P37

> ollama --version

P40

> ollama serve

> OLLAMA\_HOST=:11435 ollama serve

curl http://localhost:11434/v1/chat/completions \

    -H "Content-Type: application/json" \

    -d '{

        "model": "qwen:14b",

        "messages": [

            {

                "role": "system",

                "content": "You are a helpful assistant."

            },

            {

                "role": "user",

                "content": "你好呀！"

            }

        ]

    }'

P42

> ollama pull milkey/dmeta-embedding-zh:f16

> ollama serve

> curl http://localhost:11434/api/embeddings -d '{

  "model": "all-minilm",

  "prompt": "Here is an article about llamas..."

}'

#安装rust，注意此处如果出现错误，需要修改.bash\_profile的权限

> curl --proto '=https' --tlsv1.2 -sSf https://sh.ru\*\*up.rs | sh

#从GitHub网站上下载TEI代码，项目名称为text-embeddings-inference

> git clone xxxx.git

#进入代码目录，安装metal（如果是Intel芯片，则修改metal为mkl）

> cd text-embeddings-inference

> cargo install --path router -F metal

#启动向量服务（对于Linux系统来说，可能需安装gcc/openssl：#sudo apt-get install libssl-dev gcc -y）

#拉取模型，并启动推理服务

> model=BAAI/bge-large-zh-v1.5

> text-embeddings-inference % text-embeddings-router --model-id $model --port 8080

P45

> conda --version

> python --version

#搭建名为rag的虚拟运行环境，采用默认的Python版本

> conda create -n rag python=3.12.1

#激活名为rag的虚拟运行环境

> conda activate rag

P47

> pip install chromadb

#测试Chroma

import chromadb

client = chromadb.PersistentClient(path="./chroma\_db")

client.heartbeat()

> pip install chromadb

P48

#测试Chroma

import chromadb

client = chromadb.PersistentClient(path="./chroma\_db")

client.heartbeat()

> chroma run --path ./chroma\_db

Running Chroma

Saving data to: ./chroma\_db

Connect to chroma at: http://localhost:8000

#测试Chroma，Client/Server模式

import chromadb

chroma\_client = chromadb.HttpClient(host='localhost', port=8000)

chroma\_client.heartbeat()

P49

> echo'export PATH="/Library/PostgreSQL/15/bin:$PATH"' >> ~/.zshrc

> source ~/.zshrc

# 安装brew工具，如果已经安装，那么忽略，否则请自行安装brew工具

#查看PostgreSQL数据库的最新可安装版本，并安装，此处选择15

> brew search postgresql

> brew install postgresql@15

#设置环境变量

> echo 'export PATH="/opt/homebrew/opt/postgresql@15/bin:$PATH"' >> ~/.zshrc

> source ~/.zshrc

#启动数据库服务

> brew services start postgresql@15

P50

> cd /tmp

> git clone --branch v0.6.2 https://git\*\*\*.com/pgvector/pgvector.git

> cd pgvector

> make

> sudo make install # may need sudo

psql> create extension vector;

psql> create table mystore(id bigserial PRIMARY KEY, embedding vector(3));

P51

psql> insert into mystore(embedding) VALUES ('[1.1,0.2,3.8]');

psql> select \* from mystore

> pip install llama-index

P56

#加载与读取文档

import mimetypes

import os,configparser

def loadtext(path):

path = path.rstrip()

path = path.replace(' \n', '')

#转换绝对路径

filename = os.path.abspath(path)

#判断文档存在，并获得文档类型

filetype = ''

if os.path.isfile(filename):

filetype = mimetypes.guess\_type(filename)[0]

else:

print(f"File {filename} not found")

return None

#读取文档内容

text = ""

if filetype != 'text/plain':

return None

else:

with open(filename, 'rb') as f:

text = f.read().decode('utf-8')

return text

#这里配置了一个简单的配置器，用于读取模型名称的配置，后面要用

def getconfig():

config = configparser.ConfigParser()

config.read('config.ini')

return dict(config.items("main"))

P57

#把文档分割成知识块

import jieba,re

def split\_text\_by\_sentences(source\_text: str,

sentences\_per\_chunk: int,

overlap: int) -> List[str]:

"""

简单地把文档分割为多个知识块，每个知识块都包含指定数量的句子

"""

if sentences\_per\_chunk < 2:

raise ValueError("一个句子至少要有2个chunk！")

if overlap < 0 or overlap >= sentences\_per\_chunk - 1:

raise ValueError("overlap参数必须大于等于0，且小于sentences\_per\_chunk")

#简单化处理，用正则表达式分割句子

sentences = re.split('(?<=[。！？])\s+', source\_text)

sentences = [sentence.strip() for sentence in sentences if sentence.strip() != '']

if not sentences:

print("Nothing to chunk")

return []

#处理overlap参数

chunks = []

i = 0

while i < len(sentences):

end = min(i + sentences\_per\_chunk, len(sentences))

chunk = ' '.join(sentences[i:end])

if overlap > 0 and i > 1:

overlap\_start = max(0, i - overlap)

overlap\_end = i

overlap\_chunk = ' '.join(sentences[overlap\_start:overlap\_end])

chunk = overlap\_chunk + ' ' + chunk

chunks.append(chunk.strip())

i += sentences\_per\_chunk

return chunks

P59

import ollama, chromadb

#引入自定义模块

from load import loadtext, getconfig

from splitter import split\_text\_by\_sentences

#向量模型

embedmodel = getconfig()["embedmodel"]

#向量库

chroma = chromadb.HttpClient(host="localhost", port=8000)

chroma.delete\_collection(name="ragdb")

collection = chroma.get\_or\_create\_collection(name="ragdb")

#读取文档列表，依次处理

with open('docs.txt') as f:

lines = f.readlines()

for filename in lines:

#加载文档内容

text = loadtext(filename)

#把文档分割成知识块

chunks = split\_text\_by\_sentences(source\_text=text,

sentences\_per\_chunk=8,

overlap=0)

#对知识块依次处理

for index, chunk in enumerate(chunks):

#借助基于Ollama部署的本地嵌入模型生成向量

embed = ollama.embeddings(model=embedmodel, prompt=chunk)['embedding']

#存储到向量库Chroma中，注意这里的参数

collection.add([filename+str(index)],[embed],documents=[chunk],metadatas={"source": filename})

P60

[main]

embedmodel=milkey/dmeta-embedding-zh:f16

mainmodel=qwen:32b

if \_\_name\_\_ == "\_\_main\_\_":

while True:

query = input("Enter your query: ")

if query.lower() == 'quit':

break

else:

#从向量库Chroma中查询与向量相似的知识块

results = \

collection.query(query\_embeddings=[ollama.embeddings(model=embedmodel, prompt=query)['embedding']], n\_results=3)

#打印文档内容（Chunk）

for result in results["documents"][0]:

print("----------------------------------------------------")

print(result)

> python index.py

P61

import ollama, sys, chromadb

from load import getconfig

#嵌入模型与大模型

embedmodel = getconfig()["embedmodel"]

llmmodel = getconfig()["llmmodel"]

#向量库

chroma = chromadb.HttpClient(host="localhost", port=8000)

collection = chroma.get\_or\_create\_collection("ragdb")

while True:

query = input("Enter your query: ")

if query.lower() == 'quit':

break

else:

#生成查询向量

queryembed = ollama.embeddings(model=embedmodel,

prompt=query)['embedding']

#用查询向量检索上下文

relevantdocs = collection.query(query\_embeddings=[queryembed],

n\_results=5)["documents"][0]

docs = "\n\n".join(relevantdocs)

#生成Prompt

modelquery = f"""

请基于以下的上下文回答问题，如果上下文中不包含足够的回答问题的信息，请回答'我暂时无法回答该问题'，不要编造。

上下文：

====

{docs}

====

我的问题是：{query}

"""

#交给大模型进行生成

stream = ollama.generate(model=llmmodel, prompt=modelquery, stream=True)

#流式输出生成的结果

for chunk in stream:

if chunk["response"]:

print(chunk['response'], end='', flush=True)

P63

> python chat.py

Enter your query: 百度文心一言的主要应用场景有哪些呢？

P64

#经典的5行代码的RAG应用

#加载文档

documents = SimpleDirectoryReader("../data").load\_data()

#构造向量存储索引

index = VectorStoreIndex.from\_documents(documents)

#构造查询引擎

query\_engine = index.as\_query\_engine()

#对查询引擎提问

response = query\_engine.query('这里放入data目录中知识相关的问题')

#输出答案

print(response)

P65

import chromadb

from llama\_index.core import VectorStoreIndex,StorageContext,

from llama\_index.core import SimpleDirectoryReader,Settings

from llama\_index.core.node\_parser import SentenceSplitter

from llama\_index.vector\_stores.chroma import ChromaVectorStore

from llama\_index.llms.ollama import Ollama

from llama\_index.embeddings.ollama import OllamaEmbedding

#设置模型

Settings.llm = Ollama(model="qwen:14b")

Settings.embed\_model = \

OllamaEmbedding(model\_name="milkey/dmeta-embedding-zh:f16")

#加载与读取文档

reader = SimpleDirectoryReader(input\_files=["../../data/yiyan.txt","../../data/HR.txt"])

documents = reader.load\_data()

#分割文档

node\_parser = SentenceSplitter(chunk\_size=500, chunk\_overlap=20)

nodes = node\_parser.get\_nodes\_from\_documents(documents, show\_progress=False)

#准备向量存储

chroma = chromadb.HttpClient(host="localhost", port=8000)

chroma.delete\_collection(name="ragdb")

collection = chroma.get\_or\_create\_collection(name="ragdb", metadata={"hnsw:space": "cosine"})

vector\_store = ChromaVectorStore(chroma\_collection=collection)

#准备向量存储索引

storage\_context = StorageContext.from\_defaults(vector\_store=vector\_store)

index = VectorStoreIndex(nodes,storage\_context=storage\_context)

#构造查询引擎

query\_engine = index.as\_query\_engine()

while True:

user\_input = input("问题：")

if user\_input.lower() == "exit":

break

response = query\_engine.query(user\_input)

print("AI助手：", response.response)

P68

import chromadb

from langchain\_community.llms import Ollama

from langchain\_community.embeddings import OllamaEmbeddings

from langchain import hub

from langchain\_community.vectorstores import Chroma

from langchain\_core.output\_parsers import StrOutputParser

from langchain\_core.runnables import RunnablePassthrough

from langchain\_text\_splitters import RecursiveCharacterTextSplitter

from langchain\_community.document\_loaders import DirectoryLoader

from langchain\_community.document\_loaders import TextLoader

#模型

llm = Ollama(model="qwen:14b")

embed\_model = OllamaEmbeddings(model="milkey/dmeta-embedding-zh:f16")

#加载与读取文档

loader = DirectoryLoader('../../data/', glob="\*.txt",exclude="\*tips\*.txt",loader\_cls=TextLoader)

documents = loader.load()

#分割文档

text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=500, chunk\_overlap=20)

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

#准备向量存储

chroma = chromadb.HttpClient(host="localhost", port=8000)

chroma.delete\_collection(name="ragdb")

collection = chroma.get\_or\_create\_collection(name="ragdb", metadata={"hnsw:space": "cosine"})

db =\

Chroma(client=chroma,collection\_name="ragdb",embedding\_function=embed\_model)

#存储到向量库中，构造索引

db.add\_documents(splits)

#使用检索器

retriever = db.as\_retriever()

#构造一个RAG“链”（使用LangChain框架特有的组件与表达语言）

prompt = hub.pull("rlm/rag-prompt")

rag\_chain = (

{"context": retriever | (lambda docs: "\n\n".join(doc.page\_content for doc in docs)), "question": RunnablePassthrough()}

| prompt

| llm

| StrOutputParser()

)

while True:

user\_input = input("问题：")

if user\_input.lower() == "exit":

break

response = rag\_chain.invoke(user\_input)

print("AI助手：", response)

P71

from llama\_index.core.callbacks import (

CallbackManager,

LlamaDebugHandler,

CBEventType,

)

llama\_debug = LlamaDebugHandler(print\_trace\_on\_end=True)

callback\_manager = CallbackManager([llama\_debug])

Settings.callback\_manager = callback\_manager

index = VectorStoreIndex.from\_documents(

docs, callback\_manager=callback\_manager

)

P72

pprint.pprint(llama\_debug.get\_event\_time\_info(CBEventType.QUERY))

pprint.pprint(llama\_debug.get\_event\_pairs(CBEventType.QUERY))

P73

llama\_debug.print\_trace\_map()

P75

from llama\_index.core.callbacks import CallbackManager

from langfuse.llama\_index import LlamaIndexCallbackHandler

#设置 Langfuse平台的API Key，参考上方申请

os.environ["LANGFUSE\_SECRET\_KEY"] = "sk-\*\*\*\*"

os.environ["LANGFUSE\_PUBLIC\_KEY"] = "pk-\*\*\*\*"

os.environ["LANGFUSE\_HOST"] = "https://xx.xx.xx"

#构造Langfuse平台的回调类

langfuse\_callback\_handler = LlamaIndexCallbackHandler()

#设置到全局的callback\_manager

Settings.callback\_manager = CallbackManager([langfuse\_callback\_handler])

……

#程序退出之前注意缓存，将缓存的跟踪信息发送到Langfuse Server端

langfuse\_callback\_handler.flush()

……

P80

class BaseLLM(ChainableMixin, BaseComponent):

"""BaseLLM interface."""

@abstractmethod

def metadata(self) -> LLMMetadata:

@abstractmethod

def chat(self, messages: Sequence[ChatMessage], \*\*kwargs: Any) -> ChatResponse:

@abstractmethod

def complete(

self, prompt: str, formatted: bool = False, \*\*kwargs: Any

) -> CompletionResponse:

@abstractmethod

def stream\_chat(

self, messages: Sequence[ChatMessage], \*\*kwargs: Any

) -> ChatResponseGen:

@abstractmethod

def stream\_complete(

self, prompt: str, formatted: bool = False, \*\*kwargs: Any

) -> CompletionResponseGen:

# ===== Async Endpoints =====

@abstractmethod

async def achat(

self, messages: Sequence[ChatMessage], \*\*kwargs: Any

) -> ChatResponse:

@abstractmethod

async def acomplete(

self, prompt: str, formatted: bool = False, \*\*kwargs: Any

) -> CompletionResponse:

@abstractmethod

async def astream\_chat(

self, messages: Sequence[ChatMessage], \*\*kwargs: Any

) -> ChatResponseAsyncGen:

@abstractmethod

async def astream\_complete(

self, prompt: str, formatted: bool = False, \*\*kwargs: Any

) -> CompletionResponseAsyncGen:

P81

async def acomplete(

self, prompt: str, formatted: bool = False, \*\*kwargs: Any

) -> CompletionResponse:

@abstractmethod

async def astream\_chat(

self, messages: Sequence[ChatMessage], \*\*kwargs: Any

) -> ChatResponseAsyncGen:

@abstractmethod

async def astream\_complete(

self, prompt: str, formatted: bool = False, \*\*kwargs: Any

) -> CompletionResponseAsyncGen:

......

def \_complete(self, prompt: str, \*\*kwargs: Any) -> CompletionResponse:

client = self.\_get\_client()

all\_kwargs = self.\_get\_model\_kwargs(\*\*kwargs)

self.\_update\_max\_tokens(all\_kwargs, prompt)

**response = client.completions.create(**

**prompt=prompt,**

**stream=False,**

**\*\*all\_kwargs,**

**)**

text = response.choices[0].text

......

return CompletionResponse(

text=text,

raw=response,

logprobs=logprobs,

additional\_kwargs=self.\_get\_response\_token\_counts(response),

)

P82

from llama\_index.core.llms import ChatMessage

from llama\_index.llms.openai import OpenAI

#测试complete接口

llm = OpenAI(model='gpt-3.5-turbo-1106')

resp = llm.complete("白居易是")

print(resp)

#测试chat接口

messages = [

ChatMessage(

role="system", content="你是一个聪明的AI助手"

),

ChatMessage(role="user", content="你叫什么名字？"),

]

resp = llm.chat(messages)

print(resp)

from llama\_index.llms.ollama import Ollama

llm = Ollama(model='qwen:14b')

P83

#通过设置Settings组件更改使用的默认的大模型

llm = OpenAI(model='gpt-3.5-turbo-1106')

Settings.llm = llm

......

llm = OpenAI(temperature=0.1, model="gpt-4")

index = KeywordTableIndex.from\_documents(documents, llm=llm)

query\_engine = index.as\_query\_engine() #后面查询将使用这里定义的大模型

......

P84

from llama\_index.llms.ollama import Ollama

\_MODEL\_KWARGS = {

"base\_url":"http://localhost:11434",

"model":"qwen:14b" ,

"context\_window":4096,

"request\_timeout":60.0

}

llm = Ollama(\*\*\_MODEL\_ARGS)

from llama\_index.llms.llama\_cpp import LlamaCPP

\_MODEL\_KWARGS = {"logits\_all": True, "n\_ctx": 2048, "n\_gpu\_layers": -1}

\_GENERATE\_KWARGS = {"temperature": 0.0,"top\_p": 1.0,"max\_tokens": 500,

"logprobs": 32016,

}

model\_path = Path(download\_dir) / "selfrag\_llama2\_7b.q4\_k\_m.gguf"

llm = LlamaCPP(model\_path=str(model\_path), model\_kwargs=\_MODEL\_KWARGS,

generate\_kwargs=\_GENERATE\_KWARGS,verbose=False)

P85

from llama\_index.core import Settings

Settings.context\_window = 4096

Settings.num\_output = 256

from typing import Any

from llama\_index.core.llms import (

CustomLLM,

CompletionResponse,

CompletionResponseGen,

LLMMetadata,

)

from llama\_index.core.llms.callbacks import llm\_completion\_callback

class MyLLM(CustomLLM):

model\_name: str = "custom"

dummy\_response = "你好，我是一个正在开发中的大模型......"

#实现metadata接口

@property

def metadata(self) -> LLMMetadata:

return LLMMetadata(

model\_name=self.model\_name,

)

#实现complete接口

@llm\_completion\_callback()

def complete(self, prompt: str, \*\*kwargs: Any) -> CompletionResponse:

return CompletionResponse(text=self.dummy\_response)

#实现stream\_complete接口

@llm\_completion\_callback()

def stream\_complete(

self, prompt: str, \*\*kwargs: Any

) -> CompletionResponseGen:

response = ""

for token in self.dummy\_response:

response += token

yield CompletionResponse(text=response, delta=token)

P86

llm = MyLLM()

resp = llm.complete('你好！')

print(resp)

> python llms\_cust.py

你好，我是一个正在开发中的大模型......

#替换全局默认的大模型组件

Settings.llm = MyLLM()

P87

from llama\_index.llms.langchain import LangChainLLM

from langchain\_community.llms import QianfanLLMEndpoint

llm = LangChainLLM(llm=QianfanLLMEndpoint(model='ERNIE-Bot-4'))

Settings.llm=llm

from llama\_index.core import PromptTemplate

P88

from llama\_index.llms.openai import OpenAI

template = (

"以下是提供的上下文信息：\n"

"---------------------\n"

"{context\_str}"

"\n---------------------\n"

"根据这些信息，请回答以下问题：{query\_str}\n"

)

qa\_template = PromptTemplate(template)

prompt = qa\_template.format(context\_str='小麦15 PRO是小麦公司最新推出的6.7寸大屏旗舰手机。', query\_str='小麦15pro的屏幕尺寸是多少？')

print(prompt)

messages = qa\_template.format\_messages(context\_str='小麦15 PRO是小麦公司最新推出的6.7寸大屏旗舰手机。', query\_str='小麦15pro的屏幕尺寸是多少？')

print(messages)

P89

......

query\_engine = index.as\_query\_engine()

prompts\_dict = query\_engine.get\_prompts()

pprint.pprint(prompts\_dict.keys())

pprint.pprint(prompts\_dict["response\_synthesizer:text\_qa\_template"].get\_template())

P90

my\_qa\_prompt\_tmpl\_str = (

"以下是上下文信息。\n"

"---------------------\n"

"{context\_str}\n"

"---------------------\n"

"根据上下文信息回答问题，不要依赖预置知识，不要编造。\n"

"问题: {query\_str}\n"

"回答: "

)

my\_qa\_prompt\_tmpl = PromptTemplate(my\_qa\_prompt\_tmpl\_str)

query\_engine.update\_prompts(

{"response\_synthesizer:text\_qa\_template": my\_qa\_prompt\_tmpl}

)

......

query\_engine = index.as\_query\_engine(text\_qa\_template=my\_qa\_prompt\_tmpl)

P91

my\_qa\_prompt\_tmpl\_str = (

"以下是上下文信息。\n"

"---------------------\n"

"{my\_context\_str}\n"

"---------------------\n"

"根据上下文信息回答问题，不要依赖预置知识，不要编造。\n"

"问题: {my\_query\_str}\n"

"回答: "

)

template\_var\_mappings = {"context\_str": "my\_context\_str",

"query\_str": "my\_query\_str"}

my\_qa\_prompt\_tmpl = PromptTemplate(my\_qa\_prompt\_tmpl\_str)

#使用自定义变量来格式化Prompt模板

print(my\_qa\_prompt\_tmpl.format(my\_context\_str="......",

my\_query\_str="......"))

......

#kwargs为调用format方法时携带的关键词参数

def fn\_context\_str(\*\*kwargs):

......自定义逻辑......

return fmtted\_context

prompt\_tmpl = PromptTemplate(

qa\_prompt\_tmpl\_str, function\_mappings={"context\_str":fn\_context\_str}

)

#format参数传入fn\_context\_str变量中

prompt\_tmpl.format(context\_str="...", query\_str="...")

P93

# 用于保存嵌入后的向量

Embedding = List[float]

......

#相似度计算的3种算法：余弦相似度、点积、欧几里得距离

class SimilarityMode(str, Enum):

"""Modes for similarity/distance."""

DEFAULT = "cosine"

DOT\_PRODUCT = "dot\_product"

EUCLIDEAN = "euclidean"

......

#辅助方法：两个向量相似度比较

def similarity(

embedding1: Embedding,

embedding2: Embedding,

mode: SimilarityMode = SimilarityMode.DEFAULT,

) -> float:

......

#嵌入模型基础类

class BaseEmbedding(TransformComponent):

......

@abstractmethod

def \_get\_query\_embedding(self, query: str) -> Embedding:

@abstractmethod

async def \_aget\_query\_embedding(self, query: str) -> Embedding:

@abstractmethod

def \_get\_text\_embedding(self, text: str) -> Embedding:

@abstractmethod

async def \_aget\_text\_embedding(self, text: str) -> Embedding:

......

#模块：llama-index-embedding-openai

class OpenAIEmbedding(BaseEmbedding):

......

def \_get\_text\_embedding(self, text: str) -> List[float]:

"""Get text embedding."""

client = self.\_get\_client()

return get\_embedding(

client,

text,

engine=self.\_text\_engine,

\*\*self.additional\_kwargs,

)

......

