**简单入门20题**

**Q1:**什么是 [LangChain](https://zhida.zhihu.com/search?content_id=247484470&content_type=Article&match_order=1&q=LangChain&zhida_source=entity" \t "_blank)?

A1: LangChain 是一个用于开发由语言模型驱动的应用程序的框架。它提供了一套工具和组件，用于构建复杂的、基于语言模型的应用，如问答系统、聊天机器人等。

**Q2:** LangChain 的主要组件有哪些?

A2: LangChain 的主要组件包括:模型(Models)、提示(Prompts)、**索引(Indexes)**、链(Chains)、代理(Agents)、内存(Memory)。

**Q3**: 在 LangChain 中，什么是链(Chain)?

A3: 链是 LangChain 中的一个核心概念，它允许将多个组件(如提示、模型、解析器等)组合在一起，形成一个端到端的调用序列，用于完成复杂的任务。

**Q4: LangChain 中的代理(Agent)是什么?**

A4: 代理是一种**能够使用工具并决定下一步行动的实体**。它可以根据用户输入和当前状态，选择适当的工具来执行任务，实现更复杂的交互和问题解决。

**Q5:**解释 LangChain 中的提示模板(Prompt Template)概念。

A5: 提示模板是一种**结构化的方式来创建提示**。它允许您定义一个带有**占位符的模板**，然后在运行时用实际值填充这些占位符，从而生成完整的提示。

**Q6:** LangChain 中的内存(Memory)组件有什么作用?

A6: 内存组件用于在对话或交互过程中**存储和检索历史信息**。它允许模型或应用程序**记住之前的交互，从而实现更连贯和上下文相关的对话**。

**Q7**: 在 LangChain 中，如何实现文档检索?

A7: LangChain 提供了多种**文档加载器和文本分割器**，可以将文档加载并分割成块。然后，可以使用向量存储来索引这些文本块，并使用相似性搜索来检索相关信息。

**Q8:** LangChain 支持哪些语言模型?

A8: LangChain 支持多种语言模型，包括 OpenAI 的 [GPT](https://zhida.zhihu.com/search?content_id=247484470&content_type=Article&match_order=1&q=GPT&zhida_source=entity) 系列、[Anthropic](https://zhida.zhihu.com/search?content_id=247484470&content_type=Article&match_order=1&q=Anthropic&zhida_source=entity" \t "_blank) 的 Claude、Google 的 [PaLM](https://zhida.zhihu.com/search?content_id=247484470&content_type=Article&match_order=1&q=PaLM&zhida_source=entity" \t "_blank)、[Hugging Face](https://zhida.zhihu.com/search?content_id=247484470&content_type=Article&match_order=1&q=Hugging+Face&zhida_source=entity" \t "_blank) 的模型，以及许多开源模型如 [LLaMA](https://zhida.zhihu.com/search?content_id=247484470&content_type=Article&match_order=1&q=LLaMA&zhida_source=entity" \t "_blank) 等。

**Q9:** 什么是 LangChain 中的**输出解析器(Output Parser)**?

A9: 输出解析器是用来将语言模型的原始输出**转换为结构化格式的组件**。它可以将文本输出解析为特定的数据结构，如列表、字典或自定义对象。

**Q10:**LangChain 中的工具(Tool)是什么?

A10: 工具是代理可以使用的**函数或接口**。它们代表了代理可以执行的具体操作，如搜索网络、查询数据库、执行计算等。工具使代理能够与外部系统交互并执行实际任务。

**Q11:**解释 LangChain 中的向量存储(Vector Store)概念。

A11: 向量存储是用于存储和检索文本嵌入(文本的数值表示)的数据库。它允许快速进行相似性搜索，常用于实现文档检索、问答系统等功能。

**Q12:** LangChain 如何处理长文本输入?

A12: LangChain 使用文本分割器将长文本分割成较小的块。这些块可以单独处理或嵌入，然后根据需要进行检索和组合，以处理超出语言模型最大上下文长度的文本。

**Q13:**什么是 LangChain 中的检索器(Retriever)?

A13: 检索器是负责从数据源(如向量存储)中检索相关信息的组件。它通常用于问答系统，可以根据查询找到最相关的文档或文本**片段**。

**Q14:** LangChain 中的评估(Evaluation)功能是什么?

A14: LangChain 提供了评估模块，用于测试和评估语言模型应用的性能。它包括各种指标和方法来评估生成文本的质量、相关性和准确性。

**Q15:** 解释 LangChain 中的少样本学习(Few-shot Learning)概念。

A15: 少样本学习是指在提示中包含一些示例，以指导模型如何回答或完成任务。LangChain 的提示模板可以轻松地构建包含少量示例的提示，从而提高模型的性能。

**Q16:** LangChain 中的 SQL 数据库代理是如何工作的?

A16: SQL 数据库代理允许语言模型与 SQL 数据库交互。它可以**将自然语言查询转换为 SQL 查询，执行查询，然后将结果解释回自然语言，实现数据库的自然语言接口**。

**Q17: 在 LangChain 中，如何实现多轮对话?**

A17: 多轮对话可以通过使用**内存组件**来实现。**ConversationBufferMemory 或 ConversationSummaryMemory** 等内存类型可以存储对话历史，并在每次交互时将其纳入上下文。

**Q18:**LangChain 中的流式处理(Streaming)是什么?

A18: 流式处理允许语言模型的输出逐步生成和处理，而不是等待整个响应完成。这对于实现实时、交互式的用户界面很有用，可以提供更快的反馈。

**Q19**: 解释 LangChain 中的 **LCEL (LangChain Expression Language)**。

A19: LCEL 是 LangChain 的一种声明式 API，允许用户以更简洁和可组合的方式定义复杂的 LangChain 应用程序。它提供了一种统一的方式来组合和配置各种 LangChain 组件。

**Q20**: LangChain 如何处理敏感信息和数据安全?

A20: LangChain 提供了多种机制来处理敏感信息，包括:

1. 使用环境变量存储 API 密钥。
2. 提供**自定义回调**来审计和记录模型交互。
3. 支持本地部署的模型和私有云环境。
4. 允许实现自定义的数据处理和过滤逻辑。

**中等难度18题**

**Q1: 在 LangChain 中，如何使用 LLMChain 实现一个简单的问答系统?**

A1: 可以使用以下代码创建一个基本的问答系统:

from langchain import PromptTemplate， LLMChain

from langchain.llms import OpenAI

template = """Question: {question}

Answer: Let's approach this step-by-step:"""

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

llm\_chain = LLMChain(prompt=prompt， llm=OpenAI(temperature=0))

question = "What is the capital of France?"

print(llm\_chain.run(question))

这段代码创建了一个提示模板，然后使用 OpenAI 模型和这个模板构建了一个 LLMChain。

**Q2: 解释 LangChain 中的 [RetrievalQA](https://zhida.zhihu.com/search?content_id=247484470&content_type=Article&match_order=1&q=RetrievalQA&zhida_source=entity" \t "_blank) 链的工作原理。**

A2: RetrievalQA 链结合了检索器和问答功能。它的工作流程如下:

1. 接收用户查询
2. 使用检索器从知识库中检索相关文档
3. 将检索到的文档与原始查询结合，生成新的提示
4. 将新提示传递给语言模型生成答案
5. 返回最终答案给用户

**Q3: 如何在 LangChain 中实现文本嵌入和相似性搜索?**

A3: 可以使用以下步骤:

1. 选择嵌入模型(如 OpenAIEmbeddings)
2. 创建向量存储(如 FAISS)
3. 将文档添加到向量存储
4. 使用向量存储的相似性搜索功能

示例代码:

from langchain.embeddings import OpenAIEmbeddings

from langchain.vectorstores import FAISS

embeddings = OpenAIEmbeddings()

texts = ["Text 1"， "Text 2"， "Text 3"]

vector\_store = FAISS.from\_texts(texts， embeddings)

query = "Sample query"

docs = vector\_store.similarity\_search(query)

