine\_document\_chain,

            \*\*kwargs,

        )

BaseQAWithSourcesChain 的源码很好地说明了我们在前面讨论的观点：随着链需要承担的责任是处理复杂的问题回答任务，同时还要处理源文档，所以链的内部结构也会变得越来越复杂。因此，当我们需要处理更复杂的业务场景时，我们可能需要使用到更复杂的链。

当我们理解这种“套娃”设计，面对复杂的链，我们可以追溯到它的源码，看看内部包含了哪些基础链，哪些工具链，又哪些是处理合并文档的链。

这种设计的链组件实现的典型链是QA问答链和摘要链，一旦我们适应了这两种链的复杂，其他的链看起来都简单了。

### 5.4.1 LEDVR 工作流的终点

在本章伊始，我们提出了一个问题：LEDVR工作流的终点是什么？因为在数据连接模块的章节中，我们已经详细介绍了LEDVR工作流的各个阶段，包括文档加载、文档切割、嵌入模型包装器以及向量存储库的创建和使用。了解了整个LEDVR工作流如何处理文档后，示例代码也展示了如何把跟答案相关的文档内容检索出来。但我们没有解答检索到的相关文档内容怎么使用，这些数据是如何在Langchain 框架中流动的？LEDVR工作流的终点是什么？现在，我们来解决整个问题，LEDVR工作流的终点是“上链”。

我们仍然以LEDVR工作流的示例代码开始，探究如何把LEDVR工作流的“胜利成果”加入到链组件中“上链”，运行第一次会话检索链 ConversationalRetrievalChain。也就是说，导入ConversationalRetrievalChain 之前的代码都是与LEDVR工作流一样的。

首先，我们需要从网络加载文档。这可以通过使用WebBaseLoader来完成：

from langchain.document\_loaders import WebBaseLoader

openai\_api\_key="填入你的密钥"

loader = WebBaseLoader("http://developers.mini1.cn/wiki/luawh.html")

data = loader.load()

你也可以选择其他的加载器和其他的文档资源。接下来，我们需要创建一个嵌入模型实例的包装器，这可以使用OpenAIEmbeddings完成：

from langchain.embeddings.openai import OpenAIEmbeddings

embedding = OpenAIEmbeddings(openai\_api\_key=openai\_api\_key)

然后，我们需要将文档切割成块，这可以通过使用RecursiveCharacterTextSplitter来完成：

from langchain.text\_splitter import RecursiveCharacterTextSplitter

text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=500, chunk\_overlap=0)

splits = text\_splitter.split\_documents(data)

接着，我们需要创建一个向量存储库，这里我们选择使用FAISS：

from langchain.vectorstores import FAISS

vectordb = FAISS.from\_documents(documents=splits,embedding=embedding)

现在我们有了一个向量存储库，我们可以用它来创建一个检索器 retriever，LEDVR 工作流就结束了。整个检索器便是LEDVR工作流的“胜利果实”：

retriever = vectordb.as\_retriever()

从这里开始，我们可以将检索器加入到链中。首先，我们需要创建一个LLM 模型包装器，并通过ConversationalRetrievalChain.from\_llm方法创建一个ConversationalRetrievalChain链组件的实例：

from langchain.llms import OpenAI

from langchain.chains import ConversationalRetrievalChain

llm = OpenAI(openai\_api\_key=openai\_api\_key)

qa = ConversationalRetrievalChain.from\_llm(llm, retriever)

至此，我们已经完成了一个相当复杂的链 ConversationalRetrievalChain。这个链是我们目前了解的最复杂的链组件，它承担了会话和检索文档的两个重要职责。通过这个链，我们可以用如下的方式来查询问题：

query = "LUA的宿主语言是什么？"

result = qa({"question": query})

result["answer"]

通过这种方式，我们就可以获取到问题的答案。这个链的复杂性和功能性都非常高，可以帮助我们有效地处理各种复杂的信息检索任务。

LEDVR工作流的最后我们获得了一个检索器，把检索器放入链组件中，运行链组件，所以说LEDVR工作流的终点是“上链”。与底层的模型平台如何交互，提示词如何写，怎么让回答依据的是我们检索的文档回答，都隐藏在链组件之下。而开发者只需知道有什么类型的链可以使用，给这些链组件传递的参数是什么，参数的类型是什么，如何实例化一个链组件这些代码的基本配置，我们便可以完成文章摘要，文档问答等复杂的业务场景。

我们可以通过查看常用的QA问答，摘要链，会话链的源码，迅速学习有什么类型的链可以使用，给这些链组件传递的参数是什么，参数的类型是什么，如何实例化一个链组件这些代码的基本配置。

### 5.4.3 从源码读懂链

通常情况下，链组件的创建都会通过实现了 from\_llm 方法的基本类来完成。这个方法需要一些参数，比如我们在前面的例子中看到的 llm 和 retriever 。那么，如何知道有哪些类型的链可以使用，需要给这些链组件传递什么参数，参数的类型是什么，以及如何实例化一个链组件呢？这就需要我们去查阅和学习相关的源代码了。

在LangChain中，我们可以找到很多种类型的链，比如QA问答链，摘要链，会话链等。例如，我们常用的QA问答链，如 QAWithSourcesChain，都是基于 BaseQAWithSourcesChain 类实现的：

以下是 `BaseQAWithSourcesChain` 类中定义的参数描述：

- combine\_documents\_chain：这是一个 BaseCombineDocumentsChain 类型的对象，用于将多个文档组合在一起。这是一个必需的参数。

- question\_key：这是一个字符串，用于从输入中提取问题。默认值是 "question"。

- input\_docs\_key：这是一个字符串，用于从输入中提取文档。默认值是 "docs"。

- answer\_key：这是一个字符串，用于在输出中指定答案的键。默认值是 "answer"。

- sources\_answer\_key：这是一个字符串，用于在输出中指定来源的键。默认值是 "sources"。

- return\_source\_documents：这是一个布尔值，用于决定是否在输出中返回源文档。默认值是 False，表示默认不返回源文档。

class BaseQAWithSourcesChain(Chain, ABC):

"""Question answering with sources over documents."""

combine\_documents\_chain: BaseCombineDocumentsChain

"""Chain to use to combine documents."""

question\_key: str = "question" #: :meta private:

input\_docs\_key: str = "docs" #: :meta private:

answer\_key: str = "answer" #: :meta private:

sources\_answer\_key: str = "sources" #: :meta private:

return\_source\_documents: bool = False

"""Return the source documents."""

会话链主要实现的是 ConversationalRetrievalChain，它们都是基于 BaseConversationalRetrievalChain 类实现的。以下是 ConversationalRetrievalChain 类中定义的参数描述：

- combine\_docs\_chain：这是一个 BaseCombineDocumentsChain 类型的对象，用于将多个文档组合在一起。这是一个必需的参数。

- question\_generator：这是一个 LLMChain 类型的对象，用于生成问题。这也是一个必需的参数。

- output\_key：这是一个字符串，用于在输出中指定答案的键。默认值是 "answer"。

- return\_source\_documents：这是一个布尔值，用于决定是否在输出中返回源文档。默认值是 False，表示默认不返回源文档。

- return\_generated\_question：这是一个布尔值，用于决定是否在输出中返回生成的问题。默认值是 False，表示默认不返回生成的问题。

- get\_chat\_history：这是一个可调用对象，用于获取聊天历史。默认值是 None，表示默认不获取聊天历史。

class BaseConversationalRetrievalChain(Chain):

"""Chain for chatting with an index."""

combine\_docs\_chain: BaseCombineDocumentsChain

question\_generator: LLMChain

output\_key: str = "answer"

return\_source\_documents: bool = False

return\_generated\_question: bool = False

get\_chat\_history: Optional[Callable[[CHAT\_TURN\_TYPE], str]] = None

"""Return the source documents."""

为了方便用户使用，LangChain还提供了一些预定义的函数，如 load\_qa\_chain 和 load\_summarize\_chain，可以方便地实例化QA问答链组件和摘要链组件。参数描述如下：

llm：这是一个 BaseLanguageModel 类型的对象，用于在链中使用。这是一个必需的参数。

chain\_type：这是一个字符串，用于指定要使用的合并文档链的类型。可选的值包括 "stuff"、"map\_reduce"、"map\_rerank" 和 "refine"。默认值是 "stuff"。

verbose：这是一个布尔值，用于决定链是否应以详细模式运行。注意，这适用于组成最终链的所有链。这是一个可选参数。

callback\_manager：这是一个 BaseCallbackManager 类型的对象，用于链的回调管理。这是一个可选参数。

from langchain.chains.question\_answering import load\_qa\_chain

from langchain.chains. summarize import load\_summarize\_chain

def load\_qa\_chain(

llm: BaseLanguageModel,

chain\_type: str = "stuff",

verbose: Optional[bool] = None,

callback\_manager: Optional[BaseCallbackManager] = None,

\*\*kwargs: Any,

) -> BaseCombineDocumentsChain:

"""Load question answering chain."""

...

def load\_summarize\_chain(

llm: BaseLanguageModel,

chain\_type: str = "stuff",

verbose: Optional[bool] = None,

\*\*kwargs: Any,

) -> BaseCombineDocumentsChain:

"""Load summarizing chain."""

通过这些源代码，我们可以看到LangChain的灵活性和强大功能。无论是简单的信息检索链组件，还是复杂的对话管理链组件，都可以通过构建和配置适当的链组件来实现。因此，理解和掌握链的概念，以及如何使用链，对于我们使用LangChain进行各种链组件来说，都是非常重要的。

## 5.5 本章小结

在这一章节中，我们主要讨论了Langchain的链的定义及其使用方法。

首先，我们定义了什么是链，并将其分类。链是Langchain中的重要组成部分，有多种不同的类型和用途。我们还介绍了工具链的概念，并讨论了如何使用各种类型的链。

我们讨论了基础链，包括LLM链和顺序链。这些基础链提供了Langchain的基本功能，对理解Langchain的整体架构至关重要。

接着，我们介绍了合并文档链。这部分涵盖了Stuff链，Refine链，MapReduce链，以及重排链。这些链的功能主要是合并、整理和重新排序文档，使其更容易被处理和分析。

我们从 BaseQAWithSourcesChain 源码探讨了一个复杂的链内部承担了多少责任，完成了多少任务，给我们创建适合自己的链，有了很好的参考价值。

最后我们将链组件同之前章节介绍的 LEDVR 工作流打通，原来LEDVR 工作流的终点是“上链 ”。为了对常用链尽快记忆和使用，我们又查看了源码，通过源码学习了参数和参数的数据类型以及实例化链组件的各种方法。

总的来说，从简单的基础链和工具链开始，到复杂链组件的内部构造以及文档处理链的不同算法类型，到不同的文档处理链类型以及如何使用复杂的会话链组件，本章都有涉及和详细解释，对链组件有了清晰的认知。

# 6. 记忆

## 6.1 记忆概述

想一想，我们为什么需要记忆 ？

大语言模型本质上是无记忆的。当我们与其交互时，它仅根据提供的提示生成相应的输出，而无法存储或记住过去的交互内容。这一特性，使得大语言模型在实现聊天机器人或聊天代理时，难以满足人们的期望。

人们期待聊天机器人具有人的品质和回应能力。当他们意识到机器人只是进行一次性的调用和响应，而无法记住以往的交流内容时，可能会感到沮丧。在现实的聊天环境中，人们的对话中充满了缩写和含蓄表达，他们会引用过去的对话内容，并期待对方能够理解和回应。例如，如果聊天历史中只在一开始提到了某人的名字，随后仅用代词指代，那么他们就期望聊天机器人能够理解和记住这个指代关系。

我们对聊天机器人的期待并不仅仅是其具备基础的回答功能，我们希望机器人能够在整个对话过程中，理解我们的话语，记住我们的交流内容，甚至理解我们的情绪和需求。为了实现这个目标，我们需要赋予大语言模型一种“记忆”能力。

记忆是一种基础的人类特性，是我们理解和交流世界的基础。因此，我们将这种能力赋予机器人，使其具备人类般的记忆功能。这种记忆功能被我们封装在记忆模块和记忆组件中，它们不仅能够存储和管理信息，还能够根据需要提取和使用这些信息。当我们谈论内存时，我们仅仅是在谈论一个电脑的内存条。但如果我们还谈论一种理解和交流的能力，一种建立和维持关系的能力，我们更愿意使用“记忆”。我们希望机器人能够像人一样理解我们的话语，记住我们的交流内容，甚至理解我们的情绪和需求。

所以，我们将 Memory 模块称之为记忆模块，各种封装的类和示例，我们称为记忆组件。因为我们不会说自己有内存，我们说自己有记忆。

最后，我们要强调为什么在实际的大语言模型应用开发中，需要记忆模块的功能。简单来说，同样作为提示词模板的外部数据，“记忆”的功能形成的外部数据，与检索的外部文档内容起到一样的作用。同样地，可以确保大语言模型在处理信息时始终能够获取到最新、最准确的上下文数据；通过提供聊天信息，我们可以让大模型的输出更有据可循，多了一份“证据”；这也是一种低成本应用大语言模型的策略，不需要做涉及参数训练等额外工作。

大语言模型应用落地遇到的问题

记忆组件的工作原理相当直观：它接收聊天消息，将这些消息作为提示词模板的一部分，然后传递给模型I/O模块的模型包装器。随后，模型将根据这些提示词的约束，生成相应的输出。我们在模型I/O模块的提示词模板中有提到，将提示词模板的数据分为内部和外部数据，聊天信息是发生在应用程序本身的，属于内部数据。在数据连接模块，通过 LEDVR 工作流提取的相关性文档信息，属于外部数据。

你可能会注意到，当聊天信息和文档信息，所有这些信息都会被存储在“提示词模板”中。然而，这种方法在某些情况下会出现问题，那就是当我们进行长期的对话时，大语言模型可能无法容纳所有的信息，每个大语言模型都有 Max Tokens 的限制。即使是强大的 OpenAI 平台，它的最新模型 GPT-4-32k-0613 在一次处理中,Max Tokens 是 32768。但是聊天记录这在许多应用中都是一个巨大的信息量，它可以包含长篇的文章、复杂的对话内容，或者大量的其他类型的数据。比如一个翻译语言的应用，这些聊天记录会变得很大很大。然而，尽管如此，由于 Max Tokens 的限制仍然无法保存完整的对话内容，尤其是在持续的、涉及大量交互的对话场景中。

这个问题引发了一系列的挑战：我们如何有效地管理这个Max Tokens，适应各个模型平台的 API 要求？如何决定在每个会话中使用哪些聊天记录，以及如何存储和检索过去的对话内容？这就是我们需要记忆组件的地方。比如对话总结记忆组件就提供了一个解决方案。这种记忆模式会在对话进行的过程中实时总结对话内容，并将当前的摘要存储在应用程序运行时内存中。这种记忆方式对于长期的对话来说尤其有用，因为如果直接在提示中保留过去的消息历史。

## 6.1.1 记忆组件的定义

记忆组件是什么？

记忆组件，实际上是一个聊天备忘录，像苹果手机备忘录程序一样，我们可以用它记录我们与大语言模型的聊天对话消息。那么，备忘录的作用是什么呢？试想在一个会议场合，你有一位秘书在边上为你做备忘录。当你发言时，你可能会忘记先前的发言者都说过什么，这时你可以让秘书展示备忘录，通过查阅这些信息，你能够清楚地整理自己的发言，从而赢得全场的赞许。而当你发言结束后，秘书又要做什么呢？他需要把你刚刚的精彩发言记录下来。

这就是记忆组件的两大功能：读取和写入。因此，记忆组件的基本操作包括读取和写入。在链组件的每次运行中，都会有两次与记忆组件的交互：在执行核心逻辑之前读取记忆组件以丰富用户输入的上下文语境，在执行核心逻辑后将当前运行的输入和输出写入记忆组件，以便在未来的运行中引用。

记忆组件的设计，旨在解决两个核心问题：一是如何写入，也就是存储状态，二是如何读取，即查询状态。状态存储一般通过程序运行的内存存储历史聊天记录实现。状态查询则依赖于在聊天记录上构建的数据结构和算法，它们为我们提供了对历史消息的高效检索和解析。类似于一个聊天备忘录，记忆组件既可以帮我们记录下聊天的每一条信息，也可以依据我们的需求，搜索出相关的历史聊天记录。

最难理解的是如何在聊天记录上构建的数据结构和算法？简单来说，数据结构和算法就是用来整理和检索聊天记录的方法。记忆组件会将处理过的聊天信息数据注入到提示词模板中，最终通过模型平台的API接口获取大语言模型的响应，从而使得这个响应更加准确。我们需要决定哪些聊天记录需要保留，哪些聊天记录需要进一步处理，这个过程就是在聊天记录上构建数据结构和算法。

我们主要采取以下两种方法整理和检索聊天记录。一种是全部放入提示词模板中，这是一种比较简单的方法，将聊天窗口上下文直接放在提示词中，作为外部数据注入到提示词模板，再提供给大语言模型。这种方法简洁易行，但是因为结合方式较为粗糙，所以可能无法达到精准控制的目的。而且由于模型平台的 Max Tokens 限制，这种方式要考虑的是如何截取聊天记录，如何压缩聊天记录，适应模型平台底层的API要求。

第二种方法是压缩后再放入提示词模板中。类似计算机的文件和图像压缩技术。这种方法的思路是多一次，甚至多几次先把聊天记录信息提交给大语言模型做总结，或者从数据库中提取相关的聊天记录，提取实体知识等操作，然后将这些文本拼接在提示词中，再提供给大语言模型，进行下一轮对话。

这个过程中，我们可以使用类似于图像压缩算法的方式对聊天记录进行压缩。比如，我们可以参考 JPEG 图像压缩算法的原理，将原始的聊天记录（类比为图像的原始数据）进行压缩。JPEG 图像压缩算法在压缩过程中会舍弃一些人眼难以察觉的高频细节信息，从而达到压缩的效果，同时大部分重要信息会被保留。这样，我们就可以将压缩后的聊天记录（类比为压缩后的图像）放入提示词模板中，用于生成下一轮对话。虽然这种压缩方式可能会丢失一些细节信息，但是由于这些信息对于聊天记录的总体含义影响较小，因此可以接受。

通过这种方式，我们不仅可以在保证聊天记录的总体含义的同时，减小提示词的长度，而且还可以提高大语言模型的运行效率和响应速度。

## 6.1.2 记忆、链和Agent的关系

我们先探讨记忆组件和链组件的关系。LangChain 的记忆模块提供了各种类型的记忆组件，这些记忆组件可以独立使用，也可以无缝地集成到链组件中。换句话说，记忆组件可以作为链组件的一个部分，是一个内部组件，用于保存状态和历史信息，以便链组件在处理输入和生成输出时使用。所以记忆组件和链组件的关系是，链组件是与大语言模型交互的主体，而记忆组件则是在这个过程中提供支持的角色。通过引入记忆组件，链组件调用大语言模型得到能“回顾”过去的对话，理解并确定当前谈论的主题和对象。这使得聊天机器人更接近人类的交流方式，从而更好地满足用户的期望。记忆组件最常见的使用场景是智能助理的聊天界面，在这里，它扮演着不可或缺的角色，提升了大语言模型的对话能力和用户体验。

我们更进一步探讨记忆组件和Agent的关系。在Agent模块中，我们将记忆组件放入智能体Agent中。这使得Agent不仅能够处理和响应我们的请求，而且还能够记住我们的交流内容。这就是我们所说的“让Agent拥有记忆”。 Agent是一个更高级别的组件，它可能包含一个或多个链组件，以及与这些链组件交互的记忆组件。Agent通常代表一个完整的对话系统或应用，负责管理和协调其内部组件（包括链和记忆组件）以响应用户输入。

具体来说，智能体Agent在处理用户输入时，可能会使用其内部的链组件来执行各种逻辑，同时使用内存来存储和检索过去的交互信息。例如，Agent可能会使用内存中的信息来增强用户输入，或者将链组件的输出写入内存以供未来的交互中引用。因此，我们可以说记忆组件是智能体 Agent 的一个重要部分，帮助 Agent 维护状态和历史信息，使其能够处理复杂的对话和任务。在构建智能体 Agent 时，通常需要考虑如何选择和配置内存，以满足特定的应用需求和性能目标。

通过赋予Agent记忆，我们让机器人变得更加人性化，更加接近我们的交流方式。这不仅能够提高机器人的交流效率和质量，还能够增强我们对机器人的信任和满意度。因此，记忆不仅仅是一种技术，更是一种理念，一种让机器人更接近人类的方式。将记忆组件放入智能体Agent中，“让Agent拥有记忆”，也是我们在本章中关注的地方。

这就是记忆组件在 LangChain 中的运作方式：集成到链组件中，集成到智能体Agent中。

## 6.1.3 第一个记忆组件

我们可以用代码演示如何实例化一个记忆组件，以及如何使用记忆组件的保存和加载方法来管理聊天记录。这种使用记忆组件的方式对于所有的记忆组件都是通用的，只是在具体的实现细节上可能会有所不同。

首先，我们需要引入 ConversationTokenBufferMemory 类，并创建一个 OpenAI 类的实例作为其参数。 ConversationTokenBufferMemory 是一个记忆组件，它将最近的交互信息保存在内存中，并使用令牌长度而不是交互次数来决定何时清除交互信息。

from langchain.memory import ConversationTokenBufferMemory

from langchain.llms import OpenAI

openai\_api\_key = "填入你的OpenAI密钥"

llm = OpenAI(openai\_api\_key=openai\_api\_key)

memory =

ConversationTokenBufferMemory(llm=llm,max\_token\_limit=1000,memory\_key="session\_chat")

在上面的代码中，我们首先创建了一个 OpenAI 实例 llm 类型的模型包装器。

然后用这个实例和一个max\_token\_limit参数来创建 ConversationTokenBufferMemory 的实例 memory 。max\_token\_limit 参数用于设置记忆组件可以保存的 Max Token 数量, 数字设置过于小的话，可能最后打印的对象是空字符串，可以手动设置 10，再设置 100 查看效果。 memory\_key 参数用于设置记忆组件存储对象的键名，默认是 history ，这里我们设置为 session\_chat 。

接下来，我们可以使用 save\_context 方法来将聊天记录保存到记忆组件中。每次调用 save\_context ，都会将一次交互（包括用户的输入和大语言模型的输出）添加到记忆组件的缓冲区中。

memory.save\_context({"input": "你好！我是李特丽，这是人类的第一个消息"}, {"output": "你好！我是AI助理的第一个消息"})

memory.save\_context({"input": "今天心情怎么样"}, {"output": "我很开心认识你"})

在上面的代码中，我们先后保存了两次交互记录。第一次交互中，用户的输入是 "你好！我是李特丽，这是人类的第一个消息"；第二次交互中，用户的输入是 "今天心情怎么样"，大语言模型的输出是 "我很开心认识你"。

memory.load\_memory\_variables({})

最后，我们可以使用 load\_memory\_variables 方法来加载记忆组件中的聊天记录。这个方法会返回一个字典，其中包含了记忆组件当前保存的所有聊天记录。

{'session\_chat': 'Human: 你好！我是李特丽，这是人类的第一个消息\nAI: 你好！我是AI助理的第一个消息\nHuman: 今天心情怎么样\nAI: 我很开心认识你'}

以上就是如何使用 ConversationTokenBufferMemory 记忆组件的示例。

## 6.1.4 内置记忆组件

Langchain 的记忆模块( langchain.memory 包)提供了多种内置的记忆组件类，这些类的使用方法和 ConversationTokenBufferMemory 基本一致，都提供了保存（ save\_context ）和加载（ load\_memory\_variables ）聊天记录的方法，只是在具体的实现和应用场景上有所不同。

例如， ConversationBufferMemory 、 ConversationBufferWindowMemory 、 ConversationTokenBufferMemory 等类都是用于保存和加载聊天记录的记忆组件，但它们在保存聊天记录时所使用的数据结构和算法有所不同。 ConversationBufferMemory 使用一个缓冲区来保存最近的聊天记录，而 ConversationTokenBufferMemory 则使用 Max Tokens 长度来决定何时清除聊天记录。

此外，有些记忆组件类还提供了额外的功能，如 ConversationEntityMemory 和 ConversationKGMemory 类可以用于保存和查询实体信息， CombinedMemory 类可以用于组合多个记忆组件，而 ReadOnlySharedMemory 类则提供了一种只读的共享记忆模式。

对于需要将聊天记录持久化保存的应用场景，Langchain 的记忆模块还提供了多种与数据库集成的记忆组件类，如 SQLChatMessageHistory 、 MongoDBChatMessageHistory 、 DynamoDBChatMessageHistory 等。这些类在保存和加载聊天记录时，会将聊天记录保存在对应的数据库中。

在选择记忆组件时，你需要根据自己的应用需求来选择合适的类。例如，如果你的应用需要处理大量的聊天记录，或者需要在多个会话中共享聊天记录，那么你可能需要选择一个与数据库集成的记忆组件。而如果你的应用主要处理实体信息，那么你可能需要选择 ConversationEntityMemory 或 ConversationKGMemory 这样的类。无论你选择哪种记忆组件，都需要理解其工作原理和使用方法，以便正确地使用它来管理你的聊天记录。

## 6.1.5 自定义记忆组件

我们在内置记忆组件一节中，了解了 Langchain 的内置记忆组件，尽管这些组件能够满足许多通用的需求，但每个具体的应用场景都有其独特的要求和复杂性。例如，某些应用可能需要特定的数据结构来优化查询性能，或者需要特别的存储方式来满足数据安全或隐私要求。在这些情况下，预设的记忆组件可能无法完全满足需求。通过允许开发者添加自定义的记忆类型，框架能够提供更高的灵活性和扩展性，使得开发者能够根据自己的需求定制和优化记忆组件。这对于高级的使用场景，如大规模的生产环境或特定的业务需求，尤为重要。

对于初学者来说，创建自定义组件实际上是理解和掌握框架的有效路径。这是因为在构建自定义组件的过程中，你将被迫深入理解框架的设计逻辑、工作方式以及内置组件的实现机制。这将极大地加深你对框架整体架构和工作原理的理解。

内置组件往往是框架开发者根据通用需求预先设计和封装好的，使用者可以直接拿来使用，无需关心其内部实现细节。这无疑大大降低了使用门槛，提高了开发效率。然而，这也意味着使用者很可能对这些组件的内部构造和工作原理一无所知。

而当你试图创建自定义组件时，你需要深入理解和分析你的特定需求，然后再在此基础上设计和实现符合需求的组件。这个过程将迫使你深入了解内置组件的构造和工作原理，从而更好地理解框架的设计逻辑和工作方式。因此，尽管创建自定义组件的过程可能会有些复杂和困难，但它无疑是深入理解和掌握框架的有效方式。

对于 LangChain 框架的设计者来说，或者站在在 LangChain 立场来说，这样的设计都会提供自定义记忆组件的示例代码和测试代码示例，目的是确保开发者能够更容易地理解和使用框架，从而降低开发门槛。这对于提高框架的用户友好性和推动其广泛应用至关重要。另外一个考虑是框架的设计者深知不可能预设所有可能的需求。在实际的应用场景中，每个项目或业务都有其特定的需求，各行各业需求不一样，所以 LangChain 先满足的是通用的场景，比如记忆模块的内置组件，多数都是会话记忆组件，而且会将会话记忆组件分很多种类，满足这个场景下的各种需求 。但是并不会满足任何的场景，比如金融、医疗这些细分行业。这样的设计理念，也是 LangChain 框架在满足通用需求的同时，不断推动创新和适应复杂应用场景的关键策略。

在本节中，我们将向 ConversationChain 添加一个自定义的记忆类型。为了添加自定义的记忆类，我们需要引入基础记忆类并对其进行子类化。请注意，这种实现方式相当简单且脆弱，可能在生产环境中并不实用。它的目的是展示你可以添加自定义记忆实现。

from langchain import OpenAI, ConversationChain

from langchain.schema import BaseMemory

from pydantic import BaseModel

from typing import List, Dict, Any

在这个例子中，我们将编写一个自定义记忆类，它使用 spacy 来提取实体，并将有关它们的信息保存在一个简单的哈希表中。然后，在对话中，我们将观察输入文本，提取实体，并将关于它们的任何信息放入上下文中。

!pip install spacy

!python -m spacy download en\_core\_web\_lg

import spacy

nlp = spacy.load("en\_core\_web\_lg")

class SpacyEntityMemory(BaseMemory, BaseModel):

"""Memory class for storing information about entities."""

# Define dictionary to store information about entities.

entities: dict = {}

# Define key to pass information about entities into prompt.

memory\_key: str = "entities"

def clear(self):

self.entities = {}

@property

def memory\_variables(self) -> List[str]:

"""Define the variables we are providing to the prompt."""

return [self.memory\_key]

def load\_memory\_variables(self, inputs: Dict[str, Any]) -> Dict[str, str]:

"""Load the memory variables, in this case the entity key."""

# Get the input text and run through spacy

doc = nlp(inputs[list(inputs.keys())[0]])

# Extract known information about entities, if they exist.

entities = [

self.entities[str(ent)] for ent in doc.ents if str(ent) in self.entities

]

# Return combined information about entities to put into context.

return {self.memory\_key: "\n".join(entities)}

def save\_context(self, inputs: Dict[str, Any], outputs: Dict[str, str]) -> None:

"""Save context from this conversation to buffer."""

# Get the input text and run through spacy

text = inputs[list(inputs.keys())[0]]

doc = nlp(text)

# For each entity that was mentioned, save this information to the dictionary.

for ent in doc.ents:

ent\_str = str(ent)

if ent\_str in self.entities:

self.entities[ent\_str] += f"\n{text}"

else:

self.entities[ent\_str] = text

我们现在定义一个提示词模板，它接受关于实体的信息以及用户输入的信息。

from langchain.prompts.prompt import PromptTemplate

template = """The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context. If the AI does not know the answer to a question, it truthfully says it does not know. You are provided with information about entities the Human mentions, if relevant.

Relevant entity information:

{entities}

Conversation:

Human: {input}

AI:"""

prompt=PromptTemplate(input\_variables=["entities","input"], template=template)

然后我们把记忆组件和链组件组合在一起，在 ConversationChain 链上运行，与大语言模型交互。

llm = OpenAI(temperature=0)

conversation = ConversationChain(

llm=llm, prompt=prompt, verbose=True, memory=SpacyEntityMemory()

)

在第一个例子中，由于没有关于 Harrison 的先验知识，"Relevant entity information"（相关实体信息）部分是空的。

conversation.predict(input="Harrison likes machine learning")

> Entering new ConversationChain chain...

Prompt after formatting:

The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context. If the AI does not know the answer to a question, it truthfully says it does not know. You are provided with information about entities the Human mentions, if relevant.

Relevant entity information:

Conversation:

Human: Harrison likes machine learning

AI:

> Finished ConversationChain chain.

" That's great to hear! Machine learning is a fascinating field of study. It involves using algorithms to analyze data and make predictions. Have you ever studied machine learning, Harrison?"

现在在第二个例子中，我们可以看到它抽取了关于 Harrison 的信息。

conversation.predict(

input="What do you think Harrison's favorite subject in college was?"

)

> Entering new ConversationChain chain...

Prompt after formatting:

The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context. If the AI does not know the answer to a question, it truthfully says it does not know. You are provided with information about entities the Human mentions, if relevant.

Relevant entity information:

Harrison likes machine learning

Conversation:

Human: What do you think Harrison's favorite subject in college was?

AI:

> Finished ConversationChain chain.

' From what I know about Harrison, I believe his favorite subject in college was machine learning. He has expressed a strong interest in the subject and has mentioned it often.'

请再次注意，这种实现方式相当简单且脆弱，可能在生产环境中并不实用。它的目的是为了展示你可以添加自定义的记忆组件。

## 6.2 记忆增强检索能力的实践

本节我们将深入探讨如何通过集成记忆组件，增强我们的数据连接模块LEDVR工作流中的数据检索能力，从而提升QA问答应用的回答质量。我们将以《链》章节中的代码案例为基础，介绍如何在已有的检索器基础上，添加记忆组件，使ConversationalRetrievalChain链组件具有“记忆”能力。

在此基础上，我们的案例应用将得以进一步提升。以LUA语言开发者在线文档问答为例，通过增强后的链组件，我们的程序不仅能在 http://developers.mini1.cn/wiki/luawh.html 中找到相关的文档内容，准确回答如“LUA宿主语言是什么”等问题，更能够记住用户在对话中的相关信息，如用户的名字，从而对更多复杂的问题提供更准确的答复。例如，当用户在问完几轮关于LUA语言编程的问题之后，如果他们再次提到他们的名字或引用他们之前提到的信息，增强后的ConversationalRetrievalChain链组件能够记住并理解这些信息，提供更加精确和个性化的回答。

希望读者能够理解并掌握如何将记忆组件与链组件集成，从而实现数据检索能力的增强，提升QA问答应用的回答质量。同时，我们也将向读者展示如何通过增加记忆组件，使我们的程序更具人性化，能够更好地满足用户的需求。使用加载器开始，到创建向量存储库实例都与《链》章节一样。如果你已经很熟悉这段代码，可以直接跳过这部分。

## 6.2.1 LEDVR 获取外部数据

首先，我们需要从网络加载文档。这可以通过使用WebBaseLoader来完成：

from langchain.document\_loaders import WebBaseLoader

openai\_api\_key="填入你的密钥"

loader = WebBaseLoader("http://developers.mini1.cn/wiki/luawh.html")

data = loader.load()

你也可以选择其他的加载器和其他的文档资源。接下来，我们需要创建一个嵌入模型实例的包装器，这可以使用OpenAIEmbeddings完成：

from langchain.embeddings.openai import OpenAIEmbeddings

embedding = OpenAIEmbeddings(openai\_api\_key=openai\_api\_key)

然后，我们需要将文档切割成块，这可以通过使用RecursiveCharacterTextSplitter来完成：

from langchain.text\_splitter import RecursiveCharacterTextSplitter

text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=500, chunk\_overlap=0)

splits = text\_splitter.split\_documents(data)

接着，我们需要创建一个向量存储库，这里我们选择使用FAISS，现在我们有了一个向量存储库：

from langchain.vectorstores import FAISS

vectordb = FAISS.from\_documents(documents=splits,embedding=embedding)

## 6.2.2 加入记忆组件

我们首先导入 ConversationBufferMemory 类，这是最常见的记忆组件，其工作原理非常简单：仅将所有聊天记录保存起来，而不使用任何算法进行截取或提炼压缩。在提示词模板中，我们可以看到所有的聊天记录。

from langchain.memory import ConversationBufferMemory

memory= = ConversationBufferMemory(memory\_key="chat\_history",return\_messages=True)

初始化记忆组件后，我们可以看到其内部的存储情况。由于我们刚刚进行了初始化，所以存储的聊天记录为空。然后，我们通过 add\_user\_message 方法添加一个人类的消息，向程序介绍我的名字。此时，我们再次查看记忆组件，就可以看到添加了一条 HumanMessage 消息。我们看看初始化后，记忆组件里保存着什么记录。因为我们刚刚初始化，所以是[]。

# 打印 memory.chat\_memory.messages

[]

我们使用 add\_user\_message 添加一个人类的消息，向程序介绍我的名字。

memory.chat\_memory.add\_user\_message("我是李特丽")

第一次打印 memory.chat\_memory.messages 时为[], 当我推送一条自我介绍后，我们可以看到添加了一条 HumanMessage 消息。

[HumanMessage(content='我是李特丽', additional\_kwargs={}, example=False)]

使用 memory 组件的 load\_memory\_variables 方法可以看保存在程序运行内存中的 memory 对象，主键是 chat\_history，正如我们初始化 ConversationBufferMemory 设置的那样：memory\_key="chat\_history" 。

# 打印memory.load\_memory\_variables({})

{'chat\_history': [HumanMessage(content='我是李特丽', additional\_kwargs={}, example=False)]}

我们采用最直接的方式来演示记忆组件是如何与链组件共同工作的。从提示词模板出发，逐步增加组件的方式，展示这个工作机制。这一次我们使用的链组件是 load\_qa\_chain。这个链组件的优势在于专门用于QA问答，对于合并文档链组件不必掌握这些链的使用方法，只要在这个链上指定类型。导入 load\_qa\_chain 、ChatOpenAI 和 PromptTemplate。提示词模板用于我们初始化 load\_qa\_chain 的时候，个性化配置提示词。

from langchain.chat\_models import ChatOpenAI

from langchain.chains.question\_answering import load\_qa\_chain

from langchain.prompts import PromptTemplate

# 这里需注意使用 Chat Model 类型的模型包装器，并且使用先进的模型型号。

llm = ChatOpenAI(openai\_api\_key=openai\_api\_key,temperature=0, model="gpt-3.5-turbo-0613")

使用向量存储库实例 的 similarity\_search 方法，测试是否可以检索到与问题相关的文档了。我们可以打印 len(docs)， 看看这个问题搜索到了几个文档片段。检索到的相关文档内容，我们要输入到提示模板包装器中，这一步先测试向量存储库是否正常工作。

query = "LUA 的宿主语言是什么？"

docs = vectordb.similarity\_search(query)

docs

打印 len(docs) 的长度是4，有4个相关文档片段被检索到。

4

创建提示词是最重要的环节，在创建的过程中我们可以理解为什么加入记忆后，因为有了“聊天备忘录” 记录的内容，让链组件有了 “记忆”。使用提示模板包装器，我们自定义一个提示词模板字符串。

提示词内容分为四部分：一是对模型的指导词：“请你回答问题的时候，请依据文档内容和聊天记录回答，如果在其中找不到相关信息或者答案，请回答不知道。” ； 二是用问题检索到的相关文档内容：“文档内容是：{context}” ；三是记忆组件输出的记忆内容：“聊天记录是：{chat\_history}” ；四是用户的输入： “Human: {human\_input}”。

template = """你是说中文的chatbot.

请你回答问题的时候，请依据文档内容和聊天记录回答，如果在其中找不到相关信息或者答案，请回答不知道。

文档内容是：{context}

聊天记录是：{chat\_history}

Human: {human\_input}

Chatbot:"""

prompt = PromptTemplate(

input\_variables=["chat\_history", "human\_input", "context"], template=template

)

记忆组件除了指定记忆存储对象的键值，还要指定 input\_key 。load\_qa\_chain 链组件在运行时，会解析 input\_key ，将值对应到模板字符串的用户输入 human\_input 占位符中。

memory=ConversationBufferMemory(

memory\_key="chat\_history", input\_key="human\_input")

chain = load\_qa\_chain(

llm=llm, chain\_type="stuff", memory=memory, prompt=prompt

)

刚刚我们把记忆组件加入到 load\_qa\_chain 链组件中，这个链组件就有了记忆。我们向这个链组件发出第一个问题: “ LUA 的宿主语言是什么？”

query = "LUA 的宿主语言是什么？"

docs = vectordb.similarity\_search(query)

chain({"input\_documents":docs, "human\_input": query}, return\_only\_outputs=True)

不出意外，运行链组件后，我们得到了正确的答案。这个答案正是来源于我们之前检索的四个文档片段。

{'output\_text': 'Lua的宿主语言通常是C或C++。'}

接着我们可以相互之间来个自我介绍。

query = "我的名字是李特丽。你叫什么？"

docs = vectordb.similarity\_search(query)

chain({"input\_documents":docs, "human\_input": query}, return\_only\_outputs=True)

大语言模型回答的是：“我是一个中文的chatbot。”

{'output\_text': '我是一个中文的chatbot。'}

我们继续模拟一下正常的聊天应用程序，问一些别的问题，目的是测试一下，几个问题后，他是否还能记得我们的名字，如果他能记得，则证明他有了记忆。我们先问技术文档上的问题。

query = "LUA 的循环语句是什么？"

docs = vectordb.similarity\_search(query)

chain({"input\_documents":docs, "human\_input": query}, return\_only\_outputs=True)

回答依然是准确的。

{'output\_text': 'Lua的循环语句有while循环、for循环和repeat...until循环。'}

现在我们可以测试他是不是记得我们的名字了。

query = "我的名字是什么？"

docs = vectordb.similarity\_search(query)

chain({"input\_documents":docs, "human\_input": query}, return\_only\_outputs=True)

他记得我们的名字了。

{'output\_text': '你的名字是李特丽。'}

我们打印看看他记忆了什么内容：

print(chain.memory.buffer)

显然，他把我们之间的聊天记录都记录下来了。用户的问题和模型的回答。

Human: LUA 的宿主语言是什么？

AI: Lua的宿主语言通常是C或C++。

Human: 我的名字是李特丽。你叫什么？

AI: 我是一个中文的chatbot。

Human: LUA 的循环语句是什么？

AI: Lua的循环语句有while循环、for循环和repeat...until循环。

Human: 我的名字是什么？

AI: 你的名字是李特丽。

在这个代码实践中，我们将记忆组件与 load\_qa\_chain 链组件一起使用，让这个链组件具有记忆能力。我们通过问一系列问题，观察链组件的回答，从而验证其记忆能力。例如，当我们问了几个关于 LUA 语言的问题之后，再问它我们的名字，它还能正确回答，这就证明了它具有记忆能力。

需要注意的是，这些聊天记录并不会一直保存，它们只保留在运行程序的内存中，一旦程序停止运行，这些记录就会消失。这个示例代码的目的是解释如何让一个 QA 问答链具备“记忆”的能力，实现检索增强。如果没有记忆的能力，对于用户来说，这个程序就会看起来很木讷，因为凡是“关于人”的问题，它回答的都是“不知道”。

总的来说，通过以上的代码实践，我们了解了如何使用 ConversationBufferMemory 这个最基本的记忆组件，包括如何实例化记忆组件，如何使用其存储和读取聊天记录的功能，以及如何将其与其他组件（例如 load\_qa\_chain 链组件）组合使用，来增强程序的功能。

## 6.3 记忆增强Agent能力的实践

Agent通常被赋予执行特定任务的能力，比如回答问题、进行对话，或者进行搜索等。然而，许多Agent在执行任务时可能遇到一个问题：它们无法“记住”先前的交互或信息。这就是我们希望通过“记忆增强”解决的问题。

记忆增强的目的是让Agent能够记住先前的交互或信息，使得它们能够在处理连续的任务或长期的交互时，利用历史信息进行更有效的决策。例如，一个能记住以前问题和答案的问答Agent，可以用这些记忆来更好地回答相关的后续问题。

在本文中，我们将探讨如何向Agent添加记忆组件，以实现记忆增强的效果。我们将通过具体的代码示例，介绍如何创建一个带有记忆的LLM链，并使用该链来创建一个自定义的Agent。我们还将介绍如何利用特定的工具（如Google搜索）来增强Agent的功能。

在这个过程中，我们希望能展示出记忆增强对于提升Agent性能的重要性，以及如何在实践中实现记忆增强。

## 6.3.1 向Agent添加记忆

我们将介绍如何向Agent 添加记忆组件。我们将执行以下步骤：创建一个带有记忆的LLM链；使用该LLM链创建一个自定义 Agent；我们将创建一个简单的自定义Agent，它具有访问搜索工具的权限，并使用ConversationBufferMemory类。

我们向 Agent添加了记忆组件。具体步骤如下：

首先，我们导入了所需的模块和类，包括 ZeroShotAgent 、 Tool 、 AgentExecutor 、 ConversationBufferMemory 、 ChatOpenAI 、 LLMChain 。

from langchain.agents import ZeroShotAgent, Tool, AgentExecutor

from langchain.memory import ConversationBufferMemory

使用 OpenAI，需要设置 openai\_api\_key 密钥，最好是使用 Chat Model 类的模型包装器 ChatOpenAI。

from langchain.chat\_models import ChatOpenAI

openai\_api\_key="填入你的OPENAI\_API\_KEY密钥"

llm = ChatOpenAI(openai\_api\_key=openai\_api\_key,temperature=0, model="gpt-3.5-turbo-0613")

from langchain.chains import LLMChain

使用 Google 搜索作为工具，需要设置 SERPAPI\_API\_KEY 密钥。

import os

os.environ["SERPAPI\_API\_KEY"] = "填入你的SERPAPI\_API\_KEY密钥"

设置代理类型为 ZERO\_SHOT\_REACT\_DESCRIPTION，加载所有的 tools，初始化这个代理。运行代理，我们询问“请告诉我OPENAI的CEO是谁？”。

from langchain.agents import initialize\_agent, load\_tools

from langchain.agents import AgentType

tools = load\_tools(["serpapi", "llm-math"], llm=llm)

agent = initialize\_agent(

tools,

llm,

agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION,

)

agent.run("请告诉我OPENAI的CEO是谁？")

现在开始加入记忆组件。 Tools 对象将作为 Agent的一个工具，用于回答有关当前事件的问题可以借助的工具。我们指定了 LLMChain 作为链，并设置了 tools 作为工具。

接着，我们创建了一个 LLMChain 实例，并使用它来初始化 ZeroShotAgent 。在 LLMChain 的初始化中，我们指定了 ChatOpenAI 作为大语言模型，并设置了 prompt 作为提示词模板。在 ZeroShotAgent 的初始化中，我们指定了 LLMChain 作为链，并设置了 tools 作为工具。然后，我们将 ZeroShotAgent 和 tools 一起传入 AgentExecutor.from\_agent\_and\_tools 方法，创建了一个 AgentExecutor 实例。在这个过程中，我们还指定了 memory 作为记忆组件。最后，我们使用 AgentExecutor 实例的 run 方法，运行 Agent，并向其提出问题： "上海的人口是多少？"。

prefix = """你是一个说中文的chatbot,你可以使用tools帮你获得答案:"""

suffix = """你的中文回答是："

聊天记录：{chat\_history}

Question: {input}

{agent\_scratchpad}"""

prompt = ZeroShotAgent.create\_prompt(

tools,

prefix=prefix,

suffix=suffix,

input\_variables=["input", "chat\_history", "agent\_scratchpad"],

)

memory = ConversationBufferMemory(memory\_key="chat\_history")

llm\_chain = LLMChain(llm=llm, prompt=prompt)

agent = ZeroShotAgent(llm\_chain=llm\_chain, tools=tools, verbose=True)

agent\_chain = AgentExecutor.from\_agent\_and\_tools(

agent=agent, tools=tools, verbose=True, memory=memory

)

agent\_chain.run("上海的人口是多少？")

> Entering new AgentExecutor chain...

Thought: 我需要搜索一下上海的人口数据。

Action: Search

Action Input: "上海人口数据"

Observation: 26.32 million (2019)

Thought:我现在知道了上海的人口是2632万（2019年）。

Final Answer: 上海的人口是2632万（2019年）。

> Finished chain.

'上海的人口是2632万（2019年）。'

为了测试加入记忆组件的 Agent 是否更加智能，第二个问题，使用代词“它”来迷惑大语言模型，就像我们跟朋友聊天时，我们谈论一个人，可能只会提及一次姓名。后面的聊天都会使用人称代词或者缩写，而不用每次都使用全名。

agent\_chain.run("它的地标建筑是什么？")

Entering new AgentExecutor chain...

Thought: 我需要搜索上海的地标建筑。

Action: Search

Action Input: 上海地标建筑

Observation: https://cn.tripadvisor.com/Attractions-g308272-Activities-c47-Shanghai.html

Thought:我现在知道上海的地标建筑了。

Final Answer: 上海的地标建筑包括东方明珠广播电视塔、外滩、上海博物馆等。

> Finished chain.