......

def get\_embedding(client: OpenAI, text: str, engine: str, \*\*kwargs: Any) -> List[float]:

text = text.replace("\n", " ")

return (

client.embeddings.create(input=[text],\

model=engine, \*\*kwargs).data[0].embedding

)

P95

from llama\_index.embeddings.openai import OpenAIEmbedding

from llama\_index.core import Settings

embed\_model = OpenAIEmbedding()

embeddings = embed\_model.get\_text\_embedding(

"中国的首都是北京"

)

print(embeddings)

> python embed\_simple.py

[0.007398069836199284, -0.011682811193168163, -0.021248308941721916, -0.00428474135696888......

embeddings1 = embed\_model.get\_text\_embedding(

"中国的首都是北京"

)

embedding2 = embed\_model.get\_text\_embedding(

"中国的首都是哪里？"

)

embedding3 = embed\_model.get\_text\_embedding(

"苹果是一种好吃的水果"

)

print(embed\_model.similarity(embeddings1, embedding2))

print(embed\_model.similarity(embeddings1, embedding3))

P96

> python embed\_simple.py

0.9324159699236407

0.7942800233749084

from llama\_index.embeddings.ollama import OllamaEmbedding

embed\_model = OllamaEmbedding(model\_name="milkey/dmeta-embedding-zh:f16")

> model=BAAI/bge-large-zh-v1.5

> text-embeddings-router --model-id $model --port 8080

from llama\_index.embeddings.text\_embeddings\_inference \

import TextEmbeddingsInference

embed\_model = TextEmbeddingsInference(

model\_name="BAAI/bge-large-zh-v1.5",

timeout=60, # timeout in seconds

embed\_batch\_size=10, # batch size for embedding

)

P97

Settings.embed\_model = \

OllamaEmbedding(model\_name="milkey/dmeta-embedding-zh:f16")

......

embed\_model=OllamaEmbedding(model\_name="milkey/dmeta-embedding-zh:f16")

index = VectorStoreIndex(nodes,embed\_model=embed\_model)

......

P98

\_MODEL\_KWARGS={

"model\_name": "milkey/dmeta-embedding-zh:f16",

"embed\_batch\_size": 50

}

embed\_model = OllamaEmbedding(\*\*\_MODEL\_KWARGS)

from llama\_index.core.embeddings import BaseEmbedding

#导入自己的嵌入模型提供的模块，实现embed方法

from ... import MyModel

class MyEmbeddng(BaseEmbedding):

def \_\_init\_\_(

self,

model\_name: str = ‘MyEmbeddingModel’

\*\*kwargs: Any,

) -> None:

P99

#构造一个模型调用对象（模拟）

self.\_model = MyModel(model\_name)

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

#生成向量（模拟）

def \_get\_text\_embedding(self, text: str) -> List[float]:

embedding = self.\_model.embed(text)

return embedding

#批量生成向量（模拟）

def \_get\_text\_embeddings(self, texts: List[str]) -> List[List[float]]:

embeddings = self.\_model.embed(

[text for text in texts]

)

return embeddings

......实现其他必需的接口......

P101

class BaseNode(BaseComponent):

"""Base node Object.

Generic abstract interface for retrievable nodes

"""

class TextNode(BaseNode):

......

class Document(TextNode):

......

P102

from llama\_index.core.schema import Document

import pprint

doc = Document(text='RAG是一种常见的大模型应用范式，它通过检索—排序—生成的方式生成文本。',metadata={'title':'RAG模型介绍','author':'llama-index'})

pprint.pprint(doc.dict())

{'class\_name': 'Document',

'embedding': None,

'end\_char\_idx': None,

'excluded\_embed\_metadata\_keys': [],

'excluded\_llm\_metadata\_keys': [],

'id\_': 'dbe95286-c380-4fb4-b77a-8ca0b85735df',

'metadata': {'author': 'llama-index', 'title': 'RAG模型介绍'},

'metadata\_seperator': '\n',

'metadata\_template': '{key}: {value}',

'relationships': {},

'start\_char\_idx': None,

'text': 'RAG是一种常见的大模型应用范式，它通过检索—排序—生成的方式生成文本。',

'text\_template': '{metadata\_str}\n\n{content}'}

P104

{'ｘtitle':'RAG模型介绍','author':'llama-index'})

P105

......

doc4 = Document(

text="百度是一家中国的搜索引擎公司。",

metadata={

"file\_name": "test.txt",

"category": "technology",

"author": "random person",

},

excluded\_llm\_metadata\_keys=["file\_name"],

excluded\_embed\_metadata\_keys=["file\_name",'author'],

metadata\_seperator=" | ",

metadata\_template="{key}=>{value}",

text\_template="Metadata: {metadata\_str}\n-----\nContent: {content}",

)

print("\n全部元数据: \n",

doc4.get\_content(metadata\_mode=MetadataMode.ALL))

print("\n嵌入模型看到的 \n",

doc4.get\_content(metadata\_mode=MetadataMode.EMBED))

print("\n大模型看到的: \n",

doc4.get\_content(metadata\_mode=MetadataMode.LLM))

print("\n没有元数据： \n",

doc4.get\_content(metadata\_mode=MetadataMode.NONE))

class MetadataMode(str, Enum):

ALL = "all" #输出全部元数据

EMBED = "embed" #输出嵌入模型看到的元数据

LLM = "llm" #输出大模型看到的元数据

NONE = "none" #不需要输出元数据

P106

全部元数据:

Metadata: file\_name=>test.txt | category=>technology | author=>random person

-----

Content: 百度是一家中国的搜索引擎公司。

嵌入模型看到的

Metadata: category=>technology

-----

Content: 百度是一家中国的搜索引擎公司。

大模型看到的:

Metadata: category=>technology | author=>random person

-----

Content: 百度是一家中国的搜索引擎公司。

没有元数据：

百度是一家中国的搜索引擎公司。

from llama\_index.core.schema import Document,TextNode,MetadataMode

texts = ["This is a test","This is another test","This is a third test"]

docs = [Document(text=text) for text in texts]

P107

from llama\_index.core import SimpleDirectoryReader

#加载一个PDF文档

docs2 = \

SimpleDirectoryReader(input\_files=["../../data/Llama2PaperDataset/source\_files/llama2.pdf"]).load\_data()

print("The number of documents in docs2 is: ", len(docs2))

#加载这个目录下的所有文档（共48个TXT文档）

docs3 = \

SimpleDirectoryReader("../../data/MiniTruthfulQADataset/source\_files").load\_data()

print("The number of documents in docs3 is: ", len(docs3))

The number of documents in docs2 is: 77

The number of documents in docs3 is: 48

P108

texts = ["This is a chunk1","This is a chunk2"]

nodes = [TextNode(text=text) for text in texts]

pprint.pprint(nodes[0])

[TextNode(id\_='c5ddd431-ef5b-4126-b461-43c65beb7d67', embedding=None, metadata={}, excluded\_embed\_metadata\_keys=[], excluded\_llm\_metadata\_keys=[], relationships={}, text='This is a chunk1', start\_char\_idx=None, end\_char\_idx=None, text\_template='{metadata\_str}\n\n{content}', metadata\_template='{key}: {value}', metadata\_seperator='\n')

......

docs = [Document(text='AIGC是一种利用人工智能技术自动生成内容的方法，这些内容可以包括文本、音频、图像、视频、代码等多种形式。\n AIGC的发展得益于深度学习技术的进步，特别是自然语言处理领域的成就，使得计算机能够更好地理解语言并实现自动化内容生成。')]

#构造一个简单的数据分割器

parser = TokenTextSplitter(chunk\_size=100, chunk\_overlap=0,separator="\n")

nodes = parser.get\_nodes\_from\_documents(docs)

for i, node in enumerate(nodes):

print(f"Node {i}: {node.text}")

P109

Node 0: AIGC是一种利用人工智能技术自动生成内容的方法，这些内容可以包括文本、音频、图像、视频、代码等多种形式。

Node 1: AIGC的发展得益于深度学习技术的进步，特别是自然语言处理领域的成就，使得计算机能够更好地理解语言并实现自动化内容生成。

nodes[0].relationships[NodeRelationship.NEXT] =

RelatedNodeInfo(node\_id=nodes[1].node\_id)

P110

Node 0 relationships:

{<NodeRelationship.SOURCE: '1'>: RelatedNodeInfo(node\_id='a66cbaa8-c6b1-4dbb-8a80-0e13534b9fe5', node\_type=<ObjectType.DOCUMENT: '4'>, metadata={}, hash='c26cd260a4ec35b565b1755e4a4ecae975a962c5cd8bf8b1ddc5c66dc4004030'), <NodeRelationship.NEXT: '3'>: RelatedNodeInfo(node\_id='cd0aff6b-139b-4c69-8d25-4c5c0ab0356e', node\_type=<ObjectType.TEXT: '1'>, metadata={}, hash='c0f4a11d100ca248a094d055f700522350b0f7ba2b41c690a87be1d5ee4adea0')}

Node 1 relationships:

{<NodeRelationship.SOURCE: '1'>: RelatedNodeInfo(node\_id='a66cbaa8-c6b1-4dbb-8a80-0e13534b9fe5', node\_type=<ObjectType.DOCUMENT: '4'>, metadata={}, hash='c26cd260a4ec35b565b1755e4a4ecae975a962c5cd8bf8b1ddc5c66dc4004030'), <NodeRelationship.PREVIOUS: '2'>: RelatedNodeInfo(node\_id='72ea0260-b3d4-41cd-867e-e3410e4e20c4', node\_type=<ObjectType.TEXT: '1'>, metadata={}, hash='346fc26780860f0efbee0605bfe886ba2170c7f3acf09aa55c60c3edaa33602a')}

P111

#手工设置元数据

doc1 = Document(text="百度是一家中国的搜索引擎公司。",metadata={"file\_name": "test.txt","category": "technology","author": "random person",})

print(doc1.metadata)

#自动生成Document对象的元数据

doc2 = SimpleDirectoryReader(input\_files=["../../data/yiyan.txt"]).load\_data()

print(doc2[0].metadata)

#元数据自动继承到Node对象

parser = TokenTextSplitter(chunk\_size=100, chunk\_overlap=0,separator="\n")

nodes = parser.get\_nodes\_from\_documents(doc2)

print(nodes[0].metadata)

{'file\_name': 'test.txt', 'category': 'technology', 'author': 'random person'}

{'file\_path': '../../data/yiyan.txt', 'file\_name': 'yiyan.txt', 'file\_type': 'text/plain', 'file\_size': 1699, 'creation\_date': '2024-04-08', 'last\_modified\_date': '2024-04-07'}

{'file\_path': '../../data/yiyan.txt', 'file\_name': 'yiyan.txt', 'file\_type': 'text/plain', 'file\_size': 1699, 'creation\_date': '2024-04-08', 'last\_modified\_date': '2024-04-07'}

P112

from llama\_index.core.extractors import SummaryExtractor

llm = Ollama(model='qwen:14b')

#自动生成Document对象的元数据

docs = SimpleDirectoryReader(input\_files=["../../data/yiyan.txt"]).load\_data()

summary\_extractor = \

SummaryExtractor(llm=llm,

show\_progress=False,

prompt\_template="请生成以下内容的中文摘要：{context\_str}  
\n 摘要:",

metadata\_mode=MetadataMode.NONE)

print(summary\_extractor.extract(docs))

P113

[{'section\_summary': '摘要：本文介绍了百度研发的人工智能大模型产品——文心一言。该模型具备理解、生成、逻辑和记忆四大基础能力，适用于工作、学习、生活中的各种场景，成为高效、便捷的助手和伙伴。'}]

......

questions\_extractor = \

QuestionsAnsweredExtractor(llm=llm,show\_progress=False,metadata\_mode=MetadataMode.NONE)

print(questions\_extractor.extract(docs))

[{'questions\_this\_excerpt\_can\_answer': '1. 文心一言的最新版本升级到了什么？这对我使用它的性能有何影响？\n\n2. 文心一言如何处理复杂的逻辑难题和数学计算，它能提供哪些步骤或策略来帮助我解决这些问题？\n\n3. 对于需要长期记忆的任务，文心一言是如何保持信息的准确性和完整性的？\n\n4. 文心一言在生成文本时是否具有原创性？如果有，它的创意来源是什么？\n\n5. 我可以在哪些平台上找到并使用文心一言？它是否支持多种语言？'}]

P114

title\_extractor =\

TitleExtractor(llm=llm,show\_progress=False,metadata\_mode=MetadataMode.NONE)

print(title\_extractor.extract(docs))

[{'document\_title': '"深入解析：百度文心一言——人工智能对话助手的革新功能与广泛应用研究"\n'}]

parser = TokenTextSplitter(chunk\_size=300, chunk\_overlap=0,separator="\n")

nodes = parser.get\_nodes\_from\_documents(docs)

for node in nodes:

print(node.ref\_doc\_id)

title\_extractor = TitleExtractor(llm=llm,metadata\_mode=MetadataMode.NONE)

print("\nTitle extracted:", title\_extractor.extract(nodes))

cc919439-8a2e-4770-8e9d-f2258ce3e5dd

cc919439-8a2e-4770-8e9d-f2258ce3e5dd

cc919439-8a2e-4770-8e9d-f2258ce3e5dd

Title extracted:

[{'document\_title': '"百度文心一言：人工智能语言助手的深度解析与功能展示"\n'}, {'document\_title': '"百度文心一言：人工智能语言助手的深度解析与功能展示"\n'}, {'document\_title': '"百度文心一言：人工智能语言助手的深度解析与功能展示"\n'}]

P115

titles = extractor.extract(nodes)

for idx, node in enumerate(nodes):

node.metadata.update(titles[idx])

P118

......

#定义一个打印Document数组的方法，这个方法在后面经常使用

def print\_docs(docs:list[Document]):

print('Count of documents:',len(docs))

for index,doc in enumerate(docs):

print("-----")

print(f"Document {index}")

print(doc.get\_content(metadata\_mode=MetadataMode.ALL))

print("-----")

#设置多种不同类型的原始文档

input\_files = [

"../../data/1-news.txt",

"../../data/2-novels.docx",

"../../data/3-taxquestions.csv",

"../../data/4-taxquestions.pdf",

"../../data/5-python.md",

"../../data/6-chatdata.png",

]

reader = SimpleDirectoryReader(input\_files=input\_files)

print\_docs(reader.load\_data())

P119

#从阅读器中导入ImageReader组件

from llama\_index.readers.file import ImageReader

#图片阅读器

image\_reader = ImageReader(keep\_image=True)

reader = \

SimpleDirectoryReader(input\_files=[input\_files[5]],

file\_extractor={".png":image\_reader})

print(reader.load\_data()[0].image)

P121

......

class PSQLReader(BaseReader):

def \_\_init\_\_(self,\*args: Any,\*\*kwargs: Any,) -> None:

super().\_\_init\_\_(\*args, \*\*kwargs)

def load\_data(self,file:Path,extra\_info: Optional[Dict]=None) -> List[Document]:

with open(file) as f:

content = f.read()

#执行这个文档中的SQL语句，获得"result"

result = execute\_sql\_and\_return\_results(content)

metadata={'file\_suffix':'SQL'}

if extra\_info:

metadata = {\*\*metadata, \*\*extra\_info}

#将"result"作为"text"生成Document对象

return [Document(text=result, metadata=metadata)]

import psycopg2

def execute\_sql\_and\_return\_results(sql: str) -> str:

conn = psycopg2.connect(

host="localhost",

user="postgres",

password="\*\*\*\*\*\*",

database="postgres"

)

cur = conn.cursor()

cur.execute(sql)

results = []

for result in cur:

results.append(str(result))

conn.close()

return "\n".join(results)

P122

reader = SimpleDirectoryReader(

input\_files=['../../data/9-test.psql'], file\_extractor={".psql": PSQLReader()}

)

documents = reader.load\_data()

print\_docs(documents)

Count of documents: 1

-----

Document 0

file\_suffix: SQL

file\_path: ../../data/9-test.psql

file\_name: 9-test.psql

file\_size: 25

creation\_date: 2024-04-18

last\_modified\_date: 2024-04-18

(1, 'tom')

(2, 'george')

-----

P123

#元数据生成函数

def gen\_metadata(file):

return {"catagory": "technology", "author": "random person", "file\_name": file}

reader = \

SimpleDirectoryReader(input\_files=[input\_files[0]],file\_metadata=gen\_metadata)

print\_docs(reader.load\_data())

Count of documents: 1

-----

Document 0

catagory: technology

author: random person

file\_name: ../../data/1-news.txt

P124

from llama\_index.readers.web import SimpleWebPageReader

web\_loader = SimpleWebPageReader(html\_to\_text=True)

docs = \

web\_loader.load\_data(urls=["https://cloud.bai\*\*.com/doc/COMATE/s/rlnvnio4a"])

print\_docs(docs) #自定义的打印docs变量的方法

from llama\_index.readers.web import BeautifulSoupWebReader

#定义一个个性化的网页内容提取方式

def \_baidu\_reader(soup: Any, url: str, include\_url\_in\_text: bool = True) ->

Tuple[str, Dict[str, Any]]:

main\_content = soup.find(class\_='main')

if main\_content:

text = main\_content.get\_text()

else:

text = ''

return text, {"title": soup.find(class\_="post\_\_title").get\_text()}

web\_loader = \

BeautifulSoupWebReader(website\_extractor={"cloud.bai\*\*.com":\_baidu\_reader})

docs = \

web\_loader.load\_data(urls=["https://cloud.baidu.com/doc/COMATE/s/rlnvnio4a"])

print\_docs(docs)

P125

Count of documents: 1

-----

Document 0

URL: https://cloud.bai\*\*.com/doc/COMATE/s/rlnvnio4a

title: 产品定价

文档中心智能代码助手公有云COMATE产品定价......

......

P126

from llama\_index.readers.database import DatabaseReader

from llama\_index.core.schema import Document,TextNode,MetadataMode

db = DatabaseReader(

scheme="postgresql", # Database Scheme

host="localhost", # Database Host

port="5432", # Database Port

user="postgres", # Database User

password="\*\*\*\*\*", # Database Password

dbname="postgres", # Database Name

)

docs = db.load\_data(query="select \* from questions")

print\_docs(docs)

Count of documents: 2

-----

Document 0

id: 1, question: 文心一言是什么？, answer: 文心一言是百度公司的大模型产品，用于提供AI智能文本生成、对话与推理的能力., createtime: 2024-01-01

-----

-----

Document 1

id: 2, question: LlamaIndex框架有什么用？, answer: LlamaIndex可以用于开发基于大模型的应用程序，提供了大量可用的开发组件与工具，简化开发过程，提高开发效率, createtime: 2024-01-02

-----

P127

......

from llama\_index.core.utilities.sql\_wrapper import SQLDatabase

from sqlalchemy import text

from sqlalchemy.engine import Engine

class DatabaseReader(BaseReader):

......省略初始化代码......

def load\_data(self, query: str) -> List[Document]:

"""

查询数据库，返回Document对象

Args:

query (str): SQL语句

Returns:

List[Document]: Document对象的列表

"""

documents = []

with self.sql\_database.engine.connect() as connection:

if query is None:

raise ValueError("A query parameter is necessary to filter the data")

else:

#执行SQL语句，获得结果

result = connection.execute(text(query))

for item in result.fetchall():

doc\_str = ", ".join(

[f"{col}: {entry}" for col, entry in zip(result.keys(), item)]

)

documents.append(Document(text=doc\_str))

return documents

P128

......

class AliOSSReader(BaseReader):

def \_\_init\_\_(

self,

bucket\_name: str,

access\_key\_id: Optional[str] = None,

access\_key\_secret: Optional[str] = None,

endpoint: Optional[str] = None,

\*args: Any,

\*\*kwargs: Any,

) -> None:

#参考OSS SDK的规范初始化bucket对象

access\_key\_id = access\_key\_id or os.getenv('ACCESS\_KEY\_ID')

access\_key\_secret = \

access\_key\_secret or os.getenv('ACCESS\_KEY\_SECRET')

endpoint = endpoint or os.getenv('ENDPOINT')

if access\_key\_id and access\_key\_secret and endpoint and bucket\_name:

auth = oss2.Auth(access\_key\_id, access\_key\_secret)

self.bucket = oss2.Bucket(auth, endpoint, bucket\_name)

else:

raise ValueError("Please provide access\_key\_id, access\_key\_secret, endpoint and bucket\_name")

#加载文档，这里支持文档通配符

def load\_data(self, object\_names: List[str]) -> List[Document]:

documents = []

for object\_pattern in object\_names:

for object\_info in oss2.ObjectIterator(self.bucket):

if fnmatch.fnmatch(object\_info.key, object\_pattern):

content = self.bucket.get\_object(object\_info.key).read()

document = Document(text=content, metadata={"file\_name": object\_info.key})

documents.append(document)

return documents

......

P130

from llama\_index.core.node\_parser import SentenceSplitter

splitter = SentenceSplitter()

nodes = splitter.get\_nodes\_from\_documents([Document(text=”This is a test.\n haha!”])

P131

index = VectorStoreIndex.from\_documents(

documents,

transformations=[SentenceSplitter(chunk\_size=1024, chunk\_overlap=20)],

)

from llama\_index.core.node\_parser.text.utils import (

split\_by\_char,

split\_by\_regex,

P132

split\_by\_sentence\_tokenizer,

split\_by\_sep,

)

fn = split\_by\_sep("\n")

result = fn('Google公司介绍 \n Google是一家搜索引擎与云计算公司，总部位于美国加利福尼亚州山景城。主要产品是搜索引擎、广告服务、企业服务、云计算等。')

print(result)

print("Size of the result array:", len(result))

fn = split\_by\_sentence\_tokenizer()

result = fn('Google公司介绍 \n Google是一家搜索引擎与云计算公司，总部位于美国加利福尼亚州山景城。主要产品是搜索引擎、广告服务、企业服务、云计算等。')

print(result)

print("Size of the result array:", len(result))

fn = split\_by\_regex("[^,.;。？！]+[,.;。？！]?")

result = fn('Google公司介绍 \n Google是一家搜索引擎与云计算公司，总部位于美国加利福尼亚州山景城。主要产品是搜索引擎、广告服务、企业服务、云计算等。')

print(result)

print("Size of the result array:", len(result))

fn = split\_by\_char()

result = fn('Google公司介绍 \n Google是一家搜索引擎与云计算公司，总部位于美国加利福尼亚州山景城。主要产品是搜索引擎、广告服务、企业服务、云计算等。')

print(result)

print("Size of the result array:", len(result))

['Google公司介绍 ', '\n Google是一家搜索引擎与云计算公司，总部位于美国加利福尼亚州山景城。主要产品是搜索引擎、广告服务、企业服务、云计算等。']

Size of the result array: 2

['Google公司介绍 \n Google是一家搜索引擎与云计算公司，总部位于美国加利福尼亚州山景城。主要产品是搜索引擎、广告服务、企业服务、云计算等。']

Size of the result array: 1

['Google公司介绍 \n Google是一家搜索引擎与云计算公司，总部位于美国加利福尼亚州山景城。', '主要产品是搜索引擎、广告服务、企业服务、云计算等。']

Size of the result array: 2

['G', 'o', 'o', 'g', 'l', 'e', '公', '司', '介', '绍', ' ', '\n', ' ', 'G', 'o', 'o', 'g', 'l', 'e', '是', '一', '家', '搜', '索', '引', '擎', '与', '云', '计', '算', '公', '司', '，', '总', '部', '位', '于', '美', '国', '加', '利', '福', '尼', '亚', '州', '山', '景', '城', '。', '主', '要', '产', '品', '是', '搜', '索', '引', '擎', '、', '广', '告', '服', '务', '、', '企', '业', '服', '务', '、', '云', '计', '算', '等', '。']

Size of the result array: 74

P133

#split\_by\_sentence\_tokenizer的实现

import nltk

tokenizer = nltk.tokenize.PunktSentencetokenizer()

def split(text: str) -> List[str]:

spans = list(tokenizer.span\_tokenize(text))

sentences = []

for i, span in enumerate(spans):

start = span[0]

if i < len(spans) - 1:

end = spans[i + 1][0]

else:

end = len(text)

sentences.append(text[start:end])

return sentences

P134

import tiktoken

import re

enc = tiktoken.encoding\_for\_model("gpt-3.5-turbo")

print('length of tokens:',len(enc.encode('Google公司是一家搜索引擎公司。')))

print('length of string:',len('Google公司是一家搜索引擎公司。'))

length of tokens: 12

length of string: 18

P135

#将分割好的文本块(splits)尝试合并成接近chunk\_size参数值的块

def \_merge(self, splits: List[str], chunk\_size: int) -> List[str]:

chunks: List[str] = []

cur\_chunk: List[str] = []

cur\_len = 0

for split in splits:

split\_len = len(self.\_tokenizer(split))

# 如果在添加新的分割后超过了块大小，

# 那么需要结束当前的块，并构造一个新的块

if cur\_len + split\_len > chunk\_size:

chunk = "".join(cur\_chunk).strip()

if chunk:

chunks.append(chunk)

# 构造一个新的块,但注意保留重叠部分

# 一直弹出前一个块的第一个元素，直到：

# 1. 当前块的大小小于块重叠部分的大小

# 2. 总大小小于块大小

while cur\_len > self.chunk\_overlap or cur\_len + split\_len > chunk\_size:

# 弹出第一个元素

first\_chunk = cur\_chunk.pop(0)

cur\_len -= len(self.\_tokenizer(first\_chunk))

cur\_chunk.append(split)

cur\_len += split\_len

# 处理最后一个块

chunk = "".join(cur\_chunk).strip()

if chunk:

chunks.append(chunk)

return chunks

P136

......

docs = [Document(text="Google公司介绍 \n Google是一家搜索引擎与云计算公司 \n 总部位于美国加利福尼亚州山景城。主要产品是搜索引擎、广告服务、企业服务、云计算等。")]

splitter = TokenTextSplitter(

chunk\_size=50,

chunk\_overlap=0,

separator="\n",

backup\_separators=["。"]

)

nodes = splitter.get\_nodes\_from\_documents(docs )

print\_nodes(nodes)

P137

Count of nodes: 2

-----

Node 0

Google公司介绍

Google是一家搜索引擎与云计算公司

-----

-----

Node 1

总部位于美国加利福尼亚州山景城。主要产品是搜索引擎、广告服务、企业服务、云计算等。

-----

......