**Q4: 在 LangChain 中，如何实现自定义工具(Custom Tool)?**

A4: 可以通过继承 BaseTool 类并实现 \_run 方法来创建自定义工具:

**from** langchain.tools **import** BaseTool

**class** **CustomTool**(BaseTool):

name **=** "custom\_tool"

description **=** "A custom tool that does X"

**def** **\_run**(self， query: str) **->** str:

*# 实现工具的逻辑*

**return** f"Custom tool result for: {query}"

**async** **def** **\_arun**(self， query: str) **->** str:

*# 异步实现（可选）*

**return** f"Async custom tool result for: {query}"

**Q5: 解释 LangChain 中的 ZeroShotAgent 和 ReActAgent 的区别。**

A5:

* ZeroShotAgent: 不需要预定义的例子，可以动态选择工具。适用于**工具集可能变化**的场景。
* ReActAgent: 使用 "Reason + Act" 方法，**先推理后行动。它会生成推理步骤，然后执行操作，适合需要复杂推理的任务**。

**Q6: 如何在 LangChain 中实现长期记忆?**

A6: 可以使用 **VectorStoreRetrieverMemory** 实现长期记忆:

**from** langchain.memory **import** VectorStoreRetrieverMemory

**from** langchain.embeddings **import** OpenAIEmbeddings

**from** langchain.vectorstores **import** FAISS

embeddings **=** OpenAIEmbeddings()

vectorstore **=** FAISS**.**from\_texts([""]， embeddings)

retriever **=** vectorstore**.**as\_retriever(search\_kwargs**=**dict(k**=**1))

memory **=** VectorStoreRetrieverMemory(retriever**=**retriever)

*# 存储记忆*

memory**.**save\_context({"input": "My name is Claude"}， {"output": "Nice to meet you， Claude!"})

*# 检索记忆*

print(memory**.**load\_memory\_variables({"prompt": "What is my name?"}))

*## 经过验证，结果如下*

{'history': 'input: My name is Claude\noutput: Nice to meet you， Claude!'}

**Q7: 在 LangChain 中，如何处理流式输出?**

A7: 可以使用 CallbackHandler 来处理流式输出:

from langchain.callbacks.streaming\_stdout import StreamingStdOutCallbackHandler

from langchain.llms import OpenAI

llm = OpenAI(streaming=True， callbacks=[StreamingStdOutCallbackHandler()])

llm("Write a story about a magic forest")

这将使模型的输出实时流式显示在标准输出中。

**Q8: 解释 LangChain 中的 ConstitutionalChain 的作用和实现方式。**

A8: ConstitutionalChain 用于在语言模型的输出中强制执行某些规则或约束。它通过以下方式实现:

1. 生成初始响应
2. 使用一组预定义的规则评估响应
3. 如果违反规则，生成修订后的响应
4. 重复此过程直到满足所有规则或达到最大尝试次数

这有助于确保模型输出符合特定的道德、法律或风格准则。

**Q9: 如何在 LangChain 中实现多语言支持?**

A9: 可以通过以下方法实现多语言支持:

1. 使用支持多语言的嵌入模型(如 OpenAI 的多语言模型)
2. 为不同语言创建专门的提示模板
3. 使用翻译服务作为中间步骤
4. 选择具有多语言能力的底层语言模型

例如，可以创建一个多语言问答系统:

from langchain.llms import OpenAI

from langchain.prompts import PromptTemplate

llm = OpenAI(temperature=0)

templates = {

"en": "Answer the following question in English: {question}"，

"fr": "Répondez à la question suivante en français : {question}"，

"es": "Responda la siguiente pregunta en español: {question}"

}

def multi\_lingual\_qa(question， language):

prompt = PromptTemplate(template=templates[language]， input\_variables=["question"])

return llm(prompt.format(question=question))

**Q10: 在 LangChain 中，如何实现文档问答系统的上下文窗口大小动态调整?**

A10: 可以使用 TokenTextSplitter 并动态调整 chunk\_size:

from langchain.text\_splitter import TokenTextSplitter

from langchain.document\_loaders import TextLoader

from langchain.vectorstores import FAISS

from langchain.embeddings import OpenAIEmbeddings

def load\_docs(file\_path， chunk\_size):

loader = TextLoader(file\_path)

splitter = TokenTextSplitter(chunk\_size=chunk\_size， chunk\_overlap=0)

docs = loader.load\_and\_split(text\_splitter=splitter)

return FAISS.from\_documents(docs， OpenAIEmbeddings())

# 动态调整 chunk\_size

chunk\_size = 500 # 初始值

vectorstore = load\_docs("document.txt"， chunk\_size)

# 如果检索结果不理想，可以调整 chunk\_size 并重新加载

chunk\_size = 1000

vectorstore = load\_docs("document.txt"， chunk\_size)

**Q11: 解释 LangChain 中的 [HydeRetriever](https://zhida.zhihu.com/search?content_id=247484470&content_type=Article&match_order=1&q=HydeRetriever&zhida_source=entity" \t "_blank) 的工作原理和用途。**

A11: HydeRetriever (Hypothetical Document Embeddings) 是一种先进的检索方法，其工作原理如下:

1. 对给定的查询，使用语言模型生成一个假设的理想文档
2. 将这个假设文档嵌入向量空间
3. 使用这个嵌入来检索实际文档库中的相似文档

这种方法可以提高检索的相关性，特别是对于复杂或抽象的查询。

**Q12: 如何在 LangChain 中实现自定义的输出解析器?**

A12: 可以通过继承 BaseOutputParser 类来实现自定义输出解析器:

from langchain.schema import BaseOutputParser

class CustomOutputParser(BaseOutputParser):

def parse(self， text: str):

# 实现自定义解析逻辑

parts = text.split("， ")

return {

"name": parts[0]，

"age": int(parts[1])，

"occupation": parts[2]

}

parser = CustomOutputParser()

result = parser.parse("Alice， 30， Engineer")

print(result) # {'name': 'Alice'， 'age': 30， 'occupation': 'Engineer'}

**Q13: 在 LangChain 中，如何实现跨多个文档的共指消解(Coreference Resolution)?**

A13: 可以使用 NeuralCoref 等库结合 LangChain 的文档处理流程:

import spacy

import neuralcoref

from langchain.text\_splitter import SpacyTextSplitter

nlp = spacy.load("en\_core\_web\_sm")

neuralcoref.add\_to\_pipe(nlp)

def resolve\_coreferences(text):

doc = nlp(text)

return doc.\_.coref\_resolved

splitter = SpacyTextSplitter(chunk\_size=1000)

docs = splitter.create\_documents([long\_text])

resolved\_docs = [resolve\_coreferences(doc.page\_content) for doc in docs]

这个过程可以提高**跨文档检索**的准确性。

**Q14: 解释 LangChain 中的 GraphQLAPIChain 的用途和基本使用方法。**

A14: GraphQLAPIChain 用于与 GraphQL API 交互。它允许语言模型构建和执行 GraphQL 查询。基本使用方法如下:

from langchain.chains import GraphQLAPIChain

from langchain.llms import OpenAI

graphql\_endpoint = "https://api.example.com/graphql"

graphql\_schema = """

type Query {

user(id: ID!): User

}

type User {

id: ID!

name: String!

email: String!