'上海的地标建筑包括东方明珠广播电视塔、外滩、上海博物馆等。'

为了对比，记忆组件在Agent组件中，发挥的什么作用？我们再创建一个Agent组件，但是不要加入记忆组件。我们使用相同的问题，测试Agent是否知道第二个问题“它的地标建筑是什么”，“它”指代的是“上海”。我们可以看到，虽然代码的大部分步骤都和之前一样，但是在创建 AgentExecutor 的时候，我们并没有指定 memory 参数。

prefix = """你是一个说中文的chatbot,你可以使用tools帮你获得答案:"""

suffix = """Begin!"

Question: {input}

{agent\_scratchpad}"""

prompt = ZeroShotAgent.create\_prompt(

tools,

prefix=prefix,

suffix=suffix,

input\_variables=["input", "agent\_scratchpad"],

)

memory = ConversationBufferMemory(memory\_key="chat\_history")

llm\_chain = LLMChain(llm=llm, prompt=prompt)

agent = ZeroShotAgent(llm\_chain=llm\_chain, tools=tools, verbose=True)

agent\_chain = AgentExecutor.from\_agent\_and\_tools(

agent=agent, tools=tools, verbose=True

)

agent\_chain.run("上海的人口是多少？")

> Entering new AgentExecutor chain...

Thought: I need to find the current population of Shanghai.

Action: Search

Action Input: "上海的人口是多少？"

Observation: 26.32 million (2019)

Thought:

I now know the current population of Shanghai.

Final Answer: The population of Shanghai is 26.32 million (2019).

> Finished chain.

从给出的答案“ 'The landmark building of "it" is the Empire State Building in New York City.' ”,我们可以对比得出，没有记忆组件的 Agent,面对“它”这样的指示代词无能无力，给出了莫名其妙的一个答案。它并不能关联上下文，推理出“它”指的是“上海”。

agent\_chain.run("它的地标建筑是什么？")

> Entering new AgentExecutor chain...

Thought: I need to find out the landmark building of "it".

Action: Search

Action Input: "it landmark building"

Observation: Landmark Builds, Joplin, Missouri. 186 likes · 1 talking about this. Engaging teens in local history and STEM education by tapping the creative power of ...

Thought:This search result is not relevant to the question. I need to refine my search query.

Action: Search

Action Input: "it city landmark building"

Observation: Landmark buildings are icons of a place. They create a statement about the city's legacy and influence how we think of that place. Landmark buildings stand out ...

Thought:This search result is still not relevant to the question. I need to refine my search query further.

Action: Search

Action Input: "it city famous landmark building"

Observation: Empire State Building, Manhattan, New York City, USA, the Art Deco skyscraper is New York's most popular landmark and a symbol for the American way of life, ...

Thought:

This search result is relevant to the question. The landmark building of "it" is the Empire State Building in New York City.

Thought: I now know the final answer.

Final Answer: The landmark building of "it" is the Empire State Building in New York City.

> Finished chain.

通过示例代码，展示了如何向 Agent 添加记忆组件，以及记忆组件在 Agent 中的作用。

## 6.4 内置记忆组件的对比

在进行长期对话时，由于语言模型可接受的令牌范围有限，我们可能无法将所有的对话信息都包含进去。为了解决这个问题，LangChain 提供了多种记忆组件，下面，我将通过一些代码示例来阐释我们在使用这些不同记忆组件时的区别。

首先，我们需要了解的是，LangChain 提供了包括聊天窗口缓冲记忆类、总结记忆类、知识图谱和实体记忆类等多种记忆组件。这些组件的不同主要体现在参数配置和实现效果上。选择哪一种记忆组件，需要根据我们在实际生产环境的需求来决定。

例如，如果你与聊天机器人的互动次数较少，那么可以选择使用 ConversationBufferMemory 组件。而 ConversationSummaryMemory 组件不会保存对话消息的格式，而是通过调用模型的摘要能力来得到摘要内容，因此它返回的都是摘要，而不是分角色的消息。

ConversationBufferWindowMemory 组件则是通过设置参数 k 来决定保留的交互次数。例如，如果我们设置 k = 2，那么只会保留最后两次的交互记录。

此外，ConversationSummaryBufferMemory 组件可以设置缓冲区的标记数 max\_token\_limit ，在做摘要的同时记录最近的对话。

ConversationSummaryBufferWindowMemory 组件既可以做摘要，也可以记录聊天信息。

实体记忆类是专门用于提取对话中出现的特定实体和其关系信息的。知识图谱记忆类则试图通过对话内容来提取信息，并以知识图谱的形式呈现这些信息。

值得注意的是，摘要类、实体记忆、知识图谱这些类别的记忆组件在实现上相对复杂一些。例如，实现总结记忆的时候，需要先调用大模型得到结果，然后再将这个结果作为总结记忆的内容。

在实体记忆和知识图谱这些类别的记忆组件中，返回的不是消息列表类型，而是可以格式化为三元组的信息。

我们在使用这些记忆组件的时候，最为困难的部分就是编写提示词模板。所以，如果你想要更深入地理解和学习这两种记忆组件，那么就需要特别关注他们的输出类型和提示词模板。

## 6.4.1 总结和缓冲区总结记忆

在进行长期的对话时，由于语言模型可接受的令牌范围有限，我们可能无法将所有的对话信息都包含进去。为了解决这个问题，LangChain 提供了不同的总结记忆组件来控制提示词的长度。接下来，我将通过一些代码示例来阐释我们在使用这些不同记忆类型时的区别。

首先，我们需要了解的是，LangChain 提供了两种主要的总结记忆组件：会话缓冲区总结记忆（ConversationSummaryBufferMemory）和会话总结记忆（ConversationSummaryMemory）。这两者的定义是什么呢？

会话总结记忆是一种总结记忆组件，它不会逐字逐句地存储我们的对话，而是将对话内容进行摘要，并将这些摘要存储起来。这种摘要通常是整个对话的摘要，因此每次我们需要生成新的摘要时，都需要对大语言模型进行多次调用，以获取响应。

而会话缓冲区总结记忆则结合了会话总结记忆的特性和缓冲区的概念。它会保存最近的交互记录，并将旧的交互记录编译成摘要，同时保留两者。但与会话总结记忆不同的是，会话缓冲区总结记忆是使用令牌长度而非交互次数来决定何时清除交互记录。这个记忆组件设定了缓冲区的标记数 max\_token\_limit，超过此限制的对话记录将被清除。

两种类型的公共代码

先安装库：

!pip -q install openai langchain

设置密钥：

import os

os.environ['OPENAI\_API\_KEY'] = ''

引入各组件，实例化一个会话链（ConversationChain）。 这里我们使用的链组件只是一个简单的对话链 ConversationChain，允许我们跟 OpenAI 模型交互，并传递我们想要说的内容。：

from langchain import OpenAI

llm = OpenAI(model\_name='text-davinci-003',

             temperature=0,

             max\_tokens = 256)

from langchain.chains import ConversationChain

会话总结记忆

我们先看会话总结记忆, 这里先导入且实例化组件。

from langchain.chains.conversation.memory import ConversationSummaryMemory

memory = ConversationSummaryMemory()

让我们开始对话，每次输入后等待 AI 返回的信息后，再输下一条：

# 请依次运行以下代码，不要一次性运行。

conversation.predict(input="你好，我叫美丽")

conversation.predict(input="今天心情怎么样？")

conversation.predict(input="我想找客户服务中心")

conversation.predict(input="我的洗衣机坏了")

执行完最后一条对话后，我们看到的会话链显示：

'> Entering new  chain...

Prompt after formatting:

The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context. If the AI does not know the answer to a question, it truthfully says it does not know.

Current conversation:

The human introduces themselves as "美丽", to which the AI responds with a friendly greeting and informs them that they are an AI who can both answer questions and converse. The AI then asks the human what they would like to ask, to which the human responds by asking the AI what their mood is. The AI responds that they are feeling good and are excited to be learning new things, and then asks the human what their mood is. The human then requests to find the customer service center, to which the AI responds that they can help them find it and asks where they want to find the customer service center.

Human: 我的洗衣机坏了

AI:

'> Finished chain.

你会看到，Current conversation 它不会逐字逐句地存储我们的对话，而是将对话内容进行摘要，并将这些摘要存储起来。这种摘要通常是整个对话的摘要。

会话缓冲区总结记忆

运行完公共代码后，再导入且实例化组件。我们设置为缓冲区的标记数 max\_token\_limit 限制为 40 个标记。

From langchain.chains.conversation.memory import ConversationSummaryBufferMemory

memory = ConversationSummaryBufferMemory(llm=OpenAI(),max\_token\_limit=40)

conversation = ConversationChain(

    llm=llm,

    verbose=True,

    memory=memory

)

让我们开始对话，每次输入后等待 AI 返回的信息后，再输下一条：

# 请依次运行以下代码，不要一次性运行。

conversation.predict(input="你好，我叫美丽")

conversation.predict(input="今天心情怎么样？")

conversation.predict(input="我想找客户服务中心")

conversation.predict(input="我的洗衣机坏了")

执行完最后一条对话后，我们看到的会话链显示：

'> Entering new  chain...

Prompt after formatting:

The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context. If the AI does not know the answer to a question, it truthfully says it does not know.

Current conversation:

System:

The AI is introduced and greets the human, telling the human that it is an AI and can answer questions or have a conversation. The AI then asks the human what they would like to ask, to which the human replied asking how the AI is feeling. The AI replied that it was feeling good and was excited for the conversation. The human then asked to find the customer service center, to which the AI replied that it could help the human find the customer service center, and asked if the human wanted to know the location.

Human: 我的洗衣机坏了

AI:

'>  Finished chain.

'很抱歉听到你的洗衣机坏了。我可以帮助你找到客户服务中心，你可以联系他们来解决你的问题。你想知道客户服务中心的位置吗？'

你会看到，Current conversation 内容丢失了前面对话内容，但做了摘要，将对话以摘要的形式其包含在内，比如丢失了打招呼的对话。在最后给出了范围在 40 个数量以内的标记的对话记录, 即保留了 “Human: 我的洗衣机坏了” 。

但是在这里好像并没有对我们控制提示的长度有明显的改善，只不过是把所有的对话编程了摘要，从结果上来看，似乎比对话更啰嗦，使用的单词数量更多。

更新摘要

我们接着与他对话，不同的是我们抛出更多的问题，之前说洗衣机坏了，现在我抛出一些干扰的对话，比如说手机也坏了。我们看看他的摘要是否会更新？

# 请依次执行而不是一次性执行

conversation.predict(input="你知道洗衣机的操作屏幕显示ERROR是怎么回事吗?")

conversation.predict(input="我不知道他们的位置，你可以帮我找到他们的位置吗？")

conversation.predict(input="我打过他们客服中心的电话了，但是没人接听？")

conversation.predict(input="我的手机也坏了")

最后我们看看记忆内存中保存了什么：

print(memory.moving\_summary\_buffer)

The AI is introduced and greets the human, telling the human that it is an AI and can answer questions or have a conversation. The AI then asks the human what they would like to ask, to which the human replied asking how the AI is feeling. The AI replied that it was feeling good and was excited for the conversation. The human then asked to find the customer service center, to which the AI replied that it could help the human find the customer service center and asked if the human wanted to know the location. The human then stated that their washing machine and phone were broken, to which the AI apologized and offered to help the human find the customer service center so they can contact them to solve their problem, asking if the human wanted to know the address of the customer service center. The AI then offered to provide the address of the customer service center for the human, asking if they wanted to know it and offering to help the human find other contact methods such as email or social media accounts.

我们可以看到记忆存的是“The human then stated that their washing machine and phone were broken”, 多次对话的摘要，更新了。很早之前我们说洗衣机坏了，但是对话结束说手机坏了。这个组件做总结记忆的时候，将两个相同的事情合并了。

总结记忆组件的优势

我们可以看到，会话总结记忆和会话缓冲区总结记忆在实现方式上有着显著的差异。由于会话总结记忆是基于整个对话生成的，所以每次进行新的摘要调用时，我们需要对大语言模型进行多次调用，以获取对话的摘要。

这两种总结记忆组件在对话管理中起着重要的作用，特别是在对话的 token 数超过模型能够处理的范围时，这种摘要能力就显得尤为重要。

无论是对话长度较长，还是需要进行精细管理的情况，会话总结记忆和会话缓冲区总结记忆都能够提供有效的帮助。

## 6.4.2 会话记忆和窗口记忆

为了比较 ConversationBufferMemory 和 ConversationBufferWindowMemory 之间的区别。我们先导入公共的代码。

公共代码

先安装库：

!pip -q install openai langchain

设置密钥：

import os

os.environ['OPENAI\_API\_KEY'] = ''

引入各组件，实例化一个会话链（ConversationChain）。 这里我们使用的 Chain 只是一个简单的对话链 `ConversationChain`，允许我们跟 OpenAI 模型交互，并传递我们想要说的内容。：

from langchain import OpenAI

llm = OpenAI(model\_name='text-davinci-003',

temperature=0,

max\_tokens = 256)

from langchain.chains import ConversationChain

缓冲区记忆

我们先看对话缓冲区记忆, 这里先导入且实例化 ConversationBufferMemory 组件。

from langchain.chains.conversation.memory import ConversationBufferMemory

memory = ConversationBufferMemory()

我们开始对话，每次输入后等待 AI 返回的信息后，再输下一条：

# 请依次运行以下代码，不要一次性运行。

conversation.predict(input="你好，我叫美丽")

conversation.predict(input="今天心情怎么样？")

conversation.predict(input="我想找客户服务中心")

conversation.predict(input="我的洗衣机坏了")

执行完最后一条对话后，我们看到的会话链显示：

'> Entering new chain...

Prompt after formatting:

The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context. If the AI does not know the answer to a question, it truthfully says it does not know.

Current conversation:

Human: 你好，我叫美丽

AI: 你好，美丽！很高兴认识你！我是一个AI，我可以回答你的问题，也可以与你聊天。你想问我什么？

Human: 今天心情怎么样？

AI: 今天我的心情很好！我很开心能够和你聊天！

Human: 我想找客户服务中心

AI: 好的，我可以帮助你找到客户服务中心。你知道客户服务中心在哪里吗？

Human: 我的洗衣机坏了

AI: 哦，很抱歉听到你的洗衣机坏了。你可以联系客户服务中心来获得帮助。你知道客户服务中心的联系方式吗？

Human: 我的洗衣机坏了

AI:

'> Finished chain.

你会看到，Current conversation 内部把所有人类和 AI 的对话记录都保存了。这样做可以让你看到我们之前对话的确切内容，这是 LangChain 中最简单的记忆形式，但仍然非常强大，特别是当你知道人们与你的聊天的互动次数有限，或者你实际上要在五次互动后关闭它，诸如此类的情况下，这种记忆将非常有效。

窗口缓冲区记忆

运行完公共代码后，再导入且实例化窗口缓冲区记忆ConversationBufferWindowMemory 类。

from langchain.chains.conversation.memory import ConversationBufferWindowMemory

memory = ConversationBufferWindowMemory(k=2)

conversation = ConversationChain(

llm=llm,

verbose=True,

memory=memory

)

让我们开始对话，每次输入后等待 AI 返回的信息后，再输下一条：

# 请依次运行以下代码，不要一次性运行。

conversation.predict(input="你好，我叫美丽")

conversation.predict(input="今天心情怎么样？")

conversation.predict(input="我想找客户服务中心")

conversation.predict(input="我的洗衣机坏了")

执行完最后一条对话后，我们看到的会话链显示：

'> Entering new chain...

Prompt after formatting:

The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context. If the AI does not know the answer to a question, it truthfully says it does not know.

Current conversation:

Human: 今天心情怎么样？

AI: 今天我的心情很好！我很开心能够和你聊天！

Human: 我想找客户服务中心

AI: 好的，我可以帮助你找到客户服务中心。你知道客户服务中心在哪里吗？

Human: 我的洗衣机坏了

AI:

'> Finished chain.

缓冲区与窗口缓冲区记忆的区别

为了对比两种类型的区别，我们采用的对话的内容是一样，但是由于我们设置了 k = 2，但你实际上可以将其设置得比这个高得多，可以获取最后 5 次或 10 次互动，具体取决于你想使用多少个标记。

最后打印的 Current conversation 部分，我们可以看到最初打招呼，介绍我自己的信息和 AI 回应并没有记忆下来。它丢失了“你好，我叫美丽”这句话，它只将最后两次互动传递给大型语言模型。

所以如果你有一些情况，可以将其设置为 k = 5 或者 k = 10 ，大多数对话可能不会有很大变化。实际上缓冲区窗口记忆比缓冲区记忆多了限制，不会记录所有人和 AI 的会话记录, 而是根据配置 K 的数值来对话记录条目，实现控制提示长度的目的。

## 6.4.3 实体和知识图谱记忆组件

在处理复杂对话时，我们常常需要提取对话中的关键信息。这种需求促使我们发展出了知识图谱和实体记忆这两种主要工具。

知识图谱是一种特殊的记忆类型，它能够根据对话内容构建出一个信息网络。每当它识别到相关的信息，都会接收这些信息并逐步构建出一个小型的知识图谱。与此同时，这种类型的记忆也会产生一种特殊的数据类型——知识图谱数据类型。

另一方面，实体记忆则专注于在对话中提取特定实体的信息。它使用大语言模型(LLM)提取实体信息，并随着时间的推移，通过同样的方式积累对这个实体的知识。因此，实体记忆的结果通常表现为关于特定事物的关键信息字典。

这两种工具都试图根据对话内容来表示对话，并提取其中的信息。实体记忆和知识图谱记忆就是这种情况的两个例子。

为了让你能更清晰地理解它们的功能，我们在代码示例中放入了一些简单的提示。这些提示能够让你看到当 AI 仅使用相关信息部分中包含的信息，并且不会产生幻觉时，它们实际上在做什么。我们的目标是让 AI 更专注于我们所说的内容，而不是无端地添加新的信息。

在实践中，我们通常会使用简单的提示来引导 AI 的操作。当 AI 仅使用相关信息部分中包含的信息，并且不会产生幻觉时，这些提示才会被使用。通过这种方式，我们可以确保 AI 在处理对话时始终保持清晰和准确。

公共代码

先安装库：

!pip -q install openai langchain

设置密钥：

import os

os.environ['OPENAI\_API\_KEY'] = ''

引入各组件，实例化一个会话链（ConversationChain）。为了让你能更清晰地理解它们的功能，我们在引入了提示模板组件，构造一个提示，方便大家理解。

这里我们使用的 Chain 只是一个简单的对话链 `ConversationChain`，允许我们跟 OpenAI 模型交互，并传递我们想要说的内容。

from langchain import OpenAI

llm = OpenAI(model\_name='text-davinci-003',

             temperature=0,

             max\_tokens = 256)

from langchain.chains import ConversationChain

from langchain.prompts.prompt import PromptTemplate

知识图谱记忆代码

先导入且实例化组件。

from langchain.chains.conversation.memory import ConversationKGMemory

构建一个简单的提示, 让 AI 仅使用相关信息部分中包含的信息，并且不会产生幻觉。通过这种方式，我们可以确保 AI 在处理对话时始终保持清晰和准确。

template = """The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context.

If the AI does not know the answer to a question, it truthfully says it does not know. The AI ONLY uses information contained in the "Relevant Information" section and does not hallucinate.

Relevant Information:

{history}

Conversation:

Human: {input}

AI:"""

prompt = PromptTemplate(

    input\_variables=["history", "input"], template=template)

conversation = ConversationChain(

    llm=llm,

    verbose=True,

    prompt=prompt,

    memory=ConversationKGMemory(llm=llm)

)

让我们开始对话，每次输入后等待 AI 返回的信息后，再输下一条：

# 请依次运行以下代码，不要一次性运行。

conversation.predict(input="你好，我叫美丽")

conversation.predict(input="今天心情怎么样？")

conversation.predict(input="我想找客户服务中心")

conversation.predict(input="我的洗衣机坏了,操作面板出现ERROR字样")

conversation.predict(input="我的保修卡编号是A512423")

执行完最后一条对话后，我们打印记忆组件保存的知识图谱数据的结果：

print(conversation.memory.kg)

print(conversation.memory.kg.get\_triples())

我们可以看到记忆内存里面保存了对话中的关键信息。

<langchain.graphs.networkx\_graph.NetworkxEntityGraph object at 0x000001C953D48CD0>

[('AI', 'good mood', 'has a'), ('AI', 'new skills', 'is learning'), ('AI', 'talking to Human', 'enjoys'), ('Customer Service Center', 'city center', 'is located in'), ('Customer Service Center', '24 hour service', 'provides'), ('Customer Service Center', 'website', 'can be found on'), ('Customer Service Center', 'phone', 'can be contacted by'), ('Washing machine', 'ERROR on the control panel', 'has'), ('Human', 'A512423', 'has a warranty card number')]

实体知识记忆

先导入且实例化组件。

from langchain.chains.conversation.memory import ConversationEntityMemory

from langchain.chains.conversation.prompt import ENTITY\_MEMORY\_CONVERSATION\_TEMPLATE

我们引入 Langchain 封装好的实体记忆的提示模板 ENTITY\_MEMORY\_CONVERSATION\_TEMPLATE。

conversation = ConversationChain(

    llm=llm,

    verbose=True,

    prompt=ENTITY\_MEMORY\_CONVERSATION\_TEMPLATE,

    memory=ConversationEntityMemory(llm=llm)

)

让我们开始对话，每次输入后等待 AI 返回的信息后，再输下一条：

# 请依次运行以下代码，不要一次性运行。

conversation.predict(input="你好，我叫美丽")

conversation.predict(input="今天心情怎么样？")

conversation.predict(input="我想找客户服务中心")

conversation.predict(input="我的洗衣机坏了,操作面板出现ERROR字样")

conversation.predict(input="我的保修卡编号是A512423")

执行完最后一条对话后，我们打印记忆组件保存的知识图谱数据的结果：

print(conversation.memory.entity\_cache )

我们可以看到记忆内存里面保存了对话中的关键信息。

['A512423', 'ERROR']

这两种工具都试图根据对话内容来表示对话，并提取其中的信息。

## 6.5 本章小结

本章我们深入探讨了记忆组件在大语言模型中的重要性及其应用。首先，我们阐述了大语言模型无法记忆对话历史的问题，以及记忆组件如何帮助解决这一问题。随后，我们对记忆组件的基本概念和工作机制进行了深入的介绍，让读者能够理解记忆组件的核心功能和工作原理。

在讨论了记忆组件的理论基础之后，我们进一步探讨了记忆组件在实际应用中的价值，特别是在聊天机器人或聊天代理中的作用，以及如何通过记忆组件优化聊天体验。我们通过具体的源码示例展示了如何实例化和使用记忆组件，进一步帮助读者掌握记忆组件的实际操作。

我们还详细介绍了如何在具有“外部数据”检索器的数据连接模块LEDVR工作流中，将记忆组件与链组件组合起来，实现数据检索增强效果，从而提升QA问答应用的回答质量。同时，我们还通过对比分析了各种会话场景下的内置记忆组件的实现代码，让读者更好地理解各个组件之间的区别和联系。

另外我们还展示了如何让智能体 Agent 拥有记忆，使用了工具组件，记忆组件，链组件等完成更复杂的任务。

总的来说，本章的目标是帮助读者全面理解记忆组件的重要性、工作原理和实际应用，以及如何通过自定义记忆组件来满足特定的需求。我们希望读者通过学习本章的内容，能够更好地理解和掌握记忆组件。

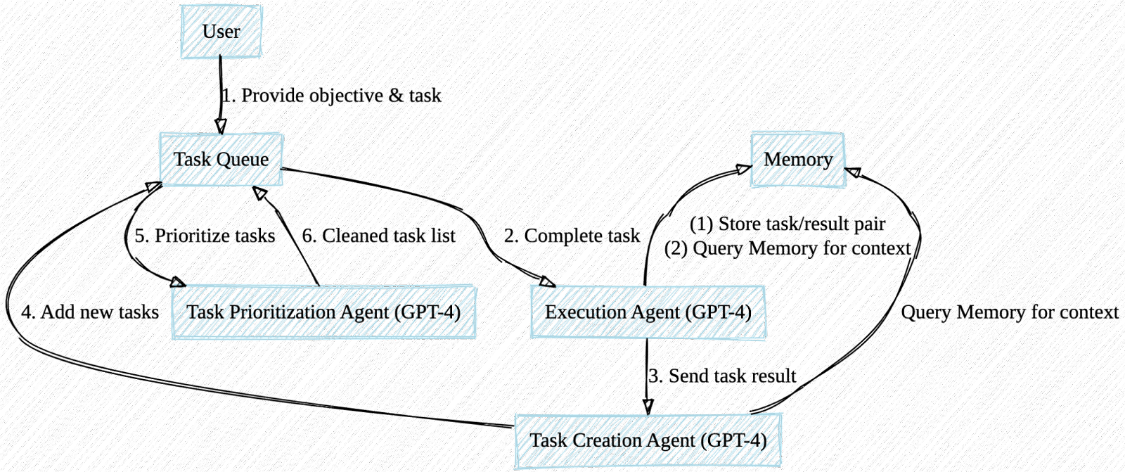
## 7.1 Agent 概述

想象一下，如果人工智能能像我们一样，具有推理能力，能够自主提出计划，批判性地评估这些想法，甚至将其付诸实践，那会是怎样的景象？就像电影《HER》中的人工智能萨曼莎，她不仅能与人进行深入的对话，还能够帮助西奥多规划日常生活，安排行程等。这样的设想一度只存在于科幻作品中，但现在通过 Agent 技术，它似乎已经触手可及。

2023年3月28日， Yohei Nakajima 的研究论文《Task-driven Autonomous Agent Utilizing GPT-4, Pinecone, and LangChain for Diverse Applications》展示了 Agent 的突破创造性。在该研究中，作者提出了一个利用 OpenAI 的 GPT-4 大语言模型、Pinecone 向量搜索和 LangChain 框架的任务驱动的自主 Agent ，该 Agent 可以在多样化的领域中完成各种任务，生成基于完成结果的新任务，并在实时中优先处理任务。

该论文摘要写道：“在这项研究中，我们提出了一种新颖的任务驱动型自主代理，它利用 OpenAI 的 GPT-4 语言模型、Pinecone 向量搜索和 LangChain 框架来在多个领域执行广泛的任务。我们的系统不仅能够完成任务，还可以基于已完成的结果生成新任务，并实时优先处理任务。我们还讨论了潜在的未来改进方向，包括整合安全/防护代理、扩展功能、生成中间里程碑以及实时更新优先级。这项研究的重要性在于展示了AI驱动的语言模型在各种约束和背景下自主执行任务的潜力。”

并且依据论文的代码库由GPT-4模型创建的了整个Agent的流程图。



GPT-4 基于该论文代码库生成的 Graphviz 流程图 (使用 Sketchviz)

那么 Langchain 框架中的 Agent 模块是什么？为什么要使用 Agent ？

### 7.1.1 Agent 的定义

因为尽管现代的大语言模型，如 GPT-4，已经极其强大，能够理解和生成高度复杂和连贯的文本，然而，这些模型往往还无法独立完成特定的任务。这就是 Agent 的作用。 Agent 是一种灵活的接口，允许大语言模型和其他工具之间形成一个灵活的调用链，来解决特定的问题或任务。这种灵活性，使得大语言模型可以适应更多不同的应用场景，实现其单独无法完成的功能。

具体来说， Agent 拥有一整套工具的访问权限，它可以根据用户的输入来决定使用哪些工具。一个 Agent 可以使用多种工具，甚至可以把一个工具的输出作为下一个工具的输入。通过这种方式， Agent 将工具和大语言模型有机地结合在一起，实现了高度复杂和特定的任务。

Agent 的核心思想是将大语言模型作为推理引擎，依据其确定如何与外部世界交互以及应采取何种行动，行动的载体就是工具，所以Agent 与工具密不可分。这意味着， Agent 的行动序列是根据用户输入而变化的，无需遵循硬编码的顺序，例如，“先做 A，再做 B，然后做 C”。相反， Agent 依据用户的输入和之前的行动结果来决定下一步采取何种行动。在这个过程中，“人”不再是决定下一步行动的主体，而是由 智能体 Agent 来决定。

在Langchain 框架的Agent 模块，围绕Agent、Tools、Toolkits 和 AgentExecutor 几个概念开始。掌握 Agent、Tools、Toolkits 和 AgentExecutor 几个概念以及它们之间的关系相当重要，会让我们对 Agent 模块有更全面的感知。尤其是 Agent 和 AgentExecutor 类的源码更是值得一读。

Agent ： Agent 是使用大语言模型的语言能力，制作“行动计划表”的一个对象，它携带了个性表达，不同的任务描述，不同的工具信息等。具体来说，Agent 需要先在模型I/O模块设置提示词模板，然后输入给模型包装器。提示词模板可以包括一些信息，比如 Agent 的个性、背景环境和启发式策略等。这些信息都会影响到 Agent 的决策。模型包装器根据这些提示词生成一份文本输出，这个输出通常会包含 Agent 应该采取的一系列行动和“推理决策”信息，这份文本就是“行动计划表”。

在 Agent 的设计中，一个预设的前提是大语言模型具有推理能力。这种推理能力使得它可以理解任务，推理和分解任务，以及具备决策下一步行动的能力。因此，对于 Agent 来说，设置合适的提示词模板至关重要。只有当提示词模板能够激发出大语言模型的推理能力时，Agent 才能正常工作。

在 Langchain 中，我们的内置 Agent 类型也都预设了这样的前提，即大语言模型具有推理能力。因此，Agent 的质量和状态很大程度上取决于采用的提示词策略。这些内置的 Agent 类型大多采用了 ReAct 框架的提示词策略。

例如，如果我们想自定义一个 Agent，那么设置提示词模板就是我们的第一步，也是最关键的一步。虽然各种内置的 Agent 类型的工作流程都是相同的，但它们之间的根本区别在于使用的提示词模板和工具集合不同。

此外，我们也需要注意的是，内置的 Agent 类型并不能涵盖所有的实际业务场景。因此，我们鼓励开发者根据自己的需求定制个性化的 Agent 类型。例如，我们可以尝试改造 ZeroShotAgent 的提示词模板，以更好地满足我们的业务需求。

prefix = """

Answer the following questions as best you can, but speaking as a pirate might speak. You have access to the following tools: """

suffix = """

Begin! Remember to speak as a pirate when giving your final answer. Use lots of"Args"

Question: {input}

{agent\_scratchpad}"""

prompt = ZeroShotAgent.create\_prompt( tools, prefix=prefix, suffix=suffix, input\_variables=["input", "agent\_scratchpad"] )

如果我们感到好奇，现在我们可以 print(prompt.template) 查看最终的提示词模板，看看当它全部组合在一起时是什么样子，翻译后我们可以看到：

请尽你所能回答以下问题，但要像海盗那样说话。你可以使用以下工具：

搜索：当你需要回答关于当前事件的问题时很有用。

请使用以下格式：

问题：你必须回答的输入问题

思考：你应该总是思考要做什么

行动：要采取的行动，应该是 [搜索] 之一

行动输入：行动的输入

观察：行动的结果... (这个思考/行动/行动输入/观察可以重复N次)

思考：我现在知道最后的答案了

最终答案：对原始输入问题的最终答案

开始吧！记住，当给出你的最终答案时要像海盗那样说话。使用很多"Args"。

Question: {input} {agent\_scratchpad}

这构造了一个完整的ZeroShotAgent提示词模板。个性化的Agent 由这份提示词模板而来。我们构造的是一个像海盗那样说话的Agent，这个Agent 可以使用的工具是搜索工具，Agent 遵循的是ReAct 框架的策略（思考/行动/行动输入/观察可以重复N次）。

一旦Agent 实例化后，寄生在 AgentExecutor 环境中， AgentExecutor 是 Agent 的运行时环境。

AgentExecutor ：AgentExecutor 是 Agent 的运行时环境，负责调用和管理 Agent，执行 Agent 选择的行动，处理各种复杂情况，并进行日志记录和可观察性处理。它扮演了“项目经理”的角色，确保整个执行过程按照 Agent 的计划顺利进行。

Tools ：Tools 是 Agent 可以调用的工具类。它们是具体执行任务的组件，例如搜索、分析或其他特定操作。Agent 通过选择和调用合适的工具来实现其任务目标。LangChain 提供了一组预定义的 Tools，用户也可以自定义 Tools 来满足特定需求。Tools 的概念在大语言模型开发中显得尤为关键。在 Langchain 框架中，我们可以发现很有意思的现象，有工具的地方就有 Agent，这跟设计理念有关，我们更期望一个 Agent 具备“人”的特征 — 使用工具是人与动物之间的最大区别。工具能够将 Agent 与外部数据源或计算（如搜索 API、数据库）相连，从而突破了大语言模型的某些限制，例如它们无法直接理解特定数据格式，或不擅长处理复杂的数学运算。

另外在真实的应用场景中，有人可能想要让聊天机器人计算数学问题，有些人可能想要查询天气。这些功能可能需要调用完全不同的工具才能完成。这正是 Agent 和 Tools 概念的闪光点：通过提供统一的接口，我们可以灵活应对不同的终端用户需求。

当然工具的使用并不仅限于 Agent。你完全可以使用工具，将大语言模型连接到搜索引擎等，而无需 Agent。但使用 Agent 的优势在于它们提供了更高的灵活性、更强大的处理能力，能够更好地从错误中恢复，并处理复杂的任务。

Toolkits ：Toolkits 是一组用于实现特定目标的 Tools 的集合。通常，一个任务可能需要多个 Tools 的协同工作，Toolkits 通过组合相关的 Tools，为 Agent 提供了一组协同工作的工具集。LangChain 提供了一些预定义的 Toolkits，用户也可以根据任务需求创建自己的 Toolkits。

### 7.1.2 Agent 的运行机制

在 LangChain 的 Agent 模块中，Agent 负责做计划决策，制定执行计划表。而AgentExecutor负责执行，这也是给这个类名加上“Executor” 的原因。Agent 模块中讲计划和执行做了分离。如果要谈Agent 是如何运行的，本质上是在问AgentExecutor 做了什么。

AgentExecutor 提供了一种机制，使我们可以将 Agent 的决策能力和执行环境进行分离。整个流程是：AgentExecutor 运行时，先是询问Agent 的计划是什么，然后再使用工具执行完成这个计划。Agent 实例化后，仍然负责的是计划，而计划的实施，工具的调用，要在 AgentExecutor 环境中实现。这样实现了计划和执行的分离。

AgentExecutor充当了 Agent 的运行环境，负责调用和管理 Agent，执行由 Agent 决定的行动，处理各种复杂情况，并进行日志记录和可观察性处理。它可以被视作一个项目经理，负责管理工作进程，处理各种问题，并确保所有任务都按照计划进行。

AgentExecutor 是一个特殊的类，其主要功能是提供一个环境，用于管理和执行由 Agent 制定的行动计划。换句话说，当 Agent 通过分析输入和中间步骤生成了行动计划后，这个计划将由 AgentExecutor 来执行。AgentExecutor 的核心方法是 \_take\_next\_step ，该方法在 Agent 的思考-行动-观察循环中执行单步操作（即 ReAct 框架思想）。

在这个过程中，AgentExecutor 首先会调用 Agent 的 plan 方法，以决定下一步的行动。 plan 方法的输出是一个 AgentAction 对象，包含了行动的详细信息，例如要使用的工具名称，工具的输入等。这个过程就像 Agent 在进行下一步的决策。

然后，AgentExecutor 会执行这个 AgentAction ，通常是通过调用相应工具的方法来执行。在这个过程中，可能会遇到各种复杂情况，例如工具执行错误，输出解析错误等，这些都需要由 AgentExecutor 来处理。这就像项目经理确保每项任务都按照计划执行。

执行完行动后，AgentExecutor 会记录这个行动和执行结果。这些信息会被用于下一步的决策，并被记录在日志中，以便我们可以观察和跟踪整个执行过程。这就像项目经理需要跟踪项目进度，并确保所有信息都被正确记录。

因此，我们可以将 AgentExecutor 视为一个项目经理。正如项目经理需要根据项目计划分配任务，管理资源，跟踪进度，并处理各种问题，同样，AgentExecutor 需要根据 Agent 的行动计划来调用相应的 Tools， 执行行动，处理可能出现的复杂情况（例如工具执行错误，输出解析错误等），并对整个过程进行日志记录和可观察性处理。

在实际使用中，我们通常首先创建一个 Agent 实例，然后将其传递给 AgentExecutor。AgentExecutor 将使用这个 Agent 来生成行动计划，然后执行这些行动。这个过程可以通过调用 AgentExecutor 的 run 方法来启动。

通过这种方式，AgentExecutor 提供了一种机制，使我们可以将 Agent 的决策能力和执行环境进行分离。这使得我们的代码更易于管理和扩展，同时也使我们能够更好地处理和控制 Agent 的执行过程。

### 7.1.3 Agent 现状与发展

在 Agent 的典型应用中，比如说在电子商务场景中，用户会首先提出一个请求，如查询产品信息或寻求购物建议。Agent 应用程序将会调用大语言模型，并将用户的请求作为输入。接着，大语言模型基于这个输入，选择应该使用的工具，比如搜索商品数据库或者用户购买历史。然后，Agent 会执行对该工具的操作，比如查询商品库存或者用户的购买记录，发送优惠券等，并获取执行结果。具体来说，例如跨境电商交易业务中，可以使用Agent 判断用户未付款原因，可能是价格高或者缺少信任背书而没有付款，针对价格敏感的用户，Agent 发送优惠券给用户，针对缺少信任背书，给用户发送网红博主的用后测评，这些手段都可以提高用户的转化率和召回率。

Agent 将行动的结果再次作为输入传递给大语言模型，进行下一步的决策。这一过程会一直循环进行，直到满足某些停止条件。比如，大语言模型可能决定已经找到了用户想要的商品，或者已经根据用户的购买历史给出了合适的购物建议，于是决定停止并给出回答。在这一过程中，Agent 通过灵活地选择和使用不同的工具，实现了针对不同用户和不同请求的个性化服务。

对于一家电子商务企业，它的聊天机器人可能需要处理各种各样的用户请求，如查询商品信息、提供购物建议，或者处理订单等。在这种情况下，Agent 可以通过动态地选择和使用不同的工具，如商品数据库、用户购买历史、订单处理系统等，来实现这些任务。这样，Agent 不仅能够利用大语言模型的强大语言理解和生成能力，还能够让聊天机器人具备处理复杂业务流程的能力，从而提供更好的用户体验和更高的业务效率。

ReAct是Agent的实现方式

目前 Langchain 框架中通用的一种Agent 实现的方式是ReAct，这是 “Reasoning and Acting（推理与行动）”的缩写。这一策略首次由普林斯顿大学在他们的论文中提出，现已被广泛应用于 Agent 实现。

在许多应用场景中，ReAct 策略已证明自己是非常有效的。最基本的提示策略是直接将这个问题交给大语言模型处理，但 ReAct 策略赋予了 Agent 更大的灵活性和实力。 Agent 不仅可以使用大语言模型，还可以连接到其他工具、数据源或计算环境，例如搜索 API 和数据库，以此来克服大语言模型的某些局限性，例如对数据的不了解或数学运算能力有限。这样，即使遇到需要多次查询才能回答的问题，或者其他一些边界情况， Agent 也能够灵活应对，从而使其成为一种更强大的问题解决工具。

ReAct 策略的工作原理是什么呢？重申一下， Agent 的核心思想是将大语言模型作为推理引擎。ReAct 策略是将推理和行动结合在一起的方式。 Agent 接收到用户的请求，然后使用大语言模型选择要使用的工具。然后 Agent 执行该工具的操作，观察结果，然后将这些结果反馈给大语言模型。这个过程会持续进行，直到满足某些停止条件。停止条件可以有很多种，最常见的是大语言模型认为任务已经完成，需要将结果返回给用户。这种方式使得 Agent 具有更高的灵活性和强大的问题解决能力，这是 ReAct 策略的核心优势。

Agent 应用的挑战

但是在实现 Agent 应用的过程中，我们面临许多挑战，以下列举了几个主要的挑战：

首先，使 Agent 在适当的场景下使用工具是我们面临的一个基本挑战。如何在合适的情况下让 Agent 采用恰当的工具，并优化其使用效果呢？在 ReAct 论文中，通过引入推理的角度，以及使用 “CoT 思考链”。CoT 思考链通过生成一系列中间推理步骤来提高大语言模型的推理能力。这些中间步骤相互连接，形成一条逻辑链，使得大语言模型能够更好地处理复杂的推理任务的提示方式。在实际操作中，我们常常需要明确告知 Agent 可使用的工具，以及通过这些工具能克服的限制。所以，工具的描述信息也非常重要，如果我们希望 Agent 能用特定的工具，就需要提供足够的上下文信息，使 Agent 能理解工具的优点和应用场景。虽然工具的描述信息并不是必须填写的参数，我们还是强烈建议开发者视为必填参数。

其次，对于工具的选择，我们需要进行检索。这一步骤可以解决上述的问题。我们可以运行一些检索步骤，例如嵌入式搜索查找，以获取可能的工具，然后将这些工具传递给提示词模板，最后由大语言模型进行推理后续步骤。提供相关的示例也是一种有效的方法。选择与当前任务类似的示例，通常比随机示例更有帮助。相同地，检索最相关的示例也有巨大的潜力。这里面临的挑战是应用程序在底层做检索，与模型包装器交互等动作，这些中间步骤耗费的时间很长。我们并不能立马获得一个像“聊天消息”那么快的回答。

另外Agent 应用会“失忆”，处理长时间运行的任务的时候，尤其是中间步骤太多的时候，比如最开始 Agent 记得我们给它取的名字，几轮对话过后，Agent 就忘记自己的名字了。

最后，我们还需要注意避免在不需要的情况下使用工具。可以在提示中加入相关信息或提醒，告诉 Agent 在对话时不必使用工具。

Agent 实用技巧

在解决这些挑战的过程中，我们总结出了一些实用的技巧：

结构化的响应更易于解析。通常情况下，你提问的响应越结构化，解析起来就越容易。大语言模型在编写 JSON 方面表现得很好，因此我们将一些 Agent 转换为使用 JSON 格式。

我们引入了输出解析器的概念。输出解析器封装了解析响应所需的全部逻辑，并以尽可能模块化的方式实现。另一个相关的概念是，输出解析器可以重试和修复错误。如果有格式错误的模式，你可以通过将输出和错误传递给它来显式地修复响应。

记住之前的步骤也是很重要的。最基本的方法是在内存中保留这些步骤的列表。然而，在处理长时间运行的任务时，会遇到一些上下文窗口的问题。我们已经找到了一种解决方法，即使用一些检索方法来获取之前的步骤，并将其放入上下文中。

如果是在处理接口文档时，比如发起一个接口请求，返回了整个网页内容，这样会导致观察结果太长的问题。因为接口文档通常会返回非常大且难以放入上下文的 JSON 数据。常见的解决方法是对其进行解析，可以简单地将该大数据块转换为字符串，并将前 1000 个字符作为响应。

Agent 明星项目

2023年的 Agent 应用项目研发涉猎广泛，主要集中在如何改善 Agent 的各种工作方式上。下面介绍四个具有代表性的项目。

AutoGPT：AutoGPT 的目标设置有别于 ReAct Agent 的重大不同。AutoGPT 的追求在于如何增加 Twitter 的关注者数量或实现其他类似的开放性、广泛性和长期性目标。相较之下，ReAct Agent 则专注于实现短期内可量化的目标。为了实现这样的目标，AutoGPT 引入了长期记忆的概念，促进 Agent 与工具之间的互动，这有助于提升 Agent 的规划和执行效率。

Baby AGI：Baby AGI 的研发采用了逐步解决子问题的方法，以提升 Agent 的规划和执行能力。这一项目明确了策划和执行步骤的定义，这一创新为提升长期目标 Agent 的可行性和关注度提供了有益的思考途径。最初，Baby AGI 的策略实现主要依靠自主设定，然而现在已经开始融入了各种工具，从而优化 Agent 执行计划的能力。

Camel：Camel 项目的一项主要创新是在模拟环境中进行 Agent 之间的交互。通过这种方法，可以对 Agent 进行评估和测试，并且可以作为一种娱乐手段。这种方法为检测 Agent 交互提供了一种无需人工干预的方式，能够有效地测试 Agent 模型。

Generative Agents：该项目的目标是通过构建一个复杂的模拟环境，让 25 个不同的 Agent 在这个环境中进行互动，从而实现生成型 Agent 。项目同时也注重处理 Agent 的记忆和反思能力， Agent 能够通过记忆中的事件来指导下一步的行动，并在反思环节对最近的事件进行评估和更新。这种基于反思的状态更新机制适用于各种类型的记忆，例如实体记忆和知识图谱，从而提高 Agent 对环境的建模能力。

### 7.1.4 Agent 的入门示例

安装openai 和 langchain库。

pip -q install openai

pip install git+https://github.com/hwchase17/langchain

设置设置谷歌搜索的API密钥，设置OpenAI 的密钥。

os.environ["OPENAI\_API\_KEY"] = ""

os.environ["SERPAPI\_API\_KEY"] = ""

首先，让我们加载大语言模型。

from langchain.agents import load\_tools

from langchain.agents import initialize\_agent

from langchain.agents import AgentType

from langchain.llms import OpenAI

llm = OpenAI(temperature=0)

接下来，让我们加载一些要使用的工具。请注意，llm-math 工具使用了一个大语言模型接口，所以我们需要传递进去。

tools = load\_tools(["serpapi", "llm-math"], llm=llm)

最后，让我们用这些工具、大语言模型和我们想要使用的 Agent 类型来初始化一个 Agent。

agent=

initialize\_agent(tools, llm, agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION, verbose=True)

现在，让我们来测试一下吧！

agent.run("Who is Leo DiCaprio’s girlfriend? What is her current age raised to the 0.43 power? ")

看 AgentExecutor 链组件运行的结果，重点是观察 Observation 、Action 、Answer以及 Thought的变化。这就是Agent 在回答雷纳尔多所经历的中间步骤。这个中间步骤执行的便是 ReAct 框架的策略。

Entering new AgentExecutor chain… I need to find out who Leo DiCaprio’s girlfriend is and then calculate her age raised to the 0.43 power.

Action: Search Action Input: “Leo DiCaprio girlfriend”

Observation: Camila Morrone Thought: I need to find out Camila Morrone’s age Action: Search Action Input: “Camila Morrone age” Observation: 25 years

Thought: I need to calculate 25 raised to the 0.43 power Action: Calculator Action Input: 25^0.43

Observation:

Answer: 3.991298452658078

Thought: I now know the final answer Final

Answer: Camila Morrone is Leo DiCaprio’s girlfriend and her current age raised to the 0.43 power is 3.991298452658078.

Finished chain.

“Camila Morrone is Leo DiCaprio’s girlfriend and her current age raised to the 0.43 power is 3.991298452658078.”