#自定义一个分割文本的函数

def my\_chunking\_tokenizer\_fn(text:str):

#跟踪是否进入本方法

print('start my chunk tokenizer function...')

sentence\_delimiters = re.compile(u'[。！？]')

sentences = sentence\_delimiters.split(text)

return [s.strip() for s in sentences if s]

"""

docs = [Document(text="\*\*\*Google公司介绍\*\*\*Google是一家搜索引擎与云计算公司\*\*\*总部位于美国加利福尼亚州山景城。主要产品是搜索引擎、广告服务、企业服务、云计算等。")]

nnode\_parser = SentenceSplitter(chunk\_size=50,

chunk\_overlap=0,

paragraph\_separator="\*\*\*",

chunking\_tokenizer\_fn=my\_chunking\_tokenizer\_fn,

secondary\_chunking\_regex = "[^,.;。？！]+[,.;。？！]?",

separator="\n")

nodes = node\_parser1.get\_nodes\_from\_documents(docs )

print\_nodes(nodes) """

P138

Count of nodes: 2

-----

Node 0

\*\*\*Google公司介绍\*\*\*Google是一家搜索引擎与云计算公司

-----

-----

Node 1

\*\*\*总部位于美国加利福尼亚州山景城。主要产品是搜索引擎、广告服务、企业服务、云计算等。

-----

P139

start my chunk tokenizer function...

Count of nodes: 2

-----

Node 0

Google公司介绍Google是一家搜索引擎与云计算公司总部位于美国加利福尼亚州山景城

-----

-----

Node 1

主要产品是搜索引擎、广告服务、企业服务、云计算等

-----

P140

......

docs = [Document(text="Google公司介绍:Google是一家搜索引擎与云计算公司。\

总部位于美国加利福尼亚州山景城。\

主要产品是搜索引擎、广告服务、企业服务、云计算等。\

百度是一家中国的搜索引擎公司。")]

splitter = SentenceWindowNodeParser(

window\_size=2,

sentence\_splitter = my\_chunking\_tokenizer\_fn

)

nodes = splitter.get\_nodes\_from\_documents(docs)

print\_nodes(nodes)

Count of nodes: 4

-----

Node 0

window: Google公司介绍:Google是一家搜索引擎与云计算公司 总部位于美国加利福尼亚州山景城 主要产品是搜索引擎、广告服务、企业服务、云计算等 百度是一家中国的搜索引擎公司

original\_text: Google公司介绍:Google是一家搜索引擎与云计算公司

Google公司介绍:Google是一家搜索引擎与云计算公司

-----

-----

Node 1

window: Google公司介绍:Google是一家搜索引擎与云计算公司 总部位于美国加利福尼亚州山景城 主要产品是搜索引擎、广告服务、企业服务、云计算等 百度是一家中国的搜索引擎公司

original\_text: 总部位于美国加利福尼亚州山景城

总部位于美国加利福尼亚州山景城

-----

P141

from llama\_index.core.node\_parser import HierarchicalNodeParser

docs = [Document(text="Google公司介绍:Google是一家搜索引擎与云计算公司。\

总部位于美国加利福尼亚州山景城。\

Google公司成立于1998年9月4日，由拉里·佩奇和谢尔盖·布林共同创立。\

主要产品是搜索引擎、广告服务、企业服务、云计算等。\

百度是一家中国的搜索引擎公司。\

百度公司成立于2000年1月1日，由李彦宏创立。")]

node\_parser = HierarchicalNodeParser.from\_defaults(

chunk\_sizes=[2048, 100, 50]

)

nodes = node\_parser.get\_nodes\_from\_documents(docs)

print\_nodes(nodes)

P143

......

docs = SimpleDirectoryReader(input\_files=["../../data/yiyan.txt"]).load\_data()

embed\_model = OllamaEmbedding(model\_name="milkey/dmeta-embedding-zh:f16")

splitter = SemanticSplitterNodeParser(

breakpoint\_percentile\_threshold=85,

sentence\_splitter = my\_chunking\_tokenizer\_fn,

embed\_model=embed\_model

)

nodes = splitter.get\_nodes\_from\_documents(docs)

print\_nodes(nodes)

start my chunk tokenizer function...

Count of nodes: 3

-----

Node 0

file\_path: ../../data/yiyan.txt

file\_name: yiyan.txt

file\_type: text/plain

......

P144

start my chunk tokenizer function...

Count of nodes: 5

-----

Node 0

file\_path: ../../data/yiyan.txt

file\_name: yiyan.txt

file\_type: text/plain

......

......

#分割Markdown格式的文档

docs = FlatReader().load\_data(Path("../../data/5-python.md"))

markdown\_parser = MarkdownNodeParser()

nodes = markdown\_parser.get\_nodes\_from\_documents(docs )

print\_nodes(nodes)

#分割HTML格式的文档

docs = FlatReader().load\_data(Path("../../data/10-google.html"))

html\_parser = HTMLNodeParser()

nodes = html\_parser.get\_nodes\_from\_documents(docs )

print\_nodes(nodes)

#分割JSON格式的文档

docs = FlatReader().load\_data(Path("../../data/11-quantum.json"))

json\_parser = JSONNodeParser()

nodes = json\_parser.get\_nodes\_from\_documents(docs )

print\_nodes(nodes)

#分割源代码文档

docs = FlatReader().load\_data(Path("../../data/8-test.py"))

code\_parser = CodeSplitter(language="python")

nodes = code\_parser.get\_nodes\_from\_documents(docs )

print\_nodes(nodes)

P146

from llama\_index.core import Document,SimpleDirectoryReader

from llama\_index.core.node\_parser import SentenceSplitter

from llama\_index.core.extractors import TitleExtractor

from llama\_index.embeddings.ollama import OllamaEmbedding

from llama\_index.llms.ollama import Ollama

from llama\_index.core.ingestion import IngestionPipeline, IngestionCache

import pprint

llm = Ollama(model='qwen:14b')

embedded\_model = \

OllamaEmbedding(model\_name="milkey/dmeta-embedding-zh:f16",

embed\_batch\_size=50)

docs = SimpleDirectoryReader(input\_files=["../../data/yiyan.txt"]).load\_data()

#构造一个数据摄取管道

pipeline = IngestionPipeline(

transformations=[

SentenceSplitter(chunk\_size=500, chunk\_overlap=0),

TitleExtractor(llm=llm, show\_progress=False)

]

)

#运行这个数据摄取管道

nodes = pipeline.run(documents=docs)

P147

......

#文本分割，相当于第一个转换器

splitter = SentenceSplitter(chunk\_size=500, chunk\_overlap=0)

nodes = splitter.get\_nodes\_from\_documents(docs)

#抽取标题，并将其设置到Node对象的元数据中，相当于第二个转换器

extractor = TitleExtractor(llm=llm, show\_progress=False)

titles = extractor.extract(nodes)

for idx, node in enumerate(nodes):

node.metadata.update(titles[idx])

......

P148

class TransformComponent(BaseComponent):

"""Base class for transform components."""

class Config:

arbitrary\_types\_allowed = True

@abstractmethod

def \_\_call\_\_(self, nodes: List["BaseNode"], \*\*kwargs: Any) -> List["BaseNode"]:

async def acall(self, nodes: List["BaseNode"], \*\*kwargs: Any) -> List["BaseNode"]:

return self.\_\_call\_\_(nodes, \*\*kwargs)

class NodeParser(TransformComponent, ABC):

"""Base interface for node parser."""

......

#转换器接口的实现

def \_\_call\_\_(self, nodes: List[BaseNode], \*\*kwargs: Any) -> List[BaseNode]:

return self.get\_nodes\_from\_documents(nodes, \*\*kwargs)

......

P149

class BaseEmbedding(TransformComponent):

"""Base class for embeddings."""

......

#转换器接口的实现

def \_\_call\_\_(self, nodes: List[BaseNode], \*\*kwargs: Any) -> List[BaseNode]:

embeddings = self.get\_text\_embedding\_batch(

[node.get\_content(metadata\_mode=MetadataMode.EMBED) for node in nodes],

\*\*kwargs,

)

for node, embedding in zip(nodes, embeddings):

node.embedding = embedding

return nodes

from llama\_index.core.schema import TransformComponent

import re

#定义一个做数据清理的转换器

class TextCleaner(TransformComponent):

def \_\_call\_\_(self, nodes, \*\*kwargs):

for node in nodes:

node.text = re.sub(r"[^\u4e00-\u9fa5A-Za-z0-9，。？！“”‘’；：【】《》（）\[\]\"\'\.\,\?\!\:\;\(\)\n\r]", "", node.text)

return nodes

......

pipeline = IngestionPipeline(

transformations=[

SentenceSplitter(chunk\_size=500, chunk\_overlap=0),

TextCleaner(), #插入自定义转换器

TitleExtractor(llm=llm, show\_progress=False)

]

)

nodes = pipeline.run(documents=docs)

......

......构造数据摄取管道，设置转换器......

#运行数据摄取管道

nodes = pipeline.run(documents=docs)

P151

......

#构造一个向量存储对象用于存储最后输出的Node对象

chroma = chromadb.HttpClient(host="localhost", port=8000)

collection = chroma.get\_or\_create\_collection(name="pipeline")

vector\_store = ChromaVectorStore(chroma\_collection=collection)

pipeline = IngestionPipeline(

transformations=[

SentenceSplitter(chunk\_size=500, chunk\_overlap=0),

TitleExtractor(llm=llm, show\_progress=False),

embedded\_model #提供一个嵌入模型用于生成向量

],

vector\_store = vector\_store #提供一个向量存储对象，用于存储最后的Node对象

)

nodes = pipeline.run(documents=docs)

#用输入问题做语义检索

results = vector\_store.query(VectorStoreQuery(

query\_str='文心一言是什么？',

similarity\_top\_k=3))

pprint.pprint(results.nodes3)

P152

......

if \_\_name\_\_ == '\_\_main\_\_':

freeze\_support()

start\_time = time.time()

nodes = pipeline.run(documents=docs,num\_workers=2)

elapsed\_time = time.time() - start\_time

print(f'Elapsed time: {elapsed\_time:.3f} s')

P153

......

#第一次运行数据摄取管道

pipeline = IngestionPipeline(

transformations=[

SentenceSplitter(chunk\_size=1000, chunk\_overlap=0),

embedded\_model

],

docstore=SimpleDocumentStore(), #需要导入SimpleDocumentStore对象

)

docs = SimpleDirectoryReader(input\_files=["../../data/sales\_tips1.txt"]).load\_data()

nodes = pipeline.run(documents=docs,show\_progress=True)

print(f'{len(nodes)} nodes ingested into vector store')

pipeline.persist("./pipeline\_storage")

P154

#第二次运行数据摄取管道

......此处省略构造数据摄取管道的代码......

pipeline.load("./pipeline\_storage")

docs = SimpleDirectoryReader(input\_files=["../../data/sales\_tips1.txt"]).load\_data()

nodes = pipeline.run(documents=docs,show\_progress=True)

print(f'{len(nodes)} nodes ingested into vector store')

P157

# 调用模型接口生成向量，此处使用批量接口

embeddings = embedded\_model.get\_text\_embedding\_batch(

[node.get\_content(metadata\_mode=MetadataMode.EMBED) for node in nodes],show\_progress=True)

#把生成的向量放到Node对象中

for node, embedding in zip(nodes, embeddings):

node.embedding = embedding

P158

nodes = embedded\_model(nodes)

......

pipeline = IngestionPipeline(

transformations=[

splitter,

embedded\_model #把嵌入模型作为转换器，将自动生成向量

]

)

#运行后将会自动生成nodes向量

nodes =pipeline.run(documents=docs,show\_progress=True)

P159

......

#model

embedded\_model = OllamaEmbedding(model\_name="milkey/dmeta-embedding-zh:f16", embed\_batch\_size=50)

Settings.embed\_model=embedded\_model

#docs

docs = [Document(text="百度文心一言是什么？文心一言是百度的大模型品牌。",metadata={"title":"百度文心一言的概念"},doc\_id="doc1"),

Document(text="什么是大模型？大模型是一种生成式推理AI模型。",metadata={"title":"大模型的概念"},doc\_id="doc2")]

splitter = SentenceSplitter(chunk\_size=100, chunk\_overlap=0)

nodes = splitter.get\_nodes\_from\_documents(docs)

#生成嵌入向量

nodes = embedded\_model(nodes)

#存储到向量库中

simple\_vectorstore = SimpleVectorStore()

simple\_vectorstore.add(nodes)

P160

#查询

result = simple\_vectorstore.query(

VectorStoreQuery(query\_embedding=embedded\_model.get\_text\_embedding('什么是文心一言'),similarity\_top\_k=1))

print(result)

VectorStoreQueryResult(nodes=None, similarities=[0.8055511120370833], ids=['a6e7fe8f-88fa-4c53-a004-bc06a4e8bb19'])

P161

#持久化存储，默认存储到当前的./storage目录中

simple\_vectorstore.persist()

simple\_vectorstore = \

SimpleVectorStore.from\_persist\_path('./storage/vector\_store.json')

P162

......

from llama\_index.vector\_stores.chroma import ChromaVectorStore

......

#此处省略Node对象的准备过程，同上

#构造一个collection对象,此处使用Server模式下的Chroma向量库

chroma = chromadb.HttpClient(host="localhost", port=8000)

collection = chroma.get\_or\_create\_collection(name="vectorstore")

#构造向量存储对象

vector\_store = ChromaVectorStore(chroma\_collection=collection)

ids = vector\_store.add(nodes)

print(f'{len(ids)} nodes ingested into vector store')

pprint.pprint(vector\_store.\_\_dict\_\_)

2 nodes ingested into vector store

{'collection\_kwargs': {},

'collection\_name': None,

'flat\_metadata': True,

'headers': None,

'host': None,

'is\_embedding\_query': True,

'persist\_dir': None,

'port': None,

'ssl': False,

'stores\_text': True}

P163

count\_result = collection.count()

print('count\_result:',count\_result)

......

#语义检索

result =\

vector\_store.query(VectorStoreQuery(query\_embedding=embedded\_model.get\_text\_embedding('什么是语言模型'),similarity\_top\_k=1))

print(result)

P165

......省略embedded\_model与向量存储对象的构造......

#加载与解析文档，这里直接构造

docs = [

Document(text="智家机器人是一种人工智能家居软件，让您的家变得更智能，让您轻松地掌控生活的方方面面。",metadata={"title":"智家机器人"},doc\_id="doc1"),

Document(text="速达飞行者是一种飞行汽车，能够让您在城市中自由翱翔，体验全新的出行方式。",metadata={"title":"速达飞行者"},doc\_id="doc2")]

#文本分割

splitter = SentenceSplitter(chunk\_size=100, chunk\_overlap=0)

nodes = splitter.get\_nodes\_from\_documents(docs)

#嵌入向量

nodes = embedded\_model(nodes)

#存储到向量库中

vector\_store.add(nodes)

#NEW：构造基于向量存储的向量存储索引对象

index = VectorStoreIndex.from\_vector\_store(vector\_store)

#测试

query\_engine = index.as\_query\_engine()

response = query\_engine.query(

"什么是速达飞行者"

)

P166

......省略用文档生成Node对象的部分，同6.3.1节......

#以下代码被注释

#嵌入向量

#nodes = embedded\_model(nodes)

#存储到向量库中

#vector\_store.add(nodes)

#NEW：构造基于向量存储的向量存储索引对象

index = VectorStoreIndex(nodes)

#测试

query\_engine = index.as\_query\_engine()

response = query\_engine.query(

"什么是速达飞行者"

)

P168

......

index = VectorStoreIndex(nodes)

pprint.pprint(index.\_\_dict\_\_)

......

#准备向量存储对象，此处采用Chroma向量库

vector\_store = ChromaVectorStore(chroma\_collection=collection)

......

#NEW：构造基于向量存储的向量存储索引对象

storage\_context = StorageContext.from\_defaults(vector\_store=vector\_store)

index = VectorStoreIndex(nodes,storage\_context=storage\_context)

P169

......

docs = [Document(text="智家机器人是一种人工智能家居软件，让您的家变得更智能，让您轻松地掌控生活的方方面面。",metadata={"title":"智家机器人"},doc\_id="doc1"),

Document(text="速达飞行者是一种飞行汽车，能够让您在城市中自由翱翔，体验全新的出行方式。",metadata={"title":"速达飞行者"},doc\_id="doc2")]

#用文档构造向量存储索引对象

vector\_index = VectorStoreIndex.from\_documents(docs)

......

query\_engine = vector\_index.as\_query\_engine()

response = query\_engine.query(

"请解释什么是区块链技术"

)

print(response)

P170

storage\_context = StorageContext.from\_defaults(vector\_store=vector\_store)

vector\_index = \

VectorStoreIndex.from\_documents(docs,storage\_context = storage\_context)

P171

......

storage\_context = \

StorageContext.from\_defaults(docstore=docstore,vector\_store=vector\_store)

#定义文本分割器与元数据抽取器

mySplitter = SentenceWindowNodeParser(window\_size=2)

myExtractor = TitleExtractor()

#将上述组件通过transformations参数传入

vector\_index = VectorStoreIndex.from\_documents(docs,

storage\_context = storage\_context,

transformations=[mySplitter,myExtractor])

nodes = vector\_index.\_vector\_store.query(VectorStoreQuery(query\_str='速达飞行者？',similarity\_top\_k=1)).nodes

pprint.pprint(nodes[0])

P172

......

"""Create index from documents. """

#接受传入的storage\_context参数，或者使用默认的storage选项

storage\_context = storage\_context or StorageContext.from\_defaults()

......

#转换器，从输入参数中获得，或者使用默认的转换器

transformations = \

transformations or transformations\_from\_settings\_or\_context(

Settings, service\_context

)

......

#运行转换器做数据摄取，生成要处理的Node对象

nodes = run\_transformations(

documents, # type: ignore

transformations,

show\_progress=show\_progress,

\*\*kwargs,

)

#用生成的Node对象构造向量存储索引对象：回到方法二

return cls(

nodes=nodes,

storage\_context=storage\_context,

callback\_manager=callback\_manager,

show\_progress=show\_progress,

transformations=transformations,

service\_context=service\_context,

\*\*kwargs,

)

......

self.\_storage\_context = storage\_context or StorageContext.from\_defaults()

self.\_vector\_store = self.\_storage\_context.vector\_store

with self.\_callback\_manager.as\_trace("index\_construction"):

if index\_struct is None:

nodes = nodes or []

index\_struct = self.build\_index\_from\_nodes(

nodes + objects # type: ignore

)

self.\_index\_struct = index\_struct

......

......

for nodes\_batch in iter\_batch(nodes, self.\_insert\_batch\_size):

nodes\_batch = \

self.\_get\_node\_with\_embedding(nodes\_batch, show\_progress)

new\_ids = self.\_vector\_store.add(nodes\_batch, \*\*insert\_kwargs)

......

P176

......这里省略了text中的大段内容......

docs = [Document(text="小麦智能健康手环是一款...",metadata={"title":"智家机器人"},doc\_id="doc1"),Document(text="速达飞行者...",metadata={"title":"速达飞行者"},doc\_id="doc2")]

doc\_summary\_index = DocumentSummaryIndex.from\_documents(docs)

......

pprint.pprint(doc\_summary\_index.get\_document\_summary("doc1"))

P177

#vector store

......构造一个存储上下文对象，用于设置向量存储......

storage\_context = StorageContext.from\_defaults(vector\_store=vector\_store)

# storage\_context：设置向量库等

# summary\_query：设置摘要生成提示

# llm：设置生成摘要的大模型

# transformations: 设置数据摄取需要的转换器

doc\_summary\_index = DocumentSummaryIndex.from\_documents(docs,

storage\_context=storage\_context,

summary\_query="用中文描述所给文本的主要内容，同时描述这段文本可以回答的一些问题。")

pprint.pprint(doc\_summary\_index.get\_document\_summary("doc1"))

P178

#构造一些不同类型的普通对象

obj1 = {"name": "小米","cpu": "骁龙","battery": "5000mAh","display": "6.67英寸"}

obj2 = ["iPhne", "小米", "华为", "三星"]

obj3 = (['A','B','C'],[100,200,300 ])

obj4 = "大模型是一种基于自然语言处理技术的生成式AI模型!"

objs= [obj1, obj2, obj3,obj4]

# 从普通对象到Node对象的映射，即生成嵌入所需要的Node对象

obj\_node\_mapping = SimpleObjectNodeMapping.from\_objects(objs)

nodes = obj\_node\_mapping.to\_nodes(objs)

# 构造对象索引

storage\_context = StorageContext.from\_defaults(vector\_store=vector\_store)

object\_index = ObjectIndex(

index=VectorStoreIndex(nodes=nodes, storage\_context=storage\_context),

object\_node\_mapping=obj\_node\_mapping,

)

#构造一个检索器，测试检索结果

object\_retriever = object\_index.as\_retriever(similarity\_top\_k=1)

results = object\_retriever.retrieve("小米手机")

print(f'results: {results}')

P179

def to\_nodes(self, objs: Sequence[OT]) -> Sequence[TextNode]:

return [self.to\_node(obj) for obj in objs]

def to\_node(self, obj: Any) -> TextNode:

return TextNode(text=str(obj))

def \_\_init\_\_(self, objs: Optional[Sequence[Any]] = None) -> None:

objs = objs or []

for obj in objs:

self.validate\_object(obj)

self.\_objs = {hash(str(obj)): obj for obj in objs}

P180

......

storage\_context = StorageContext.from\_defaults(vector\_store=vector\_store)

object\_index = ObjectIndex.from\_objects(

objs, index\_cls=VectorStoreIndex,storage\_context=storage\_context

)

P182

......

documents = SimpleDirectoryReader(

input\_files=["../../data/graph.txt"],

).load\_data()

#指定知识图谱的存储，这里使用内存存储

property\_graph\_store = SimplePropertyGraphStore()

storage\_context = StorageContext.from\_defaults(property\_graph\_store=property\_graph\_store)

#构造知识图谱索引（这里进行了本地化存储）

if not os.path.exists(f"./storage/graph\_store"):

index = PropertyGraphIndex.from\_documents(

documents,

storage\_context=storage\_context

)

index.storage\_context.persist(persist\_dir="./storage/graph\_store")

else:

print('Loading graph index...')

index = load\_index\_from\_storage(

StorageContext.from\_defaults(persist\_dir="./storage/graph\_store")

)

#构造查询引擎

query\_engine = index.as\_query\_engine(

include\_text=True, similarity\_top\_k=2

)

response = query\_engine.query(

"介绍一下西京的城市信息吧",

)

print(f"Response: {response}")

P184

#查看构造的知识图谱索引（这里打印了保存的三元组）

graph = index.property\_graph\_store.graph

pprint.pprint(graph.triplets)

P186

......省略构造Document对象......

