2.

## 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

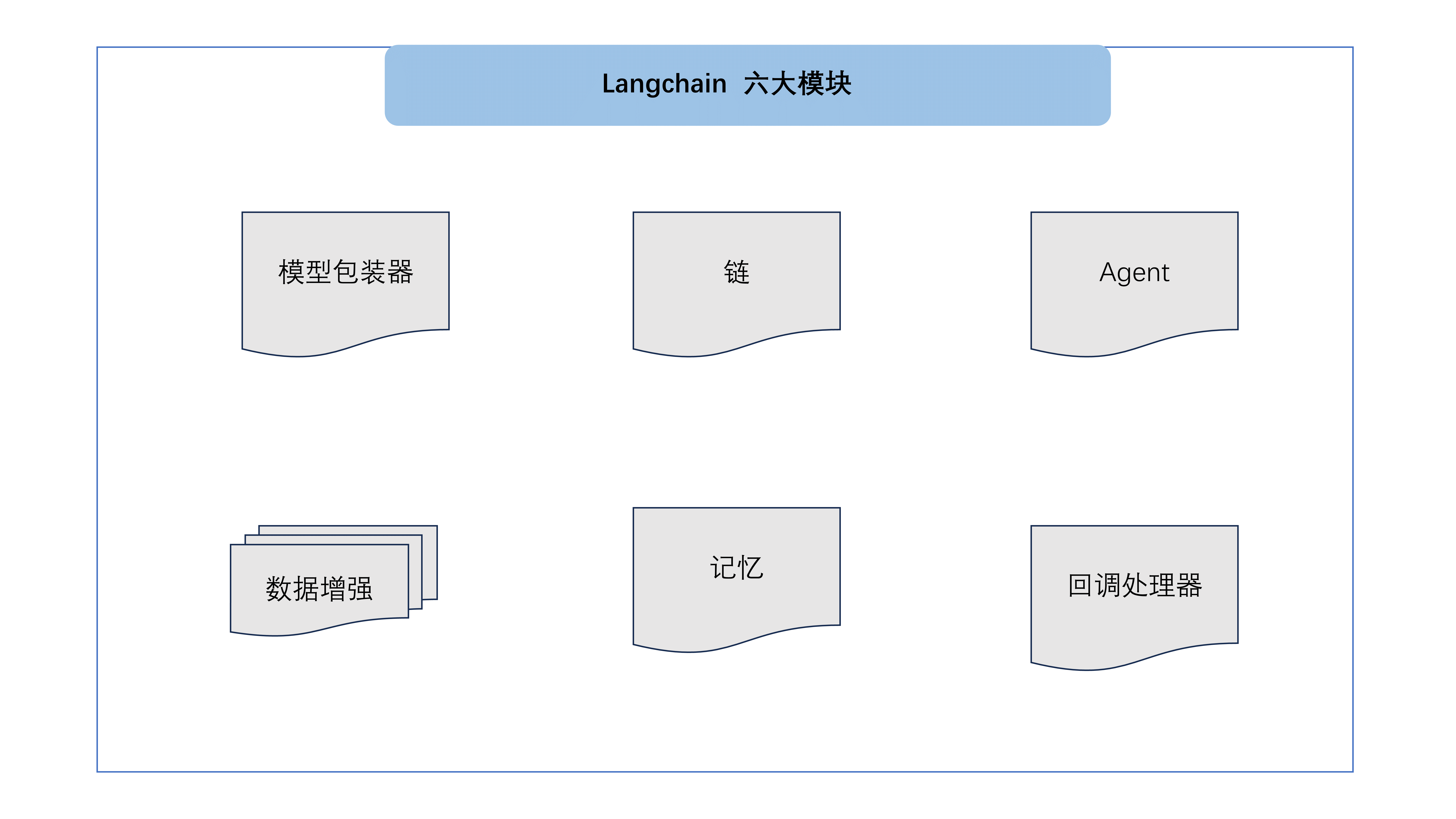
针对LLM应用开发者的需求，Langchain推出了六大核心模块。这些模块覆盖了从模型I/O到数据增强，从链到记忆，以及从Agent模块的推理决策到回调处理器的实时监控等全方位功能。借助这些模块中的包装器和组件，开发者能够更为简便地搭建和推出高级的LLM应用。

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。

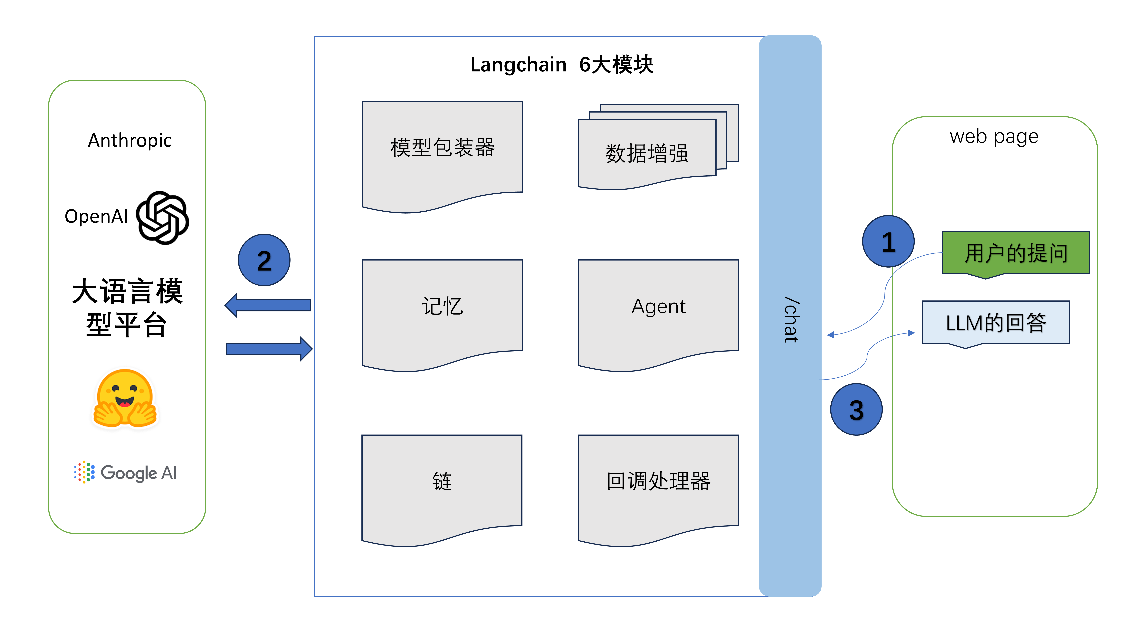


图 2‑2

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。