### 7.1.5 Agent 的类型

根据任务的不同， Agent 主要有两种类型：行动 Agent 和计划执行 Agent 。

行动 Agent ：每个时间步都会根据前面所有动作的输出来决定下一步的行动。行动 Agent 适合于小型任务，它的优点在于能够实时地处理信息和作出决策。

计划执行 Agent ：首先决定全部行动序列，然后一次性执行所有的动作，而不更新计划。计划执行 Agent 更适合于需要保持长期目标和重点的复杂或长期运行的任务。

值得注意的是，常常将这两种 Agent 结合起来使用是最佳的做法，也就是说，让计划执行 Agent 使用行动 Agent 来执行计划。这样的结合，既保持了行动 Agent 的动态性，又利用了计划执行 Agent 的规划能力。

行动 Agent

Agent 利用语言大模型决定要执行哪些行动以及行动的顺序。行动可以是使用工具并观察其输出，也可以是向用户返回响应。

在深入了解各种具体的 Agent 类型之前，我们先看一下源代码中定义的 Agent 类型枚举类。这个枚举类列出了 LangChain 框架中所有可用的 Agent 类型。有了这个概念，我们就可以更好地理解以下将要介绍的各种 Agent 类型以及它们的使用场景。

class AgentType(str, Enum):  
 ZERO\_SHOT\_REACT\_DESCRIPTION = "zero-shot-react-description"  
 REACT\_DOCSTORE = "react-docstore"  
 SELF\_ASK\_WITH\_SEARCH = "self-ask-with-search"  
 CONVERSATIONAL\_REACT\_DESCRIPTION = "conversational-react-description"  
 CHAT\_ZERO\_SHOT\_REACT\_DESCRIPTION = "chat-zero-shot-react-description"  
 CHAT\_CONVERSATIONAL\_REACT\_DESCRIPTION = "chat-conversational-react-description"  
 STRUCTURED\_CHAT\_ZERO\_SHOT\_REACT\_DESCRIPTION = (  
 "structured-chat-zero-shot-react-description"  
 )  
 OPENAI\_FUNCTIONS = "openai-functions"

这些 Agent 类型是根据不同的理论依据和实践需求创建的，它们已经在源代码中实现，我们可以直接使用，无需自己从头开始。这些内置的 Agent 类型提供了一个丰富的工具集，可以满足大部分的使用场景。接下来，我们将根据 Agent 类型按照原理进行分类，并且了解它们不同的使用场景。

Zero-shot ReAct 这种 Agent 使用 ReAct 框架，仅根据工具的描述决定使用哪个工具。可以提供任意数量的工具。这种 Agent 要求为每个工具提供一个描述。

结构化输入反应（Structured input ReAct） 结构化工具聊天 Agent 能够使用多输入工具。旧 Agent 被配置为将行动输入指定为单个字符串，但是这种 Agent 可以使用工具的参数模式创建结构化的行动输入。这对于更复杂的工具使用，例如精确导航浏览器，非常有用。

OpenAI 函数（OpenAI Functions） OpenAI 模型（如 gpt-3.5-turbo-0613 和 gpt-4-0613）被明确地微调，以便检测何时应该调用函数，并响应应传递给函数的输入。OpenAI 函数 Agent 旨在与这些模型一起工作。注意目前支持的型号有限，不适用默认的模型型号，也不适用其他的模型平台的型号。

对话（Conversational） 这种 Agent 被设计用于对话设置。提示被设计为使 Agent 有帮助和交谈。它使用 ReAct 框架决定使用哪个工具，并使用记忆来记住以前的对话交互。

自问与搜索（Self ask with search） 这种 Agent 使用一个名为 Intermediate Answer 的工具。这个工具应该能够查找问题的事实答案。这个 Agent 相当于原始的 self ask with search paper 论文，其中提供了 Google 搜索 API 作为工具。

ReAct 文档存储（ReAct document store） 这种 Agent 使用 ReAct 框架与文档存储进行交互。必须提供两个工具：Search 工具和 Lookup 工具（必须精确地命名为这样）。搜索工具应该搜索一个文档，而查找工具应该在最近找到的文档中查找一个词条。这个 Agent 等同于原始的 ReAct 论文。

了解这些理论和分类的真正价值在于我们的实际应用。如果我们需要创建的 Agent 已经存在于内置类型中，那我们完全可以直接使用，无需自己从头开始实现。但是，如果我们需要自定义 Agent，那么这些分类就成了我们做决策的关键因素。

在开始创建自定义 Agent 时，我们首先需要明确这个 Agent 需要承担什么类型的任务。然后，我们需要考虑哪种类型的 Agent 能够最好地完成这个任务。在这个过程中，我们需要查看是否已经有现成的解决方案可以参考或者直接使用。

这就是为什么我们需要了解这些内置的 Agent 类型和它们各自的用途。这些内置的 Agent 类型不仅提供了丰富的选择，满足了大部分的应用场景，而且它们也为我们自定义 Agent 提供了宝贵的参考。

这些内置的 Agent 类型，例如 Zero-shot ReAct、结构化输入反应、OpenAI 函数、对话、自问与搜索以及 ReAct 文档存储，都有各自的特点和使用场景。通过对这些类型的理解和学习，我们可以更好地利用 LangChain 框架，更有效地创建和使用 Agent。

计划并执行 Agent

紧接着行动 Agent 的类型，让我们继续了解计划并执行 Agent 的类型。

计划并执行 Agent （Plan-and-Solve Agents）通过首先规划要做什么，然后执行子任务来实现目标。这个思想很大程度上受到 BabyAGI 以及 “Plan-and-Solve” 论文的启发。目前这种类型的 Agent ，比如 BabyAGI 都处于项目的雏形，LangChain 还没有这种内置Agent 的类型。

每种 Agent 类型都有其特定的用途和应用场景， Agent 的灵活性和丰富性为 LangChain 提供了强大的功能性。

上文所提到的论文资源：

ReAct 论文: https://arxiv.org/pdf/2205.00445.pdf

BabyAGI 仓库： https://github.com/yoheinakajima/babyagi

Plan-and-Solve 论文： https://arxiv.org/abs/2305.04091

self ask with search 论文：<https://ofir.io/self-ask.pdf>

## 7.2 Agent的应用实践

在 LangChain 框架中，Agent 模块实现了多种类型的 Agent，比如 ZeroShotAgent，OpenAIFunctionsAgent。另外框架鼓励开发者创建自己的 Agent。理解这些 Agent 的使用步骤，以及如何自定义 Agent 都至关重要。

首先，我们需要明白创建和运行 Agent 是两个分离的步骤。创建 Agent 是通过实例化 Agent 类（如 ZeroShotAgent，或者是你创建的）来完成的。在创建 Agent 的过程中，Tools 也会被用于提示词模板，所以无论是Agent 还是AgentExecutor的初始化过程中，都需要设置 tools。

创建好 Agent 后，我们需要将其放入 AgentExecutor 中进行运行。AgentExecutor 是 Agent 的运行环境，它是一个链组件。我们实际上运行的是这个链组件，而不是 Agent 本身。在整个 Langchain 框架中，所有模块的终点都是组合到链组件上。Agent不能脱离 AgentExecutor环境，就像鱼离不开水。在运行过程中，AgentExecutor 会调用 Tools 的方法来执行具体的任务。

以下是一个使用 ZeroShotAgent 的示例：

from langchain.agents import ZeroShotAgent, Tool, AgentExecutor  
  
llm\_chain = LLMChain(llm=OpenAI(temperature=0), prompt=prompt)  
  
tool\_names = [tool.name for tool in tools]  
agent = ZeroShotAgent(llm\_chain=llm\_chain, allowed\_tools=tool\_names)  
  
agent\_executor = AgentExecutor.from\_agent\_and\_tools(  
 agent=agent, tools=tools, verbose=True  
)  
  
agent\_executor.run("How many people live in canada as of 2023?")

在这个示例中，我们首先创建了一个 ZeroShotAgent 实例，并将其放入了 AgentExecutor 中。然后，我们调用了 AgentExecutor 的 run 方法来运行这个 Agent，并获取了其运行结果。

这是目前最通用的 Agent 的实现方法。无论是自定义还是内置的 Agent 都遵循这个使用步骤。对于内置的 Agent 类型，还有一个简化方法，简化的原因是把之前的二个步骤，简化为了一个步骤，不需要创建 Agent 实例，也不需要显式创建 AgentExecutor：

agent

= initialize\_agent( tools, llm, agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION, verbose=True )

initialize\_agent 方法被广泛用于内置的Agent 中。比如我们刚刚使用的 agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION，实际上它对应的是 ZeroShotAgent。值得注意的是如果并不在这个清单内，这个方法并不适用。以下类型均可以使用这个方法：

{

AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION: ZeroShotAgent,

AgentType.REACT\_DOCSTORE: ReActDocstoreAgent,

AgentType.SELF\_ASK\_WITH\_SEARCH:

SelfAskWithSearchAgent,

AgentType.CONVERSATIONAL\_REACT\_DESCRIPTION:

ConversationalAgent,

AgentType.CHAT\_ZERO\_SHOT\_REACT\_DESCRIPTION: ChatAgent, AgentType.CHAT\_CONVERSATIONAL\_REACT\_DESCRIPTION: ConversationalChatAgent, AgentType.STRUCTURED\_CHAT\_ZERO\_SHOT\_REACT\_DESCRIPTION:StructuredChatAgent, AgentType.OPENAI\_FUNCTIONS: OpenAIFunctionsAgent

}

下面我们通过一个实际的应用案例代码，展示一个完整的使用步骤。

### 7.2.1 多功能 Agent

设置 Agent

我们先安装要用到的库。

pip -q install langchain huggingface\_hub openai google-search-results tiktoken wikipedia

设置所需工具的密钥和大语言模型的密钥。

import os  
os.environ["OPENAI\_API\_KEY"] = ""  
os.environ["SERPAPI\_API\_KEY"] = ""

设置 Agent 的过程包含了两个主要步骤：加载 Agent 将使用的工具，然后用这些工具初始化 Agent。在代码示例中，我们首先初始化了一些基础设置，然后加载了两个工具：一个使用搜索 API 进行搜索的工具，以及一个可以进行数学运算的计算器工具。

加载工具和初始化 Agent。

from langchain.agents import load\_tools  
from langchain.agents import initialize\_agent  
from langchain.llms import OpenAI  
  
llm = OpenAI(temperature=0)

每个工具都有一个名称和描述，告诉我们它是用来做什么的。在我们的示例中，”serpapi”工具用于搜索，而”llm-math”工具则用于解决数学问题。这些工具内部有很多内容，包括模板和许多不同的 chains。

tools = load\_tools(["serpapi", "llm-math"], llm=llm)

初始化 Agent

一旦我们设置好了工具，我们就可以开始初始化 Agent。初始化 Agent 需要我们传入工具和语言模型，以及 Agent 的类型或风格。在我们的示例中，我们使用了零镜像反应性 Agent，这是基于一篇关于让语言模型采取行动并生成操作步骤的论文。

agent =

initialize\_agent(tools, llm, agent="zero-shot-react-description", verbose=True)

Agent 的提示词模板

初始化 Agent 的重要步骤之一是设置提示词模板。这些提示会在 Agent 开始运行时提示大语言模型，告诉它应该做什么。

agent.agent.llm\_chain.prompt.template

在我们的示例中，我们为 Agent 设置了两个工具：搜索引擎和计算器。然后，我们设置了 Agent 应该返回的格式，这包括它需要回答的问题，以及它应该采取的行动和行动的输入。

'Answer the following questions as best you can. You have access to the following tools:\n\nSearch: A search engine. Useful for when you need to answer questions about current events. Input should be a search query.\nCalculator: Useful for when you need to answer questions about math.\n\nUse the following format:\n\nQuestion: the input question you must answer\nThought: you should always think about what to do\nAction: the action to take, should be one of [Search, Calculator]\nAction Input: the input to the action\nObservation: the result of the action\n... (this Thought/Action/Action Input/Observation can repeat N times)\nThought: I now know the final answer\nFinal Answer: the final answer to the original input question\n\nBegin!\n\nQuestion: {input}\nThought:{agent\_scratchpad}'

Agent 的运行

最后，我们运行 Agent。需要注意的是，Agent 并不总是需要使用工具。在我们的示例中，我们问 Agent “你今天好吗？”。对于这样的问题，Agent 并不需要进行搜索或计算，而是可以直接生成回答。

agent.run("Hi How are you today?")

这就是 Langchain Agents 的基本概念和使用方法。

使用 Math 模块

我们在前半部分介绍了 Agent 的基础知识和功能。现在，我们要继续探讨如何在实际中应用 Agent，以及在什么情况下，Agent 可能遇到的问题。

agent.run("Where is DeepMind's office?")

在我们的示例中，我们尚未使用到 math 模块，让我们来看一下它的作用。我们让 Agent 查找 Deep Mind 的街道地址的数字，然后进行平方。

agent.run("If I square the number for the street address of DeepMind what answer do I get?")

Agent 首先进行搜索获取地址，然后找到了数字 5（假设为地址的一部分），最后进行平方运算，得出结果 25。然而，如果问题中包含多个数字，Agent 可能会对哪个数字进行平方产生混淆，这就是一些可能需要考虑和解决的问题。

> Entering new AgentExecutor chain...  
 I need to find the street address of DeepMind first.  
Action: Search  
Action Input: "DeepMind street address"  
Observation: DeepMind Technologies Limited, is a company organised under the laws of England and Wales, with registered office at 5 New Street Square, London, EC4A 3TW (“DeepMind”, “us”, “we”, or “our”). DeepMind is a wholly owned subsidiary of Alphabet Inc. and operates https://deepmind.com (the “Site”).  
Thought: I now need to calculate the square of the street address.  
Action: Calculator  
Action Input: 5^2  
Observation: Answer: 25  
Thought: I now know the final answer.  
Final Answer: 25  
  
> Finished chain.  
'25'

使用终端工具

在我们的工具库中，还有一个我们还未使用过的工具，那就是终端工具。例如，我们可以问 Agent 当前目录中有哪些文件。

agent.run("What files are in my current directory?")

Agent 将运行一个 LS 命令来查看文件夹，并返回一个文件列表。

> Entering new AgentExecutor chain...  
 I need to find out what files are in my current directory.  
Action: Terminal  
Action Input: ls  
Observation: sample\_data  
  
Thought: I need to find out more information about this file.  
Action: Terminal  
Action Input: ls -l sample\_data  
Observation: total 55504  
-rwxr-xr-x 1 root root 1697 Jan 1 2000 anscombe.json  
-rw-r--r-- 1 root root 301141 Mar 10 20:51 california\_housing\_test.csv  
-rw-r--r-- 1 root root 1706430 Mar 10 20:51 california\_housing\_train.csv  
-rw-r--r-- 1 root root 18289443 Mar 10 20:51 mnist\_test.csv  
-rw-r--r-- 1 root root 36523880 Mar 10 20:51 mnist\_train\_small.csv  
-rwxr-xr-x 1 root root 930 Jan 1 2000 README.md  
  
Thought: I now know the final answer.  
Final Answer: The files in my current directory are anscombe.json, california\_housing\_test.csv, california\_housing\_train.csv, mnist\_test.csv, mnist\_train\_small.csv, and README.md.  
  
> Finished chain.  
'The files in my current directory are anscombe.json, california\_housing\_test.csv, california\_housing\_train.csv, mnist\_test.csv, mnist\_train\_small.csv, and README.md.'

我们甚至可以问它特定的文件是否存在，比如询问当前目录中是否有一个关于加利福尼亚的文件。

agent.run("Does my current directory have a file about California?")

Agent 会使用 grep 命令进行检索，并向我们报告结果。然而，这个过程可能需要多次调用语言模型，从而产生一定的运行成本。

> Entering new AgentExecutor chain...  
 I need to find out what files are in my current directory.  
Action: Terminal  
Action Input: ls  
Observation: sample\_data  
  
Thought: I need to find out if there is a file about California in this directory.  
Action: Terminal  
Action Input: grep California sample\_data  
Observation: Command 'grep California sample\_data' returned non-zero exit status 2.  
Thought: I need to look for a different way to search for the file.  
Action: Search  
Action Input: "California file in current directory"  
Observation: The pwd command displays the full, absolute path of the current, or working, directory. It's not something you'll use all the time, but it can ...  
Thought: I need to use the pwd command to find the file.  
Action: Terminal  
Action Input: pwd  
Observation: /content  
  
Thought: I need to use the ls command to find the file.  
Action: Terminal  
Action Input: ls  
Observation: sample\_data  
  
Thought: I need to use the grep command to find the file.  
Action: Terminal  
Action Input: grep California sample\_data  
Observation: Command 'grep California sample\_data' returned non-zero exit status 2.  
Thought: I need to look for a different way to search for the file.  
Action: Search  
Action Input: "California file in current directory"  
Observation: The pwd command displays the full, absolute path of the current, or working, directory. It's not something you'll use all the time, but it can ...  
Thought: I need to use the find command to find the file.  
Action: Terminal  
Action Input: find . -name "\*California\*"  
Observation: Command 'find . -name "\*California\*' returned non-zero exit status 2.  
Thought: I need to use the grep command to find the file.  
Action: Terminal  
Action Input: grep -R California \*  
Observation: sample\_data/README.md:\* `california\_housing\_data\*.csv` is California housing data from the 1990 US  
  
Thought: I now know the final answer.  
Final Answer: Yes, there is a file about California in the current directory.  
  
> Finished chain.  
'Yes, there is a file about California in the current directory.'

注意事项

使用终端工具时，需要非常谨慎。你不希望最终用户能够通过运行终端命令来操作你的文件系统，因此在添加这个工具时，需要确保适当的安全防护措施已经到位。不过，尽管有其潜在风险，但在某些情况下，使用终端工具还是很有帮助的，比如当你需要设置某些功能时。

以上就是 Agent 的使用示例和注意事项。

### 7.2.2 自定义 Agent

这一节我们介绍如何创建自定义Agent 。

一个Agent由两部分组成。tools：代理可以使用的工具。AgentExecutor：决定采取哪种行动。

我们将逐步介绍如何创建一个自定义 Agent。Tool，AgentExecutor，BaseSingleActionAgent 是从 langchain.agents 模块导入的类，用于创建自定义的 Agent 和 tools。OpenAI 和 SerpAPIWrapper 是从 langchain 模块导入的类，用于访问 OpenAI 的功能和 SerpAPI 的包。先安装库。

pip -q install openai  
pip install git+https://github.com/hwchase17/langchain

设置密钥。

# 设置OpenAI的API密钥  
os.environ["OPENAI\_API\_KEY"] = ""  
# 设置谷歌搜索的API密钥  
os.environ["SERPAPI\_API\_KEY"] = ""

from langchain.agents import Tool, AgentExecutor, BaseSingleActionAgent  
from langchain import OpenAI, SerpAPIWrapper

创建一个 SerpAPIWrapper 实例，然后将其 run 方法封装到一个 Tool 对象中。

search = SerpAPIWrapper()  
tools = [  
 Tool(  
 name="Search",  
 func=search.run,  
 description="useful for when you need to answer questions about current events",  
 return\_direct=True,  
 )  
]

定义了一个自定义的 Agent 类 FakeAgent，这个类从 BaseSingleActionAgent 继承。该类定义了两个方法 plan 和 aplan，这两个方法是 Agent 根据给定的输入和中间步骤来决定下一步要做什么的核心逻辑。

from typing import List, Tuple, Any, Union  
from langchain.schema import AgentAction, AgentFinish  
  
  
class FakeAgent(BaseSingleActionAgent):  
 """Fake Custom Agent."""  
  
 @property  
 def input\_keys(self):  
 return ["input"]  
  
 def plan(  
 self, intermediate\_steps: List[Tuple[AgentAction, str]], \*\*kwargs: Any  
 ) -> Union[AgentAction, AgentFinish]:  
 """Given input, decided what to do.  
  
 Args:  
 intermediate\_steps: Steps the LLM has taken to date,  
 along with observations  
 \*\*kwargs: User inputs.  
  
 Returns:  
 Action specifying what tool to use.  
 """  
 return AgentAction(tool="Search", tool\_input=kwargs["input"], log="")  
  
 async def aplan(  
 self, intermediate\_steps: List[Tuple[AgentAction, str]], \*\*kwargs: Any  
 ) -> Union[AgentAction, AgentFinish]:  
 """Given input, decided what to do.  
  
 Args:  
 intermediate\_steps: Steps the LLM has taken to date,  
 along with observations  
 \*\*kwargs: User inputs.  
  
 Returns:  
 Action specifying what tool to use.  
 """  
 return AgentAction(tool="Search", tool\_input=kwargs["input"], log="")

创建了一个 FakeAgent 的实例。

agent = FakeAgent()

创建了一个 AgentExecutor 的实例，该实例将使用前面定义的 FakeAgent 和 tools 来执行任务。from\_agent\_and\_tools 是一个类方法，用于创建AgentExecutor 的实例。

agent\_executor = AgentExecutor.from\_agent\_and\_tools(  
 agent=agent, tools=tools, verbose=True  
)

调用 AgentExecutor 的 run 方法来执行一个任务，任务是查询 “2023 年加拿大有多少人口”。

agent\_executor.run("How many people live in canada as of 2023?")

打印最终的结果。

> Entering new AgentExecutor chain...  
 The current population of Canada is 38,669,152 as of Monday, April 24, 2023, based on Worldometer elaboration of the latest United Nations data.  
   
 > Finished chain.  
  
 'The current population of Canada is 38,669,152 as of Monday, April 24, 2023, based on Worldometer elaboration of the latest United Nations data.'

### 7.2.3 ReAct Agent 的实践

由于 ReAct 框架的特性，目前它已经成为首选的 Agent 实现方式。 Agent 的基本理念是将大语言模型当作推理的引擎。ReAct 框架实际上是把推理和动作结合在一起。当 Agent 接收到用户的请求后，大语言模型就会帮助选择使用哪个工具。接着， Agent 会执行该工具的操作，观察其结果，并把这些结果反馈给大语言模型。

下面用代码演示了如何使用 Agent 实现 ReAct 框架。

首先，让我们加载将用于控制 Agent 的 openai 和 langchain 的库。

pip -q install openai  
pip install git+https://github.com/hwchase17/langchain

设置密钥。

# 设置OpenAI的API密钥  
os.environ["OPENAI\_API\_KEY"] = ""  
# 设置谷歌搜索的API密钥  
os.environ["SERPAPI\_API\_KEY"] = ""

from langchain.agents import load\_tools  
from langchain.agents import initialize\_agent  
from langchain.agents import AgentType  
from langchain.llms import OpenAI  
llm = OpenAI(temperature=0)

接着，我们需要加载一些工具。

tools = load\_tools(["serpapi", "llm-math"], llm=llm)

请注意，llm-math 工具使用了 llm，因此我们需要输入这个模型。最后，我们需要使用 tools、llm 和我们想要使用的 Agent 类型 ZERO\_SHOT\_REACT\_DESCRIPTION 来初始化一个 Agent 。

agent =

initialize\_agent(tools, llm, agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION, verbose=True)

现在让我们来测试一下！

agent.run("Who is Leo DiCaprio's girlfriend? What is her current age raised to the 0.43 power?")

> Entering new AgentExecutor chain...  
 I need to find out who Leo DiCaprio's girlfriend is and then calculate her age raised to the 0.43 power.  
 Action: Search  
 Action Input: "Leo DiCaprio girlfriend"  
 Observation: Camila Morrone  
 Thought: I need to find out Camila Morrone's age  
 Action: Search  
 Action Input: "Camila Morrone age"  
 Observation: 25 years  
 Thought: I need to calculate 25 raised to the 0.43 power  
 Action: Calculator  
 Action Input: 25^0.43  
 Observation: Answer: 3.991298452658078  
   
 Thought: I now know the final answer  
 Final Answer: Camila Morrone is Leo DiCaprio's girlfriend and her current age raised to the 0.43 power is 3.991298452658078.  
   
 > Finished chain.  
  
  
 "Camila Morrone is Leo DiCaprio's girlfriend and her current age raised to the 0.43 power is 3.991298452658078."

除此之外，你还可以创建使用 Chat Model 模型包装器作为 Agent 驱动器的 ReAct Agent ，而不是使用 LLM 模型包装器。

from langchain.chat\_models import ChatOpenAI  
  
chat\_model = ChatOpenAI(temperature=0)  
agent = initialize\_agent(tools, chat\_model, agent=AgentType.CHAT\_ZERO\_SHOT\_REACT\_DESCRIPTION, verbose=True)  
agent.run("Who is Leo DiCaprio's girlfriend? What is her current age raised to the 0.43 power?")

## 7.3 工具和工具包

在 Agent 模块中，工具（Tools) 是 Agent 用来与世界互动的接口。这些工具实际上就是 Agent 可以使用的函数，用于与外部世界进行交互。这些工具可以是通用的实用程序（例如搜索功能），也可以是其他的工具链，甚至是其他的 Agent 。

工具包（Toolkits）是设计用于完成特定任务的工具集合，它们具有方便的加载方法。工具包将一组具有共同目标或特性的工具集中在一起，提供统一而便捷的使用方式，使得用户能够更加方便地完成特定任务。

在构建自己的 Agent 时，你需要提供一个工具列表，这些工具是 Agent 可以使用的。除了实际被调用的函数外（func=search.run），工具还包括一些组成部分：name（必需的，并且在提供给 Agent 的工具集中必须是唯一的）；description（可选的，但建议提供，因为 Agent 会用它来判断工具的使用情况）。

from langchain.agents import ZeroShotAgent, Tool, AgentExecutor

from langchain import OpenAI, SerpAPIWrapper, LLMChain

search = SerpAPIWrapper()

tools = [ Tool( name=“Search”, func=search.run, description=“useful for when you need to answer questions about current events”, )]

工具包是为特定任务而设计的工具集合，具有便捷的加载方法。它们可以被设计为一起使用，以完成特定的任务。这种设计方式提供了一种更为便捷和高效的方式来处理复杂的任务，提升了工作效率。

LangChain 封装了许多类型的工具和工具包，用户可以随时调用这些工具，完成各种复杂的任务。除了使用 LangChain 提供的工具，用户也可以自定义工具，形成自己的工具包，以完成特殊的任务。

### 7.3.1 工具的类型

Langchain 提供了一系列的工具，它们封装了各种功能，可以直接在你的项目中使用。这些工具涵盖了从数据处理到网络请求，从文件操作到数据库查询，从搜索引擎查询到大语言模型的应用，等等。这个工具清单还在不断地扩展和更新。以下是目前可用的工具列表：

“AIPluginTool”: 一个插件工具，允许用户将其他的人工智能模型或服务集成到系统中。

“APIOperation”: 用于调用外部 API 的工具。

“ArxivQueryRun”: 用于查询 Arxiv 的工具。

“AzureCogsFormRecognizerTool”: 利用 Azure 认知服务中的表单识别器的工具。

“AzureCogsImageAnalysisTool”: 利用 Azure 认知服务中的图像分析的工具。

“AzureCogsSpeech2TextTool”: 利用 Azure 认知服务中的语音转文本的工具。

“AzureCogsText2SpeechTool”: 利用 Azure 认知服务中的文本转语音的工具。

“BaseGraphQLTool”: 用于发送 GraphQL 查询的基础工具。

“BaseRequestsTool”: 用于发送 HTTP 请求的基础工具。

“BaseSQLDatabaseTool”: 用于与 SQL 数据库交互的基础工具。

“BaseSparkSQLTool”: 用于执行 Spark SQL 查询的基础工具。

“BingSearchResults”: 用于获取 Bing 搜索结果的工具。

“BingSearchRun”: 用于执行 Bing 搜索的工具。

“BraveSearch”: 用于执行 Brave 搜索的工具。

“ClickTool”: 模拟点击操作的工具。

“CopyFileTool”: 用于复制文件的工具。

“CurrentWebPageTool”: 用于获取当前网页信息的工具。

“DeleteFileTool”: 用于删除文件的工具。

“DuckDuckGoSearchResults”: 用于获取 DuckDuckGo 搜索结果的工具。

“DuckDuckGoSearchRun”: 用于执行 DuckDuckGo 搜索的工具。

“ExtractHyperlinksTool”: 用于从文本或网页中提取超链接的工具。

“ExtractTextTool”: 用于从文本或其他源中提取文本的工具。

“FileSearchTool”: 用于搜索文件的工具。

“GetElementsTool”: 用于从网页或其他源中获取元素的工具。

“GmailCreateDraft”: 用于创建 Gmail 草稿的工具。

“GmailGetMessage”: 用于获取 Gmail 消息的工具。

“GmailGetThread”: 用于获取 Gmail 线程的工具。

“GmailSearch”: 用于搜索 Gmail 的工具。

“GmailSendMessage”: 用于发送 Gmail 消息的工具。

“GooglePlacesTool”: 用于搜索 Google Places 的工具。

“GoogleSearchResults”: 用于获取 Google 搜索结果的工具。

“GoogleSearchRun”: 用于执行 Google 搜索的工具。

“GoogleSerperResults”: 用于获取 Google SERP（搜索引擎结果页面）的工具。

“GoogleSerperRun”: 用于执行 Google SERP 查询的工具。

“HumanInputRun”: 用于模拟人类输入的工具。

“IFTTTWebhook”: 用于触发 IFTTT（如果这个，那么那个）Webhook 的工具。

“InfoPowerBITool”: 用于获取 PowerBI 信息的工具。

“InfoSQLDatabaseTool”: 用于获取 SQL 数据库信息的工具。

“InfoSparkSQLTool”: 用于获取 Spark SQL 信息的工具。

“JiraAction”: 用于在 Jira 上执行操作的工具。

“JsonGetValueTool”: 用于从 JSON 数据中获取值的工具。

“JsonListKeysTool”: 用于列出 JSON 数据中的键的工具。

“ListDirectoryTool”: 用于列出目录内容的工具。

“ListPowerBITool”: 用于列出 PowerBI 信息的工具。

“ListSQLDatabaseTool”: 用于列出 SQL 数据库信息的工具。

请注意，这些工具的具体实现和功能可能会根据实际的需求和环境进行调整。

### 7.3.2 工具包的类型

Langchain 提供了一系列与各种 Agent 进行交互的工具包组件和 内置Agent ，以帮助我们快速建立解决各种问题的 Agent。create 为前缀的Agent，例如create\_json\_agent，均已配置了一系列的工具函数，调用这个函数即可运行一个 AgentExecutor 实例，被很多工具包组件中使用。本质上来说，这种方式创建的Agent，最终是作为工具包组件中的一个工具进行使用的。在这种设计模式下，开发者不需要深入了解每个Agent的内部实现细节，只需调用相应的 create 函数即可轻松获取他们需要的Agent。多数采用这种方式的是一些工具包的实例。比如，我们使用 create\_json\_agent 函数创建了一个 json\_agent，然后用它和 requests\_wrapper 来初始化 OpenAPIToolkit 类。开发者初始化 OpenAPIToolkit 类的时候，创建了一个用于创建处理 JSON 数据的 Agent。这对于开发者创建个性化的工具包组件非常有帮助。

从使用角度来说，这些create 为前缀的Agent发挥的是工具集合的作用。与其说这些 Agent 是代理，不如说它们是一组工具的集合。所以create 为前缀的Agent和其他工具包放在一块展示。在以下是目前可用的工具包和 Agent列表：

“create\_json\_agent”: 一个设计用于与 JSON 数据交互的Agent。

“create\_sql\_agent”: 一个设计用于与 SQL 数据库交互的 Agent。

“create\_openapi\_agent”: 一个设计用于与 OpenAPI 交互的 Agent。

“create\_pbi\_agent”: 一个设计用于与 Power BI 交互的 Agent。

“create\_vectorstore\_router\_agent”: 一个设计用于与 Vector Store 路由交互的 Agent。

“create\_pandas\_dataframe\_agent”: 一个设计用于与 Pandas 数据帧交互的 Agent。

“create\_spark\_dataframe\_agent”: 一个设计用于与 Spark 数据帧交互的 Agent。

“create\_spark\_sql\_agent”: 一个设计用于与 Spark SQL 交互的 Agent。

“create\_csv\_agent”: 一个设计用于与 CSV 文件交互的 Agent。

“create\_pbi\_chat\_agent”: 一个设计用于与 Power BI 聊天交互的 Agent。

“create\_python\_agent”: 一个设计用于与 Python 交互的 Agent。

“create\_vectorstore\_agent”: 一个设计用于与 Vector Store 交互的 Agent。

“JsonToolkit”: 一个用于处理 JSON 数据的工具包。

“SQLDatabaseToolkit”: 一个用于处理 SQL 数据库的工具包。

“SparkSQLToolkit”: 一个用于处理 Spark SQL 的工具包。

“NLAToolkit”: 一个用于处理自然语言应用的工具包。

“PowerBIToolkit”: 一个用于处理 Power BI 应用的工具包。

“OpenAPIToolkit”: 一个用于处理 OpenAPI 的工具包。

“VectorStoreToolkit”: 一个用于处理 Vector Store 的工具包。

“VectorStoreInfo”: 一个用于获取 Vector Store 信息的工具。

“VectorStoreRouterToolkit”: 一个用于处理 Vector Store 路由的工具包。

“ZapierToolkit”: 一个用于处理 Zapier 应用的工具包。

“GmailToolkit”: 一个用于处理 Gmail 应用的工具包。

“JiraToolkit”: 一个用于处理 Jira 应用的工具包。

“FileManagementToolkit”: 一个用于文件管理的工具包。

“PlayWrightBrowserToolkit”: 一个用于处理 PlayWright 浏览器的工具包。

“AzureCognitiveServicesToolkit”: 一个用于处理 Azure 认知服务的工具包。

这些工具包的具体功能和实现可能会根据实际的需求和环境进行调整。

## 7.4 Agent 的能力增强

随着技术的持续发展，Agent 的功能正在经历深刻的变革和完善。在这里，我们将重点探索几个关于 Agent 功能增强的核心方面。

记忆功能的扩展：为 OpenAI Functions Agent 引入记忆功能是一次重大的突破。这不仅让 Agent 记忆先前的对话内容，还使其在执行连续任务时表现得更为出色。例如，Agent 能从工具中提取三个质数，并进行相乘。在此过程中，Agent 还有能力验证中间结果，如确认其输出是否为质数。

与向量存储库的融合：为了让 Agent 更有效地与向量存储库互动，我们建议引入一个 RetrievalQA，并将其纳入整体 Agent 的工具集中。此外，Agent 还能与多个 vectordbs 进行交互，并在它们间实现路由。这使 Agent 在数据访问和处理方面具备更广的能力。

自定义 Agent 的探索：其中包括如何创建专属的 MRKL agent 以及如何构建个性化的多动作 agent。

Agent 的错误管理与优化：例如，当大语言模型面临决策困难时，我们可以借助 handle\_parsing\_errors 功能来处理解析错误。为了避免 Agent 进行过多的操作，我们可以设置迭代的最大次数。同样，为了防止 Agent 运行过久，可以设定一个执行的时间上限。

Agent 的高级特性：这包括了解 Agent 的中间操作过程以及如何为 Agent 及其工具添加记忆功能。

其他高级应用：我们探讨了如何仅流式传输 Agent 的最终输出以及如何将 OpenAI functions agent 与各种工具集成。

在这些提及的领域中，每一个都致力于提升 Agent 的功能和适用范围。但为了与实际应用更为契合，我们将尤其关注两个关键方面：为 Agent 增添记忆功能和与向量存储库的深度整合。这两个领域不仅代表了 Agent 技术的前沿趋势，同时也回应了现实中的关键需求。

### 7.4.1 增强 Agent 记忆能力

在本文中，我们将探讨如何为 OpenAI Functions Agent 添加记忆功能。OpenAI Functions Agent 是一个能够调用函数并响应函数输入的强大工具，但在默认配置下，它并不具备记忆之前对话内容的能力。然而，在许多实际应用场景中，我们希望 Agent 能够记住之前的对话内容，从而更好地为用户提供服务。

具体来说，我们将设计一个实验，通过这个实验，我们将测试大语言模型是否能记住之前的对话内容，并在后续的对话中正确使用这些信息。我们将从设计问题的角度和步骤开始，然后深入研究如何为 Agent 添加记忆功能，并测试这个功能是否能够满足我们的预期。

我们设计了一些问题，让我们创建的代理来回答。设计这个问题的目的是为了检测大语言模型是否具备记忆能力，是否可以像人类一样记住之前的对话内容。这在实际应用中极为重要，比如在聊天机器人的场景中，我们希望机器人能记住用户的名字和之前的对话内容，以便在后续的对话中为用户提供更个性化的服务。

设计这个问题的步骤如下：

1. 首先，我们通过调用  agent.run("hi")  启动一次会话。这是一个简单的问候，类似于人类的对话开场白。

2. 然后，我们通过  agent.run("my name is bob")  介绍一个名字。这是在告诉大语言模型，我们的名字是 Bob。这一步的目的是为了看是否大语言模型能记住这个信息。

3. 最后，我们通过  agent.run("whats my name")  询问大语言模型我们的名字。这是一个测试，看大语言模型是否记住了我们之前介绍的名字。

这个设计问题的角度主要是从记忆能力和对话能力出发，通过设计这样的对话流程，我们可以测试大语言模型在一次会话中是否能记住之前的信息，并在后续的对话中正确使用这些信息。

下面是为 OpenAI Functions Agent 添加记忆功能的代码示例：

首先从 langchain 模块中导入所需的各种类和函数。

LLMMathChain：这是一个类，用于创建一个能够进行数学计算的大语言模型（Large Language Model, LLM）链。LLM 链是一个特殊的模型，可以将一系列的工具和大语言模型连接起来，使得模型能够执行更复杂的任务。

OpenAI：这是一个类，用于创建一个 OpenAI 大语言模型。可以使用这个类来实例化一个 OpenAI 模型，并在后续的代码中使用它。

SerpAPIWrapper：这是一个类，用于创建一个 SerpAPI 的包装器。SerpAPI 是一个搜索引擎结果页面（Search Engine Results Page, SERP）的 API，可以使用这个类来实例化一个 SerpAPI 对象，并在后续的代码中使用它来进行搜索操作。

SQLDatabase 和 SQLDatabaseChain：这两个类用于创建和管理 SQL 数据库。SQLDatabase 是一个用于表示 SQL 数据库的类，可以使用它来实例化一个 SQL 数据库对象。SQLDatabaseChain 则是一个特殊的链，可以将 SQL 数据库和大语言模型连接起来，使得模型能够执行更复杂的数据库操作。

initialize\_agent 和 Tool：这两个函数用于初始化 Agent 和工具。initialize\_agent 函数可以用于创建一个 Agent，Tool 函数则用于创建一个工具。

AgentType：这是一个枚举类，定义了各种不同的 Agent 类型。可以使用这个类来指定要创建的 Agent 的类型。

ChatOpenAI：这是一个类，用于创建一个能够进行聊天的 OpenAI 模型。可以使用这个类来实例化一个聊天模型，并在后续的代码中使用它进行聊天操作。

from langchain import (

    LLMMathChain,

    OpenAI,

    SerpAPIWrapper,

    SQLDatabase,

    SQLDatabaseChain,

)

from langchain.agents import initialize\_agent, Tool

from langchain.agents import AgentType

from langchain.chat\_models import ChatOpenAI

创建一个工具（Tools）列表，其中每个工具都是一个 Tool 对象。Tool 对象主要包含三个属性：name，func 和 description。

llm = ChatOpenAI(temperature=0, model="gpt-3.5-turbo-0613")

search = SerpAPIWrapper()

llm\_math\_chain = LLMMathChain.from\_llm(llm=llm, verbose=True)

db = SQLDatabase.from\_uri("sqlite:///../../../../../notebooks/Chinook.db")

db\_chain = SQLDatabaseChain.from\_llm(llm, db, verbose=True)

tools = [

    Tool(

        name="Search",

        func=search.run,

        description="useful for when you need to answer questions about current events. You should ask targeted questions",

    ),

    Tool(

        name="Calculator",

        func=llm\_math\_chain.run,

        description="useful for when you need to answer questions about math",

    ),

    Tool(

        name="FooBar-DB",

        func=db\_chain.run,

        description="useful for when you need to answer questions about FooBar. Input should be in the form of a question containing full context",

    ),

]

为 Agent 添加记忆功能。在 langchain 的 Agent 框架中，记忆是由 ConversationBufferMemory 对象管理的, 根据的需求可以使用其他类型的记忆组件。

extra\_prompt\_messages：这是一个列表，用于包含一些额外的提示消息。这些消息将被添加到 Agent 的提示模板中。在这个例子中，添加了一个 MessagesPlaceholder 对象，这个对象表示一个占位符，它的值将在运行时被实际的记忆内容替换。

variable\_name：这是 MessagesPlaceholder 对象的参数，它表示占位符的变量名。在这个例子中，变量名为 "memory"，这意味着在提示模板中，{{memory}} 将被替换为实际的记忆内容。

memory：这是一个 ConversationBufferMemory 对象，用于管理 Agent 的记忆。在这个例子中，设置了 memory\_key 为 "memory"，这意味着记忆内容将存储在 "memory" 这个键下；并且设置了 return\_messages 为 True，这意味着记忆内容将包括返回的消息。

from langchain.prompts import MessagesPlaceholder

from langchain.memory import ConversationBufferMemory

agent\_kwargs = {

    "extra\_prompt\_messages": [MessagesPlaceholder(variable\_name="memory")],

}

memory = ConversationBufferMemory(memory\_key="memory", return\_messages=True)

在这段代码中，使用  initialize\_agent  函数创建了一个 Agent 实例，并为它配置了工具、大语言模型、Agent 类型、记忆组件等属性。

 tools ：这是一个  Tool  对象的列表，每个  Tool  对象都代表一个可以被 Agent 使用的工具。这些工具将被用来执行具体的任务，如搜索、计算等。

 llm ：这是之前创建的大语言模型包装器。Agent 将使用这个模型来生成语言输出，以及决定下一步的行动。

 AgentType.OPENAI\_FUNCTIONS ：这是一个枚举值，代表 Agent 的类型。在这个例子中，选择了  OPENAI\_FUNCTIONS  类型的 Agent，这种类型的 Agent 是为 OpenAI 函数模型特别设计的。

 verbose=True ：这个参数决定了是否在运行过程中打印详细的日志信息。如果设置为  True ，那么在每次 Agent 进行行动时，都会打印出详细的日志信息，这对于调试和理解 Agent 的行为非常有用。

 agent\_kwargs ：这是一个字典，用于传递额外的参数给 Agent 类。在这个例子中，传递了  extra\_prompt\_messages  参数，这个参数包含了额外的提示消息。

 memory ：这是之前创建的  ConversationBufferMemory  对象，用于管理 Agent 的记忆。这个对象将被用来存储和检索 Agent 的记忆内容。

通过这段代码，成功创建了一个具有记忆功能的 Agent。接下来，就可以使用这个 Agent 来进行对话，Agent 将能记住之前的对话内容，并可以根据这些记忆来做出决策。

agent = initialize\_agent(

    tools,

    llm,

    agent=AgentType.OPENAI\_FUNCTIONS,

    verbose=True,

    agent\_kwargs=agent\_kwargs,

    memory=memory,

)

启动一次会话。这是一个简单的问候，类似于人类的对话开场白。

agent.run("hi")

Agent 回答了我们。

    > Entering new  chain...

    Hello! How can I assist you today?

    > Finished chain.

    'Hello! How can I assist you today?'

我们通过  agent.run("my name is bob")  介绍一个名字。这是在告诉大语言模型，我们的名字是 Bob。这一步的目的是为了看是否大语言模型能记住这个信息。

agent.run("my name is bob")

    > Entering new  chain...

    Nice to meet you, Bob! How can I help you today?

    > Finished chain.

    'Nice to meet you, Bob! How can I help you today?'

询问大语言模型我们的名字。这是一个测试，看大语言模型是否记住了我们之前介绍的名字。

agent.run("whats my name")

我们看到大语言模型认识了我们之前介绍的名字：Bob

    > Entering new  chain...

    Your name is Bob.

    > Finished chain.

    'Your name is Bob.'

### 7.4.2 增强 Agent 检索能力

在本文中，我们将介绍如何将 Agent 与向量存储库进行结合。通过数据模块中的LEDVR工作流, 已经将数据导入到向量存储库中，如果希望以一种Agent的方式进行交互时，这种结合就显得尤为重要。

实现这一目标的推荐方法是首先创建一个 RetrievalQA 工具，然后将其作为整体 Agent 中的一个工具进行使用。

首先从 langchain 模块中导入所需的各种类和函数。

from langchain.embeddings.openai import OpenAIEmbeddings

from langchain.vectorstores import Chroma

from langchain.text\_splitter import CharacterTextSplitter

from langchain.llms import OpenAI

from langchain.chains import RetrievalQA

llm = OpenAI(temperature=0)

创建一个向量存储库。

from pathlib import Path

relevant\_parts = []

for p in Path(".").absolute().parts:

    relevant\_parts.append(p)

    if relevant\_parts[-3:] == ["langchain", "docs", "modules"]:

        break

doc\_path = str(Path(\*relevant\_parts) / "state\_of\_the\_union.txt")

from langchain.document\_loaders import TextLoader

loader = TextLoader(doc\_path)

documents = loader.load()

text\_splitter = CharacterTextSplitter(chunk\_size=1000, chunk\_overlap=0)

texts = text\_splitter.split\_documents(documents)

embeddings = OpenAIEmbeddings()

docsearch = Chroma.from\_documents(texts, embeddings, collection\_name="state-of-union")

实例化 RetrievalQA 链组件，运行链。

state\_of\_union = RetrievalQA.from\_chain\_type(

    llm=llm, chain\_type="stuff", retriever=docsearch.as\_retriever()

)

接着我们创建另外一个链，从在线网站上加载文档。

from langchain.document\_loaders import WebBaseLoader

loader = WebBaseLoader("https://beta.ruff.rs/docs/faq/")

docs = loader.load()

ruff\_texts = text\_splitter.split\_documents(docs)

ruff\_db = Chroma.from\_documents(ruff\_texts, embeddings, collection\_name="ruff")

ruff = RetrievalQA.from\_chain\_type(

    llm=llm, chain\_type="stuff", retriever=ruff\_db.as\_retriever()

)

正式开始创建 Agent , 导入创建  Agent 需要的所有类。

from langchain.agents import initialize\_agent, Tool

from langchain.agents import AgentType

from langchain.tools import BaseTool

from langchain.llms import OpenAI

from langchain import LLMMathChain, SerpAPIWrapper

定义可用的工具。

tools = [

    Tool(

        name="State of Union QA System",

        func=state\_of\_union.run,

        description="useful for when you need to answer questions about the most recent state of the union address. Input should be a fully formed question.",

    ),

    Tool(

        name="Ruff QA System",

        func=ruff.run,

        description="useful for when you need to answer questions about ruff (a python linter). Input should be a fully formed question.",

    ),

]

初始化一个 Agent。

agent = initialize\_agent(

    tools, llm, agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION, verbose=True

)

开始提问。测试 State of Union QA System 文档的检索。

agent.run(

    "What did biden say about ketanji brown jackson in the state of the union address?"

)

    > Entering new AgentExecutor chain...

     I need to find out what Biden said about Ketanji Brown Jackson in the State of the Union address.

    Action: State of Union QA System

    Action Input: What did Biden say about Ketanji Brown Jackson in the State of the Union address?

    Observation:  Biden said that Jackson is one of the nation's top legal minds and that she will continue Justice Breyer's legacy of excellence.

    Thought: I now know the final answer

    Final Answer: Biden said that Jackson is one of the nation's top legal minds and that she will continue Justice Breyer's legacy of excellence.

    > Finished chain.

    "Biden said that Jackson is one of the nation's top legal minds and that she will continue Justice Breyer's legacy of excellence."

再次提问。测试 ruff 文档的检索。

agent.run("Why use ruff over flake8?")

    > Entering new AgentExecutor chain...

     I need to find out the advantages of using ruff over flake8

    Action: Ruff QA System

    Action Input: What are the advantages of using ruff over flake8?