SentenceSplitter = SentenceSplitter(chunk\_size=200,chunk\_overlap=0)

nodes = SentenceSplitter.get\_nodes\_from\_documents(documents)

#构造树索引

index = TreeIndex(nodes,num\_children=2)

#打印索引结构

print\_attrs(index.index\_struct)

P188

......构造Document对象......

nodes = SentenceSplitter.get\_nodes\_from\_documents(documents)

#构造关键词表索引，用大模型智能提取内容关键词

index = KeywordTableIndex(nodes)

#打印索引的内部结构（自定义方法）

print\_attrs(index.index\_struct)

#测试

query\_engine = index.as\_query\_engine()

response = query\_engine.query(

"文心一言的主要应用场景有哪些？",

)

print(f"Response: {response}")

P191

......

retriever = vector\_index.as\_retriever(similarity\_top\_k=1)

nodes = retriever.retrieve('文心一言的应用场景')

pprint.pprint(nodes)

query\_engine = index.as\_query\_engine()

P192

......

#构造向量索引检索器

retriever = VectorIndexRetriever(

index=vector\_index

... #其他参数

)

#构造摘要索引检索器

retriever = SummaryIndexLLMRetriever(

index=summary\_index,

choice\_batch\_size=5,

)

P193

retriever = treeindex.as\_retriever(

retriever\_mode="root",

)

from llama\_index.core.indices.tree.all\_leaf\_retriever import TreeAllLeafRetriever

#TreeRootRetriever是检索模式Root对应的类型

retriever = TreeRootRetriever(index = treeindex)

P196

object\_index = ObjectIndex.from\_objects(

objs, index\_cls=VectorStoreIndex,storage\_context=storage\_context

)

P197

......

#构造两个子检索器

synonym\_retriever = LLMSynonymRetriever(

index.property\_graph\_store,

llm=llm,

include\_text=False,

output\_parsing\_fn=parse\_fn,

max\_keywords=10,

synonym\_prompt=prompt,

path\_depth=1,

)

vector\_retriever = VectorContextRetriever(

index.property\_graph\_store,

include\_text=False,

similarity\_top\_k=2,

path\_depth=1,

)

#构造一个知识图谱检索器

retriever = PGRetriever(sub\_retrievers=[synonym\_retriever,vector\_retriever])

#也可以直接在查询引擎中指定子检索器

query\_engine = index.as\_query\_engine(

include\_text=True,

similarity\_top\_k=1,sub\_retrievers=[synonym\_retriever,vector\_retriever]

)

P200

......

#构造一个响应生成器

response\_synthesizer = get\_response\_synthesizer(

response\_mode=ResponseMode.COMPACT

)

#测试：调用响应生成器生成结果

response = response\_synthesizer.synthesize(

"你的输入问题",

nodes=nodes

)

#后面使用：在调用as\_query\_engine方法时指定

query\_engine = vector\_index.as\_query\_engine(

response\_synthesizer=response\_synthesizer

)

#后面使用：或者在直接构造查询引擎时指定

query\_engine = RetrieverQueryEngine(

retriever=retriever,

response\_synthesizer=response\_synthesizer

)

......

#使用隐式构造方法自动构造检索器与响应生成器

query\_engine = vector\_index.as\_query\_engine(streaming=True,

verbose=True,

response\_mode=ResponseMode.COMPACT)

P202

......

#此处使用内置的LlamaDebugHandler处理器进行跟踪（也可以使用Langfuse平台跟踪）

llama\_debug = LlamaDebugHandler(print\_trace\_on\_end=True)

callback\_manager = CallbackManager([llama\_debug])

Settings.callback\_manager = callback\_manager

#构造refine响应生成器

response\_synthesizer = get\_response\_synthesizer(

response\_mode="refine")

#模拟检索出的3个Node

nodes = [NodeWithScore(node=Node(text="小麦手机是一款专为满足现代生活需求而设计的智能手机。它的设计简洁大方，线条流畅，给人一种优雅的感觉"), score=1.0),

NodeWithScore(node=Node(text="小麦手机采用了最新的处理器技术，运行速度快，性能稳定，无论是玩游戏、看电影还是处理工作，都能轻松应对"), score=1.0),

NodeWithScore(node=Node(text="小麦手机还配备了高清大屏，色彩鲜艳，画面清晰，无论是阅读、浏览网页还是观看视频，都能带来极佳的视觉体验"), score=1.0)

]

#把问题和Node交给响应生成器响应生成

response = response\_synthesizer.synthesize(

"介绍一下小麦手机的优点，用中文回答",

nodes=nodes

)

print(response)

P203

def print\_events\_llm():

events = llama\_debug.get\_event\_pairs('llm')

#发生了多少次大模型调用

print(f'Number of LLM calls: {len(events)}')

#依次打印所有大模型调用的消息

for i,event in enumerate(events):

print(f'\n=========LLM call {i+1} messages===========')

pprint.pprint(event[1].payload["messages"])

print(f'\n=========LLM call {i+1} response============')

pprint.pprint(event[1].payload["response"].message.content)

print\_events\_llm()

P205

response\_synthesizer = get\_response\_synthesizer(response\_mode="compact")

get\_response\_synthesizer(response\_mode="compact")

print\_events\_llm()

P207

......

#读取文档

reader = SimpleDirectoryReader(

input\_files=["../../data/AI-survey-cn.pdf"]

)

docs = reader.load\_data()

#分割成Node

splitter = TokenTextSplitter(

chunk\_size=500,

chunk\_overlap=0,

separator="\n",

)

nodes = splitter.get\_nodes\_from\_documents(docs)

#模拟检索出的多个Node，注意不能直接用上面的Node

node\_scores = [NodeWithScore(node=node, score=1.0) for node in nodes]

#调用响应生成器，输入问题与模拟检索出的Node

response\_synthesizer = get\_response\_synthesizer(response\_mode="tree\_summarize")

response = response\_synthesizer.synthesize(

"请使用中文，文中介绍了AI Agent哪些方面的内容",

nodes=node\_scores)

print(response)

P210

#这里给Prompt模板增加一个language\_name参数

qa\_prompt\_tmpl = (

"根据以下上下文信息：\n"

"---------------------\n"

"{context\_str}\n"

"---------------------\n"

"使用{language\_name}回答以下问题\n "

"问题: {query\_str}\n"

"答案: "

)

qa\_prompt = PromptTemplate(qa\_prompt\_tmpl)

response\_synthesizer = get\_response\_synthesizer(

response\_mode="tree\_summarize",

streaming=True,

summary\_template=qa\_prompt)

......

#响应生成时，传入language\_name参数

streaming\_response = response\_synthesizer.synthesize(

"介绍一下小麦手机的优点",

nodes=nodes,

language\_name="法语"

)

P211

for text in streaming\_response.response\_gen:

#自行处理每个text变量的输出

......

class Phone(BaseModel):

name: str

description: str

features: List[str]

response\_synthesizer = get\_response\_synthesizer(response\_mode="tree\_summarize",

summary\_template=qa\_prompt,

output\_cls=Phone)

......

streaming\_response = response\_synthesizer.synthesize(

"介绍一下小麦手机",

nodes=nodes,

language\_name="英文"

)

print(streaming\_response)

P212

......

class FunnySynthesizer(BaseSynthesizer):

my\_prompt\_tmpl = (

"根据以下上下文信息：\n"

"---------------------\n"

"{context\_str}\n"

"---------------------\n"

"使用中文且幽默风趣的风格回答以下问题\n "

"问题: {query\_str}\n"

"答案: "

)

def \_\_init\_\_(

self,

llm: Optional[LLMPredictorType] = None,

) -> None:

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

llm=llm

)

self.\_input\_prompt = PromptTemplate(FunnySynthesizer.my\_prompt\_tmpl)

#必须实现的接口

def \_get\_prompts(self) -> PromptDictType:

pass

#必须实现的接口

def \_update\_prompts(self, prompts: PromptDictType) -> None:

pass

#生成响应的接口

def get\_response(

self,

query\_str: str,

text\_chunks: Sequence[str],

\*\*response\_kwargs: Any,

) -> RESPONSE\_TEXT\_TYPE:

context\_str = "\n\n".join(n for n in text\_chunks)

#此处可以自定义任何响应逻辑

response = self.\_llm.predict(

self.\_input\_prompt,

query\_str=query\_str,

context\_str=context\_str,

\*\*response\_kwargs,

)

return response

#响应接口的异步版本

async def aget\_response(

self,

query\_str: str,

text\_chunks: Sequence[str],

\*\*response\_kwargs: Any,

) -> RESPONSE\_TEXT\_TYPE:

context\_str = "\n\n".join(n for n in text\_chunks)

response = await self.\_llm.apredict(

self.\_input\_prompt,

query\_str=query\_str,

context\_str=context\_str,

\*\*response\_kwargs,

)

return response

#使用自定义的响应生成器

response\_synthesizer = FunnySynthesizer(llm=llm)

......

P215

......

#用向量索引构造查询引擎

query\_engine = vector\_index.as\_query\_engine()

response = query\_engine.query(' 客户在没有交定金之前要求出具房地产证原件，怎么办？')

print(response)

......

#query\_engine

query\_engine = vector\_index.as\_query\_engine(streaming=True)

response = query\_engine.query(' 客户在没有交定金之前要求出具房地产证原件，怎么办？')

response.print\_response\_stream()

......

#以下代码等价于query\_engine = vector\_index.as\_query\_engine()

#构造检索器

retriever = VectorIndexRetriever(

index=vector\_index,

similarity\_top\_k=2,

)

#构造响应生成器

response\_synthesizer = get\_response\_synthesizer(

streaming = True #如果需要使用流式输出

)

#组合构造查询引擎

query\_engine = RetrieverQueryEngine(

retriever=retriever,

response\_synthesizer=response\_synthesizer,

)

......

P217

......索引类中的as\_query\_engine......

def as\_query\_engine(

self, llm: Optional[LLMType] = None, \*\*kwargs: Any

) -> BaseQueryEngine:

# NOTE: lazy import

from llama\_index.core.query\_engine.retriever\_query\_engine import (

RetrieverQueryEngine,

)

retriever = self.as\_retriever(\*\*kwargs)

llm = (

resolve\_llm(llm, callback\_manager=self.\_callback\_manager)

if llm

else llm\_from\_settings\_or\_context(Settings, self.service\_context)

)

return RetrieverQueryEngine.from\_args(

retriever,

llm=llm,

\*\*kwargs,

)

P219

class MyQueryEngine(CustomQueryEngine):

response\_synthesizer: BaseSynthesizer = \

Field(default=None, description="response\_synthesizer")

retriever: BaseRetriever = \

Field(default=None, description="retriever")

def \_\_init\_\_(self, retriever: BaseRetriever, response\_synthesizer: BaseSynthesizer):

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

self.retriever = retriever

self.response\_synthesizer = response\_synthesizer

#实现必需的custom\_query接口

def custom\_query(self, query\_str: str):

nodes = self.retriever.retrieve(query\_str)

response = self.response\_synthesizer.synthesize(query\_str,nodes)

return response

P220

......先构造vector\_index对象......

retriever = vector\_index.as\_retriever(similarity\_top\_k=3)

synthesizer = get\_response\_synthesizer(llm=llm,streaming=True)

#构造自定义的查询引擎

my\_query\_engine = MyQueryEngine(retriever,synthesizer )

response = my\_query\_engine.query('你的问题')

......

qa\_prompt = PromptTemplate(

"根据以下上下文回答输入问题：\n"

"---------------------\n"

"{context\_str}\n"

"---------------------\n"

"回答以下问题，不要编造\n"

"我的问题: {query\_str}\n"

"答案: "

)

class MyLLMQueryEngine(CustomQueryEngine):

#此处直接使用大模型组件，而不是响应生成器

llm: Ollama = Field(default=None, description="llm")

retriever: BaseRetriever = Field(default=None, description="retriever")

def \_\_init\_\_(self, retriever: BaseRetriever, llm: Ollama):

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

self.retriever = retriever

self.llm = llm

def custom\_query(self, query\_str: str):

nodes = self.retriever.retrieve(query\_str)

#用检索出的Node构造上下文

context\_str = "\n\n".join([n.node.get\_content() for n in nodes])

#用上下文与查询问题组装Prompt,然后调用大模型组件响应生成

response = self.llm.complete(

qa\_prompt.format(context\_str=context\_str, query\_str=query\_str)

)

return str(response)

P222

chat\_engine = vector\_index.as\_chat\_engine(chat\_mode="condense\_question")

print(chat\_engine.chat('文心一言是什么？'))

chat\_engine.reset()

chat\_engine.chat\_repl()

......

custom\_prompt = PromptTemplate(

"""\

请根据以下的历史对话记录和新的输入问题，重写一个新的问题，使其能够捕捉对话中的所有相关上下文。

<Chat History>

{chat\_history}

<Follow Up Message>

{question}

<Standalone question>

"""

)

#历史对话记录

custom\_chat\_history = [

ChatMessage(

role=MessageRole.USER,

content="我们来讨论关于文心一言的一些问题吧",

),

ChatMessage(role=MessageRole.ASSISTANT, content="好的"),

]

#先构造查询引擎, 这里省略了构造vector\_index对象

query\_engine = vector\_index.as\_query\_engine()

#再构造对话引擎

chat\_engine = CondenseQuestionChatEngine.from\_defaults(

query\_engine=query\_engine, #对话引擎基于查询引擎构造

condense\_question\_prompt=custom\_prompt, #设置重写问题的Prompt模板

chat\_history=custom\_chat\_history, #携带历史对话记录

verbose=True,

)

chat\_engine.chat\_repl()

P224

......

def as\_chat\_engine(

self,

chat\_mode: ChatMode = ChatMode.BEST,

llm: Optional[LLMType] = None,

\*\*kwargs: Any,

) -> BaseChatEngine:

......

#先构造查询引擎

query\_engine = self.as\_query\_engine(llm=llm, \*\*kwargs)

#再构造对话引擎

if chat\_mode in [ChatMode.REACT, ChatMode.OPENAI, ChatMode.BEST]:

......

query\_engine\_tool = \

QueryEngineTool.from\_defaults(query\_engine=query\_engine)

return AgentRunner.from\_llm(

tools=[query\_engine\_tool],

llm=llm,

\*\*kwargs,

)

if chat\_mode == ChatMode.CONDENSE\_QUESTION:

return CondenseQuestionChatEngine.from\_defaults(

query\_engine=query\_engine,

llm=llm,

\*\*kwargs,

)

elif chat\_mode == ChatMode.CONTEXT:

return ContextChatEngine.from\_defaults(

retriever=self.as\_retriever(\*\*kwargs),

llm=llm,

\*\*kwargs,

)

elif chat\_mode == ChatMode.CONDENSE\_PLUS\_CONTEXT:

return CondensePlusContextChatEngine.from\_defaults(

retriever=self.as\_retriever(\*\*kwargs),

llm=llm,

\*\*kwargs,

)

elif chat\_mode == ChatMode.SIMPLE:

return SimpleChatEngine.from\_defaults(

llm=llm,

\*\*kwargs,

)

else:

raise ValueError(f"Unknown chat mode: {chat\_mode}")

......

P227

......

llm = Ollama(model='qwen:14b')

Settings.llm=llm

chat\_engine = SimpleChatEngine.from\_defaults()

chat\_engine.chat\_repl()

P228

......

#构造查询引擎

query\_engine = vector\_index.as\_query\_engine()

#构造对话引擎

chat\_engine = CondenseQuestionChatEngine.from\_defaults(

query\_engine=query\_engine,

condense\_question\_prompt=custom\_prompt,

chat\_history=custom\_chat\_history,

verbose=True,

)

......

chat\_engine = CondenseQuestionChatEngine.from\_defaults(

query\_engine=vector\_index.as\_query\_engine(),

verbose=True,

)

P231

......

chat\_engine = CondenseQuestionChatEngine.from\_defaults(

query\_engine=vector\_index.as\_query\_engine(response\_mode="refine"),

verbose=True,

)

......先准备vector\_index对象......

#也可以修改为chat\_engine=vector\_index.as\_chat\_engine(chat\_mode=”context”)

chat\_engine = ContextChatEngine.from\_defaults(

retriever=vector\_index.as\_retriever(),

llm=llm

)

chat\_engine.chat\_repl()

P233

chat\_engine = CondensePlusContextChatEngine.from\_defaults(

retriever=vector\_index.as\_retriever(similarity\_top\_k=1),

llm=llm

)

P235

chat\_engine = vector\_index.as\_chat\_engine(chat\_mode="react")

chat\_engine.chat\_repl()

......

#构造查询引擎

query\_engine = vector\_index.as\_query\_engine()

#把查询引擎"工具化"

query\_engine\_tool = QueryEngineTool.from\_defaults( query\_engine=query\_engine,

name="query\_engine",

description="用于查询文心一言的相关信息")

#将工具传入，开发一个Agent

chat\_engine =ReActAgent.from\_tools(

tools=[query\_engine\_tool]

)

chat\_engine.chat\_repl()

P239

......准备数据与索引......

class Phone(BaseModel):

""" Information & features of a phone."""

cpu: str

memory: str

storage: str

screen: str

query\_engine = index.as\_query\_engine( llm = Ollama(model='llama3:8b'),

response\_mode="tree\_summarize",

output\_cls=Phone)

response = query\_engine.query("小麦手机的主要参数是什么？")

......

P240

......

#定义响应的格式

response\_schemas = [

ResponseSchema(

name="name",

description="手机名称",

),

ResponseSchema(

name="cpu",

description="手机处理器",

),

ResponseSchema(

name="memory",

description="手机内存",

),

ResponseSchema(

name="features",

description="手机特性",

type="list",

),

]

#构造LangChain框架的输出解析器

lc\_output\_parser =\

StructuredOutputParser.from\_response\_schemas(response\_schemas)

output\_parser = LangchainOutputParser(lc\_output\_parser)

#设置大模型使用构造的输出解析器

llm = OpenAI(output\_parser=output\_parser)

#查询

query\_engine = index.as\_query\_engine(llm=llm,verbose=True)

response = query\_engine.query("小麦手机的主要参数是什么、其特性如何？")

print(response)

from llama\_index.core.prompts.default\_prompts

import DEFAULT\_TEXT\_QA\_PROMPT\_TMPL

print(output\_parser.format(DEFAULT\_TEXT\_QA\_PROMPT\_TMPL))

P246

from llama\_index.core import PromptTemplate

from llama\_index.llms.openai import OpenAI

prompt\_rewrite\_temp = """\

您是一个聪明的查询生成器。请生成与以下查询相关的{num\_queries}个查询问题 \n

注意每个查询问题都占一行 \n

我的查询：{query}

生成查询列表：

"""

prompt\_rewrite = PromptTemplate(prompt\_rewrite\_temp)

llm = OpenAI(model="gpt-3.5-turbo")

#查询转换的方法

def rewrite\_query(query: str, num: int = 3):

response = llm.predict(

prompt\_rewrite, num\_queries=num, query=query

)

# 假设大模型将每个查询问题都放在一行上

queries = response.split("\n")

return queries

print(rewrite\_query("中国目前大模型的发展情况如何？"))

P247

from llama\_index.core.indices.query.query\_transform import HyDEQueryTransform

from llama\_index.llms.openai import OpenAI

from llama\_index.core import PromptTemplate

#修改成中文Prompt

hyde\_prompt\_temp = """\

请生成一段文字来回答输入问题\n

尽可能含有更多的关键细节\n

{context\_str}

生成内容：

"""

hyde\_prompt = PromptTemplate(hyde\_prompt\_temp)

llm = OpenAI(model="gpt-3.5-turbo")

hyde = HyDEQueryTransform(llm=llm)

hyde.update\_prompts({'hyde\_prompt':hyde\_prompt})

query\_bundle = hyde.run("请介绍小麦手机的主要配置")

print(query\_bundle.\_\_dict\_\_)

......这里假设已经构造了一个城市信息查询引擎......

query\_engine = create\_city\_engine('南京市') #城市信息查询引擎

hyde\_query\_engine = TransformQueryEngine(query\_engine, hyde)

print('\nQuerying the city engine...')

response = query\_engine.query('南京市的人口是多少？经济发展如何？')

pprint\_response(response,show\_source=True)

print('\nQuerying the HyDE city engine...')

response\_hyde = hyde\_query\_engine.query("南京市的人口是多少？经济发展如何？")

pprint\_response(response\_hyde ,show\_source=True)

P250

......构造一个简单的城市信息查询引擎，代码略......

query\_engine = create\_city\_engine(['北京市','上海市'])

#转换Prompt，此处用于更新默认的Prompt

prompt\_templ = """

我们有机会从知识源中回答部分或全部问题。知识源的上下文如下，提供了之前的推理步骤。

根据上下文和之前的推理，返回一个可以从上下文中回答的问题：

1. 这个问题可以帮助回答原问题，与原问题密切相关。

2. 可以是原问题的子问题，或者是解答原问题需要的一个步骤中需要的问题。

如果无法从上下文中提取更多信息，则提供“无”作为答案。下面给出了一个示例：

-----

问题：2020年澳大利亚网球公开赛冠军获得了多少个大满贯冠军？

知识源上下文：提供了2020年澳大利亚网球公开赛冠军的名字

之前的推理：无

新问题：谁是2020年澳大利亚网球公开赛的冠军？

-----

我的问题：{query\_str}

知识源上下文：{context\_str}

之前的推理：{prev\_reasoning}

新问题：

"""

#查询转换器

step\_transformer = StepDecomposeQueryTransform(llm=llm\_openai, verbose=True)

#转换Prompt

new\_prompt = PromptTemplate(prompt\_templ)

step\_transformer.update\_prompts({'step\_decompose\_query\_prompt':new\_prompt})

#带有查询转换器的查询引擎

step\_query\_engine = MultiStepQueryEngine(query\_engine=query\_engine,

query\_transform=step\_transformer,index\_summary='这是一个关于城市的知识库，用于回答与城市信息相关的问题')

print('\nQuerying the stepcompose city engine...')

response = step\_query\_engine.query("中国首都的城市人口有多少？和上海相比呢？")

pprint\_response(response,show\_source=True)

P252

from llama\_index.question\_gen.openai import OpenAIQuestionGenerator

from llama\_index.llms.openai import OpenAI

from llama\_index.core import PromptTemplate,QueryBundle

from llama\_index.core.tools import ToolMetadata

import pprint

llm = OpenAI()

question\_gen\_prompt\_templ = """

你可以访问多个工具，每个工具都代表一个不同的数据源或API。

每个工具都有一个名称和一个描述字段，格式为JSON字典。

字典的键(key)是工具的名称，值(value)是描述。

你的目的是通过生成一系列可以由这些工具回答的子问题来帮助回答一个复杂的用户问题。

在完成任务时，请考虑以下准则：

• 尽可能具体

• 子问题应与用户问题相关

• 子问题应可通过提供的工具回答

• 你可以为每个工具都生成多个子问题

• 工具必须用它们的名称而不是描述来指定

• 如果你认为不相关，就不需要使用工具

通过调用SubQuestionList函数输出子问题列表。

## Tools

```json

{tools\_str}

```

## User Question

{query\_str}

"""

#rewriter

question\_rewriter = OpenAIQuestionGenerator.from\_defaults(llm=llm)

#转换Prompt

new\_prompt = PromptTemplate(question\_gen\_prompt\_templ)

question\_rewriter.update\_prompts({'question\_gen\_prompt':new\_prompt})

#可用的工具，注意这里只是提供工具的元数据，并未真正提供工具

tool\_choices = [

ToolMetadata(

name="query\_tool\_beijing",

description=(

"用于查询北京市各个方面的信息，如基本信息、旅游指南、城市历史等"

),

),

ToolMetadata(

name="query\_tool\_shanghai",

description=(

"用于查询上海市各个方面的信息，如基本信息、旅游指南、城市历史等"

),

),

]

print('-------------------------')

query\_str = "北京与上海的人口差距是多少？它们的面积相差多少？"

#使用generate方法生成子问题

choices = question\_rewriter.generate(

tool\_choices,

QueryBundle(query\_str=query\_str))

pprint.pprint(choices)

......

from llama\_index.core.query\_engine import SubQuestionQueryEngine

......