}

"""

chain = GraphQLAPIChain.from\_llm\_and\_api\_doc(

OpenAI(temperature=0)，

graphql\_schema，

graphql\_endpoint，

verbose=True

)

response = chain.run("Get the name of the user with ID 123")

**Q15: 如何在 LangChain 中实现基于语义的文本块分割?**

A15: 可以使用 SpacyTextSplitter 并自定义分割规则:

from langchain.text\_splitter import SpacyTextSplitter

import spacy

nlp = spacy.load("en\_core\_web\_sm")

def semantic\_split\_function(text):

doc = nlp(text)

return [sent.text for sent in doc.sents]

splitter = SpacyTextSplitter(

chunk\_size=1000，

chunk\_overlap=200，

length\_function=len，

split\_function=semantic\_split\_function

)

docs = splitter.create\_documents([long\_text])

这种方法可以确保文本块在语义上是连贯的。

PS: **Spacy**：指的是 spaCy，这是一个用于自然语言处理的 Python 库。它专门处理和分析文本，能够提供高级的文本分割、标记化、词性标注等功能。

**Q16: 在 LangChain 中，如何实现多模态(文本+图像)的问答系统?**

A16: 可以结合使用 LangChain 和 CLIP 模型来实现:

from langchain import PromptTemplate， LLMChain

from langchain.llms import OpenAI

from PIL import Image

import clip

import torch

# 加载 CLIP 模型

device = "cuda" if torch.cuda.is\_available() else "cpu"

model， preprocess = clip.load("ViT-B/32"， device=device)

def encode\_image(image\_path):

image = preprocess(Image.open(image\_path)).unsqueeze(0).to(device)

with torch.no\_grad():

image\_features = model.encode\_image(image)

return image\_features.cpu().numpy()

# 创建 LLMChain

template = """

Analyze the following image and answer the question:

Image description: {image\_description}

Question: {question}

Answer:

"""

prompt = PromptTemplate(template=template， input\_variables=["image\_description"， "question"])

llm\_chain = LLMChain(prompt=prompt， llm=OpenAI(temperature=0))

# 使用系统

image\_features = encode\_image("example.jpg")

image\_description = "A detailed description of the image based on CLIP features"

question = "What is the main object in the image?"

response = llm\_chain.run(image\_description=image\_description， question=question)

print(response)

**Q17: 解释 LangChain 中的 SQLDatabaseChain 的工作原理和如何防止 SQL 注入。**

A17: SQLDatabaseChain 允许语言模型与 SQL 数据库交互。其工作原理是:

1. 接收自然语言查询
2. 使用 LLM 将查询转换为 SQL
3. 执行 SQL 查询
4. 将结果解释回自然语言

防止 SQL 注入的方法:

1. 使用参数化查询
2. 实现自定义的 SQL 验证逻辑
3. 限制数据库用户的权限

示例:

from langchain import SQLDatabaseChain， OpenAI

from langchain.sql\_database import SQLDatabase

db = SQLDatabase.from\_uri("sqlite:///example.db")

llm = OpenAI(temperature=0)

db\_chain = SQLDatabaseChain.from\_llm(llm， db， verbose=True)

# 使用参数化查询

def safe\_query(query， params):

return db\_chain.run(f"Execute this query: {query}"， \*\*params)

result = safe\_query("SELECT \* FROM users WHERE name = :name"， {"name": "Alice"})

**Q18: 如何在 LangChain 中实现基于令牌计数的成本估算?**

A18: 可以使用自定义回调处理程序来跟踪令牌使用情况:

* **定义自定义回调处理程序**：创建一个继承自 langchain.callbacks.BaseCallbackHandler 的自定义回调处理程序，用于记录令牌的使用情况。你可以在回调中捕捉每次 API 调用的令牌消耗量。
* **集成回调处理程序**：将自定义回调处理程序与 LangChain 集成，在使用 LangChain 的过程中启用这个处理程序。
* **计算成本**：根据 OpenAI 提供的价格表，将记录下来的令牌使用量转换为相应的费用。

**from** langchain.callbacks **import** BaseCallbackHandler

**from** langchain **import** LLMChain

**from** openai **import** ChatCompletion

**class** **TokenCostTracker**(BaseCallbackHandler):

**def** \_\_init\_\_(self):

self**.**total\_tokens **=** 0

self**.**total\_cost **=** 0.0

**def** **on\_llm\_new\_token**(self， token: str， **\*\***kwargs):

*# 计算令牌数量*

self**.**total\_tokens **+=** 1

**def** **on\_llm\_end**(self， response: dict， **\*\***kwargs):

*# 假设 ada-002 每1000 tokens 需要 $0.0004*

tokens\_used **=** response['usage']['total\_tokens']

cost\_per\_token **=** 0.0004 **/** 1000

self**.**total\_cost **+=** tokens\_used **\*** cost\_per\_token

**def** **report**(self):

**return** self**.**total\_tokens， self**.**total\_cost

*# 使用自定义回调处理程序*

tracker **=** TokenCostTracker()

llm\_chain **=** LLMChain(llm**=**ChatCompletion()， callbacks**=**[tracker])

*# 执行链操作*

llm\_chain**.**run("Your input here")

*# 获取成本报告*

tokens， cost **=** tracker**.**report()

**print**(f"Total tokens used: {tokens}")

**print**(f"Estimated cost: ${cost:.4f}")

**高等难度15题**

**Q1: 解释 LangChain 中的 LCEL (LangChain Expression Language) 的工作原理，并给出一个复杂的 LCEL 表达式示例。**

A1: LCEL 是一种声明式 API，允许以可组合的方式定义复杂的 LangChain 应用。它的工作原理基于运算符重载和懒惰求值。下面是一个复杂的 LCEL 表达式示例:

from langchain.chat\_models import ChatOpenAI

from langchain.prompts import ChatPromptTemplate

from langchain.schema.output\_parser import StrOutputParser

from langchain.schema.runnable import RunnablePassthrough

from langchain.vectorstores import FAISS

from langchain.embeddings import OpenAIEmbeddings

from langchain.schema.runnable import RunnableParallel

# 假设我们已经有了一个向量存储

vectorstore = FAISS.from\_texts(["text1"， "text2"， "text3"]， OpenAIEmbeddings())

retriever = vectorstore.as\_retriever()

model = ChatOpenAI()

template = """Answer the question based only on the following context:

{context}

Question: {question}

"""

prompt = ChatPromptTemplate.from\_template(template)

chain = (

RunnableParallel(

{"context": retriever， "question": RunnablePassthrough()}

)

| prompt

| model

| StrOutputParser()

)

result = chain.invoke("What is the meaning of life?")

print(result)

这个 LCEL 表达式创建了一个复杂的链，包括并行检索、提示构建、模型调用和输出解析。

**Q2: 描述 LangChain 中的 Pydantic 集成，并解释如何创建一个自定义的 Pydantic 输出解析器来处理复杂的嵌套结构。**

A2: LangChain 使用 Pydantic 进行数据验证和序列化。要创建处理复杂嵌套结构的自定义 Pydantic 输出解析器，可以这样做:

**from** pydantic **import** BaseModel， Field

**from** typing **import** List

**from** langchain.output\_parsers **import** PydanticOutputParser

**from** langchain.prompts **import** PromptTemplate

**from** langchain.llms **import** OpenAI

**class** **Author**(BaseModel):

name: str **=** Field(description**=**"Name of the author")

birth\_year: int **=** Field(description**=**"Year the author was born")

**class** **Book**(BaseModel):

title: str **=** Field(description**=**"Title of the book")

author: Author **=** Field(description**=**"Author of the book")

publication\_year: int **=** Field(description**=**"Year the book was published")

**class** **Library**(BaseModel):

name: str **=** Field(description**=**"Name of the library")

books: List[Book] **=** Field(description**=**"List of books in the library")

parser **=** PydanticOutputParser(pydantic\_object**=**Library)

prompt **=** PromptTemplate(

template**=**"Provide information about a library and its books.\n{format\_instructions}\n{query}"，

input\_variables**=**["query"]，

partial\_variables**=**{"format\_instructions": parser**.**get\_format\_instructions()}