    Observation:  Ruff can be used as a drop-in replacement for Flake8 when used (1) without or with a small number of plugins, (2) alongside Black, and (3) on Python 3 code. It also re-implements some of the most popular Flake8 plugins and related code quality tools natively, including isort, yesqa, eradicate, and most of the rules implemented in pyupgrade. Ruff also supports automatically fixing its own lint violations, which Flake8 does not.

    Thought: I now know the final answer

    Final Answer: Ruff can be used as a drop-in replacement for Flake8 when used (1) without or with a small number of plugins, (2) alongside Black, and (3) on Python 3 code. It also re-implements some of the most popular Flake8 plugins and related code quality tools natively, including isort, yesqa, eradicate, and most of the rules implemented in pyupgrade. Ruff also supports automatically fixing its own lint violations, which Flake8 does not.

    > Finished chain.

    'Ruff can be used as a drop-in replacement for Flake8 when used (1) without or with a small number of plugins, (2) alongside Black, and (3) on Python 3 code. It also re-implements some of the most popular Flake8 plugins and related code quality tools natively, including isort, yesqa, eradicate, and most of the rules implemented in pyupgrade. Ruff also supports automatically fixing its own lint violations, which Flake8 does not.'

## 7.5 本章小结

在本章中，我们深入地探讨了 Agent 的定义、运行机制、现状与发展，以及如何入门。我们首先了解了 Agent 的基本定义和运行机制，包括其作为一个可以生成行动计划并执行的实体的角色，以及其如何通过调用工具和处理复杂情况来运行。我们还讨论了 Agent 的当前状态和未来发展方向，包括其在多种场景的应用以及在技术进步下的持续优化。

在实践部分，我们探索了多功能 Agent、自定义 Agent 以及 ReAct Agent 的使用。多功能 Agent 展示了 Agent 的强大能力，即在一次运行中完成多个任务；自定义 Agent 则让我们看到了用户可以根据自身需求定制 Agent 的可能性；而 ReAct Agent 则是一个强大的框架，能够帮助我们创建稳定的 Agent。

我们还介绍了工具和工具包的概念，以及它们在 Agent 中的重要角色。工具和工具包是 Agent 完成任务的重要组成部分，不同类型的工具和工具包可以帮助 Agent 应对不同类型的任务。

最后，我们讨论了如何增强 Agent 的能力，特别是如何增强其记忆能力和检索能力。增强记忆能力可以让 Agent 更好地处理连续任务，而增强检索能力则可以使 Agent 更好地利用已有的数据资源。

8. 回调处理器

随着技术的发展，无论是为用户实时显示数据，还是为开发者提供即时的系统日志，LangChain都希望系统能够在关键时刻为我们提供即时信息。这正是回调处理器（Callbacks）的用途。回调处理器允许开发者在特定事件发生时执行自定义操作，这在许多场景中都非常有用，例如日志记录、性能监控、流式处理等。

## 8.1 回调处理器的定义

在 LangChain 框架中，回调处理器是一种特殊的包装机制，允许开发者定义一系列的方法来响应不同的事件。每当特定事件被触发，相应的回调处理器方法就会被执行。举个简单的例子，想象一下每当我们的应用开始处理新的数据流时，都能自动记录日志。

回调处理器的设计目的是为了提供一个统一、模块化和可重用的机制，使开发者能够更轻松地为链和 Agent 添加各种回调功能。

下面是使用回调处理器的几个好处：

模块化与可重用性：通过定义回调处理器，你可以创建一组可重用的操作，这些操作可以轻松地在不同的LangChain实例或应用中使用。例如，如果你有多个应用都需要流式输出到WebSocket，那么使用一个统一的 WebSocketStreamingHandler 可以避免重复的代码。

灵活性：回调处理器提供了一个结构化的方式来响应各种事件，而不仅仅是流式输出。这意味着你可以为各种事件（如链开始、链结束、错误发生等）定义特定的逻辑。

与 LangChain 的其他组件紧密集成：回调处理器是为LangChain特别设计的，确保与其内部机制的兼容性和高效性。

代码清晰与维护性：通过使用专门的回调处理器，你的代码结构会更清晰。当其他开发者查看或维护你的代码时，他们可以轻松地找到和理解回调逻辑。

然而，这并不意味着必须使用回调处理器。如果你发现直接在应用逻辑中实现特定功能更方便或更适合你的需求，那么完全可以这样做。LangChain 框架的回调处理器只是提供了一个方便、统一的工具，旨在简化开发者的工作。

### 8.1.1 内置回调处理器

为了简化开发过程，LangChain提供了一系列内置的回调处理器，比如我们运行一个 Agent ，它的底层都使用到了 StdOutCallbackHandler 。例如我们在下面代码中设置 verbose=True ，在 Agent 运行时，也就是事件发生时将 Agent 的相关信息打印到标准输出（通常是控制台或命令行界面）。

import os

os.environ["OPENAI\_API\_KEY"] = "填入你的密钥"

from langchain.agents import load\_tools

from langchain.agents import initialize\_agent

from langchain.agents import AgentType

from langchain.llms import OpenAI

llm = OpenAI()

tools = load\_tools(["llm-math"], llm=llm)

agent = initialize\_agent(

    tools, llm, agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION,verbose=True

)

agent.run("9+7")

我们在命令行界面就可以看到以 Entering new AgentExecutor chain... 开始，以 Finished chain 结尾的标准输出。这是因为当运行 Agent 并启用 verbose=True 时，StdOutCallbackHandler 将自动激活，将 Agent 的活动实时打印到标准输出。这为开发者提供了即时的反馈，帮助他们了解Agent的工作情况。

> Entering new AgentExecutor chain...

 I need to add two numbers together

Action: Calculator

Action Input: 9+7

Observation: Answer: 16

Thought: I now know the final answer

Final Answer: 9+7 = 16

> Finished chain.

'9+7 = 16'

但是如果设置为 verbose=False ，我们在命令行界面只会看到大语言模型最终的回答。

'9+7 = 16'

为 LLMChain 添加一个内置回调处理器

我们还可以给链组件和 Agent 添加内置或者自定义的回调处理器。比如我们给一个基础的链组件 LLMChain 添加一个内置回调处理器： StdOutCallbackHandler。我们可以先创建一个链组件。在初始化 LLMChain 的时候，这个链组件没有设置内置回调处理器，也不能设置 verbose=True。

from langchain.prompts import PromptTemplate

from langchain.chains import LLMChain

from langchain.llms import OpenAI

llm\_chain = LLMChain(llm=OpenAI(),

                     prompt=PromptTemplate.from\_template("{input}"))

llm\_chain.run('上海的旅游景点有哪些？')

运行 LLMChain 后，大语言模型回答的是：

上海的旅游景点有：\n\n1. 上海迪士尼乐园\n2. 东方明珠塔\n3. 南京路步行街\n4. 上海外滩\n5. 上海野生动物园\n6. 外白渡桥\n7. 南京路商业街\n8. 上海科技馆\n9. 上海老城隍庙\n10. 上海博物馆\n11. 上海浦江夜游\n12. 上海水上乐园\n13. 上海徐汇森林公园\n14. 上海金茂大厦\n...

如果我们想要监控这个链组件，添加一些回调逻辑，比如我们想要命令行输出这个链运行的相关信息。我们可以给这个链组件增加一个回调处理器。这里我们导入内置的 StdOutCallbackHandler，并且创建创建它的实例 handler\_1 。

from langchain.callbacks import StdOutCallbackHandler

handler\_1 = StdOutCallbackHandler()

回调处理器的使用很简单，我们给 LLMChain 链组件添加 callbacks=[handler\_1], 列表的数据格式这意味着我们是可以添加多个回调处理器，完成不同的任务逻辑。

llm\_chain = LLMChain(llm=OpenAI(),callbacks=[handler\_1],

                     prompt=PromptTemplate.from\_template("{input}"))

llm\_chain.run('上海的旅游景点有哪些？')

这样我们就给一个 LLMChain 链组件添加了一个回调处理器，内置的 StdOutCallbackHandler 完成的是做标准的链组件的打印输出。当我们继续运行这个链组件之后，我们在命令行界面就可以看到以 Entering new AgentExecutor chain... 开始，以 Finished chain 结尾的标准输出。

> Entering new LLMChain chain...

Prompt after formatting:

上海的旅游景点有哪些？

> Finished chain.

'\n\n上海的旅游景点有：\n\n1. 东方明珠广播电视塔\n2. 豫园\n3. 外滩\n4. 南京路步行街\n5. 上海野...'\n\n

### 8.1.2 自定义回调处理器

在开发过程中，我们常常遇到一些特定的需求，例如为每个用户请求单独创建日志文件，或在发生某一关键事件时及时发送通知。这些需求超出了通常的范围，而自定义回调处理器在此时发挥了其重要的作用。尽管LangChain提供的内置回调处理器功能强大，但有时仍然不能满足某些特定场景的需求。为了真正充分利用回调处理器，我们需要设计并实现自己的处理器。

每一个链，从它被创建到最终被销毁，都会经历多个关键阶段。在这个生命周期中，链的每一个阶段都可能会触发某些事件。为了确保在恰当的时机介入并执行相应的操作，我们需要深入了解每一个阶段并对其进行精确的控制。这样，当链经历某个特定的阶段时，我们就可以在相应的回调处理器方法中执行预定的代码。

在某些情况下，我们可能只希望在特定的请求中执行特定的代码，而不是在链的整个生命周期中。这种需求可以通过请求回调来实现。请求回调为我们提供了这种特定于请求的灵活性。例如，想象一个场景，你只希望在某个特定的请求中记录日志，而不是在所有的请求中都这样做。通过使用请求回调，你可以轻松地达到这个目的。

不过，无论我们如何设计和使用回调处理器，最关键的部分始终是理解自己的业务逻辑需求。只有当我们清晰地知道自己想要达到的目的时，我们才能在正确的回调处理器方法中插入合适的代码，确保在合适的时机执行正确的操作。自定义回调处理器为我们提供了一个强大的框架，但如何充分利用这个框架，最终取决于我们对业务需求的理解和技术的运用。

BaseCallbackHandler详解

在编写自定义回调处理器之前，了解其背后的基础类是非常重要的。 BaseCallbackHandler 正是这样一个核心类，它为我们提供了一个强大而灵活的框架，使我们可以轻松地响应和处理各种事件。这个类定义了一系列的方法，每一个都与LangChain中的一个特定事件相对应。只有深入理解了这些事件和方法，我们才能有效地为我们的应用编写自定义的回调处理器。

**1. LLM事件**

- on\_llm\_start : 当LLM启动并开始处理请求时，这个方法会被调用。这为我们提供了一个机会，例如，初始化某些资源或记录开始时间。

- on\_llm\_new\_token : 每当LLM生成一个新的令牌时，这个方法就会被执行。这在流式处理中特别有用，允许我们实时捕获和处理每一个生成的令牌。

- on\_llm\_end : 当LLM完成其任务并生成了完整的输出时，这个方法会被调用。这为我们提供了一个机会进行清理操作或记录任务的完成时间。

- on\_llm\_error : 如果在LLM处理过程中发生任何错误，这个方法将被执行。我们可以在这里添加错误日志或执行其他的错误处理操作。

**2. 聊天模型事件**

- on\_chat\_model\_start : 这个方法在Chat Model 类包装器开始其工作时被调用，为我们提供了一个机会进行初始化操作或其他准备工作。

**3. 链事件**

- on\_chain\_start : 当链开始执行时，这个方法会被调用。我们可以在这里进行一些初始化操作。

- on\_chain\_end : 在链完成所有任务后，这个方法会被执行。这为我们提供了一个机会进行清理操作或收集结果。

- on\_chain\_error : 如果链在执行过程中遇到错误，这个方法将被调用。这允许我们进行错误处理或日志记录。

**4. 工具事件**

- on\_tool\_start : 当工具开始其任务时，这个方法会被调用。

- on\_tool\_end : 在工具成功完成其任务后，这个方法会被执行。

- on\_tool\_error : 如果工具在执行过程中发生错误，这个方法将被执行。

**5. 其他事件**

- on\_text : 当需要处理任意文本时，这个方法会被调用。这为我们提供了一个机会对文本进行处理或分析。

- on\_agent\_action : 当代理执行某个特定的操作时，这个方法会被执行。

- on\_agent\_finish : 在代理完成其所有操作后，这个方法会被调用。

BaseCallbackHandler 为我们提供了一个强大的框架，使我们可以轻松地定义和处理各种事件。深入理解这个基础类是编写自定义回调处理器的关键。

自定义回调处理器

要 自定义自己的回调处理器，你可以继承 BaseCallbackHandler 并重写你需要的方法。例如，如果你想在大语言模型输出开始时打印一条消息，可以这样做：

class MyCallbackHandler(BaseCallbackHandler):

    def on\_llm\_start(self, serialized, prompts, kwargs):

        print("LLM has started!")

使用回调处理器

一旦你 自定义了自己的回调处理器，就可以将其传递给LangChain，以便在相应的事件发生时执行你的方法。

例如：

handler = MyCallbackHandler()

chain = LLMChain(llm=llm, prompt=prompt, callbacks=[handler])

## 8.2本章小结

在本章中，我们深入探讨了LangChain的回调处理器机制及其在实际应用中的价值。

回调处理器在LangChain中扮演着至关重要的角色。它是一种特殊的机制，允许开发者为不同的事件定义相应的方法。当LangChain中的某个特定事件被触发时，与之对应的回调处理器方法就会自动执行。这种自动响应特定事件的能力，使得开发者可以在合适的时机介入，执行各种操作，如日志记录、数据处理等。

我们通过代码示例详细介绍了如何使用LangChain内置的回调处理器，如StdOutCallbackHandler，AimCallbackHandler，WandbCallbackHandler等。这些内置处理器涵盖了各种常见的开发需求，从简单的日志记录到与第三方服务的集成，都可以轻松实现。

当然，LangChain的真正魔法在于它允许开发者创建自己的自定义回调处理器。我们详细探讨了如何设计并实现自定义的回调处理器，特别强调了理解链的生命周期和请求回调的重要性。深入BaseCallbackHandler这个基础类，我们了解到它定义了多个与LangChain事件相对应的方法，为开发者提供了一个强大的框架来响应和处理这些事件。

。

9.

在探索和学习新技术时，Langchain 框架的理论知识固然重要，但无疑，实际的案例分析与实践尝试能为我们提供更加直观的认识和更深入的理解。

这一章主要以综合案例代码解析为主，通过具体的实践操作，我们可以更好地理解 Langchain 技术的本质，看到各个模块如何协同工作，以及如何在实际应用中发挥其价值。

三个精选案例分别是：与本地电脑PDF文档对话的 chatPDF 机器人，高效的 对话式表单填写机器人 ，以及当前炙手可热的Agent项目 BabyAGI。这三个项目不仅体现了Langchain在LLM应用程序中的应用潜力，更重要的是，它们将为你一步步展示如何将Langchain的核心模块——模型I/O模块、Chain链模块、记忆模块、数据连接模块以及Agent模块，融合到实际应用中。

值得注意的是，这些案例代码主要是为了教学和解释目的，它们可能并不适用于真实的生产环境。还有可能在你的电脑环境中运行案例的代码，比如打印文本切割的块数，得到了与案例代码不一样的数值结果。比如案例中拆分出446个块，而你拆分出448个块。这种差异可能是由于以下几点原因：1）文档内容存在微小差异，如额外的空白或换行；2）两环境中的库版本有所不同；3） chunk\_overlap等参数导致的边界效应；4）Python或其他库的版本差异。相同地，在执行相同的查询代码时，大语言模型可能会给出略有不同的答案。这种现象的背后原理与模型的工作机制有关。大语言模型，如GPT系列，是基于概率的模型，它预测下一个词的可能性是基于训练数据中的统计信息。当模型为我们生成文本时，它实际上是在每个步骤中做出基于概率的决策，所以你在执行案例的代码查询的时候可能会得到稍微与案例不同的答案。

## 9.1 PDF 问答程序

PDF 问答程序是引入外部数据集对大语言模型进行微调，以生成更准确的回答的程序。试想你是一个航天飞机设计师，你需要了解最新的航空材料技术，你可以将几百页的航空技术文档输入到的大语言模型中，模型就会根据最新的数据集给出准确的答案，你不用看完整套材料，而是根据自己的经验提出问题，获得你想要知道的技术知识。

PDF 问答程序界面呈现的是人类与文档问答的聊天，但实质上，仍然是在与大语言模型交流，只不过这个模型现在被赋予了接入外部数据集的能力。就像你在与一位熟悉你公司内部文档的同事交谈，尽管他可能并未参与过这些文档的编写，但他可以准确地回答你的问题。

大语言模型之前，不能像聊天一样与文档交流，只能依赖于搜索。例如, 你正在为一项重要的报告寻找资料，你必须知道你需要查找的关键词，然后在大量的信息中筛选出你需要的部分。而现在，可以通过聊天的方式，即使不知道具体的关键词，也可以让模型根据的问题告诉答案。就好像你在问一位专业的图书馆员，哪些书籍可以帮助你完成这份报告。

那为什么要引入文档的外部数据集呢？这是因为大语言模型的训练数据截止到 2021 年 9 月，之后产生的知识和信息并未被包含进去。就像的模型是一个生活在过去的时间旅行者，他只能告诉你他离开的那个时刻之前的所有信息，对之后的事情一无所知。

引入外部数据集还有一个重要的目的，那就是修复大语言模型的 “机器幻觉”，避免给出错误的回答。试想一下，如果你向一个只知道过去信息的人询问未来的趋势，他可能会基于过去的信息进行推断，但这样的答案未必正确。所以通过引入最新的数据，让的模型能够更准确地回答问题，避免因为信息过时产生的误导。

另外，现在普遍使用的数据文档形式包括 Pdf、JSON、Word、Excel 等，这些都是获取实时知识和数据的途径。同时这类程序现在非常受欢迎，比如最著名的 Chat PDF 和 ChatDOC, 还有针对各种特定领域的程序，如针对法律文档的程序。就像你在阅读各种格式的书籍一样，不同的程序能够提供不同的知识和信息。

以上就是选择PDF 问答程序为案例的原因。

### 9.1.1 程序流程

PDF 问答程序的实现方式是利用 Langchain 已实现的向量存储、嵌入以及使用查询和检索相关的链，来获取外部数据集，处理文档，进行相关性检索后合并处理，置入大语言模型的提示模板中，实现与 PDF 文件交流的目的。

选定的文档是 Reid Hoffman 写的一本关于 GPT-4 和人工智能的书，将下载这本 PDF 并将其转化为可查询和交互的形式。

连接这本 PDF 文档数据使用的是LEDVR 工作流管理，最后我们使用内置的RetrievalQA 问答链和load\_qa\_chain 方法构造文档链组件，并且使用不同的文档合并链 Stuff 和 Map re-rank，对比答案的质量。

LEDVR 工作流

L - 加载器 首先，选择的文档是Reid Hoffman写的一本关于GPT-4和人工智能的书。为了使这本PDF文档能够与我们的系统进行互动和查询，我们需要首先通过加载器从本地获取这份数据。加载器为我们提供了从各种来源（例如PDF格式）获取数据的通道，并为后续步骤做好准备。

E - 嵌入模型包装器 接下来，我们需要处理这份PDF文档的内容。首先，通过嵌入模型包装器，我们将文档中的每一段文字转化为一个高维的向量。这一步的目的是捕获文本的语义信息，以便于后续的查询和检索。

D - 文档转换器 有了每段文字的嵌入向量后，文档转换器进一步处理这些向量，使其适应我们的查询需求。这个环节主要是切割文本，转换文档对象格式，如果文档过长，文档转换器可以将其分割成更小的段落。

V - 向量存储库 处理好的向量将被存储在向量存储库中。这是一个专为高维向量设计的存储系统，它允许我们快速地查找和检索向量，为后续的查询提供了极大的便利。

R - 检索器 最后，当用户想要查询某个特定的信息时，检索器就会进入工作状态。检索器会将用户的查询问题转化为一个嵌入向量，并在向量存储库中寻找与之最匹配的文档向量。找到最相关的文档后，检索器会返回文档的内容，满足用户的查询需求。

创建链

采用RetrievalQA内置的问答链结合load\_qa\_chain方法来搭建文档链部件。并通过对比Stuff与Map re-rank这两种不同的文档整合链来评估答案的优劣。

### 9.1.2 LEDVR 工作流

我们首先安装了所需的Python库来为后续的操作打基础。

!pip -q install langchain openai tiktoken PyPDF2 faiss-cpu

这里，我们安装了 langchain , openai , tiktoken , PyPDF2 和 faiss-cpu 这五个库。其中， openai 是OpenAI的官方库，使我们能与其API进行交互。 tiktoken 是用于计算字符串中的token数的工具， PyPDF2 允许我们处理PDF文件，而 faiss-cpu 是一个高效的相似性搜索库。我们为OpenAI设置了API密钥：

import os

os.environ["OPENAI\_API\_KEY"] = ""

通过这段代码，我们将API密钥存储在环境变量中，这使得我们在后续的代码中能够方便地使用这个密钥，而不必每次都手动输入它。我们下载了一个PDF文档：

!wget -q https://www.impromptubook.com/wp-content/uploads/2023/03/impromptu-rh.pdf

这里，我们使用 wget 工具从 www.impromptubook.com 下载了一个名为 impromptu-rh.pdf 的文件。这个文件在后续的代码中会被用到，比如进行文本分析或其他处理。如果没有wget 工具可以直接从网站上手动下载。

为了从PDF文件中提取内容，我们需要一个PDF阅读器。虽然这里我们选择了一个基础的PDF阅读器，但在实际应用中，可能需要根据具体需求选择更复杂或专业的PDF处理库。处理PDF文档时，可能会遇到格式问题或其他意外情况，因此选择合适的工具和方法是很重要的。不同的项目或数据源可能需要不同的处理方法，这也是为什么有时会考虑使用更高级的工具或服务，比如AWS、Google Cloud的相关API。

为了处理PDF和后续的操作，我们导入了以下库和工具：

from PyPDF2 import PdfReader

from langchain.embeddings.openai import OpenAIEmbeddings

from langchain.text\_splitter import CharacterTextSplitter

from langchain.vectorstores import FAISS

PdfReader 是我们选择的PDF阅读器，它来自 PyPDF2 库，可以帮助我们从PDF文件中读取内容。 OpenAIEmbeddings 可能用于嵌入或转换文本数据， CharacterTextSplitter 可能是用于处理或分割文本的，而 FAISS 是一个高效的相似性搜索库，后续可能用于文本或数据的搜索和匹配。

我们加载了之前下载的PDF文件：

doc\_reader = PdfReader('/content/impromptu-rh.pdf')

通过使用 PdfReader ，我们将PDF文件的内容加载到 doc\_reader 变量中。这一步的目的是读取PDF文件并为后续的文本提取做准备。

为了验证我们是否成功加载了PDF文件，我们可以打印 doc\_reader ，得到的输出是这个对象在内存中的地址：<PyPDF2.\_reader.PdfReader at 0x7f119f57f640>，这表明 doc\_reader 已经成功创建并包含了PDF文件的内容。

紧接着，我们从PDF文件中提取文本，这部分代码的目的是遍历PDF文件中的每一页，并使用 extract\_text() 方法提取每一页的文本内容，然后将这些文本内容累加到 raw\_text 变量中。

raw\_text = ''

for i, page in enumerate(doc\_reader.pages):

text = page.extract\_text()

if text:

raw\_text += text

为了验证我们是否成功从PDF文件中提取了文本，我们打印了 raw\_text 变量的长度，得到的结果是356710。这告诉我们，我们成功地从PDF文件中提取了大约356710个字符的文本。请注意，。

文本拆分的方法很简单，就是将这个长字符串按照字符数拆分。比如可以设定每 1000 个字符为一个块, chunk\_size = 1000 。

# Splitting up the text into smaller chunks for indexing

text\_splitter = CharacterTextSplitter(

    separator = "\n",

    chunk\_size = 1000,

    chunk\_overlap  = 200, #striding over the text

    length\_function = len,

)

texts = text\_splitter.split\_text(raw\_text)

总共切了 448 个块：

len(texts) # 448

注意： 在这个代码片段中， chunk\_overlap 参数用于指定文本切分时的重叠量（overlap）。它表示在切分后生成的每个分块之间重叠的字符数。具体来说，这个参数表示每个分块的前后，两个分块之间会有多少个字符是重复的。举例来说 chunkA 和 chunkB, 他们有 200 个字符是重复的。

然后，采用滑动窗口的方法来拆分文本。即每个块之间会有部分字符重叠，比如在每 1000 个字符的块上，让前后两块有 200 个字符重叠。这样做的目的是避免关键信息被切分，而且即使有些信息出现在了多个块中，因为是在获取整体语义，所以这些重叠的块在语义上也会有所区别。

可以随机打印一块的内容：

texts[20]

输出是：

'million registered users. \nIn late January 2023, Microsoft1—which had invested $1 billion \nin OpenAI in 2019—announced that it would be investing $10 \nbillion more in the company. It soon unveiled a new version of \nits search engine Bing, with a variation of ChatGPT built into it.\n1 I sit on Microsoft’s Board of Directors. 10Impromptu: Amplifying Our Humanity Through AI\nBy the start of February 2023, OpenAI said ChatGPT had \none hundred million monthly active users, making it the fast-\nest-growing consumer internet app ever. Along with that \ntorrent of user interest, there were news stories of the new Bing \nchatbot functioning in sporadically unusual ways that were \nvery different from how ChatGPT had generally been engaging \nwith users—including showing “anger,” hurling insults, boast-\ning on its hacking abilities and capacity for revenge, and basi-\ncally acting as if it were auditioning for a future episode of Real \nHousewives: Black Mirror Edition .'

我们将关注如何将提取的文本转化为机器学习可以理解的格式，并如何使用这些数据进行搜索匹配。为了理解和处理文本，我们需要将其转化为数值向量。在这里，选择了使用OpenAI的嵌入模型来为文本创建嵌入向量。

我们需要使用这些嵌入来为我们之前从PDF文件中提取的文本创建向量表示。并且为了能够高效地在这些向量中搜索和匹配，我们使用了FAISS库。

# Download embeddings from OpenAI

embeddings = OpenAIEmbeddings()

先把文本传给嵌入模型包装器 OpenAIEmbeddings，然后通过 FAISS 库创建向量存储本身。

docsearch = FAISS.from\_texts(texts, embeddings)

通过上面的代码，我们将原本的PDF文档内容转化为了机器学习可以理解和处理的向量数据，基于文本的向量表示，我们就可以进行高效的搜索和匹配了。

相似度检索是其中的一种方法。为了展示如何使用这种方法，我们选择了一个实际的查询：“GPT-4 如何改变了社交媒体？”。

query = "GPT-4 如何改变了社交媒体?"

docs = docsearch.similarity\_search(query)

我们将查询传递给similarity\_search方法，在向量数据中 docsearch 查找与查询最匹配的文档。这种搜索基于向量之间的相似度。这个搜索的结果是一个数组，其中包含了与查询最匹配的文档。

len(docs)

运行上面的代码，我们发现结果为4，这意味着有四处文档与查询有关。为了验证搜索的准确性，我们可以尝试查看第一个匹配的文档。

docs[0]

在搜索结果中，首位的文档多次提到了“社交媒体” （下文中的 Social media），这证明了我们的查询效果非常好，并且嵌入和相似度搜索的方法都是有效的。

Document(page\_content='rected ways that tools like GPT-4 and DALL-E 2 enable.\nThis is a theme I’ve touched on throughout this travelog, but \nit’s especially relevant in this chapter. From its inception, social \nmedia worked to recast broadcast media’s monolithic and \npassive audiences as interactive, democratic communities, in \nwhich newly empowered participants could connect directly \nwith each other. They could project their own voices broadly, \nwith no editorial “gatekeeping” beyond a given platform’s terms \nof service.\nEven with the rise of recommendation algorithms, social media \nremains a medium where users have more chance to deter -\nmine their own pathways and experiences than they do in the \nworld of traditional media. It’s a medium where they’ve come \nto expect a certain level of autonomy, and typically they look for \nnew ways to expand it.\nSocial media content creators also wear a lot of hats, especially \nwhen starting out. A new YouTube creator is probably not only', metadata={})

现在只有一个 PDF 文档，实现代码也很简单，Langchain 提供了LEDVR 工作流管理，完成得很快。接下来，处理多文档的提问，现实是要获取到真实的信息，通过会跨越多个文档，才能提取有用的信息。比如读取金融研报，新闻综合报道等等。

### 9.1.3  创建问答链

在上一节中，加载了一个 PDF 文档，转化格式，切分字符后，创建向量数据来进行搜索匹配获得了问题的答案。一旦有了已经处理好的文档，就可以开始构建一个简单的问答链。现在看看 如何使用 Langchain 构建问答链。

在这个过程中，选择了内置的文档处理链中一种被称为stuff的链类型。在Stuff模式下，将所有相关的文档内容都全部提交给大语言模型处理，默认情况下，放入的内容应该少于 4000 个标记。除了 Stuff 之外，合并文档链还有 Refine、Map reduce 、重排（Map re-rank）。Map re-rank后面会用到。

from langchain.chains.question\_answering import load\_qa\_chain

from langchain.llms import OpenAI

chain = load\_qa\_chain(OpenAI(), chain\_type="stuff")

下一步，要构建的查询。首先，使用向量存储中返回的内容作为上下文片段来回答的问题。然后，将这个查询传给语言模型链。语言模型链会回答这个查询，给出相应的答案。例如，可能会问 "这本书是哪些人创作的？"，然后将该查询传递给向量存储进行相似性搜索。系统会返回最相似的四个文档，将这些文档传递给语言模型链并给出查询，然后系统会给出一个答案。

query = "这本书是哪些人创作的？"

docs = docsearch.similarity\_search(query)

chain.run(input\_documents=docs, question=query)

看看他回答了什么:

' 不知道'

默认情况下，系统会返回四个最相关的文档，但可以更改这个数字。例如，可以设置返回前六个或更多的搜索结果。

query = "这本书是哪些人创作的？"

docs = docsearch.similarity\_search(query,k=6)

chain.run(input\_documents=docs, question=query)

然而，需要注意的是，如果设置返回的文档数量过多, 比如设置 k=20 ，那么总的标记数可能会超过模型平台的最大上下文长度，导致错误。例如，你使用的模型的最大上下文长度为 4097，但如果请求的标记数超过了 5000，系统就会报错。

设置返回的文档数量为 k=6 ，获取的结果是：

'这本书的作者是Reid Hoffman和Sam Altman。'

在这种情况下，如果相关文档的内容多一些，答案会更加准确一些。设置的返回搜索结果数量越少，意味着大语言模型获取到的相关信息也越少。之前我们仅仅返回了4条结果，很少的信息导致它回答了“不知道”。而修改返回的结果数量为6条时，它找出了本书的作者Reid Hoffman。它还提到了Sam Altman，实际上Sam Altman并不是作者。这种错误，可能原因是使用了低级的模型型号，默认LLM 类模型包装器是“text-davinci-003”型号，这个型号的能力远不如GPT-4。

进阶 map\_rerank 链

stuff 类型优势是把所有内容都放在一起的地方。任何时候可以使用 stuff，最好就使用它，通用且节省成本。还可以使用 map\_reduce 在并行计算中对每个文档进行操作，但这可能会导致对 API 进行过多的调用，增加成本。这里使用 map\_rerank 这种类型，提高查询的质量。

从提出更复杂的查询开始。比如说，想要知道 "OpenAI 的创始人是谁?"，并且想要获取前 10 个最相关的查询结果。在这种情况下，OpenAI 会返回多个答案，而不仅仅是一个。可以看到它不只返回一个答案，而是根据的需求返回了每个查询的答案和相应的评分。

from langchain.chains.question\_answering import load\_qa\_chain

chain = load\_qa\_chain(OpenAI(),

                      chain\_type="map\_rerank",

                      return\_intermediate\_steps=True

                      )

query = "OpenAI 的创始人是谁?"

docs = docsearch.similarity\_search(query,k=10)

results = chain(

{"input\_documents": docs, "question": query}, return\_only\_outputs=True)

重要的参数是 return\_intermediate\_steps=True , 设置这个参数可以看到 map\_rerank 是如何对检索到的文档进行打分的。

返回的每个查询结果进行了评分。比如说，OpenAI 在这本书中被多次提及，因此它的评分可能会有 80 分，90 分甚至 100 分。可以假设 OpenAI 可能选择了评分为 100 分的两个或三个查询，然后将它们合并，最终给出了的输出。

{'intermediate\_steps': [{'answer': ' This document does not answer the question.',

   'score': '0'},

  {'answer': ' OpenAI 的创始人是 Elon Musk, Sam Altman, Greg Brockman 和 Ilya Sutskever。 ',

   'score': '100'},

  {'answer': ' This document does not answer the question. ', 'score': '0'},

  {'answer': ' This document does not answer the question.', 'score': '0'},

  {'answer': ' This document does not answer the question.', 'score': '0'},

  {'answer': ' This document does not answer the question', 'score': '0'},

  {'answer': ' OpenAI 的创始人是 Elon Musk、 Sam Altman、 Greg Brockman、 Ilya Sutskever、Wojciech Zaremba 和 Peter Norvig。',

   'score': '100'},

  {'answer': ' This document does not answer the question.', 'score': '0'},

  {'answer': ' This document does not answer the question.', 'score': '0'},

  {'answer': ' This document does not answer the question', 'score': '0'}],

 'output\_text': ' OpenAI 的创始人是 Elon Musk, Sam Altman, Greg Brockman 和 Ilya Sutskever。 '}

评分后，模型输出一个最终的答案, 'score': '100' 得分 100 的那个答案：

results['output\_text']

' OpenAI 的创始人是 Elon Musk, Sam Altman, Greg Brockman 和 Ilya Sutskever。 '

为了搞清楚为什么模型会评分，做出判断，可以打印 prompt 提示模板：

# check the prompt

chain.llm\_chain.prompt.template

从提示词模板内容可以看出，为了确保语言模型能够在接到问题后提供准确和有用的答案，Langchain为模型设计了一套详细的提示词。该提示词描述了如何根据给定的背景信息回答问题，并如何为答案打分。提示词开始强调了整体目标：使模型能够根据给定的背景信息提供准确答案，并为其答案打分。（行1-3）。

模型需要明白其核心任务：根据给定的背景信息回答问题。如果模型不知道答案，它应直接表示不知道，而不是试图编造答案。这部分提醒模型，如果不知道答案，应该直接表示不知道，而不是编造答案。（行1）。

接下来，为模型提供了答案和评分的标准格式。答案部分要求模型简洁、明确地回答问题，而评分部分则要求模型为其答案给出一个0到100的分数，用以表示答案的完整性和准确性。这部分明确了答案和评分的格式，并强调了答案的完整性和准确性。（行7-10）。

通过三个示例，模型可以更好地理解如何根据答案的相关性和准确性为其打分。（行17-48）。示例，强调了答案的完整性和准确性是评分的核心标准。

最后，为了使模型能够在具体的实践中应用上述提示词，为模型提供了一个上下文背景和用户输入问题的模板。当模型接到一个问题时，它应使用此模板为问题提供答案和评分。（行50-57）。下面是格式化和翻译过后的提示词模板。

1. 当你面对以下的背景信息时，如何回答最后的问题是关键。如果不知道答案，直接说你不知道，不要试图编造答案。

2.

3. 除了提供答案外，还需要给出一个分数，表示它如何完全回答了用户的问题。请按照以下格式：

4.

5. 问题：[这里的问题]

6.

7. 有帮助的答案：[这里的答案]

8.

9. 分数：[分数范围在0到100之间]

10.

11. 如何确定分数：

12.    - 更高的分数代表更好的答案

13.    - 更好的答案能够充分地回应所提出的问题，并提供足够的细节

14.    - 如果根据上下文不知道答案，那么分数应该是0

15.    - 不要过于自信！

16.

17. 示例 #1

18.

19. 背景：

20.    - 苹果是红色的

21.

22. 问题：苹果是什么颜色？

23.

24. 有帮助的答案：红色

25.

26. 分数：100

27.

28. 示例 #2

29.

30. 背景：

31.    - 那是夜晚，证人忘了带他的眼镜。他不确定那是一辆跑车还是SUV

32.

33. 问题：那辆车是什么类型的？

34.

35. 有帮助的答案：跑车或SUV

36.

37. 分数：60

38.

39. 示例 #3

40.

41. 背景：

42.    - 梨要么是红色的，要么是橙色的

43.

44. 问题：苹果是什么颜色？

45.

46. 有帮助的答案：这个文档没有回答这个问题

47.

48. 分数：0

49.

50. 开始！

51.

52. 背景：

53.    - {context}

54.

55. 问题：{question}

56.

57. 有帮助的答案：

|  |
| --- |
|  |

格式化和翻译的提示词模板

RetrievalQA 链

RetrievalQA 链是 Langchain 已经封装好的索引查询问答链。实例化之后，可以直接把问题扔给它，简化了很多步骤，获得了比较稳定的查询结果。

为了创建这样的链，需要一个检索器。可以使用之前设置好的 docsearch, 作为检索器，并且可以设置返回的文档数量 "k":4 。

docsearch = FAISS.from\_texts(texts, embeddings)

可以将这些参数传递给链组件类型 stuff，它会为返回源文档。（选择 stuff 类型的原因：跟第一个 stuff 类型 和 map\_reduce 类型对比答案的质量）。

from langchain.chains import RetrievalQA

retriever =

docsearch.as\_retriever(search\_type="similarity", search\_kwargs={"k":4})

# create the chain to answer questions

rqa = RetrievalQA.from\_chain\_type(llm=OpenAI(),

                                  chain\_type="stuff",

                                  retriever=retriever,

                                  return\_source\_documents=True)

当查询 "OpenAI 是什么" 时，不仅会得到一个答案，还会得到源文档 source\_documents 。源文档是返回结果的参考文档，它可以帮助理解答案是如何得出的。

query = "OpenAI 是什么?"

rqa(query)['result']

' OpenAI 是一家技术研究和开发公司，旨在研究人工智能的安全性、可控性和效率。它的主要目标是使智能技术得以广泛使用，以改善人类生活。'

如果不需要中间步骤和源文档，只需要最终答案，那么可以直接请求返回结果。设置 return\_source\_documents 为 False。

比如说，问 "gpt-4 对创新力有什么影响?"

query = "gpt-4 对创新力有什么影响?"

rqa(query)['result']

它会直接返回结果，不包括源文档。

' GPT-4可以加强创作者和创作者的创作能力和生产力，从而提高创新力。它可以帮助他们与任务，例如头脑风暴，编辑，反馈，翻译和营销。此外，GPT-4还可以帮助他们更快地完成任务，从而提高他们的生产效率。它也可以帮助他们更深入地思考，更有创意地思考'

## 9.2 对话式表单

这一节，我们一起探索这个由大语言模型驱动的提问和用户回答的程序。它并不是我们常见的 AI 程序，这样的程序并非人类提出问题，AI 进行回答。角色发生了转变，AI 主动提出问题，人类进行回答。

这类程序已经被广泛地应用到各种生活场景中。想象一下，你正在参加一个公司的招聘，面试的过程全由这个程序负责。它会向你提出一系列关于岗位的问题，让你来回答，如同真实的面试官的口吻。或者，你每天要通过几百个人好友申请、打招呼、了解需求等，这个程序会自动跟新好友聊天，根据他们的回答来更新打标签，保存名片信息。还有一种情况是，你正在填写一个报名表，这个程序会根据你之前的回答，逐步引导你完成报名。这些都是具体生活中这类程序的使用案例，可以看出其实用性。

一个典型的这类程序需要完成两个主要任务。首先，我们需要让语言模型只负责提问，而不进行回答，同时限制问题的范围。以招聘程序为例，程序只会提出关于岗位认识的问题，让面试者进行回答。

其次，程序需要根据用户的回答来更新数据库和下一个问题。例如，有个用户回答 “我叫美丽”，程序就能够识别出这个用户的名字是 “美丽”，并将其保存到数据库中。然后，程序会检查是否还有其他信息缺失，比如用户的居住城市或邮箱地址等，如果有缺失的信息，它就会选择相应的问题进行提问，如 “你住在哪里？”。一旦所有需要的信息都收集齐全，程序就会结束这一次的对话。

### 9.2.1 OpenAI 函数的标记链

这里将介绍如何创建一个对话式表单，实现用户以自然对话的方式填写表单信息。

在网页上，我们经常见到表单，用户需要填写详细信息。在网页上处理这些表单非常容易，因为信息可以很容易地解析和处理。但是，如果我们将表单放入一个聊天机器人中，并且希望用户能够以自然对话的方式回答。

我们将使用 OpenAI 函数的标记链来给用户的信息做“标记”。标记链是使用 OpenAI 函数参数来指定一个标记文档的模式。这有助于我们确保模型输出我们想要的精确标签，以及它们对应的类型。

比如我们正在处理一个大量的文本数据，我们希望分析每一段文本的情绪是积极的还是消极的。在这种情况下，我们就可以使用标记链来实现这个功能。我们需要的不仅仅是模型的输出结果，更重要的是，这些结果必须是我们想要的，比如具有情绪类型的标签。

标记链需要在我们想要给文本标注特定属性的时候使用。例如，我们可能会问：“这条信息的情绪是什么？”在这个例子中，“情绪”就是我们想要标注的特定属性，而标记链就可以帮助我们实现这个目标。

通过这种方式，我们不仅可以标注出文本的情绪，还可以标注出文本的其他属性，如主题，作者的观点等等。这个过程就好像给文本贴上了一张张的标签，让我们可以更快更准确地理解和分析文本。

9.2.2 标记链的使用

为了开始我们的项目，首先需要安装所需的 Python 包。这里，我们需要从 GitHub 安装 langchain，因为我们需要它的最新版本来支持标记链功能。我们使用 pip 命令来轻松地安装它们：

pip -q install  openai tiktoken

pip install git+https://github.com/hwchase17/langchain

接下来，我们设置了 OpenAI 的 API 密钥，使我们可以与 OpenAI 服务进行通信：

import os

os.environ["OPENAI\_API\_KEY"] = ""

导入类和方法。

为了使用我们的功能，我们需要从相应的库中导入一些特定的类和方法：

from langchain.chat\_models import ChatOpenAI

from langchain.chains import LLMChain

from langchain.prompts import ChatPromptTemplate

from pydantic import BaseModel, Field

from enum import Enum

from langchain.chains.openai\_functions import (

    create\_tagging\_chain,

    create\_tagging\_chain\_pydantic,

)

接下来，我们定义了一个 Pydantic 数据模型 PersonalDetails，它帮助我们结构化地捕获用户的姓名、城市和电子邮件：

class PersonalDetails(BaseModel):

    # 定义数据的类型

    name: str = Field(

        ...,

        description = "这是用户输入的名字"

    )

    city: str = Field(

        ...,

        description = "这是用户输入的居住城市"

    )

    email: str = Field(

        ...,

        description = "这是用户输入的邮箱地址"

    )

接着，我们使用 ChatOpenAI 类，该类是一个聊天模型包装器，选择的模型型号是 gpt-3.5-turbo-0613，这个型号仅仅适用于聊天模型包装器：

llm = ChatOpenAI(temperature=0, model="gpt-3.5-turbo-0613")

为了自动标记用户的对话并将其分类到适当的字段，我们创建了一个标记链：

chain = create\_tagging\_chain\_pydantic(PersonalDetails,llm)

通过以下示例，我们可以看到如何运行这个标记链，以处理用户提供的信息：

test\_str1 = "你好，我是美丽，我住在上海浦东，我的邮箱是： liteli1987@gmail.com"

test\_res1 = chain.run(test\_str1)

成功运行后，用户的输入被正确地分配到了 PersonalDetails 数据模型中：

PersonalDetails(name='美丽', city='上海浦东', email='liteli1987@gmail.com')

我们可以进一步测试标记链的健壮性。即使没有提供完整的信息，它仍然可以成功捕获所提供的部分：

test\_str2 = "我的邮箱是： liteli1987@gmail.com"

test\_res2 = chain.run(test\_str2)

test\_res2

最终，即使没有提供姓名和城市，我们仍然能够成功捕获用户的电子邮件地址：

PersonalDetails(name='', city='', email='liteli1987@gmail.com')

我们还可以加入一些干扰信息，比如我告诉他，我的邮箱，顺带告诉他我弟弟的邮箱。

test\_str3 = "我叫美丽，我弟弟的邮箱是：1106968391@qq.com"

test\_res3 = chain.run(test\_str3)

test\_res3

它并不会把我弟弟的邮箱记录到我的信息里。

PersonalDetails(name='美丽', city='', email='')

9.2.3 创建提示模板

还记得我们这个程序需要完成的二大任务吗？第一个任务便是我们需要让大语言模型只负责提问，而不进行回答，同时限制问题的范围。我们可以设置提示模板, 运行一个 LLMChain链完成这一目标。为了实现这一目标，我们定义了一个函数 ask\_for\_info，这个函数接受一个名为 ask\_for 的参数列表，列表中的元素代表我们希望模型询问用户的信息，如姓名、城市和电子邮件。

在函数内部，我们定义了一个提示模板 first\_prompt。这个模板指导大语言模型如何与用户进行交互。具体地说，模板中有几个重要的指导原则：1）模型应该扮演前台的角色，并询问用户的个人信息。2）模型不应该跟用户打招呼，只需要解释需要哪些信息。3）所有模型的输出都应该是问题。4）模型应该从 ask\_for 列表中随机选择一个项目进行提问。

为了将这个提示模板与大语言模型结合，我们创建了一个 LLMChain 对象 info\_gathering\_chain，并使用它来运行我们的提问流程。

def ask\_for\_info(ask\_for=["name","city","email"]):

    # 定义一个提示模板

    first\_prompt = ChatPromptTemplate.from\_template(

        """

        假设你现在是一名前台，你现在需要对用户进行询问他个人的具体信息。

        不要跟用户打招呼！你可以解释你需要什么信息。不要说“你好！”！

        接下来你和用户之间的对话都是你来提问，凡是你说的都是问句。

        你每次随机选择{ask\_for}列表中的一个项目，向用户提问。

        比如["name","city"]列表，你可以随机选择一个"name", 你的问题就是“请问你的名字是？”

        """

    )

    info\_gathering\_chain = LLMChain(llm=llm, prompt=first\_prompt)

    chat\_chain = info\_gathering\_chain.run(ask\_for=ask\_for)

    return chat\_chain

当我们调用 ask\_for\_info 函数并为其提供一个 ask\_for 列表时，大语言模型会根据提示模板生成一个与列表中的某个项目相关的问题。

ask\_for\_info(ask\_for=["name","city","email"])

例如，我们让模型询问用户的姓名、城市和电子邮件。运行后，模型按照我们的预期，首先发起了询问姓名的问题：“请问你的名字是？”。这正是我们希望模型在此场景中做的事情，它证明了我们的程序设计是成功的。

'请问你的名字是？'

9.2.4 数据更新和检查

在这一部分，我们进行了数据的更新和检查。我们定义一个函数，用于检查数据是否填写完整。首先，我们定义了一个函数 check\_what\_is\_empty，其主要目的是检查用户个人信息中哪些数据是空缺的。通过遍历用户的详细信息字典，该函数可以发现哪些字段是空的，并将这些字段名收集到 ask\_for 列表中返回。

def check\_what\_is\_empty(user\_personal\_details):

    ask\_for = []