......省略create\_city\_engine方法......

#构造两个城市信息查询引擎

query\_engine\_nanjing = create\_city\_engine('南京市')

query\_engine\_shanghai = create\_city\_engine('上海市')

#查询引擎作为工具

query\_engine\_tools = [

QueryEngineTool(

query\_engine=query\_engine\_nanjing,

metadata=ToolMetadata(

name="query\_tool\_nanjing",

description="用于查询南京市各个方面的信息，如基本信息、旅游指南、城市历史等"

),

),

QueryEngineTool(

query\_engine=query\_engine\_shanghai,

metadata=ToolMetadata(

name="query\_tool\_shanghai",

description="用于查询上海市各个方面的信息，如基本信息、旅游指南、城市历史等"

),

),

]

#构造子问题查询引擎

query\_engine = SubQuestionQueryEngine.from\_defaults(

query\_engine\_tools=query\_engine\_tools,

use\_async=True,

)

#查询

response = query\_engine.query(

"北京与上海的人口差距是多少？GDP大约相差多少？使用中文回答"

)

print(response)

P258

......先检索出Node，假设保存在nodes\_with\_scores变量中

processor = SimilarityPostprocessor(similarity\_cutoff=0.8)

filtered\_nodes = processor.postprocess\_nodes(nodes\_with\_scores)

......将filtered\_nodes变量用于响应生成

......

vector\_index = VectorStoreIndex(nodes)

query\_engine = vector\_index.as\_query\_engine(

node\_postprocessors=[

SimilarityPostprocessor(similarity\_cutoff=0.5)

]

)

......

P259

......

class MyNodePostprocessor(BaseNodePostprocessor):

def \_postprocess\_nodes(

self, nodes: List[NodeWithScore], query\_bundle: Optional[QueryBundle]

) -> List[NodeWithScore]:

pattern = r"过滤正则表达式"

filtered\_nodes = []

for node in nodes:

if not re.search(pattern, node.text):

filtered\_nodes.append(node)

nodes = filtered\_nodes

return nodes

query\_engine = vector\_index.as\_query\_engine(

node\_postprocessors=[

MyNodePostprocessor()

]

)

P260

......构造关键词过滤处理器......

processor = KeywordNodePostprocessor(required\_keywords=["小麦手机"],

exclude\_keywords=[],

lang='zh-Hans')

filtered\_nodes = processor.postprocess\_nodes(nodes\_with\_scores)

......

P261

......

docs = [Document(text="小麦手机是小麦公司最新出的第十代手机产品。\

采用了中国最先进的国产红旗CPU芯片。\

采用了6.95寸的OLED显示屏幕与5000毫安的电池容量。")]

#解析与分割文档

node\_parser = SentenceWindowNodeParser.from\_defaults(

sentence\_splitter=my\_chunking\_tokenizer\_fn,

window\_size=3,

window\_metadata\_key="window",

original\_text\_metadata\_key="original\_text",

)

nodes = node\_parser.get\_nodes\_from\_documents(docs)

vector\_index = VectorStoreIndex(nodes=nodes)

P262

#此时不指定节点后处理器

query\_engine = vector\_index.as\_query\_engine(

similarity\_top\_k=1

)

window\_response = query\_engine.query(

"小麦手机是哪个公司出品的，采用什么芯片？"

)

pprint\_response(window\_response,show\_source=True)

query\_engine = vector\_index.as\_query\_engine(

similarity\_top\_k=1,

node\_postprocessors=[

MetadataReplacementPostProcessor ( target\_metadata\_key = "window" )

],

)

P263

......

query\_engine = vector\_index.as\_query\_engine(

similarity\_top\_k=3,

node\_postprocessors=[

FixedRecencyPostprocessor ( top\_k=1, date\_key="create\_time" )

],

)

P265

......

docs = SimpleDirectoryReader(input\_files=["../../data/yiyan.txt"]).load\_data()

nodes = SentenceSplitter(chunk\_size=100,chunk\_overlap=0).get\_nodes\_from\_documents(docs)

vector\_index = VectorStoreIndex(nodes)

retriever =vector\_index.as\_retriever(similarity\_top\_k=5)

#直接检索出结果

nodes = retriever.retrieve("百度文心一言的逻辑推理能力怎么样？")

print('================before rerank================')

print\_nodes(nodes)

#使用Cohere Rerank模型重排序结果

cohere\_rerank = CohereRerank(model='rerank-multilingual-v3.0',api\_key='\*\*\*', top\_n=2)

rerank\_nodes = cohere\_rerank.postprocess\_nodes(nodes,query\_str='百度文心一言的逻辑推理能力怎么样？')

print('================after rerank================')

print\_nodes(rerank\_nodes)

P267

> model=BAAI/bge-reranker-large

> text-embeddings-router --model-id $model --port 8080

P268

import requests

from typing import List, Optional

from llama\_index.core.bridge.pydantic import Field, PrivateAttr

from llama\_index.core.postprocessor.types import BaseNodePostprocessor

from llama\_index.core.schema import NodeWithScore, QueryBundle

class BgeRerank(BaseNodePostprocessor):

url: str = Field(description="Rerank server url.")

top\_n: int = Field(description="Top N nodes to return.")

def \_\_init\_\_(self,top\_n: int,url: str):

super().\_\_init\_\_(url=url, top\_n=top\_n)

#调用TEI的Rerank模型服务

def rerank(self, query, texts):

url = f"{self.url}/rerank"

request\_body = {"query": query, "texts": texts, "truncate": False}

response = requests.post(url, json=request\_body)

if response.status\_code != 200:

raise RuntimeError(f"Failed to rerank, detail: {response}")

return response.json()

@classmethod

def class\_name(cls) -> str:

return "BgeRerank"

#实现Rank节点后处理器的接口

def \_postprocess\_nodes(

self,

nodes: List[NodeWithScore],

query\_bundle: Optional[QueryBundle] = None,

) -> List[NodeWithScore]:

if query\_bundle is None:

raise ValueError("Missing query bundle in extra info.")

if len(nodes) == 0:

return []

#调用Rerank模型

texts = [node.text for node in nodes]

results = self.rerank(

query=query\_bundle.query\_str,

texts=texts,

)

#组装并返回Node

new\_nodes = []

for result in results[0 : self.top\_n]:

new\_node\_with\_score = NodeWithScore(

node=nodes[int(result["index"])].node,

score=result["score"],

)

new\_nodes.append(new\_node\_with\_score)

return new\_nodes

P269

......

#构造自定义的节点后处理器

customRerank = BgeRerank(url="http://localhost:8080",top\_n=2)

#测试处理Node

rerank\_nodes = customRerank.postprocess\_nodes(nodes,query\_str='百度文心一言的逻辑推理能力怎么样?')

......

query\_engine = vector\_index.as\_query\_engine(

similarity\_top\_k=3,

node\_postprocessors=[customRerank],

)

P271

query\_engine = RouterQueryEngine(

　　selector = LLMSingleSelector.from\_defaults();

　　query\_engine\_tools = [......多个tool......]

}

P272

......

docs\_xiaomai = SimpleDirectoryReader(input\_files=[".../.../data/xiaomai.txt"]).load\_data()

docs\_yiyan = SimpleDirectoryReader(input\_files=[".../.../data/yiyan.txt"]).load\_data()

vectorindex\_xiaomai = VectorStoreIndex.from\_documents(docs\_xiaomai)

query\_engine\_xiaomai = vectorindex\_xiaomai.as\_query\_engine()

vectorindex\_yiyan = VectorStoreIndex.from\_documents(docs\_yiyan)

query\_engine\_yiyan = vectorindex\_yiyan.as\_query\_engine()

......

#构造第一个工具

tool\_xiaomai = QueryEngineTool.from\_defaults(

query\_engine=query\_engine\_xiaomai,

description="用于查询小麦手机的信息",

)

#构造第二个工具

tool\_yiyan = QueryEngineTool.from\_defaults(

query\_engine=query\_engine\_yiyan,

description="用于查询文心一言的信息",

)

#构造路由模块

query\_engine = RouterQueryEngine(

selector=LLMSingleSelector.from\_defaults(), #选择器

query\_engine\_tools=[ #候选工具

tool\_xiaomai,tool\_yiyan

]

)

#像使用查询引擎一样使用即可

response = query\_engine.query("什么是文心一言，用中文回答")

pprint\_response(response,show\_source=True)

P273

......

#针对同一个索引构造不同的响应类型的查询引擎

query\_engine\_quesiton =\

vectorindex\_xiaomai.as\_query\_engine(response\_mode="compact")

query\_engine\_summary =\

vectorindex\_xiaomai.as\_query\_engine(response\_mode="simple\_summarize")

#“工具化”查询引擎

tool\_question = QueryEngineTool.from\_defaults(

query\_engine=query\_engine\_quesiton,

description="用于回答事实性与细节性的问题",

)

tool\_summarize = QueryEngineTool.from\_defaults(

query\_engine=query\_engine\_summary,

description="用于回答总结性的问题",

)

#构造带有路由功能的查询引擎

query\_engine = RouterQueryEngine(

selector=LLMSingleSelector.from\_defaults(),

query\_engine\_tools=[

tool\_question,tool\_summarize

],verbose=True

)

......

P274

......

vector\_index = VectorStoreIndex(nodes)

retriever\_xiaomai = vector\_index.as\_retriever()

vector\_index2 = VectorStoreIndex(nodes2)

retriever\_yiyan = vector\_index2.as\_retriever()

tool\_xiaomai = RetrieverTool.from\_defaults(

retriever=retriever\_xiaomai,

description="用于查询小麦手机的信息",

)

tool\_yiyan = RetrieverTool.from\_defaults(

retriever=retriever\_yiyan,

description="用于查询文心一言的信息",

)

#构造带有路由功能的检索器

retriever = RouterRetriever(

selector=LLMSingleSelector.from\_defaults(),

retriever\_tools=[

tool\_xiaomai,tool\_yiyan

]

)

nodes = retriever.retrieve("什么是文心一言？")

print\_nodes(nodes)

P275

......

choices = [

"choice 1: 通过互联网查询当前实时的信息",

"choice 2: 通过大模型查询非实时信息或者创作内容",

]

choices = [

ToolMetadata(description="查询当前实时的信息"", name="web\_search"),

ToolMetadata(description="知识查询或内容创作", name="query\_engine")

]

......

choices = [

ToolMetadata(description="查询当前实时的信息", name="web\_search"),

ToolMetadata(description="知识查询或内容创作", name="query\_engine")

]

selector = LLMSingleSelector.from\_defaults()

selector\_result = selector.select(

choices, query="写一个悬疑小故事?"

)

print(selector\_result.selections)

P276

......

summary\_index =\

SummaryIndex.from\_documents(docs,chunk\_size=100,chunk\_overlap=0)

vector\_index =\

VectorStoreIndex.from\_documents(docs,chunk\_size=100,chunk\_overlap=0)

keyword\_index =\

SimpleKeywordTableIndex.from\_documents(docs,chunk\_size=100,chunk\_overlap=0)

#构造3个可用工具

summary\_tool = QueryEngineTool.from\_defaults(

query\_engine=summary\_index.as\_query\_engine(response\_mode="tree\_summarize",),

description=(

"有助于总结与小麦手机相关的问题"

),

)

vector\_tool = QueryEngineTool.from\_defaults(

query\_engine=vector\_index.as\_query\_engine(),

description=(

"适合检索与小麦手机相关的特定上下文"

),

)

keyword\_tool = QueryEngineTool.from\_defaults(

query\_engine=keyword\_index.as\_query\_engine(),

description=(

"适合使用关键词从文章中检索特定的上下文"

),

)

#构造可多选的路由查询引擎

query\_engine = RouterQueryEngine(

selector=LLMMultiSelector.from\_defaults(),

query\_engine\_tools=[

summary\_tool,vector\_tool,keyword\_tool

],verbose=True

)

response = query\_engine.query("小麦手机的屏幕特点和优势是什么")

pprint\_response(response,show\_source=True)

P278

response = query\_engine.query("小麦手机的处理器是什么？")

P279

......

from sqlalchemy import (

create\_engine,

MetaData,

Table,

Column,

String,

Integer,

select,

text

)

from sqlalchemy.orm import sessionmaker

#构造SQL查询引擎

engine =\

create\_engine("postgresql://postgres:\*\*\*\*@localhost:5432/postgres")

#构造SQLDatabase对象

sql\_database = SQLDatabase(engine, include\_tables=["customers","orders"])

from llama\_index.core.query\_engine import NLSQLTableQueryEngine

#构造SQLTable查询引擎：sql\_database、tables、llm参数

llm\_openai = OpenAI(model='gpt-3.5-turbo')

query\_engine = NLSQLTableQueryEngine(

sql\_database=sql\_database,

tables=["customers","orders"],

llm=llm\_openai

)

#测试

response = query\_engine.query("一共有多少个订单")

print(response)

P282

......

engine =\

create\_engine("postgresql://postgres:Unycp123!!@localhost:5432/postgres")

metadata\_obj = MetaData()

sql\_database = SQLDatabase(engine, include\_tables=["customers","orders"])

from llama\_index.core.indices.struct\_store.sql\_query import (

SQLTableRetrieverQueryEngine,

)

from llama\_index.core.objects import (

SQLTableNodeMapping,

SQLTableSchema,

ObjectIndex,

)

from llama\_index.core import VectorStoreIndex

#构造用于检索SQLTableSchema对象的对象索引

#table\_node\_mapping变量用于给SQLTableSchema对象与向量存储索引的Node做映射

table\_node\_mapping = SQLTableNodeMapping(sql\_database)

table\_schema\_objs = [

SQLTableSchema(table\_name="customers"),

SQLTableSchema(table\_name="orders"),

SQLTableSchema(table\_name="mystore")

]

#构造一个检索的对象索引，底层通过向量存储索引来实现语义检索

obj\_index = ObjectIndex.from\_objects(

table\_schema\_objs,

table\_node\_mapping,

VectorStoreIndex,

)

#传入retriever方法，而不是直接传入多个SQLTableSchema对象

#此处为了演示效果，设置similarity\_top\_k=1

query\_engine = SQLTableRetrieverQueryEngine(

sql\_database, obj\_index.as\_retriever(similarity\_top\_k=1)

)

response = query\_engine.query("所有订单总金额是多少？")

print(response)

P283

......

table\_retriever = obj\_index.as\_retriever(similarity\_top\_k=1)

tables = table\_retriever.retrieve("所有订单总金额是多少")

print(tables)

......

P284

......

#一个检索器，类似于使用index.as\_retriever方法生成的检索器

nl\_sql\_retriever = NLSQLRetriever(

sql\_database, tables=["customers","orders"], return\_raw=True

)

#直接构造RetrieverQueryEngine查询引擎

query\_engine = RetrieverQueryEngine.from\_args(nl\_sql\_retriever)

response = query\_engine.query(

"所有订单总金额是多少？"

)

print(response)

......

P288

......

#模型

llm =OpenAI()

embedded\_model =\

OllamaEmbedding(model\_name="milkey/dmeta-embedding-zh:f16", )

Settings.llm=llm

Settings.embed\_model=embedded\_model

#向量存储

chroma = chromadb.HttpClient(host="localhost", port=8000)

......

documents = \

LlamaParse(result\_type="markdown",language='ch\_sim').load\_data("../../data/zte-report-simple.pdf")

print(f'{len(documents)} documents loaded.\n')

#打印并观察输出的Document对象结构

pprint.pprint(documents[0].\_\_dict\_\_)

P289

......

parser = LlamaParse(result\_type="markdown",language='ch\_sim')

#把parser对象作为简单目录阅读器加载时的一个自定义的文档阅读器

documents = \

SimpleDirectoryReader("./data", file\_extractor={".pdf":parser}).load\_data()

P290

#分割Node，这里使用最简单的数据分割器

node\_parser = SimpleNodeParser()

nodes = node\_parser.get\_nodes\_from\_documents(documents)

#嵌入与索引（用Node构造）

collection = chroma.get\_or\_create\_collection(name="llamaparse\_simple")

vector\_store = ChromaVectorStore(chroma\_collection=collection)

storage\_context = StorageContext.from\_defaults(vector\_store=vector\_store)

index = VectorStoreIndex(nodes=nodes,storage\_context=storage\_context)

#构造查询引擎

query\_engine = index.as\_query\_engine(similarity\_top=10,verbose=True)

P293

......省略借助LlamaParse解析PDF文档为Document对象的过程......

#此处更改表格描述的Prompt模板

DEFAULT\_SUMMARY\_QUERY\_STR = """\

请用中文简要介绍表格内容。\

这个表格是关于什么的？给出一个非常简洁的摘要（想象你正在为这个表格添加一个新的标题和摘要），\

如果提供了上下文，那么请输出真实/现有的表格标题/说明。\

如果提供了上下文，那么请输出真实/现有的表格ID。\

还要输出表格是否应该保留的信息。\

"""

node\_parser = MarkdownElementNodeParser(summary\_query\_str=DEFAULT\_SUMMARY\_QUERY\_STR)

nodes = node\_parser.get\_nodes\_from\_documents(documents)

#分离不同的文本Node（TextNode）与索引Node（IndexNode）

base\_nodes, objects = node\_parser.get\_nodes\_and\_objects(nodes)

......此处省略构造storage\_context变量的过程......

index = VectorStoreIndex(

nodes= base\_nodes + objects,

storage\_context=storage\_context

)

query\_engine = index.as\_query\_engine(similarity\_top=10,verbose=True)

P296

from llama\_index.multi\_modal\_llms.dashscope import (

DashScopeMultiModal,

DashScopeMultiModalModels,

)

from llama\_index.multi\_modal\_llms.dashscope.utils import (

create\_dashscope\_multi\_modal\_chat\_message,

load\_local\_images

)

from llama\_index.core.base.llms.types import MessageRole

from llama\_index.core.multi\_modal\_llms.generic\_utils import load\_image\_urls

import pprint

import os

#替换成自己的阿里巴巴API Key

os.environ["DAHSCOPE\_API\_KEY"] = "sk-\*\*\*"

#加载图片

image\_documents1 = \

load\_image\_urls(["https://dashsco\*\*.oss-cn-beijing.aliyuncs.com/images/dog\_and\_girl.jpeg"])

image\_documents2 = \

load\_local\_images(["file:///Users/pingcy/本地开发/rag/data/xiaomi.png"])

#多模态大模型

dashscope\_multi\_modal\_llm = \

DashScopeMultiModal(model\_name=DashScopeMultiModalModels.QWEN\_VL\_PLUS)

#调用

chat\_message = create\_dashscope\_multi\_modal\_chat\_message(

"请概括这两张图片中的信息",

MessageRole.USER,

image\_documents1 + image\_documents2

)

chat\_response = dashscope\_multi\_modal\_llm.chat([chat\_message])

#打印结果

print(chat\_response.message.content[0]["text"])

P298

......

#多模态大模型

dashscope\_multi\_modal\_llm = \

DashScopeMultiModal(model\_name=DashScopeMultiModalModels.QWEN\_VL\_PLUS)]

#输入图片

image\_documents = \

SimpleDirectoryReader(input\_files=["../../data/xiaomi.png"]).load\_data()

#LlamaIndex框架的部分版本在此处存在漏洞，特殊处理

for doc in image\_documents:

doc.image\_url = doc.metadata["file\_path"]

#定义输出的对象

from pydantic import BaseModel

class Phone(BaseModel):

"""定义对象结构"""

name: str

cpu: str

battery: str

display: str

from llama\_index.core.program import MultiModalLLMCompletionProgram

from llama\_index.core.output\_parsers import PydanticOutputParser

#Prompt模板

prompt\_template\_str = """\

{query\_str}

请把结果作为一个Pydantic对象返回，对象格式如下:

"""

#构造MultiModalLLMCompletionProgram对象

mm\_program = MultiModalLLMCompletionProgram.from\_defaults(

output\_parser=PydanticOutputParser(Phone), #将对象类型传给输出解析器

image\_documents=image\_documents, #输入图片

prompt\_template\_str=prompt\_template\_str, #Prompt模板

multi\_modal\_llm=dashscope\_multi\_modal\_llm, #多模态大模型

verbose=True,

)

#测试

response = mm\_program(query\_str="请描述图片中的信息。")

pprint.pprint(response.\_\_dict\_\_)

P300

import chromadb

import pprint

import os

from chromadb.utils.embedding\_functions import OpenCLIPEmbeddingFunction

from chromadb.utils.data\_loaders import ImageLoader

# Chroma向量库的多模态嵌入函数与图片加载器

embedding\_function = OpenCLIPEmbeddingFunction()

image\_loader = ImageLoader()

# Chroma向量库的客户端，注意构造collection库时的区别

chroma\_client = chromadb.HttpClient(host="localhost", port=8000)

chroma\_client.delete\_collection("multimodal\_collection")

chroma\_collection = chroma\_client.get\_or\_create\_collection(

"multimodal\_collection",

embedding\_function=embedding\_function,

data\_loader=image\_loader,

)

#需要嵌入的图片列表

image\_uris = sorted([os.path.join('./jpgs/', image\_name) \

for image\_name in os.listdir('./jpgs/') ])

ids = [str(i) for i in range(len(image\_uris))]

#直接存储到Chroma向量库中，由Chroma向量库完成图片嵌入

chroma\_collection.add(ids=ids, uris=image\_uris)

retrieved = chroma\_collection.query(query\_texts=["很多辆汽车"],

include=['data'], n\_results=2)

print(retrieved['uris'])

P301

#将PDF文档解析成json对象与图片对象

def load\_docs():

parser = LlamaParse(language='ch\_sim',verbose=True)

json\_objs = parser.get\_json\_result("../../data/xiaomi14.pdf")

json\_list = json\_objs[0]["pages"]

print(f'{len(json\_list)} documents loaded.\n')

image\_list = parser.get\_images(json\_objs, download\_path="pdf\_images")

print(f'{len(image\_list)} images loaded.\n')

return json\_list,image\_list

P303

def get\_text\_nodes(json\_list: List[dict]):

text\_nodes = []

for idx, page in enumerate(json\_list):

text\_node = \

TextNode(text=page["text"], metadata={"page": page["page"]})

text\_nodes.append(text\_node)

return text\_nodes

#定义图片转换为文本的方法（使用Qwen-VL模型，参考8.5.4节的内容）

def get\_text\_of\_image(image\_path):

mm\_llm = \

DashScopeMultiModal(model\_name=DashScopeMultiModalModels.QWEN\_VL\_PLUS)

image = load\_local\_images(["file://./" + image\_path])

#调用多模态大模型

chat\_message\_local = create\_dashscope\_multi\_modal\_chat\_message(

"请详细描述图片中的信息，包括图片中的文字和图像。",

MessageRole.USER,

image

)

chat\_response = mm\_llm.chat([chat\_message\_local])

return chat\_response.message.content[0]["text"]

#生成图片对应的TextNode

def get\_image\_text\_nodes(image\_list: List[dict]):

img\_text\_nodes = []

for idx,image in enumerate(image\_list):

response = ''

#使用多模态大模型理解图片并生成文本

response = get\_text\_of\_image(image["path"])

text\_node = \

TextNode(text=str(response), metadata={"path": image["path"]})

img\_text\_nodes.append(text\_node)

return img\_text\_nodes

P304

def create\_engine():

#调用一次上面的方法解析文档、生成文本Node、生成图片Node

(json\_list,image\_list) = load\_docs()

text\_nodes = get\_text\_nodes(json\_list)

img\_text\_nodes = get\_image\_text\_nodes(image\_list)

#定义向量存储

collection = chroma.get\_or\_create\_collection(name="llamaparse\_mm")

vector\_store = ChromaVectorStore(chroma\_collection=collection)

storage\_context = StorageContext.from\_defaults(vector\_store=vector\_store)

#用文本Node和图片Node生成索引

index = VectorStoreIndex(

nodes=text\_nodes + img\_text\_nodes,

storage\_context=storage\_context

)

#构造查询引擎

query\_engine = index.as\_query\_engine(similarity\_top=5,verbose=True)

return query\_engine

P305

delete\_collection()

query\_engine = create\_engine()

while True:

query = input("\n输入你的问题 (or 'q' to quit): ")

if query == 'q':

break

if query == "":

continue

response = query\_engine.query(query)

print(response)

P309

......

prompt\_str = "请为产品设计一句简单的宣传语，我的产品是{product\_name}"

prompt\_tmpl = PromptTemplate(prompt\_str)

llm = OpenAI()

p = QueryPipeline(chain=[prompt\_tmpl, llm], verbose=True)

......