)

llm **=** OpenAI(temperature**=**0)

\_input **=** prompt**.**format\_prompt(query**=**"Describe a small library with 2 books")

output **=** llm(\_input**.**to\_string())

result **=** parser**.**parse(output)

**print**(result)

这个例子创建了一个嵌套的 Pydantic 模型结构，并使用它来解析 LLM 的输出。

**Q3: 解释 LangChain 中的 Self-querying Retriever 的工作原理，并提供一个实现示例。**

A3: Self-querying Retriever 允许检索器根据给定的查询自动构建结构化查询。其工作原理是:

1. 使用 LLM 将自然语言查询转换为结构化查询
2. 使用转换后的查询在向量存储中检索文档

实现示例:

**from** langchain.llms **import** OpenAI

**from** langchain.retrievers.self\_query.base **import** SelfQueryRetriever

**from** langchain.vectorstores **import** FAISS

**from** langchain.embeddings **import** OpenAIEmbeddings

**from** langchain.schema **import** Document

*# 创建示例文档*

docs **=** [

Document(page\_content**=**"The sky is blue"， metadata**=**{"color": "blue"， "subject": "sky"})，

Document(page\_content**=**"The grass is green"， metadata**=**{"color": "green"， "subject": "grass"})，

Document(page\_content**=**"The sun is yellow"， metadata**=**{"color": "yellow"， "subject": "sun"})，

]

*# 创建向量存储*

vectorstore **=** FAISS**.**from\_documents(docs， OpenAIEmbeddings())

*# 定义元数据字段的描述*

metadata\_field\_info **=** [

{"name": "color"， "description": "The color of the object"}，

{"name": "subject"， "description": "The subject of the sentence"}，

]

*# 创建 Self-querying Retriever*

retriever **=** SelfQueryRetriever**.**from\_llm(

llm**=**OpenAI(temperature**=**0)，

vectorstore**=**vectorstore，

document\_contents**=**"Short sentences about nature"，

metadata\_field\_info**=**metadata\_field\_info，

)

*# 使用 Self-querying Retriever*

results **=** retriever**.**get\_relevant\_documents("What color is related to the sky?")

**print**(results)

这个示例创建了一个 Self-querying Retriever，能够理解和处理与文档内容和元数据相关的复杂查询。

**Q4: 描述 LangChain 中的 Conversational Retrieval Chain 的实现细节，并解释如何优化其性能以处理大规模对话历史。**

A4: Conversational Retrieval Chain 结合了检索和对话功能。其实现细节包括:

1. 使用检索器获取相关文档
2. 将检索到的文档与对话历史和当前问题结合
3. 使用 LLM 生成回答

优化大规模对话历史的方法:

1. 使用**滑动窗口**限制历史长度
2. 实现对话**摘要**来压缩历史
3. 使用向量存储来索引对话历史

优化示例:

**from** langchain.memory **import** VectorStoreRetrieverMemory

**from** langchain.embeddings **import** OpenAIEmbeddings

**from** langchain.vectorstores **import** FAISS

**from** langchain.chains **import** ConversationalRetrievalChain

**from** langchain.chat\_models **import** ChatOpenAI

*# 创建向量存储来索引对话历史*

embeddings **=** OpenAIEmbeddings()

vectorstore **=** FAISS**.**from\_texts([""]， embeddings)

retriever **=** vectorstore**.**as\_retriever(search\_kwargs**=**dict(k**=**5))

memory **=** VectorStoreRetrieverMemory(retriever**=**retriever)

*# 创建对话检索链*

chain **=** ConversationalRetrievalChain**.**from\_llm(

llm**=**ChatOpenAI()，

retriever**=**retriever，

memory**=**memory，

)

*# 使用链进行对话*

result **=** chain({"question": "What is the capital of France?"})

**print**(result)

*# 更新记忆*

memory**.**save\_context({"input": "What is the capital of France?"}， {"output": result['answer']})

这个示例使用向量存储来有效地索引和检索对话历史，从而优化大规模对话的性能。

**Q5: 解释 LangChain 中的 Constitutional AI 的实现原理，并提供一个使用自定义规则集的示例。**

A5: Constitutional AI 在 LangChain 中通过 ConstitutionalChain 实现，其原理是:

1. 生成初始响应
2. 使用一组规则评估响应
3. 如果违反规则，生成修订后的响应
4. 重复直到满足所有规则或达到最大尝试次数

自定义规则集示例:

**from** langchain.chains.constitutional\_ai.base **import** ConstitutionalChain

**from** langchain.chains.llm **import** LLMChain

**from** langchain.prompts **import** PromptTemplate

**from** langchain.llms **import** OpenAI

llm **=** OpenAI(temperature**=**0)

*# 定义基础链*

prompt **=** PromptTemplate(

input\_variables**=**["question"]，

template**=**"Question: {question}\nAnswer: Let's approach this step-by-step:"

)

base\_chain **=** LLMChain(llm**=**llm， prompt**=**prompt)

*# 定义宪法规则*

constitution **=** """

Rule 1: Always provide factual information.

Rule 2: If unsure， express uncertainty.

Rule 3: Respect privacy and do not share personal information.

Rule 4: Promote ethical behavior and discourage harmful actions.

"""

constitutional\_chain **=** ConstitutionalChain**.**from\_llm(

chain**=**base\_chain，

constitutional\_principles**=**[constitution]，

llm**=**llm，

verbose**=**True

)

*# 使用宪法链*

response **=** constitutional\_chain**.**run("How can I hack into my neighbor's Wi-Fi?")

**print**(response)

这个示例创建了一个具有自定义规则集的 Constitutional AI 链，确保输出符合预定义的伦理和行为标准。

PS: Constitutional AI 是一种在语言模型开发和应用中，通过**引入一组规则或“宪法”来约束和指导模型行为的方法。它增强了模型输出的安全性、道德性和可控性，确保生成的内容符合预定的伦理标准或用户需求**。

**Q6: 描述 LangChain 中的 Hypothetical Document Embeddings (HyDE) 技术的工作原理，并解释如何在自定义检索系统中实现它。**

A6: HyDE 技术的工作原理是:

1. 使用 LLM 为查询生成假设性的理想文档
2. 将这个假设文档嵌入到向量空间
3. 使用这个嵌入来检索实际文档库中的相似文档

自定义实现示例:

**from** langchain.llms **import** OpenAI

**from** langchain.embeddings **import** OpenAIEmbeddings

**from** langchain.vectorstores **import** FAISS

**from** langchain.prompts **import** PromptTemplate

llm **=** OpenAI(temperature**=**0)

embeddings **=** OpenAIEmbeddings()

*# 创建 HyDE 提示*

hyde\_prompt **=** PromptTemplate(

input\_variables**=**["question"]，

template**=**"Please write a passage that answers the following question:\n\nQuestion: {question}\n\nPassage:"

)

*# 创建文档库*

docs **=** ["Document 1 content"， "Document 2 content"， "Document 3 content"]

vectorstore **=** FAISS**.**from\_texts(docs， embeddings)

**def** **hyde\_retriever**(query， k**=**3):

*# 生成假设文档*

hypothetical\_doc **=** llm(hyde\_prompt**.**format(question**=**query))

*# 嵌入假设文档*

hyde\_embedding **=** embeddings**.**embed\_query(hypothetical\_doc)

*# 使用假设文档嵌入检索相似文档*

similar\_docs **=** vectorstore**.**similarity\_search\_by\_vector(hyde\_embedding， k**=**k)

**return** similar\_docs

*# 使用 HyDE 检索器*

results **=** hyde\_retriever("What is the meaning of life?")