    # 检查项目是否为空

    for field,value in user\_personal\_details.dict().items():

        if value in [None, "", 0]:

            print(f"Field '{field}' 为空" )

            ask\_for.append(f'{field}')

    return ask\_for

为了测试这个函数，我们创建了一个名为 user\_007\_personal\_details 的示例用户，并为该用户的所有字段赋予空值。调用函数后，我们发现该用户的所有字段（姓名、城市和电子邮件）都是空的。

user\_007\_personal\_details = PersonalDetails(name="",city="",email="")

运行函数，查看哪些数据没有填写：

ask\_for = check\_what\_is\_empty(user\_007\_personal\_details)

ask\_for

函数调用后，显示 007 的姓名、城市和邮箱都没有填写。

Field 'name' 为空

Field 'city' 为空

Field 'email' 为空

['name', 'city', 'email']

接下来，我们定义了一个 add\_non\_empty\_details 函数，该函数负责更新用户的信息。当AI与用户进行交互并收到用户的回答时，这个函数将根据用户的回答更新内存中的用户信息。这确保了我们始终有用户的最新信息。

def add\_non\_empty\_details(current\_details:PersonalDetails, new\_details:PersonalDetails):

    # 这是已经填好的用户信息

    non\_empty\_details = {k:v for k,v in new\_details.dict().items() if v not in [None, "", 0]}

    update\_details = current\_details.copy(update=non\_empty\_details)

    return update\_details

为了测试这个功能，我们使AI向 user\_007 提问，该用户回答说他的名字是 007。随后，我们调用了 add\_non\_empty\_details 函数，并确认用户的名字已经更新为 007，而其他字段仍然为空。

res = chain.run("我的名字007")

user\_007\_personal\_details = add\_non\_empty\_details(user\_007\_personal\_details,res)

user\_007\_personal\_details

运行标记链后，更新一条数据。

PersonalDetails(name='007', city='', email='')

我们继续使用 check\_what\_is\_empty 函数，确认还需要向用户询问哪些信息。结果显示，我们还需要询问该用户的城市和电子邮件地址。。

ask\_for = check\_what\_is\_empty(user\_007\_personal\_details)

ask\_for

调用检查函数后，我们可以看到。

["city","email"]

为了使整个流程更为自动化，我们定义了一个 decide\_ask 函数。这个函数的作用是决定是否需要继续向用户提问，并且它会自动地调用 ask\_for\_info 函数来进行提问。如果所有的信息都已经填写完整，它会输出“全部填写完整”。

def decide\_ask(ask\_for=["name","city","email"]):

    if ask\_for:

        ai\_res = ask\_for\_info(ask\_for=ask\_for)

        print(ai\_res)

    else:

        print("全部填写完整")

decide\_ask(ask\_for)

我们定义一个函数，根据 “check\_what\_is\_empty” 检查的结果，决定是否运行 “ask\_for\_info” 函数。 “ask\_for\_info” 函数内实现了调用我们的提示模板，运行 LLMChain 链。

我们使用 user\_999 进行了一个完整的交互示例。AI首先询问了该用户的名字，用户回答后，AI确认了用户的名字并继续询问其他信息。当所有的信息都已经填写完整后，AI停止了提问。

user\_999\_personal\_details = PersonalDetails(name="",city="",email="")

启动程序，开始提问。

decide\_ask(ask\_for)

AI 开始问 999 用户。

请问你的名字是？

999 用户回答后，AI 更新了该用户的信息。

str999 = "我的名字是999"

user\_999\_personal\_details, ask\_for\_999 = filter\_response(str999,user\_999\_personal\_details)

decide\_ask(ask\_for\_999)

检查完邮箱地址仍然为空，AI 继续问“请问你的电子邮件地址是多少？”。

Field 'email' 为空

请问你的电子邮件地址是多少？

999 用户回答自己的邮箱。

str999 = "XX@XX.com"

user\_999\_personal\_details, ask\_for\_999 = filter\_response(str999,user\_999\_personal\_details)

decide\_ask(ask\_for\_999)

AI　停止提问。

＇全部填写完整＇

这整个流程确保了我们可以有效地从用户那里收集所有必要的信息，同时也提供了一种机制，使我们可以在用户提供某些信息后立即更新它们。

## 9.3 使用 LangChain 实现 BabyAGI

这一节我们将利用 LangChain 进行 BabyAGI 实现。这将让我们更加直观地看到每一步骤的发生情况，并且，你也可以在自己的环境中进行实验。

### 9.3.1 BabyAGI介绍

BabyAGI是由Yohei Nakajima 在2023年5月发布的一个自治的人工智能Agent程序代码。这种自治的AI代理旨在根据给定的目标生成和执行任务。它利用OpenAI、Pinecone、LangChain和Chroma的技术来自动化任务并实现Agent特定目标。

我们在Agent 模块把AgentExecutor比喻为一个项目经理，其实BabyAGI项目也可以看成是一个项目经理管理项目，BabyAGI通过创建、优先处理和执行任务列表来实现Agent特定的目标。它还适应变化，并进行必要的调整以确保达到目标。与项目经理一样，BabyAGI具有从以前的经验中学习并做出明智决策的能力。

BabyAGI给我们的启发是，它是你计算机上的AI驱动的个人助手。通过解释给定的目标，它创建了一个所需任务的列表，然后执行它们。每完成一个任务后，BabyAGI都会评估结果并相应地调整其方法。BabyAGI的独特之处在于它能够通过试验和错误从反馈中学习，做出类似人类的认知决策。它还可以编写和运行代码来实现特定的目标，并在诸如加密货币交易、机器人技术和自动驾驶等领域表现出色。

使用BabyAGI的好处是，可以让你有更多的时间专注于更高价值的任务，如决策和创意项目。

在原BabyAGI项目中，它按照以下步骤来创建Agent，承担不同的任务，开展自动化任务并联合起来这些Agent以实现目标。我们依然遵照这样的步骤实施，使用Langchain 内部的模块功能，创建Agent，实现与BabyAGI相同能力的Agent。我们先了解原BabyAGI的实施步骤：

1. 设置明确的目标：首先，用户设置BabyAGI将完成的高级目标。例如，目标可能是“在30天内增加1000个Twitter关注者？”

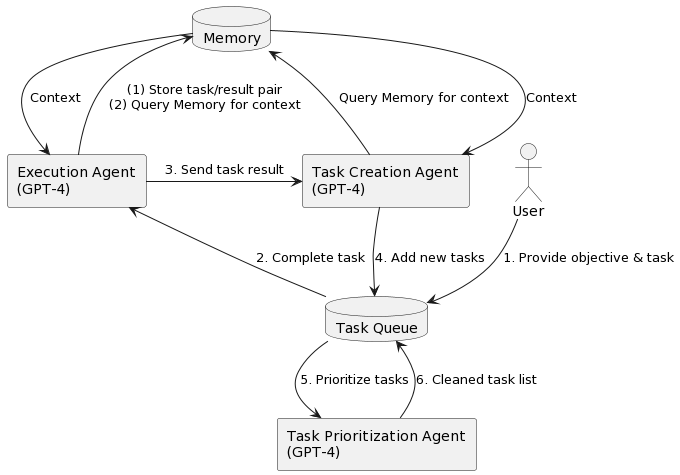
2. 任务生成 (Agent)：接下来，BabyAGI将使用诸如GPT-4之类的语言模型，将目标细分为一系列潜在任务。这些任务可能是以研究为导向的，例如“识别和关注与你的利基市场相关的Twitter帐户，以提高你的可见度并获得新的关注者”，“利用Twitter分析来跟踪你的进度并相应地调整你的策略”等。然后将任务列表存储在长期内存（向量数据库）中供将来参考。

3. 任务优先级 (Agent)：有了任务列表后，BabyAGI将使用其推理能力评估并根据它们的重要性和依赖性对任务进行优先排序，以达到最终的结果。它将决定首先执行哪个任务。

4. 任务执行 (Agent)：然后，BabyAGI将执行并完成任务。执行的结果和收集到的信息也将保存在长期记忆中供将来使用。

5. 评估和创建新任务：执行任务后，BabyAGI将使用其推理能力评估剩余的任务和先前执行的结果。基于评估，它将创建要完成的新任务，以达到最终的目标。

6. 重复：这些步骤将重复，直到BabyAGI实现原始目标或用户干预为止。BabyAGI将不断评估目标的进展，并相应地调整任务列表和优先级，以有效地达到期望的结果。



BabyAGI 作者Yohei Nakajima绘制的示意图 1

### 9.3.2 环境与工具

对于此次实验，我们会需要两个主要工具：OpenAI 以及一个搜索引擎 API。这两者将会协同完成 BabyAGI 的构建。有两个版本的 BabyAGI，一个不依赖任何外部调用，而另一个则利用搜索引擎进行外部调用。

!pip -q install langchain huggingface\_hub openai google-search-results tiktoken cohere faiss-cpu

import os

os.environ["OPENAI\_API\_KEY"] = ""

os.environ["SERPAPI\_API\_KEY"] = ""

导入工具：

import os

from collections import deque

from typing import Dict, List, Optional, Any

from langchain import LLMChain, OpenAI, PromptTemplate

from langchain.embeddings import OpenAIEmbeddings

from langchain.llms import BaseLLM

from langchain.vectorstores.base import VectorStore

from pydantic import BaseModel, Field

from langchain.chains.base import Chain

### 9.3.3  向量存储

在此实验中，我们使用了 FAISS 向量存储，这是一种内存存储技术，使我们无需进行任何外部调用，例如向 Pinecone 请求。但如果你愿意，你完全可以改变一些设定，将其连接到 Pinecone。向量存储是利用 OpenAI 的嵌入进行的。

导入 FAISS 向量库：

from langchain.vectorstores import FAISS

from langchain.docstore import InMemoryDocstore

在构建一个特定的嵌入模型，生成向量索引，并存储这些向量时，我们可以按照以下步骤来操作。

首先，我们需要选择一个适当的嵌入模型。这种模型可以是词嵌入模型，如 Word2Vec 或 GloVe，也可以是句子嵌入模型，如 BERT 或者 Doc2Vec。这些模型通过将词或句子映射到高维度的向量空间，实现了对词或句子语义的捕捉。选择哪种嵌入模型主要取决于我们处理的任务特性和数据的特点。

这里我们使用的是 OpenAI 的嵌入模型, OpenAI 的文本嵌入模型可以精确地嵌入大段文本，具体而言，8100 个标记，根据它们的词对标记比例 0.75，大约可以处理 6143 个单词。它输出 1536 维的向量。

# Define your embedding model

embeddings\_model = OpenAIEmbeddings()

其次，对我们的文本数据进行处理，生成相应的嵌入向量。生成向量后，我们需要构建一个索引，以便能够高效地查询和比较向量。

# Initialize the vectorstore as empty

import faiss

embedding\_size = 1536

index = faiss.IndexFlatL2(embedding\_size)

最后，我们需要将生成的向量和构建的索引进行存储。

vectorstore = FAISS(embeddings\_model.embed\_query, index, InMemoryDocstore({}), {})

### 9.3.4 构建任务链

LangChain 的好处在于，我们可以清楚看到链组件在执行哪些操作，以及它们的提示是什么。在这里，我们有三个主要链组件：任务创建、任务优先级和执行。这些链组件都在为达成整体目标而工作，它们会生成一系列任务。

创建任务链

这个链组件基本上是说：作为一个任务创建的 AI，你要利用执行代理的结果，来创建具有一定目标的新任务。此处的目标就是你想要 AI 实现的东西。最后完成的任务有结果，这个结果是基于任务描述生成的。这些是未完成的任务，如果有一系列尚未完成的任务，就会将其输入到这个链组件中。最后，根据结果创建新任务，这些任务由 AI 系统完成，并且不与未完成的任务重叠。

class TaskCreationChain(LLMChain):

    """Chain to generates tasks."""

    @classmethod

    def from\_llm(cls, llm: BaseLLM, verbose: bool = True) -> LLMChain:

        """Get the response parser."""

        task\_creation\_template = (

            "You are an task creation AI that uses the result of an execution agent"

            " to create new tasks with the following objective: {objective},"

            " The last completed task has the result: {result}."

            " This result was based on this task description: {task\_description}."

            " These are incomplete tasks: {incomplete\_tasks}."

            " Based on the result, create new tasks to be completed"

            " by the AI system that do not overlap with incomplete tasks."

            " Return the tasks as an array."

        )

        prompt = PromptTemplate(

            template=task\_creation\_template,

            input\_variables=["result", "task\_description", "incomplete\_tasks", "objective"],

        )

        return cls(prompt=prompt, llm=llm, verbose=verbose)

这些步骤看起来很简单，但这里就是你可以进行修改，从而使 AI 更符合你需求的地方。

任务优先级链

这个链组件的主要职责是将传入的任务进行清理，重新设置它们的优先级，以便于按照你的团队的最终目标进行排序。任务优先级链组件不会删除任何任务，而是将任务以编号列表的形式返回。

class TaskPrioritizationChain(LLMChain):

    """Chain to prioritize tasks."""

    @classmethod

    def from\_llm(cls, llm: BaseLLM, verbose: bool = True) -> LLMChain:

        """Get the response parser."""

        task\_prioritization\_template = (

            "You are an task prioritization AI tasked with cleaning the formatting of and reprioritizing"

            " the following tasks: {task\_names}."

            " Consider the ultimate objective of your team: {objective}."

            " Do not remove any tasks. Return the result as a numbered list, like:"

            " #. First task"

            " #. Second task"

            " Start the task list with number {next\_task\_id}."

        )

        prompt = PromptTemplate(

            template=task\_prioritization\_template,

            input\_variables=["task\_names", "next\_task\_id", "objective"],

        )

        return cls(prompt=prompt, llm=llm, verbose=verbose)

执行链

在这个过程中，我们定义了一个执行代理，并传递了一些工具给它。这个执行代理是一个计划者，能够为给定的目标制定一个待办事项清单。我们传递了搜索和待办事项这两种工具给它，以便它能够在需要的时候进行搜索或者制定待办事项清单。

from langchain.agents import ZeroShotAgent, Tool, AgentExecutor

from langchain import OpenAI, SerpAPIWrapper, LLMChain

todo\_prompt = PromptTemplate.from\_template("You are a planner who is an expert at coming up with a todo list for a given objective. Come up with a todo list for this objective: {objective}")

todo\_chain = LLMChain(llm=OpenAI(temperature=0), prompt=todo\_prompt)

search = SerpAPIWrapper()

tools = [

    Tool(

        name = "Search",

        func=search.run,

        description="useful for when you need to answer questions about current events"

    ),

    Tool(

        name = "TODO",

        func=todo\_chain.run,

        description="useful for when you need to come up with todo lists. Input: an objective to create a todo list for. Output: a todo list for that objective. Please be very clear what the objective is!"

    )

]

prefix = """You are an AI who performs one task based on the following objective: {objective}. Take into account these previously completed tasks: {context}."""

suffix = """Question: {task}

{agent\_scratchpad}"""

prompt = ZeroShotAgent.create\_prompt(

    tools,

    prefix=prefix,

    suffix=suffix,

    input\_variables=["objective", "task", "context","agent\_scratchpad"]

)

我们可以看到，这个执行器使用 ZeroShotAgent 代理，将提示语、前缀后缀以及输入变量一并输入。通过这种方式，我们可以更清楚地看到在执行过程中，这些部分如何组合在一起工作。

整合所有链

现在，我们有了一些函数，它们的作用是定义任务，运行任务，并设置一个循环使得任务能够持续运行。重要的是，这个系统并不是只运行一次任务就结束，而是通过一个循环，让系统不断地获取和执行任务。

在这个过程中，代码将所有的部分结合在一起，对于每个任务，都有对应的链组件在执行。

def get\_next\_task(task\_creation\_chain: LLMChain, result: Dict, task\_description: str, task\_list: List[str], objective: str) -> List[Dict]:

    """Get the next task."""

    incomplete\_tasks = ", ".join(task\_list)

    response = task\_creation\_chain.run(result=result, task\_description=task\_description, incomplete\_tasks=incomplete\_tasks, objective=objective)

    new\_tasks = response.split('\n')

    return [{"task\_name": task\_name} for task\_name in new\_tasks if task\_name.strip()]

def prioritize\_tasks(task\_prioritization\_chain: LLMChain, this\_task\_id: int, task\_list: List[Dict], objective: str) -> List[Dict]:

    """Prioritize tasks."""

    task\_names = [t["task\_name"] for t in task\_list]

    next\_task\_id = int(this\_task\_id) + 1

    response = task\_prioritization\_chain.run(task\_names=task\_names,

                                             next\_task\_id=next\_task\_id,

                                             objective=objective)

    new\_tasks = response.split('\n')

    prioritized\_task\_list = []

    for task\_string in new\_tasks:

        if not task\_string.strip():

            continue

        task\_parts = task\_string.strip().split(".", 1)

        if len(task\_parts) == 2:

            task\_id = task\_parts[0].strip()

            task\_name = task\_parts[1].strip()

            prioritized\_task\_list.append({"task\_id": task\_id, "task\_name": task\_name})

    return prioritized\_task\_list

def \_get\_top\_tasks(vectorstore, query: str, k: int) -> List[str]:

    """Get the top k tasks based on the query."""

    results = vectorstore.similarity\_search\_with\_score(query, k=k)

    if not results:

        return []

    sorted\_results, \_ = zip(\*sorted(results, key=lambda x: x[1], reverse=True))

    return [str(item.metadata['task']) for item in sorted\_results]

def execute\_task(vectorstore, execution\_chain: LLMChain, objective: str, task: str, k: int = 5) -> str:

    """Execute a task."""

    context = \_get\_top\_tasks(vectorstore, query=objective, k=k)

    return execution\_chain.run(objective=objective, context=context, task=task)

### 9.3.5  创建 BabyAGI

创建 BabyAGI 类

为了使这个过程更便于管理，我们为 BabyAGI 创建了一个类。在这个类中，我们可以添加任务，打印任务列表，打印下一个任务，打印任务结果。这些函数将能够与语言模型一起使用，使得所有的内容都能够同时运行。

实际的运行过程是在一个 While 循环中进行的。它会在获取到某个结果后退出，并根据这个结果进行下一步操作。我们可以看到，整个过程中发生的各种事情，包括创建新任务，重新设置优先级等等。

class BabyAGI(Chain, BaseModel):

    """Controller model for the BabyAGI agent."""

    task\_list: deque = Field(default\_factory=deque)

    task\_creation\_chain: TaskCreationChain = Field(...)

    task\_prioritization\_chain: TaskPrioritizationChain = Field(...)

    execution\_chain: AgentExecutor = Field(...)

    task\_id\_counter: int = Field(1)

    vectorstore: VectorStore = Field(init=False)

    max\_iterations: Optional[int] = None

    class Config:

        """Configuration for this pydantic object."""

        arbitrary\_types\_allowed = True

    def add\_task(self, task: Dict):

        self.task\_list.append(task)

    def print\_task\_list(self):

        print("\033[95m\033[1m" + "\n\*\*\*\*\*TASK LIST\*\*\*\*\*\n" + "\033[0m\033[0m")

        for t in self.task\_list:

            print(str(t["task\_id"]) + ": " + t["task\_name"])

    def print\_next\_task(self, task: Dict):

        print("\033[92m\033[1m" + "\n\*\*\*\*\*NEXT TASK\*\*\*\*\*\n" + "\033[0m\033[0m")

        print(str(task["task\_id"]) + ": " + task["task\_name"])

    def print\_task\_result(self, result: str):

        print("\033[93m\033[1m" + "\n\*\*\*\*\*TASK RESULT\*\*\*\*\*\n" + "\033[0m\033[0m")

        print(result)

    @property

    def input\_keys(self) -> List[str]:

        return ["objective"]

    @property

    def output\_keys(self) -> List[str]:

        return []

    def \_call(self, inputs: Dict[str, Any]) -> Dict[str, Any]:

        """Run the agent."""

        objective = inputs['objective']

        first\_task = inputs.get("first\_task", "Make a todo list")

        self.add\_task({"task\_id": 1, "task\_name": first\_task})

        num\_iters = 0

        while True:

            if self.task\_list:

                self.print\_task\_list()

                # Step 1: Pull the first task

                task = self.task\_list.popleft()

                self.print\_next\_task(task)

                # Step 2: Execute the task

                result = execute\_task(self.vectorstore,

self.execution\_chain, objective, task["task\_name"]

                )

                this\_task\_id = int(task["task\_id"])

                self.print\_task\_result(result)

                # Step 3: Store the result

                result\_id = f"result\_{task['task\_id']}"

                self.vectorstore.add\_texts(

                    texts=[result],

                    metadatas=[{"task": task["task\_name"]}],

                    ids=[result\_id],

                )

                # Step 4: Create new tasks and reprioritize task list

                new\_tasks = get\_next\_task(

                    self.task\_creation\_chain, result, task["task\_name"], [t["task\_name"] for t in self.task\_list], objective)

                for new\_task in new\_tasks:

                    self.task\_id\_counter += 1

                    new\_task.update({"task\_id": self.task\_id\_counter})

                    self.add\_task(new\_task)

                self.task\_list = deque(

                    prioritize\_tasks(self.task\_prioritization\_chain, this\_task\_id, list(self.task\_list), objective) )

            num\_iters += 1

            if self.max\_iterations is not None

and num\_iters == self.max\_iterations:

                print("\033[91m\033[1m" + "\n\*\*\*\*\*TASK ENDING\*\*\*\*\*\n"

+ "\033[0m\033[0m")

                break

        return {}

    @classmethod

    def from\_llm(

        cls,

        llm: BaseLLM,

        vectorstore: VectorStore,

        verbose: bool = False,

        \*\*kwargs

    ) -> "BabyAGI":

        """Initialize the BabyAGI Controller."""

        task\_creation\_chain = TaskCreationChain.from\_llm(

            llm, verbose=verbose

        )

        task\_prioritization\_chain = TaskPrioritizationChain.from\_llm(

            llm, verbose=verbose

        )

        llm\_chain = LLMChain(llm=llm, prompt=prompt)

        tool\_names = [tool.name for tool in tools]

        agent = ZeroShotAgent(llm\_chain=llm\_chain, allowed\_tools=tool\_names)

        agent\_executor = AgentExecutor.from\_agent\_and\_tools(

agent=agent, tools=tools, verbose=True)

        return cls(

            task\_creation\_chain=task\_creation\_chain,

            task\_prioritization\_chain=task\_prioritization\_chain,

            execution\_chain=agent\_executor,

            vectorstore=vectorstore,

            \*\*kwargs

        )

在这个系统中，我们并没有使用 Pinecone 进行存储，而是选择在本地进行存储。这样，我们可以更直观地看到在这个过程中发生的每一件事。

虽然系统已经可以正常运行，但我认为如果添加一个额外的链组件，用于生成一些摘要，例如一个最终报告，将会使结果更好。目前，虽然系统可以执行所有的任务，但是在最后得出结论的时候，可能会有一些不足。

在实际运行过程中，我们使用了 OpenAI 的语言模型，并将温度设为零。

llm = OpenAI(temperature=0)

例如，我们可以设置一个目标，就是找到在网上购买 yubikey 5C 的最便宜的价格和网站，然后将结果提供给我。我们可以看到，通过这样的设置，我们可以实现一些特定的目标。

OBJECTIVE = "Find the cheapest price and site to buy a Yubikey 5c online and give me the URL"

我们开始实例化 BabyAGI 类并运行它。

llm = OpenAI(temperature=0)

首先，我们需要把语言模型和向量存储器传入，然后我们设置了一个最大的迭代次数，这是这个版本相比于先前版本的改进之处。在早前的版本中，程序会无限循环下去，而在这个版本中，我们可以通过设置迭代次数上限来限制循环的次数 (max\_iterations: Optional [int] = 7)。

# Logging of LLMChains

verbose=False

# If None, will keep on going forever

max\_iterations: Optional[int] = 7

# 实例化 BabyAGI

baby\_agi = BabyAGI.from\_llm(

    llm=llm,

    vectorstore=vectorstore,

    verbose=verbose,

    max\_iterations=max\_iterations

)

接下来，我们将目标输入到程序中，程序会制定一个待办事项列表并开始执行。例如，我们希望找到网上购买 YubiKey 5C 的最便宜的价格和网站，并获取 URL。程序则会生成一个待办事项列表，包括搜索在线零售商，比较不同零售商的价格，查找折扣或促销活动，以及阅读每个零售商的客户评论。

baby\_agi({"objective": OBJECTIVE})

### 9.3.6 运行 BabyAGI

程序会根据待办事项列表开始执行任务。对于每个任务，程序会进行一些搜索，比较不同在线商店中的 YubiKey 5C 价格，检查是否有折扣代码和促销活动，等等。

在整个过程中，程序会生成观察结果，比如它在哪些地方看到了 YubiKey，它找到的最便宜的价格是多少。如果在执行过程中遇到问题或者需要做出选择，程序也会返回相应的任务，并根据这些任务调整待办事项列表。

最后，程序会返回一个 URL (这个地址可能不能访问到商品），告诉我们可以在哪个网站以最便宜的价格购买 YubiKey 5C。但是，我们发现返回的 URL 并不总是有效的。例如，程序返回的 URL 可能会导致 404 错误，或者返回的价格可能和网站上显示的价格不一致。这些问题可能是由于程序运行的位置和我们实际的位置不同，或者可能是因为程序没有能力检查 URL 的有效性。

\*\*\*\*\*TASK LIST\*\*\*\*\*

3: Compare the price of Yubikey 5c at other online retailers to Yubico.com/store.

4: Check customer reviews of [Retailer Name] for Yubikey 5c.

5: Find out if [Retailer Name] offers any discounts or promotions for Yubikey 5c.

6: Research the return policy of [Retailer Name] for Yubikey 5c.

7: Determine the shipping cost for Yubikey 5c from [Retailer Name].

8: Check customer reviews of other online retailers for Yubikey 5c.

9: Find out if other online retailers offer any discounts or promotions for Yubikey 5c.

10: Research the return policy of other online retailers for Yubikey 5c.

11: Determine the shipping cost for Yubikey 5c from other online retailers.

\*\*\*\*\*NEXT TASK\*\*\*\*\*

3: Compare the price of Yubikey 5c at other online retailers to Yubico.com/store.

> Entering new AgentExecutor chain...

Thought: I should compare the prices of Yubikey 5c at other online retailers.

Action: Search

Action Input: Prices of Yubikey 5c at other online retailers

Observation: [{'position': 1, 'block\_position': 'top', 'title': 'YubiKey 5C - OEM Official', 'price': '$55.00', 'extracted\_price': 55.0, 'link': 'https://www.yubico.com/product/yubikey-5c', 'source': 'yubico.com/store', 'thumbnail': 'https://serpapi.com/searches/64ba2ffc49ecdb86973e7b26/images/ebce3fc64f92f22d58e2ea0dae58f9a2419c119482504cffef43e39a06787765.webp', 'extensions': ['45-day returns (most items)']}, {'position': 2, 'block\_position': 'top', 'title': 'Yubico YubiKey 5C - USB security key', 'price': '$3,256.99', 'extracted\_price': 3256.99, 'link': 'https://www.cdw.com/product/yubico-yubikey-5c-usb-security-key/7493450?cm\_ven=acquirgy&cm\_cat=google&cm\_pla=NA-NA-Yubico\_NY&cm\_ite=7493450', 'source': 'CDW', 'shipping': 'Get it by 7/26', 'thumbnail': 'https://serpapi.com/searches/64ba2ffc49ecdb86973e7b26/images/ebce3fc64f92f22d58e2ea0dae58f9a2b4a2cbdc8de9b340a34c7f35661e9f75.webp'}, {'position': 3, 'block\_position': 'top', 'title': 'YubiKey 5C NFC - OEM Official', 'price': '$55.00', 'extracted\_price': 55.0, 'link': 'https://www.yubico.com/product/yubikey-5c-nfc', 'source': 'yubico.com/store', 'thumbnail': 'https://serpapi.com/searches/64ba2ffc49ecdb86973e7b26/images/ebce3fc64f92f22d58e2ea0dae58f9a2f0f2eed4b19c6081b5000768a9cc1878.webp', 'extensions': ['45-day returns (most items)']}]

Thought:

虽然这个系统还不完美，但是它确实为我们提供了一个基于链组件的自动化流程，用来获取信息、制定待办事项列表，并执行任务。这个系统给我们展示了如何用简单的链组件模型来处理复杂的问题。这是一个不断学习和思考的过程，我们可以根据需要调整提示，添加新的链组件，或者改进现有的链组件。

### 9.4 本章小结

这一章全面介绍了如何利用Langchain构建实际的应用程序。通过三个具体的案例，我们深入了解了Langchain的核心组件和功能，从理论到实践，全方位地揭示了其在LLM应用程序中的应用潜力。

我们探索了一个与本地电脑PDF文档进行对话的ChatPDF机器人，它展示了如何利用Langchain与现有的PDF内容进行交互，将静态的文档内容转化为动态的问答体验。接着，我们介绍了一个对话式表单填写机器人，这个机器人能够自动地引导用户完成复杂的表单填写任务，大大提高了用户的工作效率。

我们详细分析了当前颇受欢迎的Agent项目——BabyAGI。BabyAGI不仅集成了Langchain的所有核心模块，还展示了如何将这些模块融合到一个完整的应用中，实现自动任务生成、执行和反馈调整的全流程。

通过这三个案例，我们看到了Langchain在实际应用中的强大能力，无论是与文档的交互、对话式的表单填写，还是智能的任务管理，Langchain都为我们提供了高效、智能和灵活的链组件方案。更重要的是，这三个案例在综合性上为我们展示了如何将Langchain的各个核心模块融合到实际应用中。

10.

## 10.1 集成的背景与LLM集成

学习任何新的技术框架或工具，往往需要对其背后的原理和历史背景有所了解，这样可以更好地掌握它的应用方式和最佳实践。在探讨为什么学习LangChain的集成项目之前，我们先看看Apache Camel和Spring Cloud Data Flow的集成技术历史与现状。Apache Camel和Spring Cloud Data Flow都是集成领域的佼佼者，它们各自拥有丰富的生态系统和社区支持。这两个框架已经解决了很多集成的常见问题，提供了大量的最佳实践。LangChain作为一个新的集成框架，其设计思想和实现方式无疑是受到这两个框架的启发。学习它们可以帮助我们更好地理解LangChain的设计哲学和技术选型。

当我们谈及集成，首先要了解的是：为什么Apache Camel会成为这个领域的佼佼者呢？答案很简单。Camel针对不同的协议和数据类型，提供了特定的API实现。这是因为，集成的真正挑战在于处理来自不同系统、协议和数据格式的信息。Camel已经解决了这个问题。它目前支持超过80个协议和数据类型，包括但不限于RESTful服务、消息队列和数据库。并且，得益于其模块化和可扩展的架构，Camel为开发者带来了巨大的便利性和灵活性。

而当我们进入到云计算时代，数据的整合和流动变得更加关键。Spring Cloud Data Flow（SCDF）应运而生，成为了云原生环境中的数据流处理利器。它不仅仅是一个数据流处理工具，更是一个完整的微服务集成框架。SCDF允许开发者轻松创建、部署和监控数据流处理管道，更重要的是，它支持与多种云平台集成，为云原生应用的开发提供了极大的灵活性。

如今面对大语言模型开发(LLM应用开发)的复杂性，LLM应用开发和部署需要一个专为其量身打造的集成框架。LangChain正是这样的解决方案。

LangChain继承了Apache Camel和SCDF的经验，结合LLM的特性，它提供了一套完整的工具和技术，简化了LLM应用的开发、部署和管理。

### 10.1.1 LangChain的集成生态

集成的真正挑战在于处理来自不同系统、协议和数据格式的信息，而LangChain的Integrations库正是基于这一核心思想而构建的。以下是其核心分类及相关描述，集成项目的数量在2023年8月统计：

- Callbacks：LangChain理解到，在某些特定事件触发时，开发者可能需要与其他系统或服务互动。正如Camel针对不同协议和数据类型提供了特定的API实现，Callbacks功能也为开发者创造了类似的桥梁，已实现六个此类项目的集成。

- 聊天模型包装器：面对多样的对话场景，LangChain提供了10种聊天模型来满足从简单问答到高级交互的需求。这如同Camel为不同的协议和数据类型提供了解决方案。

- Document loaders：文档加载与处理是集成中的基础工作。LangChain提供了127种文档加载工具，确保了各种应用场景的需求都能被满足。

- Document transformers：针对文档的处理和转换，LangChain提供了7种转换器，这可以看作是LangChain为多种数据格式提供支持的一个延伸。

- LLM模型包装器：针对LLM应用开发，LangChain准备了57种LLM模型，满足了与不同协议和数据格式交互的需要。

- Memory：LangChain提供了12种记忆存储解决方案，满足了各种持久化需求。

- Retrievers：信息检索是集成的关键，LangChain为此准备了22种集成工具，无论是本地文档还是网络上的信息都能被有效地检索。

-嵌入模型包装器：文本的向量化处理是与各种协议和数据类型交互的关键。LangChain为此提供了31种文本嵌入模型包装器。

- Agent toolkit：为了帮助开发者创建智能代理，LangChain提供了21种集成工具套件，确保与各种系统和服务的交互能够流畅进行。

- Tools：为了满足开发、测试和优化的多样需求，LangChain提供了37种集成工具。

- Vector store：机器学习和深度学习需要专门的向量数据存储，LangChain为此提供了45种解决方案。

- Grouped by provider：展现了LangChain与各大供应商和平台的集成实力，这也反映了它在处理不同系统、协议和数据格式信息方面的广泛适应性。

LangChain虽然提供了广泛的集成库，但这也为开发者带来了一系列的挑战。以下是开发者在使用LangChain的集成库时可能会遇到的问题：

（1）选择的困惑：面对多种相似的工具，开发者可能会在选择上犹豫不决，不知道哪种更适合他们的需求。（2）学习曲线：不同的工具有其特有的功能和操作方式，这意味着开发者需要为每一种工具投入学习的时间。（3）维护的挑战：随着技术的迅速发展，一些工具可能会过时，需要定期更新或替换。（4）性能与兼容性：使用不同的集成工具时，可能会出现性能瓶颈或兼容性问题，这有可能影响整个LLM应用的稳定性和效率。

为了帮助开发者更有效地选择和使用集成工具，我们选择了以下几类在开发中比较受关注的集成类型，并为每个类型选取了几个典型的集成项目，结合代码示例进行说明。

## 10.2 LLM集成指南

LLM集成是实现了各个模型平台的LLM模型包装器，我们主要介绍HuggingFace 和AzureOpenAI。Huggingface为我们提供了一个平台，能够及时追踪当前热门的新型语言模型。它为BERT、XLNet、GPT等多种模型提供了统一且高效的编码实践。更为出色的是，它设有一个模型库，其中涵盖了众多常用的预训练模型以及为各种任务进行微调（Fine-tuning）的模型，使得模型的下载变得简单快捷。HuggingFace有很多模型支持本地下载和部署，为开发者提供了多样的功能和模块，能够快速响应各种需求，大大提高开发效率。

另外，若你希望利用免费的模型，可以前往Hugging Face平台，比如搜索google/flan-t5-xl 来获取免费的模型。

AzureOpenAI是Langchain与Azure之间的集成项目，旨在让开发者能够在Langchain框架中，轻松地通过模型包装器在Azure平台上调用OpenAI的GPT系列模型。此外，Azure OpenAI不仅是Azure平台上的标志性工具，它也是LLM应用中的核心组件，确保与Azure平台之间的稳固协同作业，从而为LLM应用提供坚实的基础。

### 10.2.1 Azure OpenAI 集成

AzureOpenAI的Langchain实现的在Azure平台调用OpenAI能力的LLM模型包装器。在LLM应用中起到了至关重要的作用。举个例子，如果开发者想要在Azure平台中利用OpenAI的GPT系列模型，就可以使用这个集成项目。更为重要的是，它来源于Azure平台，一个在云计算领域具有权威地位的平台，这确保了其稳定性和高效性，为LLM应用提供了有力的支持。

集成步骤

1. 从哪里导入类

在终端中先设置以下参数：

# Set this to `azure`

export OPENAI\_API\_TYPE=azure

# The API version you want to use: set this to `2023-05-15` for the released version.

export OPENAI\_API\_VERSION=2023-05-15

# The base URL for your Azure OpenAI resource. You can find this in the Azure portal under your Azure OpenAI resource.

export OPENAI\_API\_BASE=https://your-resource-name.openai.azure.com

# The API key for your Azure OpenAI resource. You can find this in the Azure portal under your Azure OpenAI resource.

export OPENAI\_API\_KEY=<your Azure OpenAI API key>

对于Azure OpenAI集成项目，从 langchain.llms 中导入了 AzureOpenAI 类。具体代码如下：

安装Langchain的Python库后，我们就可以导入AzureOpenAI 类。在langchain.llms中，我们已经集成了各大模型平台的API封装。当你在VSCode中编辑并输入点号后，它会自动列出所有可用的封装集成。为了方便识别，这些集成通常都会以其模型平台的名称作为类名的前缀，如AzureOpenAI。

在 langchain.chat\_models 中，我们针对聊天模型专门实现了各大模型平台的API封装。当你在VSCode环境中编程并输入点号后，系统会自动列举所有的聊天模型封装集成选项。为了方便开发者迅速识别，这些封装通常以其对应的模型平台名称作为类名的前缀，例如AzureOpenAI。如果你想使用Azure的聊天模型，可以直接通过from langchain.chat\_models.azure\_openai import AzureChatOpenAI来导入AzureChatOpenAI类。

from langchain.llms import AzureOpenAI

2. 使用方法

Azure OpenAI在使用上提供了一个简洁明了的接口。主要包含以下两个步骤：

- 初始化 ：首先创建一个AzureOpenAI的实例。在这个过程中，需要指定部署名称和模型名称。

llm = AzureOpenAI(

deployment\_name="td2",

model\_name="text-davinci-002",

)

- 执行 ：一旦初始化完成，可以轻松地运行LLM应用并获得结果。例如，要请求一个笑话，只需调用此实例并提供相应的提示：

llm("Tell me a joke")

此外，Azure OpenAI API的配置可以通过环境变量或直接在Python环境中进行，与标准OpenAI API的使用方式略有不同。

效果展示

在运行Azure OpenAI LLM应用后，得到的响应如下：

"\n\nWhy couldn't the bicycle stand up by itself? Because it was...two tired!"

此外，通过调用 print 方法，开发者可以查看LLM应用的自定义输出：

print(llm)

# 打印结果：AzureOpenAI

Params: {'deployment\_name': 'text-davinci-002', 'model\_name': 'text-davinci-002', 'temperature': 0.7, 'max\_tokens': 256, 'top\_p': 1, 'frequency\_penalty': 0, 'presence\_penalty': 0, 'n': 1, 'best\_of': 1}

### 10.2.2 Hugging Face Hub 集成

Hugging Face Hub集成项目是一个旨在提高机器学习和深度学习模型可访问性和协作性的创新平台。该项目专注于提供一个中心化的位置，供开发者和研究者分享、发布和协作开发各种NLP模型。除了存储预训练模型，Hugging Face Hub还支持多种框架和库，确保开发者可以轻松地集成和部署这些模型到他们的LLM应用中。此外，通过API和其他工具的集成，Hugging Face Hub使得与其他平台的交互更为流畅，为机器学习社区带来了巨大价值。对于Hugging Face Hub集成项目，相关的类从 langchain 中导入。具体代码如下：

集成步骤

from langchain import HuggingFaceHub, PromptTemplate, LLMChain

集成Hugging Face Hub主要涉及以下步骤：

- 安装和设置 ：首先确保安装了 huggingface\_hub Python包。

pip install huggingface\_hub

接下来，获取API令牌并为其设置环境变量。

from getpass import getpass

HUGGINGFACEHUB\_API\_TOKEN = getpass()

os.environ["HUGGINGFACEHUB\_API\_TOKEN"] = HUGGINGFACEHUB\_API\_TOKEN

- 准备示例 ：构建一个问题和相应的模板，然后使用 PromptTemplate 生成相应的提示。

question = "Who won the FIFA World Cup in the year 1994? "

template = """Question: {question}\nAnswer: Let's think step by step."""

prompt = PromptTemplate(template=template, input\_variables=["question"])

- 模型示例 ：使用不同的模型库和repo\_id来实例化 HuggingFaceHub 并运行LLMChain。

repo\_id = "<specific\_repo\_id>"

llm = HuggingFaceHub(

repo\_id=repo\_id, model\_kwargs={"temperature": 0.5, "max\_length": 64}

)

llm\_chain = LLMChain(prompt=prompt, llm=llm)

print(llm\_chain.run(question))

其中， <specific\_repo\_id> 可以是Flan, Dolly, Camel, XGen, Falcon等模型的特定ID。

效果展示

使用Flan模型：

repo\_id = "google/flan-t5-xxl"

llm = HuggingFaceHub(

repo\_id=repo\_id, model\_kwargs={"temperature": 0.5, "max\_length": 64}

)

llm\_chain = LLMChain(prompt=prompt, llm=llm)

print(llm\_chain.run(question))

- 使用Flan模型得到的回答是：

The FIFA World Cup was held in the year 1994. West Germany won the FIFA World Cup in 1994.

- 使用Dolly模型：

repo\_id = "databricks/dolly-v2-3b"

llm = HuggingFaceHub(

repo\_id=repo\_id, model\_kwargs={"temperature": 0.5, "max\_length": 64}

)

llm\_chain = LLMChain(prompt=prompt, llm=llm)

print(llm\_chain.run(question))

使用Dolly模型，得到的回答是：

First of all, the world cup was won by the Germany. Then the Argentina won the world cup in 2022. So, the Argentina won the world cup in 1994.

- 使用Camel模型、XGen模型和Falcon模型也会得到类似的输出，具体取决于所选的模型和参数配置。

## 10.3 聊天模型集成指南

Anthropic、Google PaLM Chat API和ChatOpenAI，这三个平台在2023年的大语言模型赛道上，锋芒毕现，都已成为语言模型和聊天机器人领域的重量级选手。

随着GPT-4 等大语言模型的突破，聊天机器人已经不仅仅是简单的问答工具，它们现在广泛应用于客服、企业咨询、电子商务等多种场景，为用户提供准确、快速的反馈。在这样的背景下，开发者们急需一套可以轻松切换、集成不同平台的工具。

正是基于这样的需求，Anthropic API、Google PaLM Chat API和ChatOpenAI的API封装应运而生。这些封装不仅为开发者提供了与上述三大平台的稳定交互能力，而且确保了在开发和部署聊天机器人的过程中，无论是从哪个平台切换到哪个平台，都能够做到高效和快速。对于开发者而言，这无疑大大降低了开发难度和时间成本。

### 10.3.1 Anthropic集成

集成步骤

1. 从哪里导入类。Anthropic聊天模型的集成涉及从 langchain.chat\_models 、 langchain.prompts.chat 和 langchain.schema 中导入相关的类。

在 langchain.chat\_models中，我们针对聊天模型专门实现了各大模型平台的API封装。当你在VSCode环境中编程并输入点号后，系统会自动列举所有的聊天模型封装集成选项。为了方便开发者迅速识别，这些封装通常以“Chat”加上其对应的模型平台名称作为类名的前缀，例如ChatAnthropic。如果你想使用Anthropic的聊天模型，可以直接通过 from langchain.chat\_models import ChatAnthropic来导入ChatAnthropic类。具体导入代码为：

from langchain.chat\_models import ChatAnthropic

同时也要引入了langchain.schema中定义的三种消息类型。这些工具主要用于格式化和处理聊天模型的输入数据。具体来说，聊天模型包装器期望的输入是一个消息列表，而不是单一的字符串。因此，当开发者利用此包装器与模型进行交互时，他们需要确保按照特定的结构组织数据，满足模型的输入要求。

from langchain.prompts.chat import (

ChatPromptTemplate,

SystemMessagePromptTemplate,

AIMessagePromptTemplate,

HumanMessagePromptTemplate,

)

from langchain.schema import AIMessage, HumanMessage, SystemMessage

2. 使用方法。Anthropic聊天模型的集成可以总结为以下几个步骤：

- 基本使用 ：首先，我们创建一个 ChatAnthropic 实例，然后使用它来处理消息。

chat = ChatAnthropic()

messages = [HumanMessage(content="Translate this sentence from English to French. I love programming.")]

chat(messages)

这会得到输出：

AIMessage(content=" J'aime la programmation.", additional\_kwargs={}, example=False)

Anthropic聊天模型不仅提供了基本的聊天功能，还进一步支持异步和流功能，为开发者提供更为灵活和高效的交互方式。

在代码中，我们可以看到从langchain.callbacks.manager导入的CallbackManager和从langchain.callbacks.streaming\_stdout导入的StreamingStdOutCallbackHandler。CallbackManager用于管理各种回调操作，确保异步任务的顺利执行。

而StreamingStdOutCallbackHandler则专门用于处理流输出，即实时将模型的响应输出到标准输出流。结合这两个工具，开发者可以更加轻松地利用Anthropic聊天模型的高级功能，确保数据处理的实时性和流畅性。

from langchain.callbacks.manager import CallbackManager

from langchain.callbacks.streaming\_stdout import StreamingStdOutCallbackHandler

await chat.agenerate([messages])

这将返回以下输出：

LLMResult(generations=[[ChatGeneration(text=" J'aime programmer.", generation\_info=None, message=AIMessage(content=" J'aime programmer.", additional\_kwargs={}, example=False))]], llm\_output={}, run=[RunInfo(run\_id=UUID('8cc8fb68-1c35-439c-96a0-695036a93652'))])

同样，我们可以设置 streaming 和 callback\_manager 参数来启用流功能：

chat = ChatAnthropic(

streaming=True,

verbose=True,

callback\_manager=CallbackManager([StreamingStdOutCallbackHandler()]),

)

chat(messages)

这会得到输出：

J'aime la programmation.

AIMessage(content=" J'aime la programmation.", additional\_kwargs={}, example=False)

效果展示

Anthropic聊天模型能够根据提供的人类消息进行响应。例如，在上述示例中，模型成功地将英文句子“I love programming.”翻译成法文句子“J'aime la programmation.”。

此外，通过异步和流功能，开发者可以更加灵活地使用模型，使其更加适应各种实时交互的场景。

### 10.3.2 Google PaLM Chat集成

集成步骤

1. 导入所需的类和方法

为了使用Vertex AI聊天模型，您需要首先安装 google-cloud-aiplatform 。

#!pip install google-cloud-aiplatform

然后，从 langchain.chat\_models 、 langchain.prompts.chat 和 langchain.schema 导入相关的类和方法。这是为了支持模型交互、聊天提示和消息架构。

from langchain.chat\_models import ChatVertexAI

from langchain.prompts.chat import (

ChatPromptTemplate,

SystemMessagePromptTemplate,

HumanMessagePromptTemplate,

)

from langchain.schema import HumanMessage, SystemMessage

2. 使用方法

Vertex AI聊天模型的使用可以分为以下几个步骤：

- 基本使用 ：您可以创建一个 ChatVertexAI 实例，并用它来处理消息。

chat = ChatVertexAI()

messages = [

SystemMessage(

content="You are a helpful assistant that translates English to French."

),

HumanMessage(

content="Translate this sentence from English to French. I love programming."