#加载文档，构造向量存储索引

docs = SimpleDirectoryReader(input\_files=["../../data/xiaomai.txt"]).load\_data()

index = VectorStoreIndex.from\_documents(docs)

P310

......

#准备组件

input = InputComponent()

llm = Ollama(model='qwen:14b')

prompt\_tmpl = PromptTemplate("对问题进行完善，输出新的问题：{query\_str}" )

retriever = index.as\_retriever(similarity\_top\_k=3)

summarizer = get\_response\_synthesizer(response\_mode="tree\_summarize")

#构造一个查询管道

p = QueryPipeline(verbose=True)

#把上面构造的模块添加进来

p.add\_modules(

{

"input": input,

"prompt": prompt\_tmpl,

"llm":llm,

"retriever": retriever,

"summarizer": summarizer,

}

)

#连接这些模块

p.add\_link("input", "prompt")

p.add\_link('prompt',"llm")

p.add\_link('llm','retriever')

p.add\_link("retriever", "summarizer", dest\_key="nodes")

p.add\_link("llm", "summarizer", dest\_key="query\_str")

output = p.run(input='小麦手机的优势是什么')

P314

from llama\_index.core.query\_pipeline import CustomQueryComponent

from pydantic import Field,BaseModel

from llama\_index.core.program import LLMTextCompletionProgram

from typing import List,Optional,Dict, Any

#自定义查询组件，可以将其插入查询管道中

class MyOutputParser(CustomQueryComponent):

#输入校验，该方法可以不实现

def \_validate\_component\_inputs(

self, input: Dict[str, Any]

) -> Dict[str, Any]:

"""校验组件的输入参数"""

return input

@property

def \_input\_keys(self) -> set:

"""定义组件的输入keys"""

return {"response"}

@property

def \_output\_keys(self) -> set:

"""定义组件的输出keys"""

return {"output"}

def \_run\_component(self, \*\*kwargs) -> Dict[str, Any]:

"""定义组件的运行逻辑"""

#在这个例子中，我们要求把查询的手机信息结构化成Phone类型的

class Phone(BaseModel):

name: str

cpu: str

memory: str

storage: str

screen: str

battery: str

features: List[str]

#给大模型提示

prompt\_template\_str = """\

根据以下内容提取结构化信息{input}\

"""

#构造一个大模型的调用模块，指定output\_cls参数

program = LLMTextCompletionProgram.from\_defaults(

output\_cls=Phone,

prompt\_template\_str=prompt\_template\_str,

verbose=True,

)

#调用program变量，注意从输入中获取key=response的内容

output = program(input=kwargs['response'])

#返回对象中必须包含\_output\_keys接口中定义的输出关键词

return {"output": output}

P315

......

p = QueryPipeline(verbose=True)

p.add\_modules(

{

"input": input,

"prompt": prompt\_tmpl,

"llm":llm,

"retriever": retriever,

"summarizer": summarizer,

'output\_parser': MyOutputParser() #增加一个自定义的模块

}

)

p.add\_link("input", "prompt")

p.add\_link('prompt',"llm")

p.add\_link('llm','retriever')

p.add\_link("retriever", "summarizer", dest\_key="nodes")

p.add\_link("llm", "summarizer", dest\_key="query\_str")

#增加与自定义的模块的连接，输入的参数为response

p.add\_link("summarizer", "output\_parser", dest\_key="response")

output = p.run(input='小麦手机的优势是什么')

print(output)

P317

from llama\_index.core.query\_pipeline import FnComponent

#定义一个组件函数

def addExtaInfo(phone: Phone) -> str:

phone\_info = f"Name: {phone.name}\nCPU: {phone.cpu}\nMemory: {phone.memory}\nStorage: {phone.storage}\nScreen: {phone.screen}\nBattery: {phone.battery}\nFeatures: {', '.join(phone.features)}"

extra\_info = "Extra information: This phone has a great camera."

phone\_str = f"{phone\_info}\n{extra\_info}"

return phone\_str

......

p = QueryPipeline(verbose=True)

p.add\_modules(

{

......

'output\_parser': MyOutputParser(),

#使用FnComponent方法添加一个新的查询组件

'post\_processor': FnComponent(fn=addExtaInfo, output\_key="output")

}

)

......

#增加连接

p.add\_link("output\_parser", "post\_processor", dest\_key="phone")

output = p.run(input='小麦手机的优势是什么')

P323

from llama\_index.core.tools import FunctionTool

def add(a: int, b: int) -> int:

return a + b

# Create a tool from the function

tool\_add = FunctionTool.from\_defaults(

fn=add,

name="tool\_add",

description="用于两个整数相加",

)

output = tool\_add.call(a=1,b=3)

print(type(output))

print(f"Output: {output.\_\_dict\_\_}")

<class 'llama\_index.core.tools.types.ToolOutput'>

Output: {'content': '4', 'tool\_name': 'add', 'raw\_input': {'args': (), 'kwargs': {'a': 1, 'b': 3}}, 'raw\_output': 4, 'is\_error': False}

P324

......准备索引......

#构造查询引擎

query\_engine = \

index.as\_query\_engine(response\_mode="compact",verbose=True,text\_qa\_template=qa\_prompt)

from llama\_index.core.tools import QueryEngineTool, ToolMetadata

#构造查询引擎工具

tool\_xiaomai = QueryEngineTool.from\_defaults(

query\_engine=query\_engine,

name="tool\_xiaomai",

description="用于小麦手机信息查询",

return\_direct=False

)

#测试工具

print(tool\_xiaomai.call(query\_str="小麦手机采用了什么型号的CPU？"))

P325

......

vector\_index = VectorStoreIndex(nodes)

#首先构造一个检索器

retriever\_xiaomai = vector\_index.as\_retriever(similarity\_K=2)

#用检索器构造一个检索工具

tool\_retriever\_xiaomai = RetrieverTool.from\_defaults(

retriever=retriever\_xiaomai,

description="用于检索小麦手机的信息",

)

print(tool\_retriever\_xiaomai.call(query\_str="小麦手机采用了什么型号的CPU？"))

......

P326

......省略数据加载......

query\_xiaomai = index1.as\_query\_engine(response\_mode="compact")

query\_ultra = index2.as\_query\_engine(response\_mode="compact")

#构造两个子工具：简单的查询引擎工具

query\_tool\_xiaomai = QueryEngineTool.from\_defaults(

query\_engine=query\_xiaomai,

name="query\_tool\_xiaomai",

description="提供小麦手机普通型号Pro/Max的信息")

query\_tool\_ultra = QueryEngineTool.from\_defaults(

query\_engine=query\_ultra,

name="query\_tool\_ultra",

description="提供小麦手机Ultra的信息")

from llama\_index.core.tools import QueryPlanTool

from llama\_index.core import get\_response\_synthesizer

from llama\_index.core.tools.query\_plan import QueryPlan, QueryNode

#构造一个查询计划工具，并传入子工具与响应生成器

response\_synthesizer = get\_response\_synthesizer()

query\_plan\_tool = QueryPlanTool.from\_defaults(

query\_engine\_tools=[query\_tool\_xiaomai, query\_tool\_ultra],

response\_synthesizer=response\_synthesizer,

)

#构造一个执行计划，执行计划由多个执行Node组成

nodes=[

QueryNode(

id=1,

query\_str="查询小麦手机普通型号Pro的信息",

tool\_name="query\_tool\_xiaomai",

dependencies=[]

),

QueryNode(

id=2,

query\_str="查询小麦手机Ultra的信息",

tool\_name="query\_tool\_ultra",

dependencies=[1]

),

QueryNode(

id=3,

query\_str="对比小麦手机普通型号Pro与小麦手机Ultrl的配置区别",

tool\_name="vs\_tool",

dependencies=[1,2]

)

]

#调用查询计划工具，并传入执行计划

output = query\_plan\_tool(nodes=nodes)

print(output)

P328

......

def \_baidu\_reader(

soup: Any, url: str, include\_url\_in\_text: bool = True

) -> Tuple[str, Dict[str, Any]]:

......

#此处省略对Web网页解析的逻辑，可参考第5章

return text, {"title": soup.find(class\_="post\_\_title").get\_text()}

#构造一个数据加载器

web\_loader =\

BeautifulSoupWebReader(website\_extractor={"cloud.baidu.com":\_baidu\_reader})

#用数据加载器构造一个按需加载工具

tool\_xiaomai = OnDemandLoaderTool.from\_defaults(

web\_loader,

name="tool\_xiaomai",

description="用于查询本地文档中的小麦手机信息",

)

#调用这个工具测试，由于web\_loader对象也需要参数，因此需要增加urls参数

output = tool\_xiaomai.call(

urls=["https://cloud.bai\*\*.com/doc/AppBuilder/s/6lq7s8lli"],

query\_str='百度云千帆appbuilder是什么？')

print(output)

P330

#工具：搜索天气情况

def search\_weather(query: str) -> str:

"""用于搜索天气情况"""

# Perform search logic here

search\_results = f"明天晴转多云，最高温度30℃，最低温度23℃。天气炎热，注意防晒哦。"

return search\_results

tool\_search = FunctionTool.from\_defaults(fn=search\_weather)

#工具：发送电子邮件

def send\_email(subject: str, recipient: str, message: str) -> None:

"""用于发送电子邮件"""

# Send email logic here

print(f"邮件已发送至 {recipient}，主题为 {subject}，内容为 {message}")

tool\_send\_mail = FunctionTool.from\_defaults(fn=send\_email)

#工具：查询客户信息

def query\_customer(phone: str) -> str:

"""用于查询客户信息"""

# Perform creation logic here

result = f"该客户信息为:\n姓名: 张三\n电话: {phone}\n地址: 北京市海淀区"

return result

tool\_generate = FunctionTool.from\_defaults(fn=query\_customer)

P331

......

#定义一个OpenAI的Agent

class MyOpenAIAgent:

#初始化参数

#tools: Sequence[BaseTool] = []，工具列表

#llm: OpenAI = OpenAI(temperature=0, model="gpt-3.5-turbo")，OpenAI的大模型

#chat\_history: List[ChatMessage] = []，聊天历史

def \_\_init\_\_(

self,

tools: Sequence[BaseTool] = [],

llm: OpenAI = OpenAI(temperature=0, model="gpt-3.5-turbo"),

chat\_history: List[ChatMessage] = [],

) -> None:

self.\_llm = llm

self.\_tools = {tool.metadata.name: tool for tool in tools}

self.\_chat\_history = chat\_history

#重置聊天历史

def reset(self) -> None:

self.\_chat\_history = []

#定义聊天接口

def chat(self, message: str) -> str:

chat\_history = self.\_chat\_history

chat\_history.append(ChatMessage(role="user", content=message))

#传入工具

tools = [

tool.metadata.to\_openai\_tool() for \_, tool in self.\_tools.items()

]

ai\_message = self.\_llm.chat(chat\_history, tools=tools).message

additional\_kwargs = ai\_message.additional\_kwargs

chat\_history.append(ai\_message)

#获取工具调用的要求

tool\_calls = additional\_kwargs.get("tool\_calls", None)

#如果调用工具，那么依次调用

if tool\_calls is not None:

for tool\_call in tool\_calls:

#调用函数

function\_message = self.\_call\_function(tool\_call)

chat\_history.append(function\_message)

#继续对话

ai\_message = self.\_llm.chat(chat\_history).message

chat\_history.append(ai\_message)

return ai\_message.content

#调用函数

def \_call\_function(

self, tool\_call: ChatCompletionMessageToolCall

) -> ChatMessage:

id\_ = tool\_call.id

function\_call = tool\_call.function

tool = self.\_tools[function\_call.name]

output = tool(\*\*json.loads(function\_call.arguments))

return ChatMessage(

name=function\_call.name,

content=str(output),

role="tool",

additional\_kwargs={

"tool\_call\_id": id\_,

"name": function\_call.name,

},

)

P333

while True:

user\_input = input("请输入您的消息：")

if user\_input.lower() == "quit":

break

response = agent.chat(user\_input)

print(response)

P334

......定义工具，同上省略......

from llama\_index.agent.openai import OpenAIAgent

from llama\_index.llms.openai import OpenAI

llm = OpenAI(model="gpt-3.5-turbo")

agent = OpenAIAgent.from\_tools(

[tool\_search, tool\_send\_mail, tool\_generate], llm=llm, verbose=True

)

P335

......

from llama\_index.core.agent import ReActAgent

from llama\_index.llms.openai import OpenAI

llm = OpenAI(model="gpt-3.5-turbo")

agent = ReActAgent.from\_tools(

[tool\_search, tool\_send\_mail, tool\_generate], llm=llm, verbose=True

)

......

......

您可以使用以下工具：

{tool\_desc}

要回答问题，请使用以下格式。

``

思考：我需要使用一个工具来帮助我回答这个问题。

行动：工具名称（{tool\_names} 之一）

行动输入：工具的输入，采用表示 kwargs 的 JSON 格式（例如 {{"text": "hello world", "num\_beams": 5}}）

`` `

请对操作输入有效的 JSON 格式。不要这样做 {{'text': 'hello world', 'num\_beams': 5}}。

如果使用此格式，您将收到以下格式的响应：

``

观察：工具响应

``

......

P337

......

from llama\_index.core.agent import AgentRunner

from llama\_index.agent.openai import OpenAIAgentWorker

llm = OpenAI(model="gpt-3.5-turbo")

openai\_step\_engine = OpenAIAgentWorker.from\_tools(

tools,

llm=llm,

verbose=True)

agent = AgentRunner(openai\_step\_engine)

......

P338

......准备3个工具：搜索天气情况、发送电子邮件、查询客户信息，省略......

tools = [tool\_search,tool\_send\_mail,tool\_customer]

#构造对象索引，在底层使用向量检索

obj\_index = ObjectIndex.from\_objects(

tools,

index\_cls=VectorStoreIndex,

)

#大模型

llm = OpenAI(model="gpt-3.5-turbo")

#开发Agent，注意提供tool\_retriever(工具检索器)，而不是工具集

agent = OpenAIAgent.from\_tools(

tool\_retriever=obj\_index.as\_retriever(similarity\_top\_k=2),

verbose=True

)

#测试：使用agent\_worker对象检索工具集，查看结果

tools = agent.agent\_worker.get\_tools('发送电子邮件')

for tool in tools:

print(f'Tool name: {tool.metadata.name}')

print(tools)

P339

Tool name: send\_email

Tool name: query\_customer

Tool name: search\_weather

Tool name: query\_customer

agent.chat('帮我查询1865120××××的客户信息')

P340

......

#对上下文建立索引，这里的上下文是一些对财务术语缩写的解释

texts = [

"Abbreviation: X = Revenue",

"Abbreviation: YZ = Risk Factors",

"Abbreviation: Z = Costs",

]

docs = [Document(text=t) for t in texts]

#一个上下文索引

context\_index = VectorStoreIndex.from\_documents(docs)

#开发一个带有上下文检索功能的Agent

context\_agent = ContextRetrieverOpenAIAgent.from\_tools\_and\_retriever(

query\_engine\_tools, #正常的工具列表

context\_index.as\_retriever(similarity\_top\_k=1),#传入上下文检索器

verbose=True)

response = context\_agent.chat("What is the YZ of March 2022?")

P342

......省略构造工具的代码......

tools = [tool\_search,tool\_send\_mail,tool\_customer]

agent = OpenAIAgent.from\_tools(tools, llm=llm, verbose=True)

#构造一个任务

task = agent.create\_task("明天南京天气如何？")

#运行这个任务

print('\n--------------')

step\_output = agent.run\_step(task.task\_id)

pprint.pprint(step\_output.\_\_dict\_\_)

#循环，直到is\_last = True

while not step\_output.is\_last:

print('\n--------------')

step\_output = agent.run\_step(task.task\_id)

pprint.pprint(step\_output.\_\_dict\_\_)

#最后输出结果

print('\nFinal response:')

response = agent.finalize\_response(task.task\_id)

print(str(response))

P343

......

steps = agent.get\_completed\_steps(task.task\_id)

for i,step in enumerate(steps):

print(f'\nStep {i+1}:')

print(step.\_\_dict\_\_)

P344

......

citys\_dict = {

'北京市':'beijing',

'南京市':'nanjing',

'广州市':'guangzhou',

'上海市':'shanghai',

'深圳市':'shenzhen'

}

# 开发城市信息查询工具

def create\_city\_tool(name:str):

......根据城市名构造对应的查询引擎,并将其包装成查询引擎工具......

#先构造一组工具，再开发一个Agent，也可以直接开发Agent

query\_engine\_tools = []

for city in citys\_dict.keys():

query\_engine\_tools.append(create\_city\_tool(city))

#开发Agent

openai\_step\_engine = OpenAIAgentWorker.from\_tools(

query\_engine\_tools,verbose=True

)

agent = AgentRunner(openai\_step\_engine)

#分步运行Agent：并要求人类给出指令

task\_message = None

while task\_message != "exit":

task\_message = input(">> 你: ")

if task\_message == "exit":

break

#根据输入问题构造任务

task = agent.create\_task(task\_message)

response = None

step\_output = None

message = None

#任务执行过程中允许人类反馈信息

#如果message为exit，那么任务被取消，并退出

#如果任务步骤返回is\_last=True，那么任务正常退出

while message != "exit" and (not step\_output or not step\_output.is\_last):

#执行下一步任务

if message is None or message == "":

step\_output = agent.run\_step(task.task\_id)

else:

#允许把人类反馈信息传入中间的任务步骤

step\_output = agent.run\_step(task.task\_id, input=message)

#如果任务没结束，那么允许用户输入

if not step\_output.is\_last:

message = input(">> 请补充任务反馈信息（留空继续，exit退出）: ")

#任务正常退出

if step\_output.is\_last:

print(">> 任务运行完成。")

response = agent.finalize\_response(task.task\_id)

print(f"Final Answer: {str(response)}")

#任务被取消

elif not step\_output.is\_last:

print(">> 任务未完成，被丢弃。")

P350

......

#读取文档，构造Node，用于生成检索评估数据集

documents = SimpleDirectoryReader(

input\_files=["../../data/citys/南京市.txt"]).load\_data()

node\_parser = SentenceSplitter(chunk\_size=1024)

nodes = node\_parser.get\_nodes\_from\_documents(documents)

for idx, node in enumerate(nodes):

node.id\_ = f"node\_{idx}"

#准备一个检索器，后面使用

vector\_index = VectorStoreIndex(nodes)

retriever = vector\_index.as\_retriever(similarity\_top\_k=2)

from llama\_index.core.evaluation import (

generate\_question\_context\_pairs,

EmbeddingQAFinetuneDataset,

)

QA\_GENERATE\_PROMPT\_TMPL = """

以下是上下文:

---------------------

{context\_str}

---------------------

你是一位专业教授。你的任务是基于以上的上下文，为即将到来的考试设置 {num\_questions\_per\_chunk} 个问题。

这些问题必须基于提供的上下文生成，并确保上下文能够回答这些问题。确保每一行都只有一个独立的问题。不要有多余解释。不要给问题编号。"

"""

print("Generating question-context pairs...")

qa\_dataset = generate\_question\_context\_pairs(

nodes,

llm=llm\_ollama,

num\_questions\_per\_chunk=1,

qa\_generate\_prompt\_tmpl=QA\_GENERATE\_PROMPT\_TMPL

)

print("Saving dataset...")

qa\_dataset.save\_json("retriever\_eval\_dataset.json")

P351

......

print("Loading dataset...")

qa\_dataset = \

EmbeddingQAFinetuneDataset.from\_json("retriever\_eval\_dataset.json")

eval\_querys = list(qa\_dataset.queries.items())

for eval\_id,eval\_query in eval\_querys[:10]:

print(f"Query: {eval\_query}")

P352

......

print("Loading dataset...")

#从保存的JSON文档中加载检索评估数据集

qa\_dataset = \

EmbeddingQAFinetuneDataset.from\_json("retriever\_eval\_dataset.json")

querys = list(qa\_dataset.queries.items())

#构造一个检索评估器，设定两个评估指标

from llama\_index.core.evaluation import RetrieverEvaluator

metrics = ["mrr", "hit\_rate"]

retriever\_evaluator = RetrieverEvaluator.from\_metric\_names(

metrics, retriever=retriever

)

#简单评估前10个评估用例

for eval\_id,eval\_query in eval\_querys[:10]:

expect\_docs = qa\_dataset.relevant\_docs[eval\_id]

print(f"Query: {eval\_query}, Expected docs: {expect\_docs}")

#评估，输入评估问题与预期检索出的Node

eval\_result = \

retriever\_evaluator.evaluate(query=eval\_query,expected\_ids=expect\_docs)

print(eval\_result)

P354

eval\_results = retriever\_evaluator.evaluate\_dataset(qa\_dataset)

......