**print**(results)

这个示例实现了一个自定义的 HyDE 检索系统，通过生成假设文档来提高检索的相关性。

**Q7: 解释 LangChain 中的 ReAct (Reasoning and Acting) 框架的工作原理，并提供一个复杂任务解决的示例实现。**

A7: ReAct 框架结合了推理和行动，其工作原理是:

1. 思考 (Reason): 分析当前情况并决定下一步行动
2. 行动 (Act): 执行决定的行动
3. 观察 (Observe): 获取行动的结果
4. 重复上述步骤直到任务完成

复杂任务解决示例:

**from** langchain.agents **import** Tool， AgentExecutor， LLMSingleActionAgent， AgentOutputParser

**from** langchain.prompts **import** StringPromptTemplate

**from** langchain **import** OpenAI， SerpAPIWrapper， LLMChain

**from** langchain.schema **import** AgentAction， AgentFinish

**from** typing **import** List， Union

**import** re

*# 定义工具*

search **=** SerpAPIWrapper()

tools **=** [

Tool(

name **=** "Search"，

func**=**search**.**run，

description**=**"useful for when you need to answer questions about current events"

)

]

*# 定义提示模板*

template **=** """Answer the following questions as best you can. You have access to the following tools:

{tools}

Use the following format:

Question: the input question you must answer

Thought: you should always think about what to do

Action: the action to take， should be one of [{tool\_names}]

Action Input: the input to the action

Observation: the result of the action

... (this Thought/Action/Action Input/Observation can repeat N times)

Thought: I now know the final answer

Final Answer: the final answer to the original input question

Begin!

Question: {input}

Thought: """

**class** **CustomPromptTemplate**(StringPromptTemplate):

template: str

tools: List[Tool]

**def** **format**(self， **\*\***kwargs) **->** str:

intermediate\_steps **=** kwargs**.**pop("intermediate\_steps"， [])

thoughts **=** ""

**for** action， observation **in** intermediate\_steps:

thoughts **+=** action**.**log

thoughts **+=** f"\nObservation: {observation}\nThought: "

kwargs["agent\_scratchpad"] **=** thoughts

kwargs["tools"] **=** "\n"**.**join([f"{tool.name}: {tool.description}" **for** tool **in** self**.**tools])

kwargs["tool\_names"] **=** "， "**.**join([tool**.**name **for** tool **in** self**.**tools])

**return** self**.**template**.**format(**\*\***kwargs)

prompt **=** CustomPromptTemplate(

template**=**template，

tools**=**tools，

input\_variables**=**["input"， "intermediate\_steps"]

)

**class** **CustomOutputParser**(AgentOutputParser):

**def** **parse**(self， llm\_output: str) **->** Union[AgentAction， AgentFinish]:

**if** "Final Answer:" **in** llm\_output:

**return** AgentFinish(

return\_values**=**{"output": llm\_output**.**split("Final Answer:")[**-**1]**.**strip()}，

log**=**llm\_output，

)

regex **=** r"Action: (.\*?)[\n]\*Action Input:[\s]\*(.\*)"

match **=** re**.**search(regex， llm\_output， re**.**DOTALL)

**if** **not** match:

**raise** **ValueError**(f"Could not parse LLM output: `{llm\_output}`")

action **=** match**.**group(1)**.**strip()

action\_input **=** match**.**group(2)

**return** AgentAction(tool**=**action， tool\_input**=**action\_input**.**strip(" ")**.**strip('"')， log**=**llm\_output)

output\_parser **=** CustomOutputParser()

llm **=** OpenAI(temperature**=**0)

llm\_chain **=** LLMChain(llm**=**llm， prompt**=**prompt)

tool\_names **=** [tool**.**name **for** tool **in** tools]

agent **=** LLMSingleActionAgent(

llm\_chain**=**llm\_chain，

output\_parser**=**output\_parser，

stop**=**["\nObservation:"]，

allowed\_tools**=**tool\_names

)

agent\_executor **=** AgentExecutor**.**from\_agent\_and\_tools(agent**=**agent， tools**=**tools， verbose**=**True)

*# 使用 ReAct 代理执行复杂任务*

result **=** agent\_executor**.**run("What is the current population of New York City， and how does it compare to Tokyo?")

**print**(result)

这个示例实现了一个基于 ReAct 框架的代理，能够执行复杂的多步骤任务。它可以搜索信息、比较数据，并生成综合回答。

**Q8: 描述 LangChain 中的 Tool-use Supervision 机制，并提供一个实现示例，展示如何监控和限制工具的使用。**

A8: Tool-use Supervision 机制允许对代理使用工具的方式进行监控和限制。实现这一机制可以通过自定义回调处理程序和修改代理的行为。

实现示例:

**from** langchain.agents **import** Tool， AgentExecutor， LLMSingleActionAgent

**from** langchain.prompts **import** StringPromptTemplate

**from** langchain **import** OpenAI， SerpAPIWrapper， LLMChain

**from** langchain.callbacks **import** BaseCallbackHandler

**from** langchain.schema **import** AgentAction， AgentFinish

**from** typing **import** Dict， List， Any

**class** **ToolUseSupervisionHandler**(BaseCallbackHandler):

**def** \_\_init\_\_(self):

self**.**tool\_use\_count: Dict[str， int] **=** {}

self**.**max\_tool\_use **=** 3

**def** **on\_tool\_start**(

self， serialized: Dict[str， Any]， input\_str: str， **\*\***kwargs: Any

) **->** None:

tool\_name **=** serialized["name"]

self**.**tool\_use\_count[tool\_name] **=** self**.**tool\_use\_count**.**get(tool\_name， 0) **+** 1

**if** self**.**tool\_use\_count[tool\_name] **>** self**.**max\_tool\_use:

**raise** **ValueError**(f"Tool {tool\_name} has been used too many times")

**def** **on\_chain\_start**(

self， serialized: Dict[str， Any]， inputs: Dict[str， Any]， **\*\***kwargs: Any

) **->** None:

**print**(f"Starting chain: {serialized['name']}")

**class** **SupervisedAgent**(LLMSingleActionAgent):

**def** **plan**(

self， intermediate\_steps: List[Tuple[AgentAction， str]]， **\*\***kwargs: Any

) **->** Union[AgentAction， AgentFinish]:

action **=** super()**.**plan(intermediate\_steps， **\*\***kwargs)

**if** isinstance(action， AgentAction):

**print**(f"Agent is using tool: {action.tool}")

**return** action

*# 设置工具和代理*

search **=** SerpAPIWrapper()

tools **=** [

Tool(

name**=**"Search"，

func**=**search**.**run，

description**=**"useful for when you need to answer questions about current events"

)

]

template **=** """Answer the following questions as best you can. You have access to the following tools:

{tools}

Use the following format:

Question: the input question you must answer

Thought: you should always think about what to do

Action: the action to take， should be one of [{tool\_names}]

Action Input: the input to the action

Observation: the result of the action

... (this Thought/Action/Action Input/Observation can repeat N times)

Thought: I now know the final answer

Final Answer: the final answer to the original input question

Begin!

Question: {input}

Thought: """

prompt **=** StringPromptTemplate(

template**=**template，

input\_variables**=**["input"， "tools"， "tool\_names"]

)

llm **=** OpenAI(temperature**=**0)

llm\_chain **=** LLMChain(llm**=**llm， prompt**=**prompt)

agent **=** SupervisedAgent(

llm\_chain**=**llm\_chain，

allowed\_tools**=**[tool**.**name **for** tool **in** tools]，

stop**=**["\nObservation:"]

)

*# 创建带有监督的执行器*

supervision\_handler **=** ToolUseSupervisionHandler()

agent\_executor **=** AgentExecutor**.**from\_agent\_and\_tools(

agent**=**agent，

tools**=**tools，

verbose**=**True，

callbacks**=**[supervision\_handler]

)

*# 执行任务*

**try**:

result **=** agent\_executor**.**run("What are the top 5 news headlines today?")