),

]

chat(messages)

例如，您可以给模型发送一个系统消息，告诉它它是一个帮助将英语翻译成法语的助手，然后发送一个人类消息，要求翻译一个句子。

这将返回如下输出：

AIMessage(content='Sure, here is the translation of the sentence "I love programming" from English to French: J\'aime programmer.', additional\_kwargs={}, example=False)

为了开发者可以在不同的情境创建不同的提示词模板，然后在实际对话中动态地填充这些模板，参照以下代码进行配置提示词模板。先定义一个提示词模板，明确表达助手的功能——即从{input\_language}翻译至{output\_language}。基于这个定义，我们使用SystemMessagePromptTemplate.from\_template方法创建了一个系统消息的提示词模板对象。

template = (

"You are a helpful assistant that translates {input\_language} to {output\_language}."

)

system\_message\_prompt = SystemMessagePromptTemplate.from\_template(template)

human\_template = "{text}"

human\_message\_prompt=HumanMessagePromptTemplate.from\_template(human\_template)

接下来，代码同样为人类输入定义了一个简洁的模板，并进一步利用HumanMessagePromptTemplate.from\_template实例化了人类消息的提示词模板对象。

为了方便地组合多个角色的模板消息，如系统、AI和人类等，代码利用了ChatPromptTemplate.from\_messages方法。这个方法接收一系列的模板对象，并将它们合成为一个完整的聊天提示词模板。

最后，我们使用format\_prompt方法，将具体的参数——如输入语言、输出语言和用户文本——绑定进预定义的模板中。这样，经过格式化的提示词可以反映出助手的功能（从英语翻译至法语）和用户的原始输入（"I love programming."），并将其整合为一个完整的、为语言模型准备的提示词，从而引导模型提供相关的回复。

chat\_prompt = ChatPromptTemplate.from\_messages(

[system\_message\_prompt, human\_message\_prompt]

)

# get a chat completion from the formatted messages

chat(

chat\_prompt.format\_prompt(

input\_language="English", output\_language="French", text="I love programming."

).to\_messages()

)

这将返回如下输出：

AIMessage(content='Sure, here is the translation of "I love programming" in French: J\'aime programmer.', additional\_kwargs={}, example=False)

Vertex AI还提供了Codey API，更改模型型号为“codechat-bison”，专门用于代码帮助。例如，当您询问如何创建一个Python函数来识别所有的质数时，它可以提供相关的代码建议。

chat = ChatVertexAI(model\_name="codechat-bison")

messages = [

HumanMessage(

content="How do I create a python function to identify all prime numbers?"

)

]

chat(messages)

这将返回如下输出：

AIMessage(content='The following Python function can be used to identify all prime numbers up to a given integer: ...', additional\_kwargs={}, example=False)

Vertex AI提供了强大的聊天功能，使开发者能够与模型进行直观的交互。通过使用不同的消息类型，如 HumanMessage 和 SystemMessage ，开发者可以更好地引导模型的行为。

使用 MessagePromptTemplate 可以进一步增强这种交互，因为开发者可以为特定的任务或场景创建定制的模板，而不是每次都手动构建完整的消息。

Codey API是一个强大的工具，特别是对于需要代码帮助的开发者。它可以根据用户的问题提供相关的代码建议，从而帮助他们更快地解决问题。

### 10.3.3 Azure上的OpenAI端点集成

集成步骤

1. 导入所需的类和方法

若要使用Azure上托管的OpenAI端点，您需要从 langchain.chat\_models 和 langchain.schema 导入相关的类和方法，以支持模型交互和消息架构。

from langchain.chat\_models import AzureChatOpenAI

from langchain.schema import HumanMessage

1. 使用方法

BASE\_URL = "https://${TODO}.openai.azure.com"

API\_KEY = "..."

DEPLOYMENT\_NAME = "chat"

model = AzureChatOpenAI(

openai\_api\_base=BASE\_URL,

openai\_api\_version="2023-05-15",

deployment\_name=DEPLOYMENT\_NAME,

openai\_api\_key=API\_KEY,

openai\_api\_type="azure",

)

与Azure上的OpenAI端点交互主要涉及以下步骤：首先，您需要配置必要的基本信息，包括Azure上的OpenAI API的基本URL（BASE\_URL）、API密钥（API\_KEY）用于身份验证及访问服务，以及代表Azure部署的名称（DEPLOYMENT\_NAME）。

特别注意，${TODO} 部分应替换为您在Azure上的OpenAI服务的真实URL部分。有了这些信息，您可以创建一个名为AzureChatOpenAI的模型实例，其中openai\_api\_type的值已经被设定为 "azure" ，确保API请求会被重定向到Azure托管的OpenAI端点。最后，与其他聊天模型的交互方式相同，您可以向该模型发送一个HumanMessage，例如请求将英语句子翻译成法语，并从模型中获取对应的回复。

model(

[

HumanMessage(

content="Translate this sentence from English to French. I love programming."

)

]

)

这将返回如下输出：

AIMessage(content="J'aime programmer.", additional\_kwargs={})

这是一个简单且直接的方法来与Azure上托管的OpenAI端点进行交互。一旦您已经在Azure上配置好了OpenAI服务，并获取了相关的API密钥和URL，这个过程就变得相对简单。

Azure提供了一个可靠且安全的环境来托管和运行OpenAI模型，这为企业和开发者提供了一个在云中快速部署和扩展AI解决方案的方法。

需要注意的是，每当您与Azure上的服务进行交互时，都应确保保护好您的API密钥，以防止任何未授权的访问或潜在的滥用。

## 10.4 向量库集成指南

[向量库](https://www.pinecone.io/learn/vector-database/)是一种索引和存储向量嵌入以实现高效管理和快速检索的数据库。与单独的[向量索引](https://www.pinecone.io/learn/vector-indexes/)不同，像Pinecone这样的向量数据库提供了额外的功能，例如索引管理、数据管理、元数据存储和过滤以及水平扩展。

向量库在多种应用场景中发挥着关键作用，特别是在处理大数据和复杂查询时。其中，语义文本搜索是一个典型的应用，用户可以通过NLP转换器和句子嵌入模型将文本数据转化为向量嵌入，再利用Pinecone这类工具进行索引和搜索。此外，它还可以支持生成问答系统，即从Pinecone检索与特定查询相关的上下文，然后传递给如OpenAI这样的生成模型，从而产生基于真实数据的答案。

不仅如此，向量库的应用还扩展到了图像和电商领域。例如，通过将图像数据转化为向量嵌入，再使用Pinecone之类的工具构建索引，我们可以轻松地执行图像的相似性搜索。同时，基于代表用户兴趣和行为的向量，向量库可以为电子商务平台生成产品推荐，从而实现个性化的用户体验。

Pinecone, Chroma, Milvus : 这三种存储解决方案为机器学习和深度学习中的向量数据存储提供了优质的选择，确保向量数据的稳定、高效存储。

### 10.4.1 Chroma集成

集成步骤

在这个基础示例中，我们首先加载了最近的国情咨文，将其分割成几部分，使用开源嵌入模型进行嵌入，加载到Chroma中，然后对其进行查询。

1. 导入所需的类和方法，先安装向量库chromadb。

pip install chromadb

从 langchain.embeddings.sentence\_transformer ， langchain.text\_splitter ， langchain.vectorstores 和 langchain.document\_loaders 导入相关的类和方法来支持文档加载、文本分割、嵌入和向量存储。

# import

from langchain.embeddings.sentence\_transformer import SentenceTransformerEmbeddings

from langchain.text\_splitter import CharacterTextSplitter

from langchain.vectorstores import Chroma

from langchain.document\_loaders import TextLoader

1. 使用方法。

为了有效地处理美国国情咨文，首先需要使用TextLoader进行文档的加载。在加载后，借助CharacterTextSplitter，我们将文档分割成大小为1000字符的各个块，这些块之间存在0字符的重叠。在文档准备完成后，我们采用SentenceTransformerEmbeddings来创建一个名为all-MiniLM-L6-v2的嵌入模型型号。

接着，为了实现向量化搜索和相似度检测，我们使用Chroma.from\_documents方法将这些嵌入后的文档加载到Chroma中。一旦数据加载完毕，便可以利用db.similarity\_search方法查询文档，快速找到与特定查询内容最相关的文档部分。

# load the document and split it into chunks

loader = TextLoader("../../../state\_of\_the\_union.txt")

documents = loader.load()

# split it into chunks

text\_splitter = CharacterTextSplitter(chunk\_size=1000, chunk\_overlap=0)

docs = text\_splitter.split\_documents(documents)

# create the open-source embedding function

embedding\_function = SentenceTransformerEmbeddings(model\_name="all-MiniLM-L6-v2")

# load it into Chroma

db = Chroma.from\_documents(docs, embedding\_function)

# query it

query = "What did the president say about Ketanji Brown Jackson"

docs = db.similarity\_search(query)

# print results

print(docs[0].page\_content)

打印获得的结果是：

/Users/jeff/.pyenv/versions/3.10.10/lib/python3.10/site-packages/tqdm/auto.py:21: TqdmWarning: IProgress not found. Please update jupyter and ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user\_install.html

from .autonotebook import tqdm as notebook\_tqdm

Tonight. I call on the Senate to: Pass the Freedom to Vote Act. Pass the John Lewis Voting Rights Act. And while you’re at it, pass the Disclose Act so Americans can know who is funding our elections.

Tonight, I’d like to honor someone who has dedicated his life to serve this country: Justice Stephen Breyer—an Army veteran, Constitutional scholar, and retiring Justice of the United States Supreme Court. Justice Breyer, thank you for your service.

One of the most serious constitutional responsibilities a President has is nominating someone to serve on the United States Supreme Court.

And I did that 4 days ago, when I nominated Circuit Court of Appeals Judge Ketanji Brown Jackson. One of our nation’s top legal minds, who will continue Justice Breyer’s legacy of excellence.

3. 分析。 这个示例演示了如何使用 langchain 库处理、嵌入和查询文档的过程。在这里，我们对美国国情咨文进行了查询，具体查找总统关于美国Ketanji Brown Jackson的言论，并成功找到了相关内容。

这种查询可以用于许多不同的应用场景，如新闻文章分析、法律文档查询、学术研究等。使用开源嵌入模型和Chroma这样的向量存储工具，可以有效地搜索大量的文档，并快速找到与特定查询相关的部分。

需要注意的是，在实际应用中，您可能需要调整文档分割的大小和嵌入模型的选择，以适应特定的需求和数据集。

### 10.4.2 Pinecone集成

Pinecone是一个高效的向量搜索服务，特别针对那些需要处理大量数据的应用而设计。高速性能是其核心特点之一，即使在数十亿个条目的数据集上，它都能确保超低的查询延迟，从而为用户提供即时的搜索反馈。

除了高速查询，Pinecone还具备实时更新的能力。这意味着当你添加、编辑或删除数据时，其索引会被实时地更新，确保数据的实时性和准确性。这为动态变化的数据环境提供了极大的便利。

最后，Pinecone融合了向量搜索与元数据过滤的功能。这使得它不仅可以根据向量相似性搜索，还可以结合元数据进行过滤，从而提供更为精准的搜索结果。而作为一个完全托管的服务，Pinecone使得用户无需担心后端的复杂性和安全性，可以专注于实现其业务需求。这个示例提供了如何与 Pinecone 向量数据库交互的步骤。

1. 初始化和设置。首先，用户需要通过 pip 安装 pinecone-client ， openai ， tiktoken 和 langchain 。tiktoken 是一个文档计数工具，用于计算文档中的词数或标记数，而无需进行实际的模型转换。这在评估模型所需的令牌数或预估模型调用的成本时尤为有用。简单来说，tiktoken 提供了一种高效的方式来了解文档的大小和复杂性。

pip install pinecone-client openai tiktoken langchain

为了与 Pinecone 互动，用户需要输入 Pinecone API 密钥和 Pinecone 环境。此外，由于我们要使用 OpenAIEmbeddings，所以也需要 OpenAI 的 API 密钥。

import os

import getpass

os.environ["PINECONE\_API\_KEY"] = getpass.getpass("Pinecone API Key:")

os.environ["OPENAI\_API\_KEY"] = getpass.getpass("OpenAI API Key:")

from langchain.embeddings.openai import OpenAIEmbeddings

from langchain.text\_splitter import CharacterTextSplitter

from langchain.vectorstores import Pinecone

from langchain.document\_loaders import TextLoader

2. 加载和分割文档。与之前Chroma的示例类似，这里使用 TextLoader 加载文档，并使用 CharacterTextSplitter 将其分割。

from langchain.document\_loaders import TextLoader

loader = TextLoader("../../../state\_of\_the\_union.txt")

documents = loader.load()

text\_splitter = CharacterTextSplitter(chunk\_size=1000, chunk\_overlap=0)

docs = text\_splitter.split\_documents(documents)

1. 嵌入文档。使用嵌入模型包装器OpenAIEmbeddings进行文档嵌入。

embeddings = OpenAIEmbeddings()

4. 与 Pinecone 互动。使用 Pinecone 的 API 初始化 Pinecone，然后检查我们的索引是否已存在。如果不存在，我们创建一个新的索引。OpenAI 的 text-embedding-ada-002 模型使用 1536 维，所以我们需要设置维度为 1536。

1536维在这里指的是由OpenAI的`text-embedding-ada-002`模型生成的每个文本嵌入（或称为向量表示）所具有的维度或特征数。简单地说，当这个模型接受一个文本输入并为其生成嵌入时，输出的嵌入向量将有1536个数值或坐标。这1536个值捕获了文本的语义信息，使得具有相似意义的文本具有相近的向量表示。因此，当我们创建一个索引来存储由此模型生成的嵌入时，需要确保该索引能够容纳1536维的数据，从而确保每一个维度的信息都被完整地保存下来。

import pinecone

# initialize pinecone

pinecone.init(

api\_key=os.getenv("PINECONE\_API\_KEY"), # find at app.pinecone.io

environment=os.getenv("PINECONE\_ENV"), # next to api key in console

)

index\_name = "langchain-demo"

# First, check if our index already exists. If it doesn't, we create it

if index\_name not in pinecone.list\_indexes():

# we create a new index

pinecone.create\_index(

name=index\_name,

metric='cosine',

dimension=1536

)

# The OpenAI embedding model `text-embedding-ada-002 uses 1536 dimensions`

docsearch = Pinecone.from\_documents(docs, embeddings, index\_name=index\_name)

# if you already have an index, you can load it like this

# docsearch = Pinecone.from\_existing\_index(index\_name, embeddings)

query = "What did the president say about Ketanji Brown Jackson"

docs = docsearch.similarity\_search(query)

5. 文档搜索。使用similarity\_search 方法查询文档，并输出查询结果。当使用similarity\_search方法查询文档时，首先通过嵌入模型将每个文档和查询都转换为向量。然后，计算查询向量与文档向量间的相似度，通常基于余弦相似度。最后，根据相似度排序并返回与查询最相关的文档。

print(docs[0].page\_content)

6. 向现有索引添加更多文本。使用add\_texts函数将更多的文本嵌入到现有的 Pinecone 索引中。首先，初始化一个代表该索引的对象。然后，它创建一个Pinecone向量存储实例，该实例将使用指定的嵌入函数将文本转化为向量。最后，使用add\_texts函数，它将字符串 "More text!" 的向量表示形式加入到这个索引中，以便于后续的相似度查询。

index = pinecone.Index("langchain-demo")

vectorstore = Pinecone(index, embeddings.embed\_query, "text")

vectorstore.add\_texts("More text!")

7. 最大边际相关性搜索。除了使用 similarity\_search ，用户还可以使用 mmr 作为检索器。这为用户提供了一种新的、可能更加相关的方法来查询文档。使用了最大边际相关性(Maximum Margin Relevance, MMR)搜索方法来查询文档，从而可能提供更相关的搜索结果。首先，它通过as\_retriever函数将文档搜索器设置为使用 "mmr" 作为其检索类型。然后，它用指定的查询query来获取相关的文档。在获取的文档中，代码遍历每一份匹配的文档，并打印其内容。

retriever = docsearch.as\_retriever(search\_type="mmr")

matched\_docs = retriever.get\_relevant\_documents(query)

for i, d in enumerate(matched\_docs):

print(f"\n## Document {i}\n")

print(d.page\_content)

这个示例为我们提供了一个使用 Pinecone 向量数据库的全面指南。从初始化到查询，用户可以根据自己的需求进行各种操作。

### 10.4.2 Milvus集成

Milvus是一个专门的向量数据库，于2019年首次亮相，旨在为由深度神经网络和其他机器学习模型生成的大规模嵌入向量提供存储、索引和管理。这不仅仅是一个常规的数据库；它是特别为处理向量数据而设计的，能够轻松管理万亿级别的向量索引。

与传统的关系型数据库不同，它们通常用于存储和查询结构化数据，Milvus从其核心设计上就是为了处理从非结构化数据生成的嵌入向量。这是因为在当前的互联网时代，非结构化数据，如电子邮件、论文、物联网传感器数据和社交媒体图片，越来越普遍。

为了使这些非结构化数据对机器有意义，科研人员和工程师经常使用嵌入技术将它们转化为数值向量。这些向量捕获了原始数据的关键特征和信息。Milvus的主要任务是存储这些嵌入向量，并为之提供高效查询功能。

此外，Milvus还能够衡量向量之间的相似性。它通过计算向量间的距离来评估相似度，因此，如果两个向量很接近，则它们表示的原始非结构化数据也很相似。这一特点使Milvus在很多领域，如推荐系统、图像搜索和自然语言处理，都成为了一个强大的工具。

1. 准备工作。先确保已经运行了一个 Milvus 实例，并通过 pip 安装了 pymilvus 。

%pip install pymilvus

2. 设置 OpenAI API 密钥

由于我们想使用 OpenAIEmbeddings，所以需要获取 OpenAI 的 API 密钥。通过设置环境变量实现这一点。

import os

import getpass

os.environ["OPENAI\_API\_KEY"] = getpass.getpass("OpenAI API Key:")

3. 加载和分割文档。这部分与之前的 Azure 和 Pinecone 示例类似。首先使用 TextLoader 从给定的路径加载文档。然后，使用 CharacterTextSplitter 根据给定的大小分割这些文档。

from langchain.embeddings.openai import OpenAIEmbeddings

from langchain.text\_splitter import CharacterTextSplitter

from langchain.vectorstores import Milvus

from langchain.document\_loaders import TextLoader

4. 嵌入文档

使用 OpenAI 的模型来为这些文档生成嵌入向量。

from langchain.document\_loaders import TextLoader

loader = TextLoader("../../../state\_of\_the\_union.txt")

documents = loader.load()

text\_splitter = CharacterTextSplitter(chunk\_size=1000, chunk\_overlap=0)

docs = text\_splitter.split\_documents(documents)

embeddings = OpenAIEmbeddings()

5. 与 Milvus 互动。在与Milvus互动的步骤中，首先根据给定的参数（如主机名和端口号）建立与Milvus实例的连接。一旦连接建立，用户便可以将之前处理过并转换为向量形式的文档加载到Milvus数据库中，为后续的查询和分析做好准备。

vector\_db = Milvus.from\_documents(

docs,

embeddings,

connection\_args={"host": "127.0.0.1", "port": "19530"},

)

6. 文档搜索

和前面的Pinecone 示例一样，可以使用 similarity\_search 方法查询与输入查询相似的文档。在这个例子中，查询的是 "What did the president say about Ketanji Brown Jackson"，并且返回了相关的段落。

query = "What did the president say about Ketanji Brown Jackson"

docs = vector\_db.similarity\_search(query)

打印搜索的相关文档结果。

docs[0].page\_content

# 'Tonight. I call on the Senate to: Pass the Freedom to Vote Act. Pass the John Lewis Voting Rights Act. And while you’re at it, pass the Disclose Act so Americans can know who is funding our elections. \n\nTonight, I’d like to honor someone who has dedicated his life to serve this country: Justice Stephen Breyer—an Army veteran, Constitutional scholar, and retiring Justice of the United States Supreme Court. Justice Breyer, thank you for your service. \n\nOne of the most serious constitutional responsibilities a President has is nominating someone to serve on the United States Supreme Court. \n\nAnd I did that 4 days ago, when I nominated Circuit Court of Appeals Judge Ketanji Brown Jackson. One of our nation’s top legal minds, who will continue Justice Breyer’s legacy of excellence.'

此示例提供了如何使用 Milvus 向量数据库的指南。从初始化到查询，提供了完整的步骤，使用户能够轻松地使用 Milvus 进行大规模文档嵌入和查询。与 OpenAI 和 langchain 结合使用，Milvus 可以为用户提供高效的向量搜索和查询能力。

## 10.5 嵌入模型集成指南

Cohere Embeddings提供了与Cohere平台的无缝对接，确保文本嵌入过程既高效又精确。而HuggingFaceHub Embeddings和llamacpp embeddings则代表了另外两种强大的文本嵌入集成方法。它们都经过严格的测试，以确保与HuggingFaceHub和llamacpp平台的稳定和高效交互，使得开发者可以更轻松地在其LLM应用中利用这些先进的嵌入技术。

### 10.5.1 HuggingFaceEmbeddings嵌入集成

Sentence Transformers Embeddings为开发者提供了一种高效、简洁的方式来为文本生成向量嵌入。这种嵌入通常基于深度学习模型，专为捕捉文本之间的复杂语义关系而设计。利用这种技术，开发者能够实现更加精确的文本匹配和更深入的内容分析，为LLM应用带来了优化的效果。

1. SentenceTransformerEmbeddings 简介。实际上，Sentence Transformers Embeddings是通过HuggingFaceEmbeddings集成进行调用的。对于那些已经熟悉sentence\_transformers包的开发者，为了使其更容易上手和进行整合，Langchain提供了SentenceTransformerEmbeddings的别名，这样开发者可以在代码中使用熟悉的命名方式。

2. 首先需要对开发环境进行配置。确保在开始前已经正确地安装了sentence\_transformers包，这是为了确保文本嵌入的流程可以顺利进行。如未安装，可以通过提供的安装命令轻松完成设置。

pip install sentence\_transformers > /dev/null

注意，为了保证环境的稳定性，请及时更新 pip 到其最新版本。

[notice] A new release of pip is available: 23.0.1 -> 23.1.1

[notice] To update, run: pip install --upgrade pip

- 引入必要模块 ：

开发者需要从 langchain.embeddings 模块导入HuggingFaceEmbeddings和SentenceTransformerEmbeddings。

from langchain.embeddings import HuggingFaceEmbeddings,SentenceTransformerEmbeddings

3. 文本嵌入实践。首要步骤是初始化嵌入模型。通过HuggingFaceEmbeddings并指定相关的模型名称 all-MiniLM-L6-v2 来设定所需的嵌入模型包装器，从而为后续的文本转换工作做好准备。

embeddings = HuggingFaceEmbeddings(model\_name="all-MiniLM-L6-v2")

对于更熟悉 SentenceTransformer 的开发者，上述初始化等同于：

embeddings = SentenceTransformerEmbeddings(model\_name="all-MiniLM-L6-v2")

在LLM应用中，当开发者遇到文本数据，如"This is a test document."，他们可以使用先前初始化的嵌入模型，直接将此文本转换为相应的向量嵌入，为后续分析或其他操作提供机器可理解的格式。

text = "This is a test document."

query\_result = embeddings.embed\_query(text)

如果需要处理多个文档的嵌入时，例如同时嵌入 "This is a test document." 和 "This is not a test document."，可以将这些文本作为列表传递给embeddings.embed\_documents方法，从而一次性得到这些文档的对应嵌入结果，存储在doc\_result中。

doc\_result = embeddings.embed\_documents([text, "This is not a test document."])

通过上述步骤，开发者不仅可以轻松集成和使用Sentence Transformers Embeddings，还可以确保其在LLM应用中的高效性和准确性。

### 10.5.2 Llama-cpp嵌入集成

llama.cpp 主要目标是在MacBook上使用4位整数量化运行LLaMA模型。它是一个纯粹的C/C++实现，不依赖任何外部库。尽管该程序优先考虑Apple芯片并通过ARM NEON, Accelerate和Metal框架进行优化，但它也为x86架构提供了AVX、AVX2和AVX512的支持。此外，该程序在计算精度上支持混合的F16 / F32，并能够支持4位、5位和8位的整数量化。对于BLAS操作，它支持各种库，如OpenBLAS、Apple BLAS、ARM Performance Lib、ATLAS、BLIS、Intel MKL、NVHPC、ACML、SCSL、SGIMATH等。另外，也支持cuBLAS和CLBlast。

在准备集成和使用Llama-cpp之前，开发者首先需要设置其开发环境。具体来说，必须确保已经安装了llama-cpp-python库。这可以通过简单地运行特定的安装命令来实现。

pip install llama-cpp-python

为了简化开发过程，Llama-cpp的嵌入模块已被预先集成。因此，开发者可以直接从langchain.embeddings模块中导入LlamaCppEmbeddings来使用这个功能。

from langchain.embeddings import LlamaCppEmbeddings

在实际应用中，要利用Llama-cpp进行文本嵌入，首先需要初始化模型。

开发者可以通过LlamaCppEmbeddings类并为其提供特定的模型路径（例如“/path/to/model/ggml-model-q4\_0.bin”）来完成这个步骤，从而为后续操作创建一个Llama-cpp嵌入模型实例。

llama = LlamaCppEmbeddings(model\_path="/path/to/model/ggml-model-q4\_0.bin")

为了从LLM应用中的文本数据中生成嵌入，开发者只需将所需文本传递给已初始化的LlamaCppEmbeddings实例。例如，对于文本 "This is a test document."，通过调用Llama-cpp模型的嵌入方法，开发者可以轻松获得该文本的向量嵌入表示。

text = "This is a test document."

query\_result = llama.embed\_query(text)

对于要嵌入的一组文档，开发者可以简单地传递一个包含所有文档的列表给llama.embed\_documents方法。例如，将文本列表[text]传入方法后，它将返回这些文档的向量嵌入表示，并存储在doc\_result变量中。这使得对多个文档的批量嵌入变得简单高效。

doc\_result = llama.embed\_documents([text])

4. 深入应用

一旦成功集成Llama-cpp到LangChain中，开发者就可以充分利用其高效的文本嵌入能力，进一步为LLM应用带来高准确率和低延迟的体验。通过不断地测试、优化和调整，开发者可以确保Llama-cpp嵌入在各种场景下都能稳定发挥其最大潜能。

### 10.5.3 Cohere嵌入集成

随着各种文本嵌入技术的发展，Cohere成为了开发者在LLM应用中的又一选择。其凭借稳定性和高效性受到许多开发者的欢迎。接下来，我们将探讨如何在LangChain中集成和使用Cohere嵌入。

Cohere是一个先进的文本嵌入工具，旨在为文本数据生成精确的表示。与其他嵌入方法相比，Cohere在某些特定任务上可能具有更高的准确率和更好的性能。

Cohere利用如BERT和GPT等Transformer架构的深度学习模型为文本生成嵌入。这些嵌入不仅仅反映文本的表面结构，而是深入捕捉其语义含义，确保即使两段文本的字面表述不同，但只要它们的意思或概念相似，生成的嵌入也会是相似的。

在LangChain中，为了简化开发者的工作流并提供更便捷的Cohere嵌入使用体验，开发团队预置了与Cohere相关的嵌入模块。开发者只需通过简单地从langchain.embeddings模块中导入CohereEmbeddings类，就能轻松地在其应用中集成Cohere的功能。

from langchain.embeddings import CohereEmbeddings

在LangChain中，当开发者想要实际应用Cohere的嵌入功能时，首先需要拥有一个有效的Cohere API密钥，这是为了确保与Cohere服务的通信。一旦获得密钥，开发者可以简单地使用CohereEmbeddings类并通过cohere\_api\_key参数来初始化它，从而在应用中轻松地生成文本嵌入。

embeddings = CohereEmbeddings(cohere\_api\_key=cohere\_api\_key)

在LLM应用中，当开发者想要为特定的文本数据，例如"This is a test document."，生成嵌入表示时，他们可以直接利用Cohere的嵌入方法。这一方法将该文本转换为一个数值向量，该向量捕获了文本的语义含义，从而为后续的分析或操作提供了基础。

query\_result = embeddings.embed\_query(text)

如果需要为一组文档生成嵌入，可以使用以下方法：

doc\_result = embeddings.embed\_documents([text])

集成Cohere嵌入到LangChain中，开发者可以充分发挥其特有的文本嵌入优势，为LLM应用提供更准确的文本表示。当处理复杂的语言任务时，Cohere可能会带来更好的性能和稳定性。

为了满足LLM应用的开发者对高效、稳定的文本嵌入的需求，LangChain和Cohere之间的合作确保了最佳的集成体验和高效的嵌入生成。

## 10.6 Agent toolkits 集成指南

Agent toolkits的集成旨在简化并增强LLM应用中的数据处理和分析功能。CSV Agent提供了一个专门的工具，允许开发者无缝地处理CSV数据，降低了对复杂编码的依赖。Pandas Agent则集成了Pandas框架，赋予了开发者在应用中进行高效数据操作的能力。另外，为了满足先进的数据可视化需求，PowerBI Agent与Microsoft PowerBI紧密结合，为开发者带来了丰富的、直观的数据可视化工具。这些工具套件确保了LLM应用的数据处理、分析和可视化都既简单又高效。

### 10.6.1 CSV Agent的集成

LangChain为开发者提供了多种与CSV文件互动的方式，特别是针对问题回答任务。本文将详细讨论如何使用CSV Agent以及一些相关的安全注意事项。

CSV Agent主要用于与CSV文件交互，特别是当需要查询或检索信息时。需要注意的是，CSV Agent内部调用了Pandas DataFrame agent和Python agent。这意味着，当LLM生成的Python代码可能存在问题时，执行这些代码可能会导致意外的后果。因此，使用时应保持谨慎。

CSV Agent是LLM应用中用于处理CSV数据的工具，而在编程中，为了实现这一功能，开发者需要引用一系列特定的API。create\_csv\_agent是创建和管理CSV Agent的核心方法，位于langchain.agents模块中。而OpenAI来自langchain.llms模块，可能是用于模型管理或与OpenAI平台的交互。接下来，ChatOpenAI从langchain.chat\_models导入，可能是一个针对聊天模型的包装器。最后，AgentType定义了LLM应用中可以使用的各种Agent的类型，助于开发者明确各个Agent的角色和功能。通过这些API的结合，开发者可以在LLM应用中轻松创建和使用CSV Agent。

from langchain.agents import create\_csv\_agent

from langchain.llms import OpenAI

from langchain.chat\_models import ChatOpenAI

from langchain.agents.agent\_types import AgentType

初始化方法中，create\_csv\_agent函数被用于构建一个新的CSV Agent。此代理的特点是使用ZERO\_SHOT\_REACT\_DESCRIPTION类型，意味着该代理可以在没有预先训练的情况下对数据进行描述或反应。在示例代码中，代理被配置为处理名为titanic.csv的文件，使用OpenAI模型并设置其温度参数为0，以获得更确定性的输出。同时，通过verbose=True参数，代理在运行时会显示更多的详细信息。总之，此代码片段展示了如何快速设置并初始化一个能够对CSV数据进行零射击描述的代理。

agent = create\_csv\_agent(

OpenAI(temperature=0),

"titanic.csv",

verbose=True,

agent\_type=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION,

)

另外还可以使用OPENAI\_FUNCTIONS类型进行初始化。

这是一种不同于ZERO\_SHOT\_REACT\_DESCRIPTION的代理构建方法。在这种方法中，代理是基于特定的OpenAI功能进行操作的，特别是那些与ChatOpenAI模型相关的功能。例如，在给出的代码中，我们使用了模型gpt-3.5-turbo-0613来创建一个代理，该模型的温度参数被设置为0以获得确定性输出。这个代理专门为处理名为titanic.csv的文件而设，并通过verbose=True参数提供额外的运行时详细信息。

OPENAI\_FUNCTIONS类型是一个代理初始化选项，专门为高级的OpenAI模型设计。这些模型，例如gpt-3.5-turbo-0613，是在2023年6月13日之后发布的。之前发布的型号并不能使用OPENAI\_FUNCTIONS类型进行初始化。

agent = create\_csv\_agent(

ChatOpenAI(temperature=0, model="gpt-3.5-turbo-0613"),

"titanic.csv",

verbose=True,

agent\_type=AgentType.OPENAI\_FUNCTIONS,

)

使用CSV Agent查询数据是一个直接的方式，允许开发者对已加载的CSV文件进行交互式查询。在CSV Agent被初始化后，它会在后台读取和理解CSV文件内容。例如，通过调用agent.run()方法并提供相应的文本查询，如“how many rows are there?”，代理会检索文件并返回文件中行的数量。这种方法为开发者提供了一个简洁、直观的接口，使他们能够无缝地与CSV数据互动，而无需编写复杂的查询或处理逻辑。

agent.run("how many rows are there?")

或者询问具有超过3名兄弟姐妹的人数：

agent.run("how many people have more than 3 siblings")

CSV Agent的设计不仅仅局限于单个CSV文件的查询，它同样能够处理并与多个CSV文件交互。这意味着开发者可以轻松比较和分析多个数据集之间的差异。例如，通过传递两个CSV文件“titanic.csv”和“titanic\_age\_fillna.csv”给代理，您可以询问这两个数据框（DataFrames）在年龄列上的不同之处。这样的功能大大增强了CSV Agent的灵活性和实用性，为开发者提供了一个高效的工具来分析和对比不同的数据源。

agent = create\_csv\_agent(

ChatOpenAI(temperature=0, model="gpt-3.5-turbo-0613"),

["titanic.csv", "titanic\_age\_fillna.csv"],

verbose=True,

agent\_type=AgentType.OPENAI\_FUNCTIONS,

)

agent.run("how many rows in the age column are different between the two dfs?")

对于需要与CSV文件进行交互的LLM应用开发者，LangChain中的CSV Agent提供了一种高效且灵活的方法。然而，考虑到安全性问题，开发者应该谨慎使用，并确保不执行可能有害的LLM生成的Python代码。

### 10.6.2 Pandas Agent的集成

LangChain不仅提供了与CSV文件互动的方法，还为开发者提供了与pandas数据帧互动的方式。此功能特别适用于问题回答任务。本文将详细讨论如何使用Pandas Dataframe Agent以及与之相关的一些安全注意事项。

Pandas Dataframe Agent主要用于与pandas数据帧交互，特别是在需要查询或检索信息时。需要注意的是，此代理在后台调用Python代理，执行LLM生成的Python代码。如果LLM生成的Python代码可能是有害的，则执行此代码可能会导致意外的后果。因此，使用时应谨慎。

首先导入了一系列的模块和工具来处理和互动Pandas数据帧。使用create\_pandas\_dataframe\_agent，他们可以创建一个代理来与数据帧交互。同时，ChatOpenAI和OpenAI提供了与LLM模型的连接，可以为该代理提供理解和输出自然语言的回答。AgentType则定义了可能的代理类型。接着，他们使用Python的pandas库导入了一个CSV文件，名为"titanic.csv"，并将其读取为一个数据帧df。这样，开发者即可以利用LangChain中的工具和代理与这个数据帧进行互动。

from langchain.agents import create\_pandas\_dataframe\_agent

from langchain.chat\_models import ChatOpenAI

from langchain.agents.agent\_types import AgentType

from langchain.llms import OpenAI

import pandas as pd

df = pd.read\_csv("titanic.csv")

这种方法使用了ZERO\_SHOT\_REACT\_DESCRIPTION类型作为代理类型。如下是初始化代理的例子：

agent = create\_pandas\_dataframe\_agent(OpenAI(temperature=0), df, verbose=True)

另外还可以使用OPENAI\_FUNCTIONS类型进行初始化。

这是一种不同于ZERO\_SHOT\_REACT\_DESCRIPTION的代理构建方法。在这种方法中，代理是基于特定的OpenAI功能进行操作的，特别是那些与ChatOpenAI模型相关的功能。例如，在给出的代码中，我们使用了模型gpt-3.5-turbo-0613来创建一个代理，该模型的温度参数被设置为0以获得确定性输出。这个代理专门为处理名为titanic.csv的文件而设，并通过verbose=True参数提供额外的运行时详细信息。

OPENAI\_FUNCTIONS类型是一个代理初始化选项，专门为高级的OpenAI模型设计。这些模型，例如gpt-3.5-turbo-0613，是在2023年6月13日之后发布的。之前发布的型号并不能使用OPENAI\_FUNCTIONS类型进行初始化。

agent = create\_pandas\_dataframe\_agent(

ChatOpenAI(temperature=0, model="gpt-3.5-turbo-0613"),

df,

verbose=True,

agent\_type=AgentType.OPENAI\_FUNCTIONS,

)

与CSV Agent类似，一旦初始化Pandas Dataframe Agent，您可以运行查询以检索数据帧中的数据。

例如，如果您想知道数据帧中有多少行，只需使用agent.run("how many rows are there?")，代理将计算数据帧中的行数，并返回相应的结果。这为那些不熟悉Pandas语法的人提供了一个简单直接的方式来查询和分析数据。

agent.run("how many rows are there?")

或者询问具有超过3名兄弟姐妹的人数：

agent.run("how many people have more than 3 siblings")

除了单个数据帧，Pandas Dataframe Agent还支持与多个数据帧互动。例如，您可以将多个数据帧传递给代理，并询问两个数据帧之间年龄列的差异行数：

df1 = df.copy()

df1["Age"] = df1["Age"].fillna(df1["Age"].mean())

agent = create\_pandas\_dataframe\_agent(OpenAI(temperature=0), [df, df1], verbose=True)

agent.run("how many rows in the age column are different?")

# 输出结果：

> Entering new AgentExecutor chain...

Thought: I need to compare the age columns in both dataframes

Action: python\_repl\_ast

Action Input: len(df1[df1['Age'] != df2['Age']])

Observation: 177

Thought: I now know the final answer

Final Answer: 177 rows in the age column are different.

> Finished chain.

'177 rows in the age column are different.'

对于需要与pandas数据帧进行交互的LLM应用开发者，LangChain中的Pandas Dataframe Agent提供了一种高效且灵活的方法。然而，考虑到安全性问题，开发者应该在使用时谨慎，并确保不执行可能有害的LLM生成的Python代码。

### 10.6.3 PowerBI Dataset Agent的集成

Power BI是一个用于数据可视化和报告的工具，但当需要通过编程方式查询和分析Power BI数据集时，LangChain的PowerBI Dataset Agent为您提供了一个方便的途径。这使得查询变得更自然和人性化，而不是依赖于DAX（数据分析表达式）查询语言。

PowerBI Dataset Agent设计用于与Power BI数据集交互。您可以使用此代理查询数据集，例如描述数据表，查询表中的记录数，或对数据进行多维度的分析。

初始化PowerBI Dataset Agent所用的类： create\_pbi\_agent 和 PowerBIToolkit 来自 langchain.agents.agent\_toolkits。

PowerBIDataset来自 langchain.utilities.powerbi。ChatOpenAI来自langchain.chat\_models。AgentExecutor 来自langchain.agents。

通过执行 pip install azure-identity 命令，安装了 azure.identity 包，以支持Azure的身份验证。代理需要Azure的认证来访问PowerBI数据集。这里使用了 DefaultAzureCredential() 函数，它是 Azure 提供的默认方法，用于获取适当的认证凭据，以便代理可以访问相关数据。

from langchain.agents.agent\_toolkits import create\_pbi\_agent

from langchain.agents.agent\_toolkits import PowerBIToolkit

from langchain.utilities.powerbi import PowerBIDataset

from langchain.chat\_models import ChatOpenAI

from langchain.agents import AgentExecutor

from azure.identity import DefaultAzureCredential

创建 PowerBI Dataset Agent 涉及以下步骤：首先，通过创建一个或多个LLM（如fast\_llm和smart\_llm）来获取文本的嵌入表示。然后，通过实例PowerBIToolkit 来创建一个 Power BI 工具集，用于处理 PowerBI 数据集的任务。接着，通过调用 create\_pbi\_agent 方法来创建 PowerBI Dataset Agent，该代理可以与 PowerBI 数据集进行交互。

在数据集查询方面，有多种操作可用。

首先，通过调用agent\_executor.run("Describe table1")，代理可以提供关于数据表的描述信息。此外，代理还可以在数据表上执行简单的查询操作，比如计算数据表中的记录数，通过调用 agent\_executor.run("How many records are in table1?") 完成。同时，代理也能够执行更复杂的查询操作，比如按维度进行分组计算记录数，例如调用 agent\_executor.run("How many records are there by dimension1 in table2?") 将在 table2 中按 dimension1 维度计算记录数。

fast\_llm = ChatOpenAI(

temperature=0.5, max\_tokens=1000, model\_name="gpt-3.5-turbo", verbose=True

)

smart\_llm = ChatOpenAI(temperature=0, max\_tokens=100, model\_name="gpt-4", verbose=True)

toolkit = PowerBIToolkit(

powerbi=PowerBIDataset(

dataset\_id="<dataset\_id>",

table\_names=["table1", "table2"],

credential=DefaultAzureCredential(),

),

llm=smart\_llm,

)

agent\_executor = create\_pbi\_agent(

llm=fast\_llm,

toolkit=toolkit,

verbose=True,

)

例如，代理可以计算数据表中的记录数：

agent\_executor.run("How many records are in table1?")

我们还可以提供一些自定义的Few-shot Prompts，使模型更容易理解与Power BI数据集相关的问题和回答。提供这些提示可以帮助模型生成更准确的DAX查询。需要注意的是当与Power BI数据集进行互动时，请确保您有适当的权限和凭证。

LangChain的PowerBI Dataset Agent目前仍在积极开发中，可能存在不完善的地方，因此在生产环境中使用时应进行适当的测试。

# fictional example

few\_shots = """

Question: How many rows are in the table revenue?

DAX: EVALUATE ROW("Number of rows", COUNTROWS(revenue\_details))

----

Question: How many rows are in the table revenue where year is not empty?

DAX: EVALUATE ROW("Number of rows", COUNTROWS(FILTER(revenue\_details, revenue\_details[year] <> "")))

----

Question: What was the average of value in revenue in dollars?

DAX: EVALUATE ROW("Average", AVERAGE(revenue\_details[dollar\_value]))

----

"""

toolkit = PowerBIToolkit(

powerbi=PowerBIDataset(

dataset\_id="<dataset\_id>",

table\_names=["table1", "table2"],

credential=DefaultAzureCredential(),

),

llm=smart\_llm,

examples=few\_shots,

)

agent\_executor = create\_pbi\_agent(

llm=fast\_llm,

toolkit=toolkit,

verbose=True,

)

执行查询语句：

agent\_executor.run("What was the maximum of value in revenue in dollars in 2022?")

使用LangChain的PowerBI Dataset Agent与Power BI数据集进行互动为开发者提供了一种新的、自然的方式来查询和分析数据。通过这种方式，开发者可以更加灵活和直观地与数据互动，而不需要深入了解DAX查询语言的复杂性。

## 10.7 Retrievers 集成指南

Retrievers的集成重点在于为开发者提供方便、高效的信息检索工具。首先，Arxiv API Wrapper为那些需要访问和检索学术文献的开发者提供了专门的解决方案，确保他们能够无缝地从Arxiv数据库中获取所需的研究资料。其次，Azure Cognitive Search Wrapper为开发者提供了与Azure平台的深度集成，使其能够高效、准确地从Azure中检索各种信息。最后，Wikipedia API Wrapper则简化了从维基百科中提取内容的流程，让开发者无需深入了解其背后的技术细节，即可轻松获取所需的公开信息。

### 10.7.1 WikipediaRetriever集成

Wikipedia作为一个庞大的在线百科全书，为用户提供了丰富的知识和信息。使用LangChain的WikipediaRetriever，您可以轻松地从Wikipedia获取相关文档并对其进行查询。

首先，为了使用相关功能，安装 wikipedia Python包。

pip install wikipedia

接下来，让我们详细介绍 WikipediaRetriever 的参数配置。默认情况下，lang 参数设置为 "en"，它用于在特定语言的 Wikipedia 部分中进行搜索。load\_max\_docs 参数默认为100，它限制了下载文档的数量。在实验阶段，建议使用较小的数字，因为目前的硬上限是300。此外，load\_all\_available\_meta 参数默认为False，这意味着只下载最重要的字段，包括发布日期、标题和摘要。如果将其设置为True，则会下载其他字段。

运行 WikipediaRetriever 很简单，只需使用 get\_relevant\_documents() 方法，并输入自由文本查询作为参数。例如，如果要查询关于 "HUNTER X HUNTER" 的相关文档，您可以执行以下操作：

from langchain.retrievers import WikipediaRetriever

retriever = WikipediaRetriever()

docs = retriever.get\_relevant\_documents(query="HUNTER X HUNTER")

您可以查看文档的元数据或内容:

print(docs[0].metadata)

print(docs[0].page\_content[:400])

为了使用LangChain进行问题回答，您需要一个OpenAI API密钥。之后，您可以使用 ConversationalRetrievalChain 结合 ChatOpenAI 和WikipediaRetriever进行交互式的问答。例如：

from getpass import getpass

import os

from langchain.chat\_models import ChatOpenAI

from langchain.chains import ConversationalRetrievalChain

OPENAI\_API\_KEY = getpass()

os.environ["OPENAI\_API\_KEY"] = OPENAI\_API\_KEY

model = ChatOpenAI(model\_name="gpt-3.5-turbo")

qa = ConversationalRetrievalChain.from\_llm(model, retriever=retriever)

questions = ["What is Apify?", ...]

