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？”

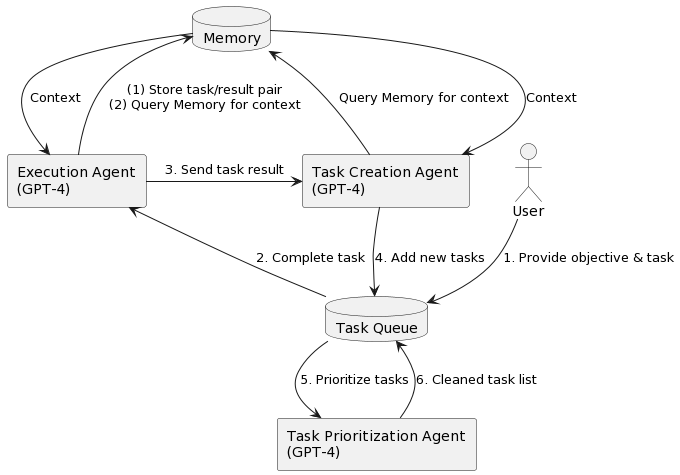
2. 任务生成 (Agent)：接下来，BabyAGI将使用诸如GPT-4之类的语言模型，将目标细分为一系列潜在任务。这些任务可能是以研究为导向的，例如“识别和关注与你的利基市场相关的Twitter帐户，以提高你的可见度并获得新的关注者”，“利用Twitter分析来跟踪你的进度并相应地调整你的策略”等。然后将任务列表存储在长期内存（向量数据库）中供将来参考。

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。