**print**(result)

**except** **ValueError** **as** e:

**print**(f"Supervision triggered: {e}")

详细日志。

**Q9: 解释 LangChain 中的 Structured Output Parser 的工作原理，并提供一个复杂的示例，展示如何处理嵌套的JSON结构和自定义验证逻辑。**

A9: Structured Output Parser 用于将 LLM 的文本输出解析为结构化数据。它的工作原理是:

1. 定义期望的输出结构
2. 创建一个提示，指导 LLM 生成符合该结构的输出
3. 使用解析器将 LLM 的输出转换为结构化数据

处理嵌套 JSON 结构和自定义验证的示例:

from langchain.output\_parsers import PydanticOutputParser

from langchain.prompts import PromptTemplate

from langchain.llms import OpenAI

from pydantic import BaseModel， Field， validator

from typing import List， Optional

class Author(BaseModel):

name: str = Field(description="Name of the author")

birth\_year: int = Field(description="Year the author was born")

@validator('birth\_year')

def valid\_year(cls， v):

if v < 0 or v > 2023:

raise ValueError("Birth year must be between 0 and 2023")

return v

class Book(BaseModel):

title: str = Field(description="Title of the book")

author: Author = Field(description="Author of the book")

publication\_year: int = Field(description="Year the book was published")

genres: List[str] = Field(description="List of genres the book belongs to")

@validator('publication\_year')

def valid\_publication\_year(cls， v， values， \*\*kwargs):

if 'author' in values and v < values['author'].birth\_year:

raise ValueError("Publication year cannot be before author's birth year")

return v

class Library(BaseModel):

name: str = Field(description="Name of the library")

books: List[Book] = Field(description="List of books in the library")

founded\_year: Optional[int] = Field(description="Year the library was founded")

@validator('founded\_year')

def valid\_founded\_year(cls， v):

if v and (v < 1000 or v > 2023):

raise ValueError("Founded year must be between 1000 and 2023")

return v

parser = PydanticOutputParser(pydantic\_object=Library)

prompt = PromptTemplate(

template="Provide information about a library and its books.\n{format\_instructions}\n{query}"，

input\_variables=["query"]，

partial\_variables={"format\_instructions": parser.get\_format\_instructions()}

)

llm = OpenAI(temperature=0.7)

\_input = prompt.format\_prompt(query="Describe a fictional library with 2 science fiction books")

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

try:

structured\_output = parser.parse(output)

print(structured\_output)

except ValueError as e:

print(f"Validation error: {e}")

这个示例创建了一个复杂的嵌套结构(Library -> Books -> Authors)，包含自定义的验证逻辑，如检查年份的有效性和出版年不能早于作者出生年等。

**Q10: 描述 LangChain 中的 Multi-Modal Chain 的实现方法，并提供一个示例，展示如何结合文本和图像处理来完成复杂任务。**

A10: Multi-Modal Chain 结合了多种模态的处理能力，如文本和图像。实现这种链需要集成专门的图像处理模型(如 CLIP)和文本处理模型。

示例实现:

**import** torch

**from** PIL **import** Image

**from** transformers **import** CLIPProcessor， CLIPModel

**from** langchain **import** PromptTemplate， LLMChain

**from** langchain.llms **import** OpenAI

**from** langchain.chains **import** SequentialChain

*# 加载 CLIP 模型*

clip\_model **=** CLIPModel**.**from\_pretrained("openai/clip-vit-base-patch32")

clip\_processor **=** CLIPProcessor**.**from\_pretrained("openai/clip-vit-base-patch32")

**def** **process\_image**(image\_path):

image **=** Image**.**open(image\_path)

inputs **=** clip\_processor(images**=**image， return\_tensors**=**"pt"， padding**=**True， truncation**=**True)

**with** torch**.**no\_grad():

image\_features **=** clip\_model**.**get\_image\_features(**\*\***inputs)

**return** image\_features**.**squeeze()**.**tolist()

*# 创建图像描述链*

image\_prompt **=** PromptTemplate(

input\_variables**=**["image\_features"]，

template**=**"Describe the image based on these features: {image\_features}"

)

image\_chain **=** LLMChain(llm**=**OpenAI(temperature**=**0)， prompt**=**image\_prompt)

*# 创建问答链*

qa\_prompt **=** PromptTemplate(

input\_variables**=**["image\_description"， "question"]，

template**=**"Based on this image description: {image\_description}\n\nAnswer the following question: {question}"

)

qa\_chain **=** LLMChain(llm**=**OpenAI(temperature**=**0)， prompt**=**qa\_prompt)

*# 创建多模态链*

multi\_modal\_chain **=** SequentialChain(

chains**=**[image\_chain， qa\_chain]，

input\_variables**=**["image\_features"， "question"]，

output\_variables**=**["text"]

)

*# 使用多模态链*

image\_path **=** "path/to/your/image.jpg"

image\_features **=** process\_image(image\_path)

question **=** "What is the main object in the image?"

result **=** multi\_modal\_chain({

"image\_features": image\_features，

"question": question

})

**print**(result["text"])

这个示例创建了一个多模态链，它首先使用 CLIP 模型处理图像，然后使用 LLM 生成图像描述，最后基于图像描述回答问题。

**Q11: 解释 LangChain 中的 Few-Shot Learning Chain 的实现原理，并提供一个示例，展示如何动态生成和选择最优的少样本示例。**

A11: Few-Shot Learning Chain 利用少量示例来提高模型的性能。其实现原理是:

1. 准备一组示例
2. 根据当前任务动态选择最相关的示例
3. 将选定的示例与当前输入结合，生成提示
4. 使用 LLM 处理增强后的提示

动态生成和选择最优少样本示例的示例:

**from** langchain **import** PromptTemplate， LLMChain， FewShotPromptTemplate

**from** langchain.llms **import** OpenAI

**from** langchain.embeddings **import** OpenAIEmbeddings

**from** langchain.vectorstores **import** FAISS

*# 创建示例库*

examples **=** [

{"input": "What is the capital of France?"， "output": "The capital of France is Paris."}，

{"input": "Who wrote Romeo and Juliet?"， "output": "Romeo and Juliet was written by William Shakespeare."}，

{"input": "What is the boiling point of water?"， "output": "The boiling point of water is 100 degrees Celsius (212 degrees Fahrenheit)."}

]

*# 创建嵌入模型和向量存储*

embeddings **=** OpenAIEmbeddings()

vectorstore **=** FAISS**.**from\_texts([ex["input"] **for** ex **in** examples]， embeddings)

*# 创建示例选择器*

**def** **select\_examples**(query， k**=**2):

relevant\_examples **=** vectorstore**.**similarity\_search(query， k**=**k)

**return** [examples[doc**.**metadata["index"]] **for** doc **in** relevant\_examples]

*# 创建 Few-Shot 提示模板*

example\_prompt **=** PromptTemplate(

input\_variables**=**["input"， "output"]，

template**=**"Human: {input}\nAI: {output}"

)

dynamic\_prompt **=** FewShotPromptTemplate(

example\_selector**=**select\_examples，

example\_prompt**=**example\_prompt，

prefix**=**"Answer the following question based on these examples:"，

suffix**=**"Human: {input}\nAI:"，

input\_variables**=**["input"]

)

*# 创建 Few-Shot Learning Chain*

llm **=** OpenAI(temperature**=**0)

few\_shot\_chain **=** LLMChain(llm**=**llm， prompt**=**dynamic\_prompt)

*# 使用 Few-Shot Learning Chain*

question **=** "What is the largest planet in our solar system?"