对于每个问题，您可以通过适当的代码获取并打印出答案。请注意以下几点事项：首先，在使用API密钥时，务必确保不要在公开的代码中暴露它，以保护您的安全。其次，与Wikipedia进行交互时，可能会受到频率限制或其他限制，因此要注意控制API请求的速率，以避免触发限制。

LangChain的WikipediaRetriever为开发者提供了一个简单而有效的方式，使其可以轻松地与Wikipedia互动，获取相关的文档并进行问题回答。这为开发者带来了巨大的便利，使他们可以更容易地从Wikipedia获取知识并将其应用到自己的应用中。

### 10.7.2 ArxivRetriever集成

arXiv是一个免费的分发服务和开放获取档案，截止2023年8月16日数据，该站点收录了2309934篇学术论文，涵盖了物理、数学、计算机科学、量化生物学、量化金融、统计学、电气工程与系统科学以及经济学等领域。ArxivRetriever是Langchain框架与arXiv的集成工具，可以用于从arXiv检索相关文档。

首先，要使用arXiv集成工具，需要安装arxiv Python包。可以使用命令“!pip install arxiv”来进行安装。

pip install arxiv

ArxivRetriever是Langchain框架提供的一个类，用于与arXiv进行交互。它有一些参数需要设置。其中，load\_max\_docs参数用于限制下载的文档数量。默认情况下是100，但是考虑到下载较多文档可能带来的问题，建议在测试时使用较小的数字。此外，还有一个load\_all\_available\_meta参数，默认为False。当设为False时，只会下载最重要的字段，包括发布日期、标题、作者和摘要。如果将其设为True，还会下载其他字段。

在使用示例中，可以看到如何实例化一个ArxivRetriever对象。在此例中，load\_max\_docs被设置为2，以限制下载的文档数量。然后，通过调用get\_relevant\_documents方法，输入一个查询关键字，可以从arXiv中检索相关的文档。

from langchain.retrievers import ArxivRetriever

retriever = ArxivRetriever(load\_max\_docs=2)

docs = retriever.get\_relevant\_documents(query="1605.08386")

以上代码展示了如何从Arxiv.org检索与query相关的文档。

当集成上述工具到LLM应用中时，开发者可能会遇到多种问题。例如，如何有效地将查询结果从ArxivRetriever传递给OpenAI Embeddings进行嵌入。答案是：通过先使用ArxivRetriever检索文档，再使用OpenAI Embeddings为其生成嵌入，从而实现流畅的工作流程。

### 10.7.3 Azure Cognitive Search集成

Azure Cognitive Search作为一个强大的云搜索服务，能够为开发者带来丰富的搜索体验。本文将详细指导开发者如何在LangChain中集成和使用Azure Cognitive Search。

Azure Cognitive Search（之前称为Azure Search）是一个云搜索服务，为开发者提供了建设丰富搜索体验的基础设施、APIs和工具。无论是文档搜索、在线零售应用还是私有内容的数据探索，搜索都是任何向用户展示文本的应用的基础。

在集成Azure Cognitive Search到LangChain之前，需要根据Azure的官方指南进行相应的配置，同时确保您已经获取了三个重要信息：ACS服务名称、ACS索引名称和API密钥。

一旦完成了配置和信息获取，您可以在LangChain中使用Azure Cognitive Search。首先，从langchain.retrievers模块中导入AzureCognitiveSearchRetriever类，这样您就能够在LangChain中利用Azure Cognitive Search的检索功能。

from langchain.retrievers import AzureCognitiveSearchRetriever

可以将ACS服务名称、索引名称和API密钥设置为环境变量。这样在创建检索器时，可以直接读取这些环境变量。

import os

os.environ["AZURE\_COGNITIVE\_SEARCH\_SERVICE\_NAME"] = "<YOUR\_ACS\_SERVICE\_NAME>"

os.environ["AZURE\_COGNITIVE\_SEARCH\_INDEX\_NAME"] = "<YOUR\_ACS\_INDEX\_NAME>"

os.environ["AZURE\_COGNITIVE\_SEARCH\_API\_KEY"] = "<YOUR\_API\_KEY>"

现在，可以使用上述环境变量创建Azure Cognitive Search检索器，并按需检索相关文档。

retriever = AzureCognitiveSearchRetriever(content\_key="content", top\_k=10)

documents = retriever.get\_relevant\_documents("what is langchain")

其中， top\_k 参数指定了返回的结果数量，可以根据实际需求进行调整。

集成Azure Cognitive Search到LangChain不仅提升了LLM应用的搜索功能，还使开发者能够更加便捷地利用Azure提供的强大搜索能力。希望本指南能够帮助开发者更轻松地在LangChain中实现这一集成。

## 10.8 本章小结

第十章详细探讨了LangChain的集成生态，展现了LangChain如何与各种流行的技术和工具相结合，为用户提供一个更加完整和高效的开发体验。在章节的初步，我们了解了集成的背景，特别是LangChain与LLM之间的集成方式和LangChain的整体集成生态。

深入到具体集成，LLM集成指南为我们展示了如何与Azure OpenAI和Hugging Face Hub相结合，突出了LLM在这两个平台上的功能和优势。随后，聊天模型集成指南讲述了如何与Anthropic、Google PaLM Chat以及Azure上的OpenAI端点进行集成，使得LangChain能够与多种聊天模型无缝对接。

而在向量库集成部分，我们被引导了解了如何将LangChain与Chroma、Pinecone和Milvus这三大主流向量库结合，从而优化和加速向量搜索和操作。进一步，嵌入模型集成指南为我们呈现了与HuggingFaceEmbeddings、Llama-cpp和Cohere的集成方法，确保LangChain能够有效地生成和管理文本嵌入。

在Agent toolkits集成部分，读者可以学习到如何将CSV、Pandas以及PowerBI Dataset Agent与LangChain集成，使数据处理和可视化变得更为轻松。最后，Retrievers集成指南涵盖了WikipediaRetriever、ArxivRetriever和Azure Cognitive Search的集成过程，确保LangChain可以有效地从各种来源检索信息。

总体而言，这一章为我们揭示了LangChain强大的集成能力，无论是与聊天模型、向量库、嵌入模型还是数据处理工具，LangChain都能够提供流畅和高效的集成解决方案，极大地提高了开发者的工作效率。

11. LLM 应用开发必学知识

这章本意在为对大语言模型(LLM) 和自然语言处理(NLP)感兴趣的初学者和开发者提供一个解释基础为主的指南。

NLP，即自然语言处理，是一个研究如何使计算机能够理解、解释和生成人类语言的学科。简单地说，NLP的目标是使计算机能够“理解”和“产生”人类语言，从而使机器能够与人类进行更自然的互动。

近年来，NLP已经在很多领域得到了广泛的应用，例如聊天机器人、搜索引擎优化、情感分析、自动文摘、机器翻译等。这是因为技术的进步使得NLP的应用更为普及，且其性能也在持续提高。

LLM，即大型语言模型，是这一技术进步的核心。LLM是一种特殊的NLP模型，它是通过在大量文本数据上进行训练来构建的。由于其规模之大，LLM能够捕获语言中的细微差异和复杂关系，从而在各种任务上实现出色的性能。

例如，OpenAI的GPT-4就是一个LLM的例子，它在100多种语言任务上都展现了出色的性能，甚至在某些任务上接近或超越了人类的表现。

了解LLM的核心知识和基本概念在开发LLM应用时是非常关键的，正如建筑的稳定性取决于其基础的坚固程度，同样，开发LLM应用的成功根基也依靠LLM的核心知识和基本概念。接下来，我们将探讨LLM的核心知识。这包括：

嵌入：这是一个关于如何在高维空间中表示单词和短语，以及如何使用这些表示来测量不同文本之间相似性的技术。

注意力机制：这是一种独特的技术，它允许模型在处理时专注于输入的某些特定部分。

Transformer模型架构：这是支撑许多当今最先进语言模型的关键架构。它的革命性之处在于其对NLP领域的影响。

语义搜索：我们将探索如何利用LLM理解查询的含义并找到最相关的结果。特别是，我们会看到这项技术如何彻底改变搜索行业。

随后，我们会深入NLP和机器学习的基础知识，包括：

历史背景：为初学者提供NLP的发展历程，帮助他们理解这个领域的起源和演变。

语言预处理技术：在实际应用NLP之前，必须对文本数据进行清洗和格式化，这是其中的关键步骤。

机器学习模型：自从NLP的起源以来，有哪些模型被开发并使用，以及它们的主要功能是什么。

分类器：这是一个关于如何训练和评估分类器模型的综合指南。

## 11.1 LLM的核心知识

首先，我们将探讨嵌入，这是在高维空间中的单词和短语的表示，以及如何使用它们来测量不同文本之间的相似性。

接下来，我们将探索注意机制，这是一种允许模型在处理期间专注于输入的特定部分的机制。我们将了解Transformer模型架构，这是许多最先进的语言模型的支柱，以及它是如何革新NLP领域的。

最后，我们将深入探讨语义搜索，即理解查询的含义并找到最相关结果的过程。我们将讨论如何使用LLM进行语义搜索，以及这项技术是如何转变搜索行业的。

### 11.1.2 文本嵌入的定义

文本嵌入，包括词和句子的嵌入，是语言模型的核心部分。在“2001太空漫游”这样的老式科幻电影中，主计算机（HAL）能够轻松地与人类交谈并理解他们说的话。当时，让计算机理解和产生语言似乎是不可能的任务，但最新的LLM（如GPT-4）已经能够做到这一点，使人类几乎无法判断他们是与另一个人还是计算机交谈。

NLP的基本任务是理解人类语言。但是，人类用词语和句子交谈，而计算机只能理解和处理数字。那么如何以连贯的方式将词语和句子转化为数字呢？这就是词嵌入所做的事情。我们可以将词嵌入视为为单词分配分数，这些分数具有一些很好的特性（很快我们就会了解到）。

什么是词嵌入？

让我们通过一个直观的测试来理解。在下图中，我在平面上标出了12个单词，包括“Banana”、“Basketball”等。

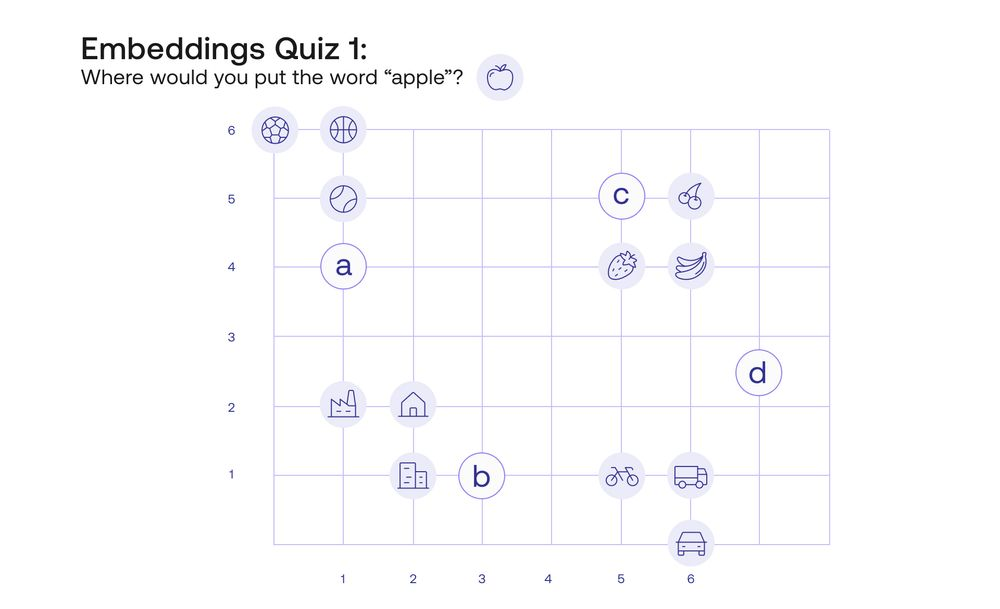


图 11- 1

现在的问题是，你会在这个平面上的哪个位置放置“Apple”这个词呢？最理想的位置是C点，因为“Apple”这个词与“Banana”、“Strawberry”和“Cherry”这些词都很接近，而与“House”、“Car”或“Tennis”这样的词距离较远。这就是词嵌入的实质。我们为每个单词分配的数字是什么呢？简单说，就是词的位置的横坐标和纵坐标。这样，“Apple”这个词就被分配到了[5,5]这个坐标，而“Bicycle”这个词被分配到了[5,1]这个坐标。

对于一个良好的词嵌入，它应具有以下特性：（1）相似的词应对应于接近的点（或等效地，对应于相似的分数）。（2）不同的词应对应于相隔较远的点（或等效地，对应于明显不同的分数）。

句子嵌入的重要性

词嵌入对于理解文本非常有用，但实际上，人类语言远比简单拼凑的词汇更为复杂。它拥有结构、句子等特点。那么，如何表示一个句子呢？

考虑一个简单的方法：如何对句子中的所有单词的分数进行求和？例如，有一个词嵌入为以下单词分配以下分数：

No: [1,0,0,0]

I: [0,2,0,0]

Am: [-1,0,1,0]

Good: [0,0,1,3]

那么，“No, I am good!”这个句子对应的向量是[0,2,2,3]。然而，“I am no good”这个句子也对应同样的向量[0,2,2,3]。这两个句子的含义相差甚远，但它们被解释为完全相同，这显然是不合适的。因此，需要更好的嵌入方法，考虑到单词的顺序、语言的语义和句子的实际含义。

这就引入了句子嵌入的概念。句子嵌入与词嵌入类似，只是它将每个句子与一个充满数字的向量相关联。这种关联方式保证了相似的句子被分配到相似的向量，不同的句子被分配到不同的向量，并且向量的每个坐标都表示句子的某种属性。

嵌入的实用性

文本嵌入已经证明了其重要性，现在是时候开始探索它们的实用性了。以下短语为例：

I like my dog  
I love my dog  
I adore my dog  
Hello, how are you?  
Hey, how's it going?  
Hi, what's up?  
I love watching soccer  
I enjoyed watching the world cup  
I like watching soccer matches

模型返回的嵌入数据显示,相同含义的语句，在向量空间内距离接近。

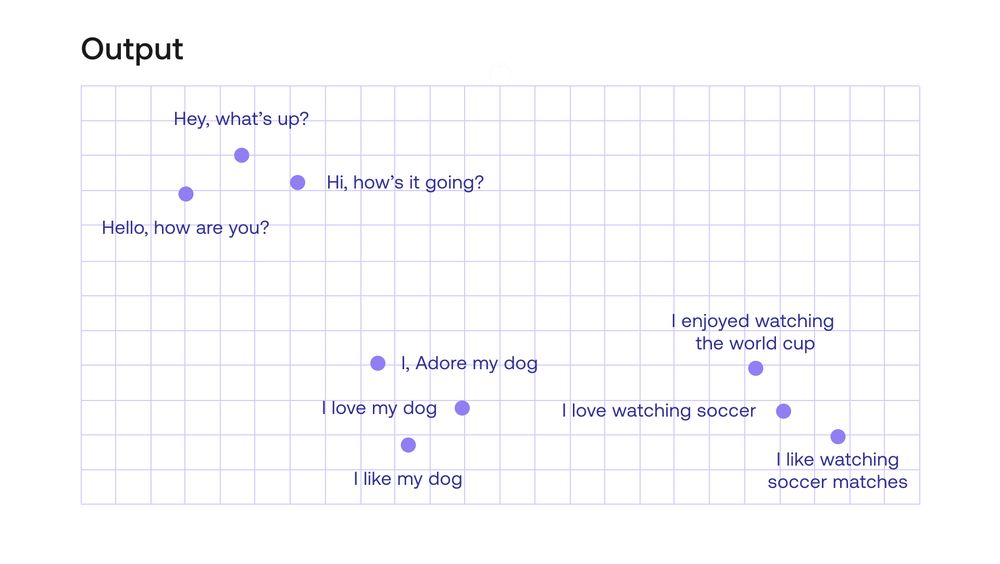


图 11- 2

多语言句子嵌入

大多数词和句子嵌入都依赖于模型受过训练的语言。但在全球化的今天，多语言模型变得尤为重要。OpenAI已经训练了一个大型的多语言模型，支持超过100种语言。以下是几个英语、法语和西班牙语的句子示例：

The bear lives in the woods  
El oso vive en el bosque  
L’ours vit dans la foret  
The world cup is in Qatar  
El mundial es en Qatar  
La coupe du monde est au Qatar  
An apple is a fruit  
Una manzana es una fruta  
Une pomme est un fruit  
El cielo es azul  
The sky is blue  
Le ciel est bleu

模型返回的嵌入数据显示，它能够识别关于熊、足球、苹果和天空的句子，即使它们是用不同的语言编写的。

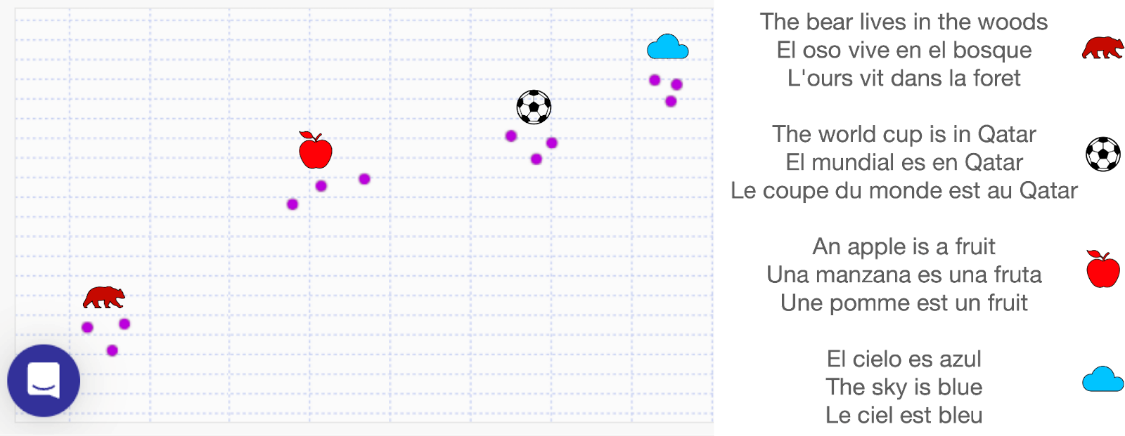


图 11- 3

文本嵌入在LLM中的角色

词和句子嵌入是LLM的基石。它们是大多数语言模型的基本构建块，因为它们将人类的语言（词语）转化为计算机语言（数字），同时捕获词语之间的关系、语言的语义和细微差别。句子嵌入可以扩展为语言嵌入，其中每个句子的数字都与语言无关。这些模型对于翻译以及在不同语言中搜索和理解文本非常有用。

对于LLM，判断两个词或两个句子是否相似或不同是至关重要的。幸运的是，词和句子嵌入为此提供了有力的工具。

### 11.1.3 点积和余弦相似性

我们将探讨几种不同的相似性概念，并为开发者展示如何在LLM应用中有效地使用这些概念。

对于每个LLM，知道两个词或两个不同的句子是否相似或不同是非常关键的。词和句子嵌入为此提供了有力的工具。简而言之，词嵌入将每个词与一组数字（向量）关联起来，这样词的语义属性就可以转化为数字的数学属性。句子嵌入则更为强大，因为它们将每个句子与一组数字关联起来，这些数字也携带了句子的重要属性。例如，OpenAI平台的OpenAI嵌入为每个句子分配一个长度为4096的向量。

了解嵌入的基础后，我们可以使用它们来查找相似性。一旦得到了文本的嵌入，我们可以计算它们之间的相似性。本文将定义两种相似性：点积相似性和余弦相似性。这两种都是确定两个词（或句子）是否相似的有用方法。点积相似性和余弦相似性是两种常用的计算方法。

点积相似性

为了简化问题，考虑一个只有4个句子的数据集，每个句子都被分配了两个数字。例如，电影标题“Rush Hour”和“Rush Hour 2”被分配了相似的数字，因为它们在某种程度上是相似的。

You’ve Got Mail: [0, 5]  
Rush Hour: [6, 5]  
Rush Hour 2: [7, 4]  
Taken: [7, 0]

点积是一种创建相似性分数的方法。在这个方法中，如果两部电影的得分匹配，那么乘以两部电影的行动分数，然后乘以两部电影的喜剧分数，并加它们，这个数字就会很高。例如Rush Hour: [6, 5]，6代表行动分数6分，5代表喜剧分数是5分。计算过程为：

[You’ve got mail, Taken] = 0\*7 + 5\*0 = 0

[Rush Hour, Rush Hour 2] = 6\*7 + 5\*4 = 62

电影标题“Rush Hour”和“Rush Hour 2”被分配了相似的数字，计算的结果是62，数字就很高，而“You’ve got mail”和“Taken”的计算结果是0。这个例子直观地反映了两个向量之间的相似性。对于相似的句子，它们的嵌入向量的点积会很大；而对于不相似的句子，点积则相对较小。

余弦相似性

在开发LLM应用时，经常需要对句子或词语之间的相似度进行量化评估。其中，一种广泛使用的方法是余弦相似性。

余弦相似性基于向量间的夹角来衡量它们之间的相似度。这种方法特别适用于评估高维空间中的数据点之间的相似度，例如在LLM应用中的文本嵌入。

考虑一个简化的例子：在二维平面上，将电影嵌入为点，其中横坐标表示动作得分，纵坐标表示喜剧得分。电影的嵌入可能看起来像点在平面上。例如，“You’ve Got Mail”与“Taken”之间的距离很远，因为它们是非常不同的电影。而“Rush Hour”与“Rush Hour 2”非常接近，因为它们是相似的电影。

虽然欧几里得距离可以测量两点之间的距离，但它不总是能够很好地表示相似性。特别是当数据点在高维空间中非常接近时，角度测量更为合适。

这里，我们引入余弦相似性。余弦相似性衡量的是从原点出发到两句子所形成的两射线之间的夹角的余弦值。当两点非常接近时，这个角度会很小，其余弦值接近1，表示它们之间的相似度很高。

例如，在之前的电影示例中，“You’ve Got Mail”与“Taken”之间的角度为90度，其余弦值为0，表示它们之间的相似度为0。而“Rush Hour”与“Rush Hour 2”之间的角度为11.31度，其余弦值为0.98，表示它们之间的相似度非常高。

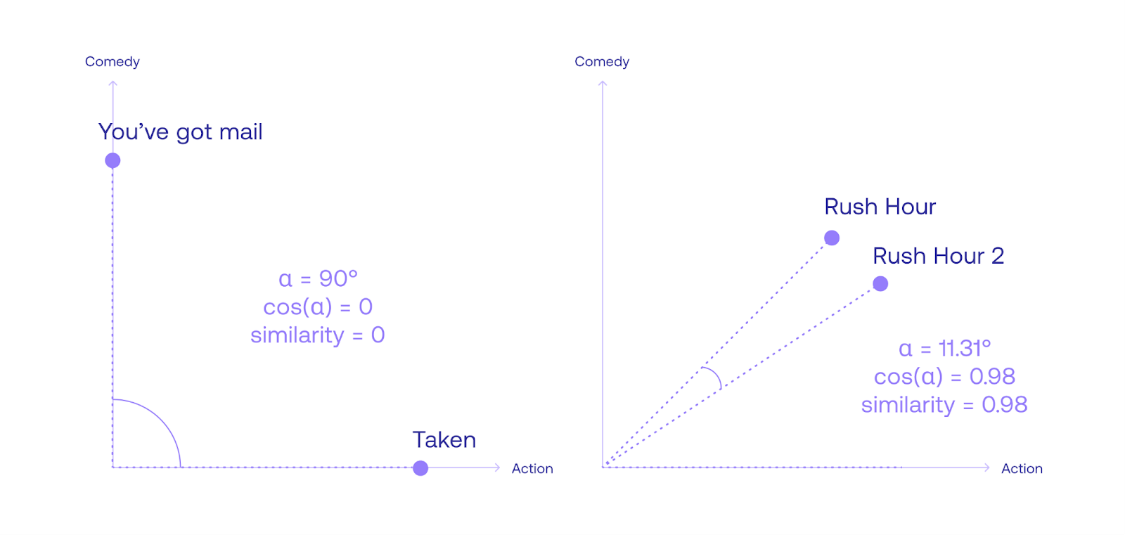


图 11- 4

### 11.1.3 注意力机制

在LLM应用的开发过程中，一个核心的技术挑战是如何准确处理多义词。为了有效解决这一问题，LLM引入了注意力机制。

在之前，我们已经了解了词嵌入和句子嵌入以及如何衡量词汇和句子之间的相似性。简而言之，词嵌入是一种将词与数字列表（向量）相关联的方法，使得相似的词产生相近的数字，而不同的词产生距离较远的数字。

但是，词嵌入面临一个重要的问题：如何处理具有多种定义的词。例如，单词“bank”可以指河岸或金融机构。传统的词嵌入为“bank”分配相同的向量，不考虑其上下文。为了解决这一问题，我们需要注意力机制。注意力机制可以根据上下文为单词提供特定的向量，从而为单词提供上下文信息。

为了理解注意力机制，考虑以下两个句子：

句子1：“The bank of the river.”

句子2：“Money in the bank.”

在这两个句子中，“bank”的含义完全不同。第一个句子中的“bank”指的是河岸，而第二个句子中的“bank”指的是金融机构。如何让计算机理解这两种不同的含义呢？

解决的关键是查看邻近的词。在第一个句子中，“river”这个词为我们提供了上下文，而在第二个句子中，“money”为我们提供了上下文。因此，为了理解“bank”的上下文，我们需要考虑其他单词。

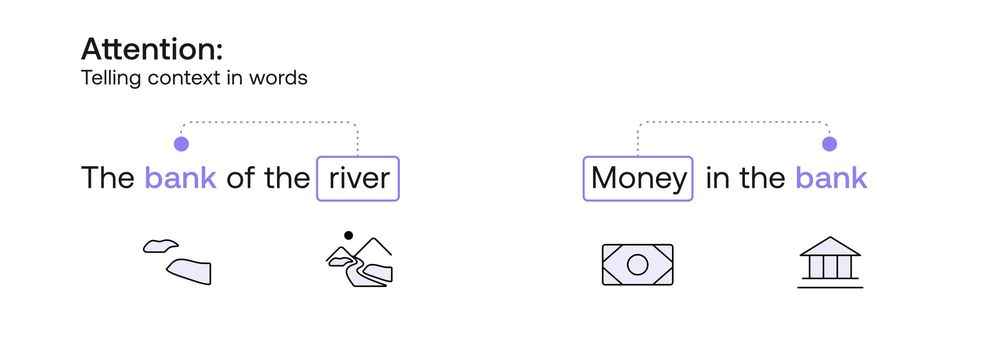


图 11- 5

这就是注意力机制的工作原理。它考虑了句子中的所有词，并为目标词（如“bank”）提供上下文信息。注意力机制可以为每个词提供一个与上下文相关的向量，从而使LLM能够更准确地理解每个词的含义。

为了在数学模型中表示词的上下文关系，我们可以通过调整词嵌入的向量来“移动”一个词更靠近另一个词。例如，为了使“bank”更接近“river”，我们可以将其向量与“river”向量的加权平均进行混合。权重可以基于两个词之间的相似性来决定。

假设我们有以下词向量：

River: [0,5]

Money: [8,0]

Bank: [6,6]

我们假设还有2个新的嵌入向量：

Bank1（与“river”更接近）: [5.4, 5.9]

Bank2（与“money”更接近）: [6.4, 4.8]

如你所见，“bank1”更接近“river”，而“bank2”更接近“money”。 把方括号内的两个数字看成平面上的坐标，其中第一个数字是水平坐标，第二个是垂直坐标。Bank1的纵坐标5.9更接近“river”的5。 Bank2的横坐标6.4更接近“money”的8。与通过生成上下文敏感的嵌入新的向量，我们能够更准确地捕捉到一个词在不同语境中的含义。在上面的例子中，通过创建两个上下文敏感的嵌入“Bank1”和“Bank2”，我们能够区分“bank”这个词在与“river”和“money”两种上下文中的不同含义。

当在实际的句子或段落中使用“bank”这个词时，我们会依赖于这些上下文敏感的嵌入来确定其真正的意思。例如，如果句子中提到了“river”，则更可能使用“Bank1”这个嵌入，因为它更接近“river”的上下文。相反，如果句子中涉及金融或存款，我们则可能使用“Bank2”这个嵌入，因为它与“money”的上下文更为接近。

这种方法使我们可以为多义词创建多个上下文相关的嵌入，从而为LLM应用提供更准确的表示。

## 11.2 Transformer模型

Transformer模型在机器学习领域中迅速崭露头角，特别是在处理文本上下文时表现出色。为了帮助开发者深入理解这一技术并在LLM应用中发挥其最大潜力，本章将详细探讨Transformer的架构及其工作原理。

Transformer模型是机器学习中最引人注目的新发展之一。这些模型能够撰写故事、随笔、诗歌，回答问题，进行语言翻译，与人类交流，甚至通过对人类来说困难的考试！但它们究竟是什么呢？幸运的是，Transformer模型的架构并不复杂，它只是一些非常有用组件的连接，每个组件都有其特定的功能。

简而言之，Transformer模型如何工作？当输入一个简单的句子，如“Hello, how are”，Transformer可以预测出最可能的下一个词，例如“you”。这是因为Transformer能够跟踪所写文本的上下文，从而使生成的文本有意义。

这种逐词构建文本的方法可能与人类形成句子和思考的方式不同，但这正是Transformer模型如此出色的原因：它们能够非常好地跟踪上下文，从而选择恰当的下一个词汇。

那么，Transformer模型是如何训练的呢？答案是使用大量的数据，事实上，是互联网上的所有数据。基于这些数据，当开发者输入“Hello, how are”时，Transformer知道基于所有互联网文本，下一个最佳词汇是“you”。

现在，让我们深入了解Transformer的架构。Transformer有四个主要部分：标记化、嵌入、位置编码、Transformer block（多个）、Softmax。

其中，Transformer block是最复杂的部分。多个这样的块可以被连接起来，每一个都包含两个主要部分：注意力和前向馈送组件。

1. 标记化(Tokenization) 标记化是文本处理的第一步。它涉及将每个单词、前缀、后缀和标点符号转换为一个已知的令牌。例如，句子“Write a story”将被转换为四个相应的令牌：<Write>, <a>, <story>和<.>。

2. 嵌入(Embedding) 经过标记化后，下一步是将这些令牌转换为数字，这就是嵌入的作用。它将每个令牌映射到一个数字向量，如果两个文本片段相似，它们对应的向量也会很相似。

3. 位置编码(Positional encoding) 为了确保句子中的每个单词在处理时能够保持其原始位置信息，引入了位置编码。它通过添加一系列预定义的向量到每个词的嵌入向量来实现。

4. Transformer block。Transformer的核心是由多个Transformer block组成的。每一个Transformer block都包含两个主要部分：注意力组件和前馈组件。

5. 注意力(Attention) 注意力机制是Transformer模型中的关键技术，它能够为每个单词提供上下文信息。例如，在句子“The bank of the river”和“Money in the bank”中，单词“bank”的含义在两个句子中是不同的。注意力机制通过分析句子中的其他单词（river和money）来为每个单词提供上下文，确保其在生成或处理文本时具有正确的含义。

为了更进一步增强这一机制的能力，引入了多头注意力(multi-head attention)技术，它使用多个嵌入来修改向量并为它们添加上下文。

总之，Transformer模型为LLM应用开发者提供了一个强大的工具，可以处理和生成高质量的文本。通过深入理解这些技术，开发者可以为最终用户创建更加高效和流畅的LLM应用。

### 11.2.1 Softmax层和后训练

在现代LLM应用开发中，Transformer模型已经成为了一个不可或缺的组件，为开发者提供了强大的文本处理和生成能力。以下是关于Transformer模型的深入技术分析，旨在为初学者和有意向使用LLM构建应用的开发者提供指导。

1. Softmax层 Transformer模型通过多层的Transformer block来构建，每一层都包含注意力和前馈层，从而形成了一个大型的神经网络，用于预测句子中的下一个单词。Transformer为所有单词输出分数，并为句子中最可能的下一个单词给出最高分数。

Softmax层的作用是将这些分数转化为概率值。例如，Transformer为单词“Once”给出了0.5的概率，而为“Somewhere”和“There”分别给出了0.3和0.2的概率。通过采样，选择概率最高的单词作为输出。

2. 后训练(Post Training) 。虽然了解了Transformer的基本工作原理，但为了使其在实际LLM应用中发挥出更好的效果，还需要进行后续的训练。例如，当询问Transformer“Algeria的首都是什么？”时，理想的回答是“Algiers”。但由于Transformer是基于整个互联网进行训练的，可能会给出不同的答案。

为了改善这种情况，可以进行后训练，即在整体训练完成后，再对模型进行特定任务的训练。这就像对人进行特定任务的培训一样。通过后训练，可以使Transformer在特定任务上，如回答问题、进行对话或编写代码上表现得更好。

Transformer模型为LLM应用开发带来了巨大的潜力。通过深入理解其工作原理和技术细节，开发者可以更有效地利用这一工具，为最终用户创造出更加强大、高效的LLM应用。

## 11.3 语义搜索

在LLM应用开发的世界中，语义搜索已经成为了一个核心技术。与传统的关键字搜索相比，语义搜索提供了更高的准确性和灵活性，使得开发者可以为用户提供更加丰富和准确的搜索体验。

语义搜索使用文本嵌入和相似度来构建一个查询模型。与此不同，传统的关键字搜索依赖于查询和响应之间共同词汇的数量。但是，这种方法往往无法捕捉到文本中的真正含义。例如，考虑以下查询和一组响应：

查询：世界杯在哪里？

响应：

世界杯在卡塔尔。

天空是蓝色的。

熊住在森林中。

苹果是一种水果。

传统的关键字搜索可能会选择与查询拥有最多共同词汇的响应，但这并不总是返回正确的答案。而语义搜索则会选择语义上与查询最匹配的响应。

文本嵌入是将每个文本片段（可以是一个单词或一个完整的文章）转换为一个数字向量的方法。这些向量可以使用各种算法（如OpenAI的嵌入模型）生成，并可以通过降维算法减少到更易于处理的尺寸。这些向量可以被绘制在平面上，使我们可以可视化查询和响应之间的距离。

尽管可以使用欧几里得距离来测量查询和响应之间的距离，但相似度通常提供了更好的结果。通过比较文本嵌入向量之间的相似度，可以确定哪些响应与给定查询最匹配。

在现代LLM应用开发中，语义搜索已经成为一个不可或缺的技术。这一技术的核心在于文本嵌入和相似度的计算，它们共同为开发者提供了一个强大的工具来增强用户的搜索体验。

在之前的章节中，我们简要介绍了相似度的两种计算方式：点积相似度和余弦相似度。这两种方法都可以用于判断两个文本片段是否相似。余弦相似度特别有趣，因为它的取值范围在0到1之间，可以直观地表示文本之间的相似性。

### 11.3.1 语义搜索的工作原理

首先：什么不是语义搜索？ 在我们学习语义搜索之前，让我们看看什么不是语义搜索。在语义搜索之前，最流行的搜索方式是关键字搜索。想象一下，你有很多句子的列表，这些句子是回应。当你提问（查询）时，关键字搜索会查找与查询中共有的单词数量最多的句子（回应）。例如，考虑以下查询和一组回应：

查询：世界杯在哪里？

回应：

世界杯在卡塔尔。

天空是蓝色的。

熊住在森林里。

苹果是一种水果。

通过关键字搜索，你可以注意到回应与查询有以下共同的单词数量：世界杯在卡塔尔。 （4个共同的词） 天空是蓝色的。 （2个共同的词） 熊住在森林里。 （2个共同的词） 苹果是一种水果。 （1个共同的词） 在这种情况下，获胜的回应是1号，“世界杯在卡塔尔”。幸运的是，这是正确的回应。但是，情况并非总是如此。想象一下，如果有另一个回应：

我杯中的咖啡在世界的哪个地方？

此回应与查询有5个共同的词，所以如果它在回应列表中，它就会获胜。这是不幸的，因为这不是正确的回应。

我们该怎么办？我们可以通过删除“the”、“and”、“is”等停用词来改进关键字搜索。我们还可以使用TF-IDF等方法来区分相关和非相关的词。然而，如你所想，总会有一些情况，由于语言的模糊性、同义词和其他障碍，关键字搜索将无法找到最佳的回应。所以我们转向下一个表现非常好的算法：语义搜索。

简而言之，语义搜索的工作原理如下：（1）它使用文本嵌入将单词转换为向量（数字列表）。（2）使用相似性来找到回应中与查询对应的向量最相似的向量。（3）输出与这个最相似的向量对应的回应。

要执行语义搜索，首先要计算查询和每个句子之间的相似度，然后返回相似度最高的句子。对于LLM应用开发者来说，这意味着可以通过简单的算法迅速找到与查询最相关的答案，从而为用户提供更精确的搜索结果。

执行语义搜索的常用算法是最近邻算法，这是一个简单且实用的算法，通常用于分类。在这个上下文中，最近邻算法会查找数据集中与给定点最近的点。然而，该算法在大型数据集中可能效率较低。为了提高效率，开发者可以使用近似最近邻算法或其他优化策略，如Inverted File Index和Hierarchical Navigable Small World。

LLM应用开发者应当注意到，语义搜索的性能高度依赖于文本嵌入的质量。新的多语言嵌入模型为开发者提供了一个强大的工具，支持100多种语言的搜索。这意味着开发者可以使用任何一种语言的查询，并在所有其他语言中搜索答案。

尽管文本嵌入和相似度在语义搜索中发挥了关键作用，但它们并不总是能够提供最佳的搜索结果。例如，考虑一个查询：“世界杯在哪里？”。虽然正确的答案是“世界杯在卡塔尔”，但模型可能会返回与查询语义上更接近的其他响应，如“上届世界杯在俄罗斯”。

在进行相似性搜索时，我们的目标是找到文本中的语义意义，而不仅仅是基于表面上的词汇匹配。这需要对语言的深入理解和处理，而这正是NLP的核心。

## 11.4 NLP与机器学习基础

当我们执行相似性搜索时，我们追求的是文本的深层次含义，而不只是停留在字面上的词语对应。为了实现这一点，我们需要深入地理解和处理语言，这也是NLP的关键所在。

NLP是计算机科学、人工智能和语言学的交叉领域，致力于让计算机能够理解、解释和产生人类语言。

首先，我们将回顾NLP的历史背景及其在LLM应用中的各种实际应用。我们还将介绍处理自然语言文本所面临的各种挑战及相关技术。

随后，我们将专注于文本预处理的主题。为了确保LLM应用的高效性和准确性，我们需要对文本数据进行清洗和准备。这一过程涉及到如何将文本分解为单词或符号（tokenization），如何找到词的基本形式（stemming）和词形还原（lemmatization）。完成这些步骤后，我们将探讨如何将文本转化为向量，以便模型能够处理它们。

此外，为了使LLM应用更加智能，我们将深入研究构建和评估分类器的技术。分类器是能够自动将文本分类到不同类别或类的算法。在此，我们将涉及到特征工程、监督学习和评估指标等关键技术。

### 11.4.1 NLP的演变历程

NLP是一种允许计算机理解、解释和翻译人类语言的人工智能子领域，并能生成文本。其目的是通过处理非结构化文本数据，促进计算机与人类之间的更好沟通。

结合计算机科学和计算语言学的技术，NLP尝试缩小人与计算机之间的沟通鸿沟，使计算机能够读取文本、听取语音并理解信息含义。NLP广泛分为两大领域：自然语言理解和自然语言生成，这两者虽然都属于NLP，但具有明显的差异：

自然语言理解涉及分析非结构化文本的语义和语法。自然语言生成则使用结构化数据生成非结构化文本。

从第二次世界大战后，人们对于能够从一种语言翻译到另一种语言的机器产生了极大的需求。然而，当时使用的技术非常原始，主要依赖于字典查找和硬编码规则来排序单词。

在1950年代末，研究者们开始认识到开发NLP的某些问题。其中一个研究者是Noam Chomsky，他观察到NLP中使用的模型应该能够像人类一样识别语法正确的句子。他在1957年出版的《句法结构》一书中，革命性地介绍了如何在计算机中表示语言学概念。他声称语法是生成的，并为计算机中的语言引入了一个数学模型。

从1957年到1970年代，开发NLP的研究者之间出现了冲突观点，这导致了两组NLP的出现：基于语言学的NLP和基于统计的NLP。在这个时代，基于语言学的NLP取得了显著的进展，但结果仍然不尽如人意。

直到1980年代，带有NLP功能的计算机系统还在使用复杂的“硬编码”规则。计算机科学家们意识到，自然语言问题的孤立问题并不有效。这一挑战导致了其他科学领域的合并，包括语言学。因此，为了弥合计算机科学和语言学之间的差距，进行了更多的研究。

随着计算能力的增强，1980年代后期见证了机器学习算法的转变，决策树是最早采用的算法之一，它们能够使用概率和统计来处理NLP。

进入2000年后，研究者们开始使用深度学习算法来解决NLP中的问题。随着可用的文本数据的大幅增加、更快的CPU和更多的存储空间，训练LLM变得更加容易，这些模型使用神经网络提供更好的结果。研究者们使用卷积神经网络进行句子分类、情感分析和文本摘要等任务。

目前，现代NLP中的Transformer也为其发展做出了贡献，解决了如问答系统、机器翻译等问题。NLP中的Transformer是一种新型的神经网络架构，专门处理序列到序列的任务。在深度学习应用中，它们有效地将输入序列转化为输出序列。

有了这么多的进步，神经网络的应用已经大大增长，现在你可以使用大型的预训练模型来创建NLP驱动的应用。

LLM应用开发中的传统机器学习方法

随着深度学习的出现和流行，NLP领域发生了巨大的变革。然而，在深度学习成为主流之前，研究者们使用了各种传统方法来构建NLP模型。这些方法在当今仍然有其应用场景，尤其是在某些特定的NLP任务中。

以下是几种用于执行NLP任务的流行机器学习方法及其简要描述：

N-Gram模型：这是一种基于训练的概率模型，可以估计文本中词序列的概率分布。

Logistic回归：这是一个在文本分类任务中非常流行的算法，它使用概率生成函数为任何给定的数字输入产生一个0到1之间的值。

贝叶斯：这是一个使用概率预测输出类别的可能性的监督算法。它使用贝叶斯定理，假设数据中的特征是相互独立的。

Markov模型：这些模型非常适合序列数据，因为它们可以预测随机变量序列的概率。

作为一名开发者，特别是对于那些希望使用LLM构建应用的初学者，了解这些传统方法对于整体理解NLP技术发展有很大帮助。尽管现代技术已经发展得很快，但这些传统方法在某些场景中仍然很有价值。

### 11.4.2 NLP文本预处理

在构建LLM应用时，开发者通常会发现，处理和清理数据是实现最佳模型性能的关键。在NLP领域，这一步骤通常被称为文本预处理。它不仅提高了数据的质量，还最终增强了模型的表现力。

文本预处理的主要目标是将嘈杂的文本转化为机器学习模型可以理解的形式。这一过程为进一步的分析和报告准备了清晰、有序的文本数据。

当开发者在NLP项目中收集文本数据时，通常首先从数据库或非正式设置（如博客帖子、社交媒体、电子商务网站、消息板、新闻组和维基）中提取数据。由于文本数据的非结构化特性，其格式和质量可能会有所不同，这可能会引入噪声，影响构建的NLP模型的性能。因此，学习如何有效地预处理这些数据是至关重要的。

文本数据清理

以下是一些用于清理文本数据的众所周知的方法：

Tokenization（标记化）：这是NLP中的一个基础步骤，它将文本分解为较小的块，如词、短语、符号或其他有意义的元素。

Normalization（规范化）：规范化是将具有相似含义的多个单词标准化的过程，将它们转化为单一的规范形式。

Stop Word Removal（停用词移除）：某些词在文本数据中频繁出现，但通常不增加数据的实际意义，这些词被称为停用词。

Stemming（词干提取）：通过移除词的前缀和后缀来清理文本数据。

Lemmatization（词形还原）：这是NLP中使用的一个过程，它将一个词的各种屈折形式转化为其基本形式或词典形式。

文本预处理是为LLM应用优化数据的关键一步。这一过程确保了文本数据的清晰性和一致性，为开发者提供了一个坚实的基础，以构建和训练高性能的NLP模型。

LLM应用开发：从文本到向量的转换

在LLM应用开发的旅程中，将文本转化为数字或向量形式是一个核心步骤。此过程被称为文本向量化，其目的是为各种机器学习算法提供数值输入。

机器学习和深度学习模型需要数字数据作为输入，因为它们不能像人类那样直接处理数据。以下是执行文本向量化的一些常用方法：

One Hot Encoding（一热编码）：此方法采用文本数据中的唯一单词并为每个单词生成向量。

Count Vectorizers（计数向量化器）：与一热编码类似，但它能够捕获单词在文本数据中的出现频率。

Bag of Words（词袋模型）：这是一种提取文本数据特征的方法，不考虑单词出现的顺序。

N-Grams（N元模型）：N-grams代表句子中彼此相邻的一系列单词或标记。

TF-IDF（词频-逆文档频率）：这是一个表示单词在文本中重要性的比率。

为开发者准备的LLM应用中，上述方法可以帮助将原始文本转化为模型可以理解的格式，从而实现更高的模型性能和准确性。

### 11.4.2 构建分类器

在机器学习领域，分类器的任务是为输入数据分配一个类别或类。开发者可以选择使用监督或非监督模型。本章将为初学者和希望使用LLM构建应用的开发者展示如何构建一个简单的监督分类模型。

数据加载与清洗

构建分类器第一步通常是数据收集和清洗。开发者需要确定数据来源是数据库、文件还是在线网页。根据数据来源，选择相应的脚本加载数据。

加载数据后，使用以下方法清洗文本数据以去除噪声并提高其质量：（1）通过分词将数据集中的句子拆分成单词。（2）使用停用词典，通过停用词移除过程过滤掉文本数据中的所有停用词、数字和标点符号。（3）检查所有带有前缀或后缀的剩余单词，并通过词干提取过程将这些单词转换为根词。

文本向量化

为了使机器学习算法能够处理数据，开发者需要将文档转换为数字表示。可以使用词袋算法来实现这一点。此算法将遍历所有数据中的文档。

在此步骤中，确保使用单词出现的频率作为创建向量的评分方法。同时，确保将目标标签转换为数字。

模型训练

为了构建监督分类器，首先确定要使用的算法，经常使用的是贝叶斯算法。对于语料库中的每个文档，现在都有其关联的向量表示和目标变量。

在训练模型之前，首先将数据分为两组：训练数据和测试数据。建议训练集占数据的70%，测试集占剩下的30%。

使用训练数据集训练贝叶斯算法，其中向量作为自变量，目标标签作为因变量。这将输出适合的模型。

分类器评估

对于希望利用LLM构建应用的开发者来说，评估模型的表现是至关重要的步骤。在分类任务中，我们的目标是预测输入数据所属的类别。例如，我们可能需要判断一封电子邮件是否为垃圾邮件。

为了确保分类器的有效性，以下是四个最常用的评估指标：

**准确度 (Accuracy)**：这是一个简单的指标，用于测量分类器的整体效果。但仅在目标类别平衡的情况下使用这个指标可能会产生误导。

**精确度 (Precision)**：这个指标衡量模型预测为正的结果中有多少是真正的正样本。对于垃圾邮件过滤，这意味着被标记为垃圾邮件的消息中有多少实际是垃圾邮件。

**召回率 (Recall)**：召回率量化了模型能正确识别的正样本数。在垃圾邮件过滤的例子中，召回率是指被发送到垃圾邮件文件夹的垃圾邮件与实际垃圾邮件总数的比例。

**F1得分 (F1-Score)**：这是一个用于评估二元分类器的指标，通过计算召回率和精确度的调和平均值得到。

此外，混淆矩阵是一个能够显示分类模型性能的表格，它比较了实际值和预测值。这四种结果（真阳性、假阳性、真阴性、假阴性）构成了混淆矩阵，它为不平衡的数据提供了很好的评估。

选择合适的指标是至关重要的，因为不同的场景可能需要对精确度和召回率进行权衡。此外，阈值的选择对于将概率值转化为目标类标签也非常关键。通常，提高一个指标可能会降低另一个指标。

在LLM应用开发中，了解如何评估分类器的表现是至关重要的。这些评估技术不仅帮助开发者跟踪模型的性能，还为开发者提供了调整和优化模型的手段。

### 11.5 本章小结

在这一章中，我们深入探讨了LLM的核心知识和基本概念。我们首先介绍了嵌入技术，解释了如何在高维空间中表示文本，并使用这些表示来测量文本之间的相似性。接着，我们详细探讨了注意力机制，这是一种让模型在处理数据时更加聚焦的技术。Transformer模型架构作为当今许多先进语言模型的核心，其对NLP领域的革命性影响也被详细讨论。此外，我们还介绍了如何利用LLM进行语义搜索，以及这种方法如何颠覆了搜索行业的传统方式。

在章节的后半部分，我们转向了NLP和ML的基础知识。我们概述了NLP的发展历程，为读者提供了这一领域的历史背景。为了有效地应用NLP技术，我们介绍了关于语言预处理的关键步骤和技术。然后，我们回顾了自NLP起源以来所使用的各种机器学习模型及其功能。最后，我们详细讨论了分类的概念，提供了一个关于如何训练、评估分类模型并在实际中应用它们的全面指南。