# build documents

docs =SimpleDirectoryReader(input\_files = ['../../data/citys/南京市.txt']).load\_data()

# define generator, generate questions

dataset\_generator = RagDatasetGenerator.from\_documents(

documents=docs,

llm=llm\_ollama,

num\_questions\_per\_chunk=1, # 设置每个Node都生成的问题数量

show\_progress=True,

question\_gen\_query="您是一位老师。您的任务是为即将到来的考试设置{num\_questions\_per\_chunk}个问题。这些问题必须基于提供的上下文生成，并确保上下文能够回答这些问题。确保每一行都只有一个独立的问题。不要有多余解释。不要给问题编号。"

)

#以下代码只需要运行一次

print('Generating questions from nodes...\n')

rag\_dataset = dataset\_generator.generate\_dataset\_from\_nodes()

rag\_dataset.save\_json('./rag\_eval\_dataset.json')

#从本地文档中加载并查看

print('Loading dataset...\n')

rag\_dataset = LabelledRagDataset.from\_json('./rag\_eval\_dataset.json')

for example in rag\_dataset.examples:

print(f'query: {example.query}')

print(f'answer: {example.reference\_answer}')

P356

......这里省略构造查询引擎的过程......

query\_engine = \_create\_doc\_engine('Nanjing')

#两个重要的输入参数

query = "南京的气候怎么样？"

response = query\_engine.query(query)

# 评估忠实度的评估器

evaluator = FaithfulnessEvaluator()

eval\_result = evaluator.evaluate\_response(query=query, response=response)

print(f'faithfulness score: {eval\_result.score}\n')

# 评估相关性的评估器（综合了上下文相关性与答案相关性）

evaluator = RelevancyEvaluator()

eval\_result = evaluator.evaluate\_response(query=query,response=response)

print(f'relevancy score: {eval\_result.score}\n')

# 评估上下文相关性的评估器

evaluator = ContextRelevancyEvaluator()

eval\_result = evaluator.evaluate\_response(query=query,response=response)

print(f'context relevancy score: {eval\_result.score}\n')

# 评估答案相关性的评估器

evaluator = AnswerRelevancyEvaluator()

eval\_result = evaluator.evaluate\_response(query=query,response=response)

print(f'answer relevancy score: {eval\_result.score}\n')

# 评估正确性的评估器，注意输入了reference

evaluator = CorrectnessEvaluator()

eval\_result = evaluator.evaluate\_response(query=query,response=response,

reference='南京的气候属于较典型的北亚热带季风气候。这里四季分明，冬夏温差较大，年平均气温为16.4℃，最冷月（1月）平均气温为3.1℃，最热月（7月）平均气温为28.4℃。南京降水丰富，年平均降水量约为1144毫米，且全年有大约112.9天的降雨日。冬季受西伯利亚高压或蒙古高压控制，盛行东北风；夏季则分为初夏多雨的梅雨季节和盛夏的伏旱天气两段。')

print(f'correctness score: {eval\_result.score}\n')

# 评估答案与标准答案的语义相似度（基于embedding）的评估器，注意输入了reference

evaluator = SemanticSimilarityEvaluator()

eval\_result = evaluator.evaluate\_response(query=query,response=response,

reference='南京四季分明，冬夏温差较大，冬季受西伯利亚高压或蒙古高压控制，盛行东北风；夏季则分为初夏多雨的梅雨季节和盛夏的伏旱天气两段。')

print(f'semantic similarity score: {eval\_result.score}\n')

P358

......这里省略构造查询引擎的过程......

query\_engine = \_create\_doc\_engine('Nanjing')

# 构造多个响应评估器

faithfulness\_evaluator= FaithfulnessEvaluator()

relevancy\_evaluator = RelevancyEvaluator()

correctness\_evaluator = CorrectnessEvaluator()

similartiy\_evaluator = SemanticSimilarityEvaluator()

# 加载数据集

rag\_dataset = LabelledRagDataset.from\_json('./rag\_eval\_dataset.json')

from llama\_index.core.evaluation import BatchEvalRunner

import asyncio

#构造一个批量评估器

runner = BatchEvalRunner(

{"faithfulness": faithfulness\_evaluator,

"relevancy": relevancy\_evaluator,

"correctness": correctness\_evaluator,

"similarity": similartiy\_evaluator},

workers = 4

)

#为了提高性能，采用异步并行的评估方法，调用批量评估器

#输入：查询引擎、批量的query，批量的reference

#这里对响应评估数据集中的前十个评估用例进行评估

async def evaluate\_queries():

eval\_results = await runner.aevaluate\_queries(

query\_engine,

queries=[example.query for example in rag\_dataset.examples][:10],

reference=[example.reference\_answer for example in rag\_dataset.examples][:10],

)

return eval\_results

eval\_results = asyncio.run(evaluate\_queries())

#打印评估结果

import pandas as pd

def display\_results(eval\_results):

data = {}

for key, results in eval\_results.items():

scores = [result.score for result in results]

scores.append(sum(scores) / len(scores))

data[key] = scores

data["query"] = [result.query for result in eval\_results["faithfulness"]]

data["query"].append("【Average】")

df = pd.DataFrame(data)

print(df)

display\_results(eval\_results)

P360

GUIDELINES = [

"答案应该完全回答了输入问题。",

"答案应该避免模糊或含糊不清的用词。",

"答案应该在可能时使用明确的统计数据或数字。"

]

evaluators = [

GuidelineEvaluator(guidelines=guideline,

eval\_template=myprompts.MY\_GUILD\_EVAL\_TEMPLATE)

for guideline in GUIDELINES

]

for guideline, evaluator in zip(GUIDELINES, evaluators):

eval\_result = evaluator.evaluate\_response(

query= "南京有多少人口？南京的气候怎么样？",

response=response

)

print("====================================")

print(f"Guideline: {guideline}")

print(f"Pass: {eval\_result.passing}")

print(f"Feedback: {eval\_result.feedback}")

P364

......

#设置模型(略)与准备向量库

......

vector\_store = ChromaVectorStore(chroma\_collection=collection)

#准备需要的知识文档

eval\_documents = \

SimpleDirectoryReader("../../data/MiniTruthfulQADataset/source\_files/").load\_data()

#准备用于评估的数据集

eval\_questions = \

LabelledRagDataset.from\_json("../../data/MiniTruthfulQADataset/rag\_dataset.json")

#准备评估组件

faithfulness = FaithfulnessEvaluator()

relevancy = RelevancyEvaluator()

correctness = CorrectnessEvaluator()

for chunk\_size in [128,1024,2048]:

avg\_time, avg\_faithfulness, avg\_relevancy,average\_correctness = \

evaluate\_response\_time\_and\_accuracy(chunk\_size)

print(f"Chunk size {chunk\_size} \n

Average Response time: {avg\_time:.2f}s \n

Average Faithfulness: {avg\_faithfulness:.2f}\n

Average Relevancy: {avg\_relevancy:.2f}\n

Average Correctness: {average\_correctness:.2f}")

P365  
......

def evaluate\_response\_time\_and\_accuracy(chunk\_size,num\_questions=100):

total\_response\_time = 0

total\_faithfulness = 0

total\_relevancy = 0

total\_correctness = 0

#构造查询引擎

node\_parser = SentenceSplitter(chunk\_size=chunk\_size, chunk\_overlap=0)

nodes = \

node\_parser.get\_nodes\_from\_documents(eval\_documents, show\_progress=True)

storage\_context = StorageContext.from\_defaults(vector\_store=vector\_store)

print(f"\nTotal nodes: {len(nodes)}")

vector\_index = VectorStoreIndex(nodes,storage\_context=storage\_context)

query\_engine = vector\_index.as\_query\_engine(top\_K=3)

#对指定问题集进行一次评估

for index, question in enumerate(eval\_questions.examples[:num\_questions]):

print(f"\nStart evaluating question【{index}】: {question.query}")

start\_time = time.time()

response\_vector = query\_engine.query(question.query)

#计算响应时间

elapsed\_time = time.time() - start\_time

#评估忠实度

faithfulness\_result = faithfulness.evaluate\_response(

query=question.query, response=response\_vector

).score

#评估相关性

relevancy\_result = relevancy.evaluate\_response(

query=question.query, response=response\_vector

).score

#评估正确性，此处需要提供参考答案

correctness\_result = correctness.evaluate\_response(

query=question.query,response=response\_vector,

reference = question.reference\_answer

).score

total\_response\_time += elapsed\_time

total\_faithfulness += faithfulness\_result

total\_relevancy += relevancy\_result

total\_correctness += correctness\_result

print(f"Response time: {elapsed\_time:.2f}s, Faithfulness: {faithfulness\_result}, Relevancy: {relevancy\_result}, Correctness: {correctness\_result}")

#计算平均得分

average\_response\_time = total\_response\_time / num\_questions

average\_faithfulness = total\_faithfulness / num\_questions

average\_relevancy = total\_relevancy / num\_questions

average\_correctness = total\_correctness / num\_questions

return average\_response\_time, average\_faithfulness, average\_relevancy,average\_correctness

P369

......

def read\_csv\_file(file\_path):

nodes = []

with open(file\_path, 'r') as file:

csv\_reader = csv.reader(file)

for row in csv\_reader:

question = row[0]

answer = row[1]

node = TextNode(text=question, metadata={"answer":answer})

#嵌入时不带入答案

node.excluded\_embed\_metadata\_keys = ["answer"]

node.text\_template = "{content}\n{metadata\_str}\n"

nodes.append(node)

return nodes

# Example usage:

csv\_file\_path = "../../data/questions.csv"

nodes = read\_csv\_file(csv\_file\_path)

#打印嵌入内容与大模型生成的内容

for node in nodes:

print('Embed content:')

print(node.get\_content(metadata\_mode=MetadataMode.EMBED))

print('LLM content:')

print(node.get\_content(metadata\_mode=MetadataMode.LLM))

P370

node\_postprocessors=[

MetadataReplacementPostProcessor ( target\_metadata\_key = "window" )

],

P373

......

reader = SimpleDirectoryReader(input\_files=["../../data/citys/南京市.txt"])

documents = reader.load\_data()

#使用分层Node解析器在3个不同粒度上解析

node\_parser = HierarchicalNodeParser.from\_defaults(

chunk\_sizes = [2048, 512, 128],

chunk\_overlap=0)

nodes = node\_parser.get\_nodes\_from\_documents(documents)

print(f'{len(nodes)} nodes created.\n')

#可以用以下代码查看生成的叶子Node(128)和根Node(2048)的数量

#from llama\_index.core.node\_parser import get\_leaf\_nodes, get\_root\_nodes

#leaf\_nodes = get\_leaf\_nodes(nodes)

#root\_nodes = get\_root\_nodes(nodes)

#print(f'leaf nodes: {len(leaf\_nodes)}')

#print(f'root nodes: {len(root\_nodes)}')

#使用Chroma向量库

collection = chroma.get\_or\_create\_collection(name="auto\_retrieve")

vector\_store = ChromaVectorStore(chroma\_collection=collection)

#注意此处使用文档存储组件把解析出的所有Node全部添加到docstore对象中

docstore = SimpleDocumentStore()

docstore.add\_documents(nodes)

#在叶子Node层构造向量存储索引

storage\_context = StorageContext.from\_defaults(vector\_store=vector\_store,docstore=docstore)

leaf\_index = VectorStoreIndex(

nodes=leaf\_nodes,

storage\_context=storage\_context

)

#在叶子Node层构造检索器

leaf\_retriever = leaf\_index.as\_retriever(similarity\_top\_k=1)

#在叶子Node检索器之上构造自动合并检索器

retriever = AutoMergingRetriever(leaf\_retriever,

storage\_context,

verbose=True,

simple\_ratio\_thresh = 0.1)

P374

......

leaf\_nodes = leaf\_retriever.retrieve("南京市有哪些主要的旅游景点")

print('\n---------------leaf nodes------------------\n')

print\_nodes(leaf\_nodes)

nodes = retriever.retrieve("南京市有哪些主要的旅游景点")

print('\n---------------nodes------------------\n')

print\_nodes(nodes)

P378

......

catalog\_prompt\_temp = """\

你是一个聪明的内容分类器。请把我的内容归类到以下类别之一：

-------

基本

历史

经济

文化

交通

旅游

其他

-------

我的内容是：{text}

直接输出类别，不要有多余说明。

"""

catalog\_prompt = PromptTemplate(catalog\_prompt\_temp)

#构造简单的函数生成一个元数据

def get\_catalog(text: str):

catalog = llm.predict(

catalog\_prompt, text = text

)

return catalog

#定义一个用于数据摄取的转换器，把这个元数据插入Node中

class MetadataRicher(TransformComponent):

def \_\_call\_\_(self, nodes, \*\*kwargs):

for node in nodes:

node.metadata["catalog"] = get\_catalog(node.text)

return nodes

......

......

city\_docs = SimpleDirectoryReader(input\_files=["../../data/citys/南京市.txt"]).load\_data()

#构造普通的向量索引，注意这里生成了元数据

def create\_vector\_index():

P379

collection = chroma.get\_or\_create\_collection(name=f"autoretrieve")

vector\_store = ChromaVectorStore(chroma\_collection=collection)

if not os.path.exists(f"./storage/vectorindex/autoretrieve"):

#加载数据

pipeline = IngestionPipeline(

transformations=[

SentenceSplitter(chunk\_size=500, chunk\_overlap=0),

MetadataRicher(), #插入自定义转换器

]

)

nodes = pipeline.run(documents=city\_docs)

storage\_context = StorageContext.from\_defaults(vector\_store=vector\_store)

vector\_index = VectorStoreIndex(nodes,storage\_context=storage\_context)

vector\_index.storage\_context.persist(

persist\_dir=f"./storage/vectorindex/autoretrieve")

else:

print('Loading vector index...\n')

storage\_context = StorageContext.from\_defaults(

persist\_dir=f"./storage/vectorindex/autoretrieve",

vector\_store=vector\_store)

vector\_index = load\_index\_from\_storage(storage\_context=storage\_context)

return vector\_index

index = create\_vector\_index()

retriever = index.as\_retriever()

P380

from llama\_index.core.retrievers import VectorIndexAutoRetriever

from llama\_index.core.vector\_stores.types import MetadataInfo, VectorStoreInfo

vector\_store\_info = VectorStoreInfo(

content\_info="中国城市各方面的信息与介绍",

metadata\_info=[

MetadataInfo(

name="catalog",

type="str",

description=(

"""

信息目录,只能是以下之一:基本、历史、经济、文化、交通、旅游、

其他。

"""

),

)

],

)

auto\_retriever = VectorIndexAutoRetriever(

index, vector\_store\_info=vector\_store\_info,verbose=True,similarity\_top\_k=3

)

nodes=auto\_retriever.retrieve("介绍一些关于南京经济情况的信息")

print\_nodes(nodes)

P382

......

#加载原始文档

print('Loading documents...\n')

city\_docs = SimpleDirectoryReader(input\_files=[

"../../data/citys/南京市.txt",

"../../data/citys/北京市.txt",

"../../data/citys/上海市.txt",

"../../data/citys/广州市.txt"]).load\_data()

#设置固定的index\_id，后面用于构造对应的摘要Node

for doc in city\_docs:

doc.metadata['index\_id'] = os.path.splitext(doc.metadata["file\_name"])[0]

P383

#给每个Document对象都生成一个摘要（summary），用于构造摘要Node

def generate\_docs\_summary():

#持久化存储

def save\_summary\_txt(doc\_id, summary\_txt):

summary\_dir = "./storage/summary\_txt"

if not os.path.exists(summary\_dir):

os.makedirs(summary\_dir)

summary\_file = os.path.join(summary\_dir, f"{doc\_id}.txt")

with open(summary\_file, "w") as f:

f.write(summary\_txt)

#加载

def load\_summary\_txt(doc\_id):

summary\_dir = "./storage/summary\_txt"

summary\_file = os.path.join(summary\_dir, f"{doc\_id}.txt")

if os.path.exists(summary\_file):

with open(summary\_file, "r") as f:

return f.read()

return None

#借助SummaryIndex组件生成摘要，也可以用大模型快速生成摘要

def generate\_summary\_txt(doc):

summary\_index = SummaryIndex.from\_documents([doc])

query\_engine = summary\_index.as\_query\_engine(

llm=llm\_dash,

response\_mode="tree\_summarize")

summary\_txt = query\_engine.query("请用中文生成摘要")

summary\_txt = str(summary\_txt)

save\_summary\_txt(doc.metadata["index\_id"], summary\_txt)

return summary\_txt

#给每个Document对象都生成摘要，将其保存到元数据中，且不参与嵌入和大模型输入

print('Generate document summary...\n')

for doc in city\_docs:

doc\_id = doc.metadata["index\_id"]

summary\_txt = load\_summary\_txt(doc\_id)

if summary\_txt is None:

summary\_txt = generate\_summary\_txt(doc)

#这个摘要在原始文档Node中不参与嵌入与生成

doc.metadata["summary\_text"] = summary\_txt

doc.excluded\_embed\_metadata\_keys = ["summary\_text"]

doc.excluded\_llm\_metadata\_keys = ["summary\_text"]

generate\_docs\_summary()

P384

#构造二级向量索引，针对所有Document对象，并持久化存储，避免重复构造

def create\_vector\_index():

splitter = SentenceSplitter(chunk\_size=500,chunk\_overlap=0)

nodes = splitter.get\_nodes\_from\_documents(city\_docs)

collection = chroma.get\_or\_create\_collection(name=f"details\_citys")

vector\_store = ChromaVectorStore(chroma\_collection=collection)

#构造向量索引，通过持久化存储避免重复构造

if not os.path.exists(f"./storage/vectorindex/allcitys"):

print('Creating vector index...\n')

storage\_context = StorageContext.from\_defaults(vector\_store=vector\_store)

vector\_index = VectorStoreIndex(nodes,

storage\_context=storage\_context)

vector\_index.storage\_context.persist(

persist\_dir=f"./storage/vectorindex/allcitys")

else:

print('Loading vector index...\n')

storage\_context = StorageContext.from\_defaults(

persist\_dir=f"./storage/vectorindex/allcitys",

vector\_store=vector\_store)

vector\_index = load\_index\_from\_storage(

storage\_context=storage\_context)

return vector\_index

docs\_index = create\_vector\_index()

P385

#辅助函数：针对单个doc，构造一个基于摘要的IndexNode（索引Node）

def create\_doc\_index\_node(doc):

#取出摘要内容

summary\_txt = doc.metadata["summary\_text"]

filters = MetadataFilters(

filters=[

MetadataFilter(

key="index\_id",

operator=FilterOperator.EQ, value=doc.metadata["index\_id"]

),

]

)

#构造单个文档的摘要Node，注意此处的obj是用于进行递归检索的检索器

#摘要Node为IndexNode类型的

index\_node = IndexNode(

index\_id = doc.metadata["index\_id"],

text = summary\_txt,

metadata = doc.metadata,

obj = docs\_index.as\_retriever(filters = filters)

)

return index\_node

P386

#构造摘要索引

def create\_summary\_index():

summary\_collection = chroma.get\_or\_create\_collection(

name=f"summary\_allcitys")

vector\_store = ChromaVectorStore(chroma\_collection=summary\_collection)

#所有的摘要Node

index\_nodes = []

for doc in city\_docs:

index\_node = create\_doc\_index\_node(doc)

index\_nodes.append(index\_node)

#构造基于摘要Node的索引，注意摘要Node用objects参数

print('Creating summary index (for recursive retrieve)...\n')

storage\_context =   
 StorageContext.from\_defaults(vector\_store=vector\_store)

vector\_index = VectorStoreIndex(

objects=index\_nodes,

storage\_context=storage\_context)

return vector\_index

summary\_index = create\_summary\_index()

P387

print('Creating query engine...\n')

#构造一个基于摘要索引的查询引擎

retriever = summary\_index.as\_retriever(similarity\_top\_k=1, verbose=True)

query\_engine = RetrieverQueryEngine(retriever)

print('Query executing...\n')

response = query\_engine.query('上海市的人口多少')

pprint\_response(response,show\_source=True)

P389

names = ['c-rag','self-rag','kg-rag']

files = ['../../data/c-rag.pdf','../../data/self-rag.pdf','../../data/kg-rag.pdf']

P390

......此处省略import部分与模型准备部分......

#采用Chroma向量库

chroma = chromadb.HttpClient(host="localhost", port=8000)

collection = chroma.get\_or\_create\_collection(name="agentic\_rag")

vector\_store = ChromaVectorStore(chroma\_collection=collection)

#创建针对某个文档的Tool Agent

def create\_tool\_agent(file,name):

#分割文档，生成Node对象

print(f'Starting to create tool agent for 【{name}】...\n')

docs =SimpleDirectoryReader(input\_files = [file]).load\_data()

splitter = SentenceSplitter(chunk\_size=500,chunk\_overlap=50)

nodes = spltter.get\_nodes\_from\_documents(docs)

#构造向量索引，并持久化存储

if not os.path.exists(f"./storage/{name}"):

print('Creating vector index...\n')

storage\_context = StorageContext.from\_defaults (vector\_store=vector\_store)

vector\_index = VectorStoreIndex(nodes,

storage\_context= storage\_context)

vector\_index.storage\_context.persist(persist\_dir= f"./storage/{name}")

else:

print('Loading vector index...\n')

storage\_context = StorageContext.from\_defaults(

persist\_dir=f"./storage/ {name}",

vector\_store=vector\_store)

vector\_index = load\_index\_from\_storage(

storage\_context=storage\_context)

#构造基于向量索引的查询引擎

query\_engine = vector\_index.as\_query\_engine(similarity\_top\_k=5)

# Create a summary index

summary\_index = SummaryIndex(nodes)

summary\_engine = summary\_index.as\_query\_engine(

response\_mode="tree\_summarize")

#转换为工具

query\_tool = QueryEngineTool.from\_defaults(

query\_engine=query\_engine,

name=f'query\_tool',

description=f'Use if you want to query details about {name}')

summary\_tool = QueryEngineTool.from\_defaults(

query\_engine=summary\_engine,

name=f'summary\_tool',

description=f'Use ONLY IF you want to get a holistic summary of the documents. DO NOT USE if you want to query some details about {name}.')

#创建一个Tool Agent

tool\_agent = ReActAgent.from\_tools([query\_tool,summary\_tool],

verbose=True,

system\_prompt=

f"""

You are a specialized agent designed to answer queries about {name}.You must ALWAYS use at least one of the tools provided when answering a question; DO NOT rely on prior knowledge. DO NOT fabricate answer.

"""

)

return tool\_agent

P392

#创建不同文档的Tool Agent

print('===============================================\n')

print('Creating tool agents for different documents...\n')

tool\_agents\_dict = {}

for name, file in zip(names, files):

tool\_agent = create\_tool\_agent(file, name)

tool\_agents\_dict[name] = tool\_agent

#将Tool Agent进行“工具化”

print('===============================================\n')

print('Creating tools from tool agents...\n')

all\_tools = []

for name in names:

agent\_tool = QueryEngineTool.from\_defaults(

#注意，Agent本身也是一种查询引擎，所以可以直接转换为工具

query\_engine=tool\_agents\_dict[name],

#这个工具的名称

name=f"tool\_{name.replace("-", "")}",

#描述这个工具的作用和使用方法

description=f"Use this tool if you want to answer any questions about {name}."