result **=** few\_shot\_chain**.**run(question)

**print**(result)

这个示例实现了一个动态的 Few-Shot Learning Chain，它根据输入查询选择最相关的示例，并使用这些示例来增强 LLM 的性能。

**Q12: 描述 LangChain 中的 Modular Transformation Chain 的概念和实现方法，并提供一个示例，展示如何构建一个可定制的多步骤数据处理管道。**

A12: Modular Transformation Chain 是一种灵活的链结构，允许定义一系列可组合的数据转换步骤。其实现方法包括:

1. 定义独立的转换模块
2. 创建一个管理这些模块的主链
3. 允许动态配置和重排转换步骤

示例实现:

**from** langchain.chains.base **import** Chain

**from** langchain.chains **import** LLMChain

**from** langchain.prompts **import** PromptTemplate

**from** langchain.llms **import** OpenAI

**from** typing **import** Dict， List

**class** **ModularTransformationChain**(Chain):

**def** \_\_init\_\_(self， modules: List[TransformationModule]):

self**.**modules **=** modules

super()**.**\_\_init\_\_()

**def** **\_call**(self， inputs: Dict[str， str]) **->** Dict[str， str]:

current\_input **=** inputs["input"]

**for** module **in** self**.**modules:

current\_input **=** module({"input": current\_input})["output"]

**return** {"output": current\_input}

@property

**def** **input\_keys**(self) **->** List[str]:

**return** ["input"]

@property

**def** **output\_keys**(self) **->** List[str]:

**return** ["output"]

*# 定义转换模块*

**def** **lowercase\_transform**(text: str) **->** str:

**return** text**.**lower()

**def** **remove\_punctuation\_transform**(text: str) **->** str:

**import** string

**return** text**.**translate(str**.**maketrans(""， ""， string**.**punctuation))

**def** **summarize\_transform**(text: str) **->** str:

llm **=** OpenAI(temperature**=**0)

prompt **=** PromptTemplate(

input\_variables**=**["text"]，

template**=**"Summarize the following text:\n\n{text}\n\nSummary:"

)

chain **=** LLMChain(llm**=**llm， prompt**=**prompt)

**return** chain**.**run(text)

*# 创建转换模块*

lowercase\_module **=** TransformationModule("Lowercase"， lowercase\_transform)

remove\_punctuation\_module **=** TransformationModule("Remove Punctuation"， remove\_punctuation\_transform)

summarize\_module **=** TransformationModule("Summarize"， summarize\_transform)

*# 创建可定制的转换链*

modular\_chain **=** ModularTransformationChain([

lowercase\_module，

remove\_punctuation\_module，

summarize\_module

])

*# 使用转换链*

input\_text **=** "The quick brown fox jumps over the lazy dog. It was the best of times， it was the worst of times."

result **=** modular\_chain({"input": input\_text})

**print**(result["output"])

*# 动态重新配置链*

modular\_chain**.**modules **=** [summarize\_module， lowercase\_module]

result **=** modular\_chain({"input": input\_text})

**print**(result["output"])

这个示例实现了一个模块化的转换链，允许定义和组合不同的文本处理步骤。每个模块都是独立的，可以轻松地添加、删除或重新排序。这种结构使得创建复杂的、可定制的数据处理管道变得简单。

**Q13: 解释 LangChain 中的 Async Agent 的实现原理，并提供一个示例，展示如何构建一个能够并行执行多个任务的异步代理系统。**

A13: Async Agent 利用 Python 的异步编程特性来并行执行多个任务，提高效率。其实现原理包括:

1. 定义异步工具和操作
2. 创建支持异步执行的代理
3. 使用异步运行时来并行处理多个任务

示例实现:

**import** asyncio

**from** langchain.agents **import** Tool， AgentExecutor， LLMSingleActionAgent

**from** langchain.prompts **import** StringPromptTemplate

**from** langchain **import** OpenAI， LLMChain

**from** langchain.schema **import** AgentAction， AgentFinish

**from** typing **import** List， Union， Dict， Any

**import** aiohttp

**class** **AsyncSearchTool**(Tool):

name **=** "AsyncSearch"

description **=** "useful for when you need to answer questions about current events"

async **def** **\_arun**(self， query: str) **->** str:

async **with** aiohttp**.**ClientSession() **as** session:

async **with** session**.**get(f"https://api.duckduckgo.com/?q={query}&format=json") **as** response:

data **=** await response**.**json()

**return** data['Abstract']

**class** **AsyncCalculatorTool**(Tool):

name **=** "AsyncCalculator"

description **=** "useful for when you need to perform mathematical calculations"

async **def** **\_arun**(self， expression: str) **->** str:

**return** str(eval(expression))

async\_tools **=** [AsyncSearchTool()， AsyncCalculatorTool()]

**class** **AsyncAgentExecutor**(AgentExecutor):

async **def** **\_arun**(self， inputs: Dict[str， str]) **->** Dict[str， str]:

agent\_outcome **=** await self**.**agent**.**aplan(

intermediate\_steps**=**[]， **\*\***inputs

)

**if** isinstance(agent\_outcome， AgentFinish):

**return** agent\_outcome**.**return\_values

tasks **=** []

**for** action **in** agent\_outcome:

**if** isinstance(action， AgentAction):

tool **=** self**.**tools\_lookup[action**.**tool]

tasks**.**append(asyncio**.**create\_task(tool**.**arun(action**.**tool\_input)))

observations **=** await asyncio**.**gather(**\***tasks)

final\_result **=** await self**.**agent**.**aplan(

intermediate\_steps**=**list(zip(agent\_outcome， observations))， **\*\***inputs

)

**return** final\_result**.**return\_values

**class** **AsyncAgent**(LLMSingleActionAgent):

async **def** **aplan**(

self， intermediate\_steps: List[tuple[AgentAction， str]]， **\*\***kwargs: Any

) **->** Union[AgentAction， AgentFinish]:

*# 实现异步计划逻辑*

llm\_output **=** await self**.**llm\_chain**.**arun(

intermediate\_steps**=**intermediate\_steps， **\*\***kwargs

)

**return** self**.**output\_parser**.**parse(llm\_output)

*# 设置异步代理*

llm **=** OpenAI(temperature**=**0)

prompt **=** StringPromptTemplate(

template**=**"Answer the following questions as best you can. You have access to the following tools:\n\n{tools}\n\nUse the following format:\n\nQuestion: the input question you must answer\nThought: you should always think about what to do\nAction: the action to take， should be one of [{tool\_names}]\nAction Input: the input to the action\nObservation: the result of the action\n... (this Thought/Action/Action Input/Observation can repeat N times)\nThought: I now know the final answer\nFinal Answer: the final answer to the original input question\n\nBegin!\n\nQuestion: {input}\nThought:"，

input\_variables**=**["input"， "tools"， "tool\_names"]

)

llm\_chain **=** LLMChain(llm**=**llm， prompt**=**prompt)

agent **=** AsyncAgent(

llm\_chain**=**llm\_chain，

allowed\_tools**=**[tool**.**name **for** tool **in** async\_tools]，

output\_parser**=**None *# 为简化示例，省略了输出解析器的实现*

)

async\_executor **=** AsyncAgentExecutor**.**from\_agent\_and\_tools(

agent**=**agent，

tools**=**async\_tools，

verbose**=**True

)

*# 使用异步代理*

async **def** **main**():

result **=** await async\_executor**.**arun("What is the population of New York City， and what is 15% of that number?")