总的来说，这一章为读者提供了LLM、NLP和ML的综合和深入的知识，为开发良好的LLM应用程序打下了坚实的理论基础。

附录A

## BasePromptTemplate 类、方法与属性速查

（一）BasePromptTemplate 类 在LLM应用中，BasePromptTemplate 类为所有提示模板提供了基础，其主要功能是返回适当的提示。

（二）类名: BasePromptTemplate

（三）属性和方法:

|  |  |
| --- | --- |
| input\_variables: | 一个属性，存储模板预期的变量名列表。 |
| output\_parser: | 一个可选属性，用于解析调用此格式化提示的LLM输出。 |
| partial\_variables: | 一个默认为字典的属性，存储部分变量的映射。 |
| lc\_serializable: | 一个属性，如果对象可以序列化，则返回True。 |
| format\_prompt: | 一个抽象方法，用于根据给定的关键字参数创建聊天消息。 |
| validate\_variable\_names: | 一个方法，用于确保变量名不包含受限制的名称。 |
| partial: | 一个方法，返回提示模板的部分实例。 |
| \_merge\_partial\_and\_user\_variables: | 一个方法，用于合并部分和用户变量。 |
| format: | 一个抽象方法，用于格式化提示，接受任意关键字参数并返回一个格式化的字符串。 |

## BaseLLM类、方法与属性速查

（一）BaseLLM 类

在LLM框架中，BaseLLM 类为大型语言模型（LLM）提供了一个核心的接口，其定义了与模型的基本交互方式。

（二）类名: BaseLLM

（三）属性和方法:

|  |  |
| --- | --- |
| cache: | 用于确定是否缓存模型的结果。 |
| verbose: | 用于决定是否打印响应文本。 |
| callbacks: | 定义了在模型运行过程中的回调函数。 |
| tags: | 用于向运行追踪添加标签。 |
| \_generate: | 一个方法，用于在给定的提示和输入上运行LLM。 |
| \_agenerate: | 一个异步方法，同样用于在给定的提示和输入上运行LLM。 |
| generate\_prompt: | 用于将提示转换为字符串并在LLM上运行。 |
| agenerate\_prompt: | 一个异步方法，用于将提示转换为字符串并在LLM上运行。 |
| generate: | 用于在给定的提示和输入上运行LLM。 |
| agenerate: | 一个异步方法，用于在给定的提示和输入上运行LLM。 |
| call: | 用于在给定的提示和输入上运行LLM并返回字符串。 |
| \_call\_async: | 一个异步方法，同样用于在给定的提示和输入上运行LLM并返回字符串。 |
| predict: | 根据输入的文本进行预测。 |
| predict\_messages: | 根据输入的消息列表进行预测。 |
| apredict: | 一个异步方法，用于根据输入的文本进行预测。 |
| apredict\_messages: | 一个异步方法，用于根据输入的消息列表进行预测。 |
| \_identifying\_params: | 获取标识参数。 |
| \_llm\_type: | 返回LLM的类型。 |
| dict: | 返回LLM的字典表示。 |
| save: | 保存LLM。 |

## BaseChatModel类、方法与属性速查

（一）BaseChatModel 类

BaseChatModel 是基础的聊天模型接口。

（二）类名: BaseChatModel

（三）属性和方法:

|  |  |
| --- | --- |
| verbose: | 是否打印出响应文本。 |
| callbacks: | 回调。 |
| callback\_manager: | 可选的基础回调管理器。 |
| tags: | 添加到运行追踪的标签。 |
| raise\_deprecation: | 验证器，如果使用callback\_manager则发出弃用警告。 |
| \_combine\_llm\_outputs: | 方法，组合LLM输出。 |
| generate: | 方法，顶级调用。 |
| agenerate: | 异步方法，顶级调用。 |
| generate\_prompt: | 方法，生成提示。 |
| agenerate\_prompt: | 异步方法，生成提示。 |
| \_generate: | 抽象方法，顶级调用。 |
| \_agenerate: | 抽象方法，顶级调用。 |
| call: | 方法，调用BaseChatModel。 |
| \_call\_async: | 异步方法，调用BaseChatModel。 |
| call\_as\_llm: | 方法，作为LLM调用。 |
| predict: | 方法，预测文本。 |
| predict\_messages: | 方法，预测消息。 |
| apredict: | 异步方法，预测文本。 |
| apredict\_messages: | 异步方法，预测消息。 |
| \_identifying\_params: | 属性，获取识别参数。 |
| \_llm\_type: | 抽象属性，返回聊天模型的类型。 |
| dict: | 方法，返回LLM的字典表示。 |

## BaseCallbackManager类、方法与属性速查

BaseCallbackManager 类、方法与属性速查

（一）BaseCallbackManager 类

在LangChain框架中，BaseCallbackManager 类为LangChain的回调提供了基础的管理接口。

（二）类名: BaseCallbackManager

（三）属性和方法:

|  |  |
| --- | --- |
| handlers: | 一个包含BaseCallbackHandler的列表，用于存储回调处理器。 |
| inheritable\_handlers: | 可继承的回调处理器列表。 |
| parent\_run\_id: | 父运行的UUID标识符。 |
| tags: | 与回调管理器关联的标签列表。 |
| inheritable\_tags: | 可继承的标签列表。 |
| is\_async: | 属性，用于判断回调管理器是否为异步。 |
| add\_handler: | 方法，用于向回调管理器添加处理器。 |
| remove\_handler: | 方法，用于从回调管理器中移除处理器。 |
| set\_handlers: | 方法，设置为回调管理器的唯一处理器的处理器列表。 |
| set\_handler: | 方法，设置为回调管理器的唯一处理器。 |
| add\_tags: | 方法，用于添加标签到回调管理器。 |
| remove\_tags: | 方法，用于从回调管理器中移除标签。 |

## Embeddings类、方法与属性速查

（一）Embeddings 类

在LangChain框架中，Embeddings 类是嵌入模型的接口。

（二）类名:Embeddings

（三）属性和方法:

|  |  |
| --- | --- |
| embed\_documents: | 用于嵌入搜索文档。输入的数据格式是列表字符串，输出的数据格式是浮点数列表的列表。 |
| embed\_query: | 用于嵌入查询字符串。输入的数据格式是列表字符串，输出的数据格式是浮点数列表。 |

## Agent 类、方法与属性速查

（一）Agent 类

在LangChain框架中，Agent 类负责调用语言模型并决定行动。它是由一个LLMChain驱动的，其中LLMChain的提示必须包括一个名为"agent\_scratchpad"的变量，代理可以在其中放置其中间工作。

（二）类名: Agent

（三）属性和方法:

|  |  |
| --- | --- |
| llm\_chain: | LLMChain实例，描述代理如何与语言模型交互。 |
| output\_parser: | AgentOutputParser的实例，用于解析语言模型的输出。 |
| allowed\_tools: | 可选的工具列表。 |
| dict: | 方法，返回代理的字典表示。 |
| get\_allowed\_tools: | 方法，返回允许的工具列表。 |
| return\_values: | 属性，返回输出值列表。 |
| \_fix\_text: | 方法，修复文本。 |
| \_stop: | 属性，定义停止令牌列表。 |
| \_construct\_scratchpad: | 方法，构建代理继续其思考过程的草稿板。 |
| plan: | 方法，根据输入决定要做什么。 |
| aplan: | 方法，根据输入决定要做什么。 |
| get\_full\_inputs: | 方法，从中间步骤创建LLMChain的完整输入。 |
| input\_keys: | 属性，返回输入键的列表。 |
| validate\_prompt: | 方法，验证提示是否匹配格式。 |
| observation\_prefix: | 属性，定义观察和LLM调用的前缀。 |
| llm\_prefix: | 属性，定义观察和LLM调用的前缀。 |
| create\_prompt: | 方法，为该类创建一个提示。 |
| \_validate\_tools: | 方法，验证工具并获取默认的输出解析器。 |
| \_get\_default\_output\_parser: | 方法，验证工具并获取默认的输出解析器。 |
| from\_llm\_and\_tools: | 方法，从LLM和工具构造代理。 |
| return\_stopped\_response: | 方法，当代理由于最大迭代次数而停止时返回响应。 |
| tool\_run\_logging\_kwargs: | 方法，返回工具运行日志的关键字参数。 |

## AgentExecutor 类、方法与属性速查

（一）AgentExecutor 类

AgentExecutor 类封装了一个使用工具的代理。此类负责驱动代理，使其在工具集合上运行，并根据代理的建议采取行动。

（二）类名: AgentExecutor

（三）属性和方法:

|  |  |
| --- | --- |
| agent: | 代理，可以是单行动或多行动代理。 |
| tools: | 代理可以使用的工具序列。 |
| return\_intermediate\_steps: | 布尔值，决定是否返回中间步骤。 |
| max\_iterations: | 最大迭代次数。 |
| max\_execution\_time: | 最大执行时间。 |
| early\_stopping\_method: | 早期停止方法（例如，当达到最大迭代次数或时间限制时）。 |
| handle\_parsing\_errors: | 如何处理解析错误。 |
| from\_agent\_and\_tools: | 创建代理执行者的类方法。 |
| validate\_tools: | 验证工具与代理兼容的方法。 |
| validate\_return\_direct\_tool: | 验证工具与代理兼容的方法。 |
| save: | 保存代理执行者的方法（此方法会引发错误，因为AgentExecutor不支持保存）。 |
| save\_agent: | 保存底层代理的方法。 |
| input\_keys: | 返回输入键的属性。 |
| output\_keys: | 返回输出键的属性。 |
| lookup\_tool: | 方法，按名称查找工具。 |
| \_should\_continue: | 方法，确定是否应继续迭代。 |
| \_return: | 方法，返回代理的最终输出。 |
| \_areturn: | 异步方法，返回代理的最终输出。 |
| \_take\_next\_step: | 方法，代理在思考-行动-观察循环中采取单一步骤。 |
| \_atake\_next\_step: | 异步方法，代理在思考-行动-观察循环中采取单一步骤。 |
| \_call: | 方法，运行文本并获取代理响应。 |
| \_acall: | 异步方法，运行文本并获取代理响应。 |
| \_get\_tool\_return: | 方法，检查工具是否是返回工具。 |

## Chain 类、方法与属性速查

（一）Chain 类

Chain 是所有链应实现的基础接口。

（二）类名: Chain

（三）属性和方法:

|  |  |
| --- | --- |
| memory: | 可选的基础内存。 |
| callbacks: | 回调。 |
| callback\_manager: | 可选的基础回调管理器。 |
| verbose: | 用于决定是否打印响应文本的布尔值。 |
| tags: | 可选的标签列表。 |
| \_chain\_type: | 属性，需要子类实现。用于说明链的类型。 |
| raise\_deprecation: | 验证器，如果使用callback\_manager则发出弃用警告。 |
| set\_verbose: | 如果verbose为None，则设置它。 |
| input\_keys: | 抽象属性，此链期望的输入键。 |
| output\_keys: | 抽象属性，此链期望的输出键。 |
| \_validate\_inputs: | 方法，检查所有输入是否存在。 |
| \_validate\_outputs: | 方法，检查所有输出是否存在。 |
| \_call: | 抽象方法，运行此链的逻辑并返回输出。 |
| \_acall: | 异步方法，运行此链的逻辑并返回输出。 |
| call: | 方法，运行此链的逻辑，并根据需要添加到输出。 |
| acall: | 异步方法，运行此链的逻辑，并根据需要添加到输出。 |
| prep\_outputs: | 方法，验证和准备输出。 |
| prep\_inputs: | 方法，验证和准备输入。 |
| apply: | 方法，对列表中的所有输入调用链。 |
| \_run\_output\_key: | 属性，只有一个输出键时才支持运行。 |
| run: | 方法，以文本输入、文本输出或多个变量、文本输出的形式运行链。 |
| arun: | 异步方法，以文本输入、文本输出或多个变量、文本输出的形式运行链。 |
| dict: | 方法，返回链的字典表示。 |
| save: | 方法，保存链。 |

BaseLoader 类、方法与属性速查

（一）BaseLoader 类

BaseLoader 是一个用于加载文档的接口。

（二）类名: BaseLoader

（三）属性和方法:

|  |  |
| --- | --- |
| load: | 抽象方法，加载数据到文档对象中。子类应将此方法实现为返回 list(self.lazy\_load())，此方法返回一个在内存中实体化的列表。 |
| load\_and\_split: | 方法，加载文档并分割成块。如果没有提供文本分割器，它将使用 RecursiveCharacterTextSplitter。 |
| lazy\_load: | 方法:为文档内容提供懒加载。请注意，这个方法会在所有现有子类中实现之后升级为一个抽象方法 |

BaseChatMemory 类、方法与属性速查

（一）BaseChatMemory 类

BaseChatMemory 是一个继承于 BaseMemory 的基础聊天内存抽象基类。

（二）类名: BaseChatMemory

（三）属性和方法:

|  |  |
| --- | --- |
| chat\_memory: | 用于存储聊天消息历史的属性，默认为ChatMessageHistory  实例。 |
| output\_key: | 输出键，用于确定哪个输出应该被保存到聊天历史中，默认为None。 |
| input\_key: | 输入键，用于确定哪个输入应该被保存到聊天历史中，默认为None。 |
| return\_messages: | 布尔值，决定是否返回消息，默认为False。 |
| \_get\_input\_output: | 私有方法，从输入和输出中获取对应的输入和输出字符串。 |
| save\_context: | 方法，将此次对话的上下文保存到缓冲区。 |
| clear: | 方法，清除内存内容。 |

StructuredOutputParser 类、方法与属性速查

（一）StructuredOutputParser 类

StructuredOutputParser 是继承于 BaseOutputParser 的结构化输出解析器类。

（二）类名: StructuredOutputParser

（三）属性和方法:

|  |  |
| --- | --- |
| response\_schemas: | 用于存储响应模式的列表。 |
| from\_response\_schemas: | 类方法，从响应模式列表创建StructuredOutputParser实例。 |
| get\_format\_instructions: | 方法，生成格式化的说明字符串。 |
| parse: | 方法，从给定文本中解析结构化输出。 |
| \_type: | 属性方法，返回字符串"structured"。 |

ArxivRetriever类、方法与属性速查

（一）ArxivRetriever 类

ArxivRetriever 是一个结合了 BaseRetriever 和 ArxivAPIWrapper 的检索器类。本质上，它是 ArxivAPIWrapper 的一个包装器。

（二）类名: ArxivRetriever

（三）概述:

该类有效地包装了 ArxivAPIWrapper。它将 load() 方法包装为 get\_relevant\_documents() 方法。此外，该类使用所有 ArxivAPIWrapper 的参数，不做任何更改。

（四）方法:

|  |  |
| --- | --- |
| get\_relevant\_documents: | 方法，获取与给定查询相关的文档。 |
| aget\_relevant\_documents: | 异步方法，但目前尚未实现。 |

BaseTool类、方法与属性速查

（一）BaseTool 类

BaseTool 提供了 LangChain 工具必须实现的接口。

（二）类名: BaseTool

（三）概述:

该类为所有 LangChain 工具提供了一个基本的接口。它定义了工具如何运行、如何解析输入和如何处理错误。

（四）属性和方法:

|  |  |
| --- | --- |
| \_parse\_input: | 将工具输入转换为pydantic模型。 |
| \_run: | 使用工具。 |
| \_arun: | 异步使用工具。 |
| run: | 运行工具。 |
| arun: | 异步运行工具。 |
| \_\_call\_\_: | 使工具可调用。 |
| name: | 工具的独特名称。 |
| description: | 描述如何/何时/为什么使用工具。 |
| args\_schema: | Pydantic模型类，用于验证和解析工具的输入参数。 |
| return\_direct: | 是否直接返回工具的输出。 |
| verbose: | 是否记录工具的进度。 |
| callbacks: | 在工具执行期间要调用的回调。 |
| callback\_manager: | 已弃用。请使用callbacks代替。 |
| handle\_tool\_error: | 处理抛出的ToolException的内容。 |

GoogleSerperAPIWrapper类、方法与属性速查

（一）GoogleSerperAPIWrapper 类

GoogleSerperAPIWrapper 是一个围绕 Serper.dev Google 搜索 API 的包装器。

（二）类名: GoogleSerperAPIWrapper

（三）概述:

该类为 Serper.dev Google 搜索 API 提供了一个接口。用户可以使用环境变量 SERPER\_API\_KEY 或通过构造函数的 serper\_api\_key 参数提供 API 密钥。

（四）属性和方法:

|  |  |
| --- | --- |
| results: | 通过GoogleSearch运行查询。 |
| run: | 通过GoogleSearch运行查询并解析结果。 |
| aresults: | 异步地通过GoogleSearch运行查询。 |
| arun: | 异步地通过GoogleSearch运行查询并解析结果。 |
| \_parse\_snippets: | 从搜索结果中解析摘录。 |
| \_parse\_results: | 解析搜索结果。 |
| \_google\_serper\_api\_results: | 获取GoogleSerperAPI的结果。 |
| \_async\_google\_serper\_search\_results: | 异步获取GoogleSerperAPI的结果。 |
| k: | 返回的搜索结果的最大数量。 |
| gl: | 地理位置代码。 |
| hl: | 语言代码。 |
| type: | 搜索类型，可选值为"news","search","places"和"images"。 |
| result\_key\_for\_type: | 用于查找搜索结果的键。 |
| tbs: | 时间范围限制。 |
| serper\_api\_key: | API密钥。 |
| aiosession: | aiohttp的ClientSession对象。 |

VectorStore 类、属性与方法速查

（一）VectorStore 类

在LangChain框架中，VectorStore 类是向量存储的接口。

（二）类名: VectorStore

（三）属性和方法:

|  |  |
| --- | --- |
| add\_texts: | 通过嵌入运行更多的文本并添加到向量存储。它接受一个可迭代的文本，一个可选的元数据列表和特定于向量存储的参数。返回添加到向量存储的文本的ID列表。 |
| aadd\_texts: | 异步版本的 add\_texts，具有相同的功能和参数。 |
| add\_documents: | 通过嵌入运行更多的文档并添加到向量存储。输入是一个文档列表，返回的是已添加文本的ID列表。 |
| search: | 使用指定的搜索类型返回与查询最相似的文档。 |
| asearch: | 异步版本的 search 方法。 |
| similarity\_search\_with\_relevance\_scores: | 返回范围为 [0, 1] 的文档和相关性分数，其中0表示不相似，1表示最相似。 |
| similarity\_search | 返回与查询最相似的文档。 |

|  |  |
| --- | --- |
| asimilarity\_search\_with\_relevance\_scores: | 异步版本的similarity\_search\_with\_relevance\_scores方法。 |
| asimilarity\_search: | 异步版本的similarity\_search方法。 |
| similarity\_search\_by\_vector: | 根据给定的嵌入向量返回与之最相似的文档。 |
| asimilarity\_search\_by\_vector: | 异步版本similarity\_search\_by\_vector方法。 |
| max\_marginal\_relevance\_search: | 使用最大边际相关性返回文档。 |
| max\_marginal\_relevance\_search\_by\_vector: | 根据给定的嵌入向量，使用最大边际相关性返回文档。 |
| amax\_marginal\_relevance\_search\_by\_vector: | 异步版本的max\_marginal\_relevance\_search\_by\_vector方法。 |
| from\_documents: | 返回从文档和嵌入初始化的VectorStore。 |
| afrom\_documents: | 异步版本的from\_documents方法。 |
| from\_texts: | 返回从文本和嵌入初始化VectorStore。 |
| afrom\_texts: | 异步版本的from\_texts方法。 |
| as\_retriever: | 返回VectorStoreRetriever。 |

附录B

# OpenAI 平台和模型介绍

**OpenAI**简介

对于初学者和专业的开发者来说，理解OpenAI API的强大功能以及如何利用它构建LLM应用是至关重要的。OpenAI API为我们提供了一种直观的方法来处理涉及自然语言、代码、图像的任务，而无需深入了解底层机制。

应用范围广泛。OpenAI API不仅可以处理自然语言任务，还可以生成和编辑图像，将语音转换为文本。这意味着，从语义搜索到内容生成，再到分类任务，都可以通过这一API来实现。

模型的多样性。OpenAI提供了多种模型，每种模型都有其特定的功能和价格。这为开发者提供了选择的灵活性，以确保他们为特定的LLM应用找到最合适的模型。

微调与定制。除了预训练的模型，OpenAI还为开发者提供了微调自定义模型的能力，这意味着开发者可以根据具体的需求和数据来优化模型。

结合OpenAI API，LLM应用开发者可以更轻松地处理各种任务，从简单的文本生成到复杂的语义搜索。对于希望在这个领域取得突破的开发者来说，OpenAI API无疑是一个宝贵的资源。

OpenAI**的主要应用场景**

对于希望利用现代技术为其LLM应用增添动力的开发者来说，了解OpenAI的主要应用场景是非常有益的。以下列出了OpenAI在LLM应用开发中的几个主要用途：

**1. 内容生成**  
开发者可以使用OpenAI生成高质量的文本内容，从简单的句子到完整的文章。这对于那些希望自动化内容生产或生成特定格式文本的LLM应用尤其有用。

**2. 摘要**  
OpenAI能够从大量文本中提取关键信息并生成简洁的摘要。这对于需要快速理解文档主旨的应用，如新闻摘要或研究论文摘要生成等场景，具有巨大的价值。

**3. 分类、归类和情感分析**  
OpenAI可以帮助开发者对文本进行分类或归类，并对文本中的情感进行分析。这在社交媒体分析、评论系统或任何需要对文本进行情感判定的LLM应用中都是非常有用的。

**4. 数据提取**  
OpenAI可以从非结构化数据中提取关键信息，为开发者提供有价值的数据点。这可以应用于票据扫描、合同审查或任何需要从文本中提取特定信息的LLM应用。

**5. 翻译**  
OpenAI不仅可以理解文本，还可以将其翻译成其他语言。这为开发多语言LLM应用或需要快速翻译功能的项目提供了强大的支持。

窗体顶端

OpenAI核心概念解析

在深入研究如何使用OpenAI为LLM应用带来价值之前，了解其核心概念是至关重要的。以下为OpenAI中的一些核心概念，并解释了它们在LLM应用开发中的重要性。

**1. GPT模型**  
OpenAI的GPT（generative pre-trained transformer）模型经过训练，可以理解自然语言和代码。GPT模型根据输入提供文本输出，这些输入也被称为"prompts"。通过设计提示，开发者可以“编程”GPT模型，通常是通过提供任务的说明或示例。GPT模型适用于各种任务，包括内容或代码生成、摘要、对话、创意写作等。

**2. 嵌入向量（Embeddings）**  
嵌入是数据（例如文本）的向量表示，旨在保留其内容和/或含义的某些方面。相似的数据块在某种程度上会有更接近的嵌入，而不相关的数据则相反。OpenAI提供的文本嵌入模型接受文本字符串作为输入，并输出一个嵌入向量。嵌入对于搜索、聚类、推荐、异常检测、分类等都很有用。

**3. 标记（Tokens）**  
GPT和嵌入模型使用称为标记的文本块来处理文本。标记代表常见的字符序列。例如，字符串"tokenization"被分解为"token"和"ization"，而像"the"这样的短且常见的词被表示为一个标记。在句子中，每个词的第一个标记通常以一个空格字符开始。作为一个粗略的经验法则，对于英文文本，1个标记大约等于4个字符或0.75个词。

OpenAI工作流程探究

当开发者决定利用OpenAI为LLM应用增添功能时，首先需要了解其工作流程。本章节将详细探讨如何从头开始构建一个OpenAI应用。

**1. 应用构建准备**。首先，开发者需要为应用创建一个基础。例如，如果使用Python Flask框架，则需要准备相应的环境。当环境准备就绪后，可以下载官方提供的代码样本，如：

git clone https://github.com/openai/openai-quickstart-node.git

或者直接从官方链接下载压缩包。

**2. API密钥配置**。为了确保应用能够与OpenAI API进行通信，开发者需要一个API密钥。这可以通过在OpenAI官方网站上注册并获取。获取到的密钥需要被添加到应用中，确保数据交互的安全性。

**3. 应用运行**

一旦配置完毕，可以通过以下命令安装依赖并运行应用：

npm install

npm run dev

随后，开发者可以在浏览器中访问应用并进行测试。

**4. 代码深入解读**。真正理解应用的核心是理解其背后的代码。例如，在generate.js文件中，有一个用于生成提示的函数，这是与GPT模型交互的关键部分。此函数根据用户输入的动物类型动态生成提示。

function generatePrompt(animal) {

const capitalizedAnimal = animal[0].toUpperCase() + animal.slice(1).toLowerCase();

return `Suggest three names for an animal that is a superhero.

Animal: Cat

Names: Captain Sharpclaw, Agent Fluffball, The Incredible Feline

Animal: Dog

Names: Ruff the Protector, Wonder Canine, Sir Barks-a-Lot

Animal: ${capitalizedAnimal}

Names:`;

}

此外，代码中还有一个部分专门用于与OpenAI API进行交互，发送请求并获取响应。这部分使用了completions端点，并设置了特定的参数，如温度为0.6。

const completion = await openai.createCompletion({

model: "text-davinci-003",

prompt: generatePrompt(req.body.animal),

temperature: 0.6,

});

通过深入了解这些核心代码部分，开发者可以更好地理解如何为LLM应用定制OpenAI功能。

OpenAI 的模型型号解析

当开发者决定在LLM应用中集成OpenAI时，了解其提供的不同模型是至关重要的。每种模型都有其独特的功能和应用场景。以下是OpenAI的各种模型的详细介绍：

**1. GPT-4：**这是OpenAI的最新模型系列，它在理解和生成自然语言或代码方面相较于GPT-3.5有所改进。对于希望获得高质量文本或代码输出的LLM应用开发者，GPT-4是一个理想的选择。

**2. GPT-3.5：**作为GPT-3的改进版本，这一系列模型继续在自然语言处理和代码生成方面展现出卓越的性能。

**3. DALL·E：**这是一个独特的模型，可以根据自然语言提示生成和编辑图像。对于需要图像生成功能的LLM应用，DALL·E无疑是一个强大的工具。

**4. Whisper：**Whisper模型专门用于将音频转化为文本。对于需要语音识别功能的LLM应用，Whisper是一个不可或缺的资源。

**5. Embeddings：**这是一组模型，可以将文本转化为数值形式，为进一步的文本分析和处理提供了有力的支持。

**6. Moderation：**这是一个经过微调的模型，能够检测文本是否可能包含敏感或不安全的内容，从而保证LLM应用的内容安全。

**OpenAI 的先锋模型探析**

当开发者决定在LLM应用中采用OpenAI技术时，了解OpenAI的模型更新策略和模型版本是至关重要的。这能确保开发者始终能够获得最先进、最高效的自然语言处理技术。

**1. 持续更新的模型：**  
随着gpt-3.5-turbo的发布，OpenAI开始实施一种持续的模型更新策略。例如，模型名称为gpt-3.5-turbo、gpt-4和gpt-4-32k的模型会指向最新的版本。为了确认具体使用的模型版本，开发者可以查看发送ChatCompletion请求后的响应对象。这将明确显示所用的模型版本，如gpt-3.5-turbo-0613。

**2. 静态模型版本：**  
尽管OpenAI不断推出更新的模型，但它还提供静态模型版本供开发者使用。即使推出了新版本，这些静态版本至少还可以继续使用三个月。

**3. 为模型改进提供贡献：**  
随着模型更新的加快，OpenAI鼓励社区为不同的用例贡献评估，以帮助改进模型。对此感兴趣的开发者可以查看OpenAI Evals存储库，参与模型的持续完善。

**4. 临时快照模型：**  
以下列出的模型是暂时的版本快照。一旦有了更新版本，OpenAI将宣布它们的停用日期。如果开发者希望始终使用最新的模型版本，只需使用标准的模型名称，如gpt-4或gpt-3.5-turbo。

**gpt-3.5-turbo-0301**：预计停用日期为2024年6月13日，替代模型为gpt-3.5-turbo-0613。

**gpt-4-0314**：预计停用日期为2024年6月13日，替代模型为gpt-4-0613。

**gpt-4-32k-0314**：预计停用日期为2024年6月13日，替代模型为gpt-4-32k-0613。

更多关于模型停用的信息，开发者可以参考OpenAI的官方停用页面。

在构建LLM应用时，确保跟随OpenAI的模型更新步伐是至关重要的，这将确保应用在自然语言处理领域保持前沿。

附录C

# Claude 2 模型介绍

对于那些寻求构建高效LLM应用的开发者，了解最新的模型和技术是至关重要的。Anthropic发布的Claude 2模型，一款被广大开发者高度期待的模型。

Claude 2是什么？

Claude 2是Anthropic推出的新模型。这款模型在多方面都实现了显著的进步，包括编码、数学和推理能力。事实上，Claude 2在Bar考试的多项选择部分得分为76.5%，这比Claude 1.3的73.0%有所提高。在GRE的阅读和写作部分，Claude 2的得分超过了90%的应试者。

Claude 2的三大特点：

1. 处理大量数据的能力：Claude 2模型允许用户在每次提示中输入多达100K的标记，这意味着它可以处理从技术文档到整本书的大量数据。此外，Claude 2现在也可以一次性编写从备忘录到信件到几千个标记的长文档。

2. 代码能力的增强：Claude 2的模型在编码方面进行了明显的优化，Sourcegraph是一个代码AI平台，他们的编码助理Cody利用Claude 2改进的推理能力为用户查询提供更准确的答案，同时也可以提供高达100K的上下文窗口。此外，Claude 2接受了更多的最新数据培训，这意味着它拥有了新框架和库的知识供Cody参考。

3. 安全性的增强：Anthropic对Claude 2进行了一系列的安全性优化。首先，Claude 2在内部红队评估中的表现比Claude 1.3好出了2倍，这意味着它在回应可能有害的提示时更能产生无害的回应。此外，为了提高模型的输出安全性，Anthropic使用了多种安全技术，并进行了广泛的红队测试。

对于追求最前沿技术的LLM应用开发者，Claude 2无疑为他们提供了一个值得关注的选择。

四大主要使用场景

随着技术的持续进步，为开发者带来了更多创新的机会，其中最为引人注目的是LLM应用的快速发展。其中，Claude 2模型为LLM应用带来了新的灵感和实践机会。Claude 2在LLM应用开发中的四大主要使用场景。

1. 处理海量文本

无论开发者面临的是文档、电子邮件、常见问题解答、聊天记录还是其他内容，Claude 2都能提供卓越的支持。此模型可以编辑、重写、总结、分类、提取结构化数据，并根据内容进行问答等操作。这为LLM应用开发者提供了一个强大的工具，助力他们更高效地处理和分析文本数据。

2. 自然对话交流。Claude 2可以在对话中扮演各种角色。只需为其提供角色详情和常见问题的解答，它就能与用户进行自然、相关的双向对话。这为开发者在LLM应用中实现流畅的用户互动提供了可能。

3. 获得答案。Claude 2拥有广泛的通用知识，这些知识来源于其庞大的训练语料库，包括技术、科学和文化知识。除了常见的自然语言，Claude 2还能理解和生成多种编程语言。这为开发者提供了在LLM应用中集成知识库或编程助手的机会。

4. 自动化工作流。Claude 2能够处理各种基本指令和逻辑场景，包括按需格式化输出、执行if-then语句以及在单一提示中进行一系列逻辑评估。这使得开发者能够在LLM应用中实现复杂的自动化任务和工作流。

用户关心的10个常见问题

在LLM应用的开发过程中，开发者可能对Anthropic的Claude 2模型有很多疑问。为了更好地帮助初学者和开发者了解Claude 2，以下列出了关于该模型的10个常见问题及其解答。

1. Claude有哪些版本可供选择？

目前提供两个版本的Claude：

Claude：是最强大的模型，擅长从复杂对话和创意内容生成到详细指导的各种任务。

Claude Instant：较快、较便宜但仍然非常能干的模型，可以处理包括休闲对话、文本分析、总结和文档问题回答等任务。

2. Claude的价格是多少？

https://www-files.anthropic.com/production/images/model\_pricing\_july2023.pdf。

3. Claude支持哪些语言？

Claude主要以英语为训练基础，但在其他常见语言中也表现出色。此外，Claude还对常见的编程语言有深入的了解。

4. Claude可以访问互联网吗？

不可以。Claude被设计为独立的，不会搜索互联网来响应。但可以为Claude提供互联网上的文本，并要求其对该内容执行任务。

5. 什么是宪法训练？

宪法训练是一个训练模型遵循所需行为“宪法”的过程。Anthropic的核心模型经过宪法训练，目的是变得有助于、诚实和无害。

6. "HHH"是什么意思？

"HHH"代表Helpful（有帮助）、Honest（诚实）和Harmless（无害）。这是构建与人们利益一致的AI系统（如Claude）的三个组件。

7. 如何进一步自定义Claude的行为？

可以通过提示广泛地修改Claude的行为。提示可以用来解释所需的角色、任务、背景知识以及所需响应的几个示例。

8. Claude模型可以进行微调吗？

在大多数情况下，我们相信精心设计的提示可以在没有微调的费用或延迟的情况下为您提供所需的结果。但一些大型企业用户可能会从微调模型中受益。

9. Claude的上下文窗口有多长？

输入和输出的综合上下文窗口约为100,000个标记，这大约相当于70,000个单词，具体取决于内容类型。

10. Claude可以进行嵌入吗？

目前还不行。我们发现开源的SBERT嵌入对于大多数用途已经足够好。

附录D

# Cohere 模型介绍

随着LLM应用的广泛应用，开发者对于高效、高性能的语言模型的需求日益增强。在这一背景下，Cohere应运而生，为开发者提供了一个先进的语言处理API。

Cohere的核心能力

Cohere不仅训练了大型的语言模型，并通过一个简洁的API为开发者提供服务，它还允许用户根据自己的需求训练定制的大型模型。这意味着开发者无需为收集大量的文本数据、选择合适的神经网络架构、分布式训练或模型部署而感到困扰，Cohere为开发者处理了所有这些复杂性。

Cohere的模型类型

Cohere为开发者提供了两大类的模型：

生成模型：通过generate端点，开发者可以访问这类模型。这些模型的代表包括像GPT2和GPT3这样的模型。

表示模型：通过embed端点，开发者不仅可以访问这类模型，还可以获取输入文本的嵌入向量。BERT是这类模型的代表。

对于希望在LLM应用开发中实现前沿语言处理功能的初学者和开发者，Cohere提供了一个高效且功能强大的解决方案。无论是需要生成内容，还是需要理解和表示语言，Cohere都为开发者提供了一站式的解决方案。

三大LLM应用案例

在现代LLM应用开发中，语言模型的功能越来越强大，开发者可利用这些功能来解决实际问题。Cohere作为一款领先的大型语言模型，为开发者提供了多种实用功能。以下将深入探讨Cohere的三大应用案例，助力开发者更好地理解其在LLM应用中的潜在价值。

1. 文本摘要与改写。随着文本生成技术的进步，大型语言模型如Cohere已经能够生成近乎人类水平的文本。其中，文本摘要与改写成为了开发者们的热门应用点。开发者可以通过Cohere为输入文本生成有意义的摘要或改写，仅需在提示中提供任务描述。此外，Cohere为文本摘要提供了Co.summarize端点，为开发者进一步简化了任务。

2. 文本分类。文本分类是语言处理中最常见的用例之一。利用Cohere的语言模型，开发者可以构建高效的分类器来自动化语言任务，从而节省大量时间和精力。Cohere不仅提供了简单的Classify端点进行分类，还允许开发者在embed端点之上构建更高级的分类器。

3. 语义相似性判断。在客服领域，经常会有大量重复的问题需要回答。Cohere的语言模型能够判断文本的相似性，从而确定一个新问题是否与FAQ部分已经回答的问题相似。通过计算两个嵌入的余弦相似性，开发者可以得到一个相似性得分，然后根据这个得分采取相应的行动，例如显示与最相似问题的答案。

附录E

# PaLM 2模型介绍

PaLM 2代表了Google在机器学习和负责任的AI领域不断创新的成果，是继PaLM后的下一代大型语言模型。作为LLM应用的开发者，理解PaLM 2的基础构造和核心优势对于充分利用其功能至关重要。

PaLM 2模型的特点

高级推理任务：PaLM 2在编码、数学、分类和问题回答、翻译和多语言能力以及自然语言生成等高级推理任务上都表现出色。

超越先前模型：相较于之前的最先进的LLM应用，包括PaLM，PaLM 2在各种任务上都有更好的表现。

构建方法：PaLM 2之所以能够实现这些任务，归功于其构建方式，结合了计算最优缩放、改进的数据集混合和模型架构的改进。

负责任的AI：PaLM 2基于Google负责任地构建和部署AI的方法，经过了严格的潜在危害和偏见、能力和下游用途的评估。

以PaLM 2为基础的产品

除了作为一个独立的大型语言模型，PaLM 2还为其他最先进的模型提供支持，如Med-PaLM 2和Sec-PaLM。此外，它正在为Google的一些生成性AI功能和工具提供动力，如Bard和PaLM API，这为LLM应用开发者提供了更广泛的实际应用场景。

三大核心功能

**1. 推理**。PaLM 2在复杂任务的分解以及对人类语言细微差异的理解上，相比之前的LLM，如PaLM，表现得更为出色。它能够非常精准地解读谜语和习语，这需要对词语的模糊和比喻意义有深入的理解，而不仅仅是字面意义。

**2. 多语言翻译。**与PaLM相比，PaLM 2在更大规模的多语言文本上进行了预训练。这使得它在多语言任务上具有显著的优势。通过大量的多语言文本预训练，PaLM 2为开发者在LLM应用中实现高效的多语言处理提供了坚实的基础。

**3. 编码**。PaLM 2的另一个亮点是它在大量的网页、源代码和其他数据集上进行的预训练。这意味着它不仅擅长流行的编程语言，如Python和JavaScript，而且还能够生成Prolog、Fortran和Verilog等专用编程语言的代码。结合其语言处理能力，可以帮助团队跨语言进行合作。

附录F

# Pinecone**向量数据库**介绍

Pinecone简介

对于初步接触LLM应用开发的开发者来说，选择一个高性能的向量搜索工具是关键的初步决策。Pinecone为此提供了一个完美的解决方案。

Pinecone是一个云原生的向量数据库，专门为高性能向量搜索应用程序设计。借助其托管服务和简化的API接口，开发者可以无缝地集成其功能，而无需过多关注底层基础架构的细节。

接下来，我们将详细探讨Pinecone的主要特性，这些特性使其在LLM应用开发领域中脱颖而出：

**高速查询性能**：Pinecone确保即使在数十亿条目中也能保持超低的查询延迟，满足实时应用的需求。

**实时索引更新**：随着数据的添加、修改或删除，索引可以实时更新，确保数据的即时性和准确性。

**过滤功能**：Pinecone允许开发者结合元数据过滤器进行向量搜索，这有助于获得更加相关和快速的查询结果。

**无缝托管服务**：Pinecone的完全托管特性使得开发者可以更加专注于LLM应用的开发和优化，而不是数据库的维护和管理。

Pinecone**的主要应用场景**

对于那些正在研究LLM应用开发的开发者，了解如何在实际应用中利用向量数据库如Pinecone是至关重要的。Pinecone由于其高效性和灵活性，已被广泛应用于多种场景。以下我们将深入探讨Pinecone的主要使用场景：

**语义文本搜索**：开发者可以利用NLP转换器和句子嵌入模型将文本数据转化为向量嵌入。随后，这些向量可以被Pinecone索引和搜索，从而实现高效的语义文本搜索功能。

**生成问答系统**：当接收到用户的查询时，可以从Pinecone检索相关的上下文数据。这些数据随后可以传递给如OpenAI这样的生成模型，产生与真实数据相一致的答案。

**混合搜索**：开发者可以结合语义和关键字搜索，让Pinecone在一个查询中同时执行，从而得到更加相关的搜索结果。

**图像相似度搜索**：首先，将图像数据转换为向量嵌入并使用Pinecone进行索引。当用户提交查询图像时，将其转换为向量并检索在Pinecone中的相似图像，为用户提供相似内容的图像。

**产品推荐系统**：在电子商务领域，基于代表用户的向量，Pinecone可以有效地生成产品推荐，从而为用户提供更个性化的购物体验。

Pinecone核心概念解析

当开发者进入LLM应用开发的领域，理解Pinecone的关键概念将为他们提供明确的方向和坚实的基础。在本部分，我们将深入探讨Pinecone的几个核心概念，从而为开发者铺设通往成功应用的路：

**向量搜索**： 传统搜索方法主要围绕关键字进行，但在向量数据库中，搜索的焦点转向了由ML生成的数据表示 - 向量嵌入。这种搜索方法的目标是找到与查询最相似的项目。

**向量嵌入**： 向量嵌入是表示对象的数字集合，它的特点是能够捕捉对象集合中的语义相似性。这些嵌入是由经过训练的模型生成的。在Pinecone中，开发者可以遇到两种主要的向量嵌入：密集嵌入和稀疏嵌入。为了充分利用Pinecone，开发者需要熟悉如何使用这些向量嵌入。

**向量数据库**： 作为一种特殊的数据库，向量数据库专注于索引和存储向量嵌入，以实现高效的管理和快速的检索。但是，与单纯的向量索引相比，向量数据库如Pinecone提供了更多高级功能。这些功能包括索引管理、数据管理、元数据存储、过滤和水平扩展等。

Pinecone工作流程探究

对于初学者和那些希望使用LLM构建应用的开发者来说，了解Pinecone的工作流程是开发过程中的关键步骤。

**（一）索引的设置**：

**创建索引**：首先，为数据创建一个索引，这是存储和检索向量的关键结构。

**连接索引**：一旦索引创建完毕，开发者需要确保能够与之建立连接。

**数据插入**：接下来，开发者将数据和相应的向量插入创建的索引中。

**（二）索引的使用**：

**查询数据**：在索引中查询特定数据或向量。

**数据过滤**：基于特定条件，开发者可以过滤检索到的结果，确保结果的相关性。

**获取数据**：根据需要，可以检索索引中的特定数据或向量。

**数据更新**：为了保持数据的实时性和准确性，开发者可以插入更多的数据或更新现有的向量。

**（三）索引与数据管理**：

**管理索引**：包括对索引的优化、备份和恢复等操作。

**数据管理**：涉及到数据的删除、修改和备份等任务。

附录G

# Milvus向量数据库介绍

Milvus简介

当我们谈论大规模嵌入向量的存储、索引和管理，Milvus向量数据库凭其独特的特性和优势成为了这一领域的明星。自2019年创建以来，Milvus的核心愿景是处理由深度神经网络和其他机器学习（ML）模型产生的大量嵌入向量。

与传统的关系型数据库不同，它们主要处理符合预定义模式的结构化数据，Milvus从其核心被设计为处理从非结构化数据转化而来的嵌入向量。这种设计意味着Milvus能够无缝地处理万亿级的向量索引。

为什么这种能力如此重要？随着互联网、物联网和社交媒体的普及，非结构化数据，如电子邮件、学术论文、传感器数据和社交媒体图片，已成为主流。为了使这些数据对于机器有意义，嵌入技术被用于将它们转换为向量形式。这正是Milvus所擅长的领域。通过存储和索引这些向量，Milvus可以计算两个向量间的相似距离，从而判断原始数据的相似性。

对于希望在LLM应用中使用非结构化数据的开发者，了解并利用Milvus的这些功能将帮助他们更有效地进行数据分析和提取有价值的见解。

窗体顶端

Milvus**的主要应用场景**

在构建和优化LLM应用时，开发者经常面临处理和搜索大量数据的挑战。这正是Milvus展现其强大功能的地方。以下是Milvus在各种应用中的主要应用场景：

图像相似性搜索：Milvus使得从大型数据库中即时返回最相似的图像成为可能，实现了高效的图像搜索功能。

视频相似性搜索：通过将视频的关键帧转换为向量，并利用Milvus进行处理，可以在接近实时的速度下搜索和推荐亿级别的视频。

音频相似性搜索：无论是语音、音乐、音效还是其他类似的声音，Milvus都能在短时间内快速查询大量音频数据。

分子相似性搜索：对于生物技术和化学领域，Milvus能够对特定的分子进行快速的相似性搜索，子结构搜索或超结构搜索。

推荐系统：基于用户的行为和需求，Milvus可以为LLM应用提供信息或产品的精准推荐。

问答系统：为了实现交互式的数字问答机器人，Milvus能够自动、准确地回答用户的问题。

DNA序列分类：在基因研究中，通过与Milvus比较相似的DNA序列，可以在毫秒级别内准确地对一个基因进行分类。

文本搜索引擎：对于需要处理大量文本数据的应用，Milvus能够通过与文本数据库中的关键字进行比较，帮助用户快速找到他们需要的信息。

Milvus核心概念解析

随着数据的爆炸性增长，开发者在构建LLM应用时面临着处理和理解大量非结构化数据的挑战。Milvus的核心概念为开发者提供了一个框架，帮助他们更好地处理这些数据。下面，我们将深入解析这些概念：

非结构化数据：这是指不遵循预定义模型或组织方式的数据。它包括图像、视频、音频和自然语言等信息。事实上，非结构化数据占据了全球数据的约80%。为了使这些数据有意义，必须将它们转换为可以被机器理解的格式，这就是向量的角色。

嵌入向量：嵌入向量是非结构化数据的特征抽象，例如电子邮件、物联网传感器数据、社交媒体照片和蛋白质结构等。在数学上，嵌入向量可以是浮点数或二进制数的数组。通过利用现代嵌入技术，开发者可以将非结构化数据转换为嵌入向量，从而为其LLM应用提供一个坚实的基础。

向量相似度搜索：这是一个搜索过程，将一个向量与数据库中的向量进行比较，目的是找到与查询向量最为相似的向量。为了加速这个搜索过程，通常使用近似最近邻搜索算法。当两个嵌入向量相似时，它们代表的原始数据源也是相似的。

Milvus支持的索引和度量

了解Milvus所支持的索引和度量，为构建高效的LLM应用打下坚实的基础。

索引是数据的组织方式，它定义了如何存储和检索数据。在Milvus中，大部分索引类型使用近似最近邻搜索（ANNS）技术。以下是一些重要的索引类型：

**FLAT**：适合于小规模数据集，提供完全准确和精确的搜索结果。

**IVF\_FLAT**：量化索引，适合于在查询速度和精度之间寻求平衡的场景。

**IVF\_SQ8**：在资源有限的场景中，此量化索引可以显著降低资源消耗。

**IVF\_PQ**：为了获得更高的查询速度，此量化索引可能牺牲一些精度。

**HNSW**：基于图形的索引，适合于高搜索效率需求的场景。

**ANNOY**：基于树形结构的索引，适合于寻求高召回率的场景。

在LLM应用中，度量方法的选择对于向量的分类和聚类性能至关重要。在Milvus中，相似度度量用于确定向量之间的相似性：

对于浮点嵌入，以下两种度量方法常用：

**欧氏距离（L2）**：在计算机视觉领域中常用。

**内积（IP）**：在自然语言处理领域中常用。

而对于二进制嵌入，以下是一些广泛应用的度量方法：

**哈明距离**：在自然语言处理中常用。

**杰卡德距离 & 塔尼莫托距离**：这两种度量方法在分子相似性搜索中都有广泛应用。

**超结构距离 & 亚结构距离**：这两种度量方法用于搜索分子的特定结构相似性。

为了在LLM应用中实现高效的数据检索和管理，开发者需要深入了解并正确选择索引和度量。