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LLM）。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。