**print**(result)

asyncio**.**run(main())

这个示例实现了一个异步代理系统，能够并行执行搜索和计算任务。它使用 asyncio 来管理并发操作，大大提高了处理复杂查询的效率。

**Q14: 描述 LangChain 中的 Recursive Summarization 技术的实现方法，并提供一个示例，展示如何处理超长文本并生成分层摘要。**

A14: Recursive Summarization 是一种处理超长文本的技术，它通过递归地summarize文本块来生成分层摘要。实现步骤包括:

1. 将长文本分割成较小的块
2. 对每个块进行summary
3. 将这些summary再次组合和summarize，直到达到所需的长度

示例实现:

**from** langchain.text\_splitter **import** RecursiveCharacterTextSplitter

**from** langchain.llms **import** OpenAI

**from** langchain.chains.summarize **import** load\_summarize\_chain

**from** langchain.docstore.document **import** Document

**def** **recursive\_summarize**(text， max\_tokens**=**1000， chunk\_size**=**2000， chunk\_overlap**=**200):

llm **=** OpenAI(temperature**=**0)

text\_splitter **=** RecursiveCharacterTextSplitter(

chunk\_size**=**chunk\_size，

chunk\_overlap**=**chunk\_overlap

)

**def** **summarize\_text**(text):

docs **=** [Document(page\_content**=**t) **for** t **in** text\_splitter**.**split\_text(text)]

chain **=** load\_summarize\_chain(llm， chain\_type**=**"map\_reduce")

summary **=** chain**.**run(docs)

**return** summary

summary **=** summarize\_text(text)

**while** len(summary**.**split()) **>** max\_tokens:

summary **=** summarize\_text(summary)

**return** summary

*# 使用示例*

long\_text **=** """

[在这里插入一个非常长的文本，可能是一篇长文章或书籍内容]

"""

hierarchical\_summary **=** recursive\_summarize(long\_text)

**print**(hierarchical\_summary)

*# 生成更详细的分层摘要*

**def** **hierarchical\_summarize**(text， levels**=**3， chunk\_size**=**2000， chunk\_overlap**=**200):

llm **=** OpenAI(temperature**=**0)

text\_splitter **=** RecursiveCharacterTextSplitter(

chunk\_size**=**chunk\_size，

chunk\_overlap**=**chunk\_overlap

)

**def** **summarize\_level**(texts， level):

**if** level **==** 0 **or** len(texts) **==** 1:

**return** texts[0]

summaries **=** []

**for** t **in** texts:

docs **=** [Document(page\_content**=**t)]

chain **=** load\_summarize\_chain(llm， chain\_type**=**"stuff")

summaries**.**append(chain**.**run(docs))

**return** summarize\_level(summaries， level **-** 1)

chunks **=** text\_splitter**.**split\_text(text)

**return** summarize\_level(chunks， levels)

detailed\_summary **=** hierarchical\_summarize(long\_text)

**print**(detailed\_summary)

这个示例实现了两种递归summarization方法。第一种方法简单地reduce文本直到达到指定的token数量。第二种方法创建了一个分层的summary，允许在不同级别的细节上查看文本的summary。

**Q15: 解释 LangChain 中的 Confidence-Based Agent Routing 的概念和实现方法，并提供一个示例，展示如何构建一个基于置信度的多代理决策系统。**

A15: Confidence-Based Agent Routing 是一种根据代理的置信度来路由任务的技术。实现步骤包括:

1. 创建多个专门的代理
2. 实现一个元代理来评估每个专门代理的置信度
3. 基于置信度选择最合适的代理来处理任务

示例实现:

**from** langchain.agents **import** Tool， AgentExecutor， LLMSingleActionAgent

**from** langchain.prompts **import** StringPromptTemplate

**from** langchain **import** OpenAI， LLMChain

**from** langchain.schema **import** AgentAction， AgentFinish

**from** typing **import** List， Union， Dict， Any

**class** **ConfidenceBasedAgent**(LLMSingleActionAgent):

**def** \_\_init\_\_(self， llm\_chain， tools， confidence\_threshold**=**0.7):

self**.**llm\_chain **=** llm\_chain

self**.**tools **=** tools

self**.**confidence\_threshold **=** confidence\_threshold

**def** **plan**(self， intermediate\_steps， **\*\***kwargs):

llm\_output **=** self**.**llm\_chain**.**run(intermediate\_steps**=**intermediate\_steps， **\*\***kwargs)

confidence， action **=** self**.**parse\_output(llm\_output)

**if** confidence **>=** self**.**confidence\_threshold:

**return** AgentAction(tool**=**action["tool"]， tool\_input**=**action["tool\_input"]， log**=**llm\_output)

**else**:

**return** AgentFinish(return\_values**=**{"output": "I'm not confident enough to answer this question."}， log**=**llm\_output)

**def** **parse\_output**(self， llm\_output):

*# 简化的输出解析，实际应用中应该更健壮*

parts **=** llm\_output**.**split("\n")

confidence **=** float(parts[0]**.**split(":")[1]**.**strip())

action **=** eval(parts[1]**.**split(":"， 1)[1]**.**strip())

**return** confidence， action

*# 定义专门的代理*

**def** **create\_specialized\_agent**(name， expertise):

prompt **=** StringPromptTemplate(

template**=**f"You are an expert in {expertise}. Answer the following question:\n\n{{question}}\n\nProvide your confidence level (0-1) and the action to take:\nConfidence: <confidence>\nAction: {{tool: '<tool\_name>'， tool\_input: '<input>'}}"，

input\_variables**=**["question"]

)

llm\_chain **=** LLMChain(llm**=**OpenAI(temperature**=**0)， prompt**=**prompt)

**return** ConfidenceBasedAgent(llm\_chain**=**llm\_chain， tools**=**[])

math\_agent **=** create\_specialized\_agent("Math Agent"， "mathematics")

history\_agent **=** create\_specialized\_agent("History Agent"， "history")

science\_agent **=** create\_specialized\_agent("Science Agent"， "science")

*# 创建元代理*

meta\_prompt **=** StringPromptTemplate(

template**=**"Given the following question， determine which specialized agent is best suited to answer it:\n\nQuestion: {question}\n\nChoose from: Math Agent， History Agent， Science Agent\n\nProvide your decision as:\nAgent: <agent\_name>"，

input\_variables**=**["question"]

)

meta\_chain **=** LLMChain(llm**=**OpenAI(temperature**=**0)， prompt**=**meta\_prompt)

**class** **MetaAgent**(LLMSingleActionAgent):

**def** \_\_init\_\_(self， llm\_chain， agents):

self**.**llm\_chain **=** llm\_chain

self**.**agents **=** agents

**def** **plan**(self， intermediate\_steps， **\*\***kwargs):

llm\_output **=** self**.**llm\_chain**.**run(**\*\***kwargs)

chosen\_agent **=** self**.**parse\_output(llm\_output)

**return** AgentAction(tool**=**chosen\_agent， tool\_input**=**kwargs["question"]， log**=**llm\_output)

**def** **parse\_output**(self， llm\_output):

**return** llm\_output**.**split(":")[1]**.**strip()

meta\_agent **=** MetaAgent(llm\_chain**=**meta\_chain， agents**=**[math\_agent， history\_agent， science\_agent])

*# 创建工具*

tools **=** [

Tool(name**=**"Math Agent"， func**=**math\_agent**.**plan)，

Tool(name**=**"History Agent"， func**=**history\_agent**.**plan)，

Tool(name**=**"Science Agent"， func**=**science\_agent**.**plan)，

]

*# 创建代理执行器*

agent\_executor **=** AgentExecutor**.**from\_agent\_and\_tools(agent**=**meta\_agent， tools**=**tools， verbose**=**True)

*# 使用多代理系统*

result **=** agent\_executor**.**run("What is the square root of 144?")

**print**(result)

result **=** agent\_executor**.**run("Who was the first president of the United States?")

**print**(result)

result **=** agent\_executor**.**run("What is the boiling point of water?")

**print**(result)

（文章结束）