)

all\_tools.append(agent\_tool)

#创建Top Agent

print('Creating top agent...\n')

top\_agent = OpenAIAgent.from\_tools(tools=all\_tools,

verbose=True,

system\_prompt="""You are an agent designed to answer queries over a set of given papers.Please always use the tools provided to answer a question.Do not rely on prior knowledge.DO NOT fabricate answer""" )

P393

top\_agent.chat\_repl()

P394

#构造工具检索器

print('===============================================\n')

print('Creating tool retrieve index...\n')

obj\_index = ObjectIndex.from\_objects(

all\_tools,

index\_cls=VectorStoreIndex,

)

tool\_retriever = obj\_index.as\_retriever(similarity\_top\_k=2,verbose=True)

......

top\_agent = OpenAIAgent.from\_tools(tool\_retriever=tool\_retriever,

　　　　　　　　　　　　　　　　　　　　　verbose=True,

system\_prompt="""You are an agent designed to answer queries over a set of given papers.Please always use the tools provided to answer a question.Do not rely on prior knowledge.""")

......

P395

tools\_needed = tool\_retriever.retrieve("What is the Adaptive retrieval in the c-RAG?")

print('Tools needed to answer the question:')

for tool in tools\_needed:

print(tool.metadata.name)

P397

citys\_dict = {

'北京市':'beijing',

'南京市':'nanjing',

'广州市':'guangzhou',

'上海市':'shanghai',

'深圳市':'shenzhen'

}

llm\_openai = OpenAI(model='gpt-3.5-turbo')

embedded\_model\_openai = OpenAIEmbedding(model\_name="text-embedding-3-small", embed\_batch\_size=50)

Settings.llm = llm\_openai

Settings.embed\_model = embedded\_model\_openai

def rewrite\_query(query: str, num: int = 3):

""" 将query转换为num个查询问题"""

prompt\_rewrite\_temp = """\

您是一个查询生成器，根据我的输入问题生成多个查询问题。

请生成与以下输入问题相关的{num\_queries}个查询问题 \n

注意每个查询问题都占一行 \n

我的输入问题：{query}

生成查询列表：

"""

prompt\_rewrite = PromptTemplate(prompt\_rewrite\_temp)

response = llm\_openai.predict(

prompt\_rewrite, num\_queries=num, query=query

)

# 假设大模型将每个查询问题都放在一行上

queries = response.split("\n")

return queries

P398

......

def create\_vector\_index\_retriever(name:str):

#解析Document为Node

city\_docs = \

SimpleDirectoryReader(input\_files=[f"../../data/citys/{name}.txt"  
]).load\_data()

splitter = SentenceSplitter(chunk\_size=500,chunk\_overlap=0)

nodes = splitter.get\_nodes\_from\_documents(city\_docs)

#存储到向量库Chroma中

collection = \ chroma.get\_or\_create\_collection(name=f"agent\_{citys\_dict[name]}")

vector\_store = ChromaVectorStore(chroma\_collection=collection)

#首次运行时构造向量索引，完成后进行持久化存储，以后直接加载

if not os.path.exists(f"./storage/vectorindex/{citys\_dict[name]}"):

print('Creating vector index...\n')

storage\_context = StorageContext.from\_defaults(vector\_store=vector\_store)

vector\_index = VectorStoreIndex(nodes,storage\_context=　storage\_context)

vector\_index.storage\_context.persist(persist\_dir=f"./　storage/vectorindex/{citys\_dict[name]}")

else:

print('Loading vector index...\n')

storage\_context = StorageContext.from\_defaults(

persist\_dir=f"./storage/vectorindex/{citys\_dict[name]}",

vector\_store=vector\_store)

vector\_index = load\_index\_from\_storage(storage\_context=storage\_context)

#返回向量检索器

vector\_retriever = vector\_index.as\_retriever(similarity\_top\_k=3)

return vector\_retriever

P399

def create\_kw\_index\_retriever(name:str):

city\_docs =\

SimpleDirectoryReader(input\_files=[f"../../data/citys/ {name}.txt"]).load\_data()

splitter = SentenceSplitter(chunk\_size=500,chunk\_overlap=0)

nodes = splitter.get\_nodes\_from\_documents(city\_docs)

if not os.path.exists(f"./storage/keywordindex/{citys\_dict[name]}"):

print('Creating keyeword index...\n')

#构造关键词表索引

kw\_index = KeywordTableIndex(nodes)

kw\_index.storage\_context.persist(

persist\_dir=f"./storage/keywordindex/ {citys\_dict[name]}")

else:

print('Loading keyeword index...\n')

storage\_context = StorageContext.from\_defaults(

persist\_dir=f"./storage/keywordindex/ {citys\_dict[name]}")

kw\_index = load\_index\_from\_storage(storage\_context= storage\_context)

#返回关键词检索器

kw\_retriever = kw\_index.as\_retriever(num\_chunks\_per\_query=5)

return kw\_retriever

P400

async def run\_queries(queries, retrievers):

tasks = []

#对于每个问题，每个检索器都进行检索

for query in queries:

for i, retriever in enumerate(retrievers):

tasks.append(retriever.aretrieve(query))

task\_results = await tqdm.gather(\*tasks)

#保存每次检索的结果

results\_dict = {}

for i, (query, query\_result) in enumerate(zip(queries, task\_results)):

results\_dict[(query, i)] = query\_result

return results\_dict

def rerank\_results(results\_dict, similarity\_top\_k: int = 3):

k = 60.0

fused\_scores = {}

text\_to\_node = {}

#计算不同Node的文本内容评分

for nodes\_with\_scores in results\_dict.values():

for rank, node\_with\_score in enumerate(

sorted(

nodes\_with\_scores, key=lambda x: x.score or 0.0, reverse=True

)

):

text = node\_with\_score.node.get\_content()

text\_to\_node[text] = node\_with\_score

if text not in fused\_scores:

fused\_scores[text] = 0.0

fused\_scores[text] += 1.0 / (rank + k)

#重排序

reranked\_results = dict(

sorted(fused\_scores.items(), key=lambda x: x[1], reverse=True)

)

# 构造重排序的Node并返回前*K*个Node

reranked\_nodes: List[NodeWithScore] = []

for text, score in reranked\_results.items():

reranked\_nodes.append(text\_to\_node[text])

reranked\_nodes[-1].score = score

return reranked\_nodes[:similarity\_top\_k]

P401

class FusionRetriever(BaseRetriever):

#基于多个检索器构造融合检索器

#参数：检索器列表与top\_k

def \_\_init\_\_(

self,

retrievers: List[BaseRetriever],

similarity\_top\_k: int = 3,

) -> None:

self.\_retrievers = retrievers

self.\_similarity\_top\_k = similarity\_top\_k

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

#实现检索方法

def \_retrieve(self, query\_bundle: QueryBundle) -> List[NodeWithScore]:

#查询转换

querys = rewrite\_query(query\_bundle.query\_str,num=3)

#调用辅助方法得到全部检索结果

results\_dict = asyncio.run(run\_queries(querys, self.\_retrievers))

#使用RRF算法重排序

final\_results = rerank\_results(results\_dict, similarity\_top\_k=self.\_similarity\_top\_k)

return final\_results

P402

def run\_main():

query = "南京市有多少人口，是怎么分布的？"

#构造两个检索器

vector\_retriever = create\_vector\_index\_retriever('南京市')

kw\_retriever = create\_kw\_index\_retriever('南京市')

#构造融合检索器

fusion\_retriever = FusionRetriever(

[vector\_retriever, kw\_retriever],

similarity\_top\_k=3)

#构造查询引擎

query\_engine = RetrieverQueryEngine(fusion\_retriever)

#查询

response=query\_engine.query(query)

pprint\_response(response,show\_source=True)

if \_\_name\_\_ == "\_\_main\_\_":

run\_main()

P403

......

def run\_main():

from llama\_index.core.retrievers import QueryFusionRetriever

query = "南京市有多少人口，是怎么分布的？"

#构造两个检索器

vector\_retriever = create\_vector\_index\_retriever('南京市')

kw\_retriever = create\_kw\_index\_retriever('南京市')

#使用现成的QueryFusionRetriever类型的融合检索器

fusion\_retriever = QueryFusionRetriever(

[vector\_retriever, kw\_retriever],

similarity\_top\_k=3,

num\_queries=1, # set this to 1 to disable query generation

mode="reciprocal\_rerank",

use\_async=True,

verbose=True

)

#构造查询引擎

query\_engine = RetrieverQueryEngine(fusion\_retriever)

#查询

response=query\_engine.query(query)

pprint\_response(response)

P407

......

docs = SimpleDirectoryReader(input\_files=["../../data/c-rag.pdf"]).load\_data()

def create\_base\_index():

splitter = SentenceSplitter(chunk\_size=1024,chunk\_overlap=0)

nodes = splitter.get\_nodes\_from\_documents(docs)

#设置每个Node的id为固定值

for idx,node in enumerate(nodes):

node.id\_ = f"node\_{idx}"

collection = chroma.get\_or\_create\_collection(name=f"crag")

vector\_store = ChromaVectorStore(chroma\_collection=collection)

if not os.path.exists(f"./storage/vectorindex/crag"):

print('Creating vector index...\n')

storage\_context = StorageContext.from\_defaults(vector\_store=vector\_store)

vector\_index = VectorStoreIndex(nodes,storage\_context=storage\_context)

vector\_index.storage\_context.persist(persist\_dir=f"./storage/vectorindex/crag")

else:

print('Loading vector index...\n')

storage\_context = StorageContext.from\_defaults(

persist\_dir=f"./storage/vectorindex/crag",

vector\_store=vector\_store)

vector\_index = load\_index\_from\_storage(storage\_context=storage\_context)

return vector\_index,nodes

#构造父Node

base\_index,base\_nodes = create\_base\_index()

P408

def create\_subnodes\_index(base\_nodes):

#构造两个不同粒度的分割器

sub\_chunk\_sizes = [128, 256]

sub\_node\_parsers = [SentenceSplitter(chunk\_size=subsize,chunk\_overlap=0) for subsize in sub\_chunk\_sizes]

all\_nodes = []

#对每一个父Node都进行分割

for base\_node in base\_nodes:

for n in sub\_node\_parsers:

#使用get\_nodes\_from\_documents方法生成子Node

sub\_nodes = n.get\_nodes\_from\_documents([base\_node])

for sn in sub\_nodes:

#子Node是IndexNode类型的，并用父Node的id作为index\_id

indexnode\_sn = \

IndexNode.from\_text\_node(sn, base\_node.node\_id)

all\_nodes.append(indexnode\_sn)

#父Node也作为IndexNode对象放入all\_nodes对象中

all\_nodes.append(IndexNode.from\_text\_node(base\_node, base\_node.node\_id))

#构造子Node的向量索引

collection = chroma.get\_or\_create\_collection(name=f"crag-subnodes")

vector\_store = ChromaVectorStore(chroma\_collection=collection)

if not os.path.exists(f"./storage/vectorindex/crag-subnodes"):

print('Creating subnodes vector index...\n')

storage\_context = StorageContext.from\_defaults (vector\_store=vector\_store)

vector\_index = VectorStoreIndex(all\_nodes, storage\_context=storage\_context)

vector\_index.storage\_context.persist(

persist\_dir=f"./storage/vectorindex/ crag-subnodes")

else:

print('Loading subnodes vector index...\n')

storage\_context = StorageContext.from\_defaults(

persist\_dir=f"./storage/vectorindex/ crag-subnodes",

vector\_store=vector\_store)

vector\_index = load\_index\_from\_storage(storage\_context= storage\_context)

return vector\_index, all\_nodes

#构造子Node与索引

sub\_index,sub\_nodes = create\_subnodes\_index(base\_nodes)

P410

return vector\_index, all\_nodes

#构造子Node与索引

sub\_index,sub\_nodes = create\_subnodes\_index(base\_nodes)

for sn in sub\_nodes:

indexnode\_sn = IndexNode.from\_text\_node(sn, base\_node.node\_id)

all\_nodes.append(indexnode\_sn)

#准备子Node层的检索器

sub\_retriever = sub\_index.as\_retriever(similarity\_top\_k=2)

#准备一个所有Node的id与Node的对应关系字典

#这个字典用于在递归检索时，根据index\_id快速地找到对应的对象

sub\_nodes\_dict = {n.node\_id: n for n in sub\_nodes}

#构造递归检索器

recursive\_retriever = RecursiveRetriever(

"root\_retriever",

retriever\_dict={"root\_retriever": sub\_retriever},

node\_dict=sub\_nodes\_dict,

verbose=True,

)

#用递归检索器构造查询引擎

recursive\_query\_engine = RetrieverQueryEngine.from\_args (recursive\_retriever)

#测试

response = recursive\_query\_engine.query("please explain the concept of Action Trigger in c-rag?

pprint\_response(response)

P412

......

#根据基础Node构造摘要Node

def create\_summary\_nodes(base\_nodes):

#构造一个元数据抽取器

extractor = SummaryExtractor(summaries=["self"], show\_progress=True)

summary\_dict = {}

#为了避免重复抽取，进行持久化存储

if not os.path.exists(f"./storage/metadata/summarys.json"):

print('Extract new summary...\n')

#抽取元数据，建立从Node到元数据的词典

summarys = extractor.extract(base\_nodes)

for node,summary in zip(base\_nodes,summarys):

summary\_dict[node.node\_id] = summary

with open('./storage/metadata/summarys.json', "w") as fp:

json.dump(summary\_dict, fp)

else:

print('Loading summary from storage...\n')

with open('./storage/metadata/summarys.json', "r") as fp:

summary\_dict = json.load(fp)

#根据摘要构造摘要Node，注意使用IndexNode类型

all\_nodes = []

for node\_id, summary in summary\_dict.items():

all\_nodes.append(IndexNode(text=summary["section\_summary"], index\_id=node\_id))

#加入基础Node

all\_nodes.extend(IndexNode.from\_text\_node(base\_node, base\_node.node\_id) for base\_node in base\_nodes)

#构造摘要Node层的索引

collection = chroma.get\_or\_create\_collection (name=f"crag-summarynodes")

vector\_store = ChromaVectorStore(chroma\_collection=collection)

if not os.path.exists(f"./storage/vectorindex/ crag-summarynodes"):

print('Creating summary nodes vector index...\n')

storage\_context = StorageContext.from\_defaults (vector\_store=vector\_store)

vector\_index = VectorStoreIndex(all\_nodes,storage\_context=storage\_context)

vector\_index.storage\_context.persist(persist\_dir= f"./storage/vectorindex/crag-summarynodes")

else:

print('Loading summary nodes vector index...\n')

storage\_context = StorageContext.from\_defaults (persist\_dir=f"./storage/vectorindex/crag-summarynodes",

vector\_store=vector\_store)

vector\_index = load\_index\_from\_storage(storage\_context= storage\_context)

return vector\_index, all\_nodes

P413

......

summary\_index,summary\_nodes = create\_summary\_nodes(base\_nodes)

summary\_retriever = summary\_index.as\_retriever(similarity\_top\_k=2)

summary\_nodes\_dict = {n.node\_id: n for n in summary\_nodes}

#构造一个递归检索器

recursive\_retriever = RecursiveRetriever(

"root\_retriever",

retriever\_dict={"root\_retriever": summary\_retriever},

node\_dict=summary\_nodes\_dict,

verbose=True,

)

recursive\_query\_engine = RetrieverQueryEngine.from\_args (recursive\_retriever)

response = recursive\_query\_engine.query("please explain the concept of Action Trigger in c-rag?)

pprint\_response(response)

......

P414

......

extractor = QuestionsAnsweredExtractor(questions=5, show\_progress=True)

......

P416

......

url = ['https://qw\*\*lm.github.io/zh/blog/qwen1.5/']

#此处更改默认的摘要Prompt为中文

DEFAULT\_SUMMARY\_QUERY\_STR = """\

尽可能结合上下文，用中文详细介绍表格内容。\

这个表格是关于什么的？给出一个摘要说明（想象你正在为这个表格添加一个新的标题和摘要），\

如果提供了上下文，请输出真实/现有的表格标题/说明。\

如果提供了上下文，请输出真实/现有的表格ID。\

"""

#加载网页到docs变量

web\_loader = SimpleWebPageReader()

docs = web\_loader.load\_data(url)

#分割成Node，并持久化存储

node\_parser = UnstructuredElementNodeParser(

summary\_query\_str=DEFAULT\_SUMMARY\_QUERY\_STR)

if nodes\_save\_path is None or not os.path.exists (nodes\_save\_path):

raw\_nodes = node\_parser.get\_nodes\_from\_documents(docs)

pickle.dump(raw\_nodes, open(nodes\_save\_path, "wb"))

else:

raw\_nodes = pickle.load(open(nodes\_save\_path, "rb"))

P418

#解析其中的IndexNode类型的索引Node，找到其指向的Node，然后给该Node中的表格生成对应的查询引擎

raw\_nodes\_dict = {doc.id\_: doc for doc in raw\_nodes}

query\_engine\_dict = {}

nonbase\_node\_ids = set()

for node in raw\_nodes:

#如果是索引Node

if isinstance(node, IndexNode):

#找到索引Node指向的表格Node

child\_node = raw\_nodes\_dict[node.index\_id]

#把表格Node转换成pandas.DataFrame类型的

df = node\_to\_df(child\_node)

#构造一个基于此DataFrame对象的查询引擎

df\_query\_engine = PandasQueryEngine(df)

#将索引Node与查询引擎链接起来

query\_engine\_dict[node.index\_id] = df\_query\_engine

#记录已经被索引Node引用的Node，后面将其去除

nonbase\_node\_ids.add(node.index\_id)

#去除已经被索引Node引用的Node，剩下的Node用于构造一级向量索引

base\_nodes = []

for node in raw\_nodes:

if node.node\_id not in nonbase\_node\_ids:

base\_nodes.append(node)

P419

#把Node中的内容转换为一个DataFrame对象

node\_table\_save\_path = './storage/nodes/qwen1.5de\_id}.pkl'

def node\_to\_df(node):

prompt\_rewrite\_temp = """\

你是一个数据清洗工具。请去除内容中前面的说明部分，仅保留表格输出。不要多余解释和多余空格。不要修改和编造表格。\n

内容：{content}

表格：

"""

prompt\_rewrite = PromptTemplate(prompt\_rewrite\_temp)

llm = OpenAI(model="gpt-3.5-turbo")

node\_table\_save\_file = node\_table\_save\_path.format (node\_id=node.id\_)

if not os.path.exists(node\_table\_save\_file):

response = llm.predict(

prompt\_rewrite, content=node.get\_content (metadata\_mode='llm')

)

pickle.dump(response, open(node\_table\_save\_file, "wb"))

else:

response = pickle.load(open(node\_table\_save\_file, "rb"))

#把输出的Markdown表格文本转换为Pandas数据分析组件的DataFrame对象

df = pd.read\_csv(io.StringIO(response), sep="|", engine="python")

return df

P420

#给基础Node构造一级向量索引

collection = chroma.get\_or\_create\_collection(name=f"qwen1.5")

vector\_store = ChromaVectorStore(chroma\_collection=collection)

if not os.path.exists(f"./storage/vectorindex/qwen1.5"):

print('Creating vector index...\n')

storage\_context = StorageContext.from\_defaults (vector\_store=vector\_store)

vector\_index = VectorStoreIndex(base\_nodes,

storage\_context= storage\_context)

vector\_index.storage\_context.persist(

persist\_dir=f"./storage/ vectorindex/qwen1.5")

else:

print('Loading vector index...\n')

storage\_context = StorageContext.from\_defaults(

persist\_dir=f"./storage/ vectorindex/qwen1.5",

vector\_store=vector\_store)

vector\_index = load\_index\_from\_storage (storage\_context=storage\_context)

#构造一级检索器

vector\_retriever = vector\_index.as\_retriever (similarity\_top\_k=2)

#构造递归检索器，实现递归检索

recursive\_retriever = RecursiveRetriever(

"vector",

retriever\_dict={"vector": vector\_retriever},

#node\_dict=node\_mappings,

query\_engine\_dict=query\_engine\_dict,

verbose=True,

)

P421

query\_engine = RetrieverQueryEngine.from\_args(recursive\_retriever)

response = query\_engine.query('HumanEval基准测试中，哪些模型参与了测试？平均分是多少？最高分是多少？')

pprint\_response(response)

P422

......

#创建针对某个文档的Agent

def create\_file\_agent(file,name):

print(f'Starting to create tool agent for 【{name}】...\n')

#......省略构造文档对应的向量索引对象的过程......

# vector\_index = ...

#构造查询引擎

query\_engine = vector\_index.as\_query\_engine(similarity\_top\_k=3)

# 构造摘要索引

summary\_index = SummaryIndex(nodes)

summary\_engine = summary\_index.as\_query\_engine(

response\_mode="tree\_summarize")

# 将查询引擎“工具化”

query\_tool = QueryEngineTool.from\_defaults(

query\_engine=query\_engine,

name=f'query\_tool',

description=f'Use if you want to query details about {name}')

summary\_tool = QueryEngineTool.from\_defaults(

query\_engine=summary\_engine,

name=f'summary\_tool',

description=f'Use ONLY IF you want to get a holistic summary of the documents. DO NOT USE if you want to query some details about {name}.')

# 创建文档Agent

file\_agent = ReActAgent.from\_tools([query\_tool,summary\_tool],

verbose=True,

system\_prompt=f"""You are a specialized agent designed to answer queries about {name}.You must ALWAYS use at least one of the tools provided when answering a question; do NOT rely on prior knowledge.DO NOT fabricate answer."""

)

return file\_agent

agent = create\_file\_agent('../../data/citys/南京市.txt','Nanjing')

agent.chat\_repl()

P424

names = ['Nanjing','Beijing','Shanghai']

files = ['../../data/citys/南京市.txt','../../data/citys/北京市.txt','../../data/citys/上海市.txt']

#创建不同的文档Agent

print('===============================================\n')

print('Creating file agents for different documents...\n')

file\_agents\_dict = {}

for name, file in zip(names, files):

file\_agent = create\_file\_agent(file, name)

file\_agents\_dict[name] = file\_agent

P425

print('===============================================\n')

print('Creating top level nodes from tool agents...\n')

index\_nodes = []

query\_engine\_dict = {}

#给每个文档都构造一个索引Node，用于搜索

for name in names:

doc\_summary = f"这部分内容包含关于城市{name}的维基百科文章。如果您需要查找城市{name}的具体事实，请使用此索引。\n如果您想分析多个城市，请不要使用此索引。"

node = IndexNode(

index\_id = name,

text=doc\_summary,

)

index\_nodes.append(node)

#把index\_id与真正的Agent对应起来，用于在递归检索时查找

#注意Agent也是一种查询引擎

query\_engine\_dict[name] = file\_agents\_dict[name]

#构造一级索引与检索器

top\_index = VectorStoreIndex(index\_nodes)

top\_retriever = top\_index.as\_retriever(similarity\_top\_k=1)

#构造递归检索器，从上面的top\_retriever对象开始

#传入query\_engine\_dict变量，用于在递归检索时找到二级文档Agent

recursive\_retriever = RecursiveRetriever(

"vector",

retriever\_dict={"vector": top\_retriever},

query\_engine\_dict=query\_engine\_dict,

verbose=True,

)

P426

query\_engine = RetrieverQueryEngine.from\_args(recursive\_retriever)

response = query\_engine.query('南京市有哪些著名的旅游景点呢？')

print(response)

P437

import os

import pickle

import chromadb

from multiprocessing import Lock

from multiprocessing.managers import BaseManager

from llama\_index.core import Settings,SimpleDirectoryReader, VectorStoreIndex,StorageContext, load\_index\_from\_storage

from llama\_index.embeddings.ollama import OllamaEmbedding

from llama\_index.embeddings.openai import OpenAIEmbedding

from llama\_index.vector\_stores.chroma import ChromaVectorStore

from llama\_index.llms.ollama import Ollama

from llama\_index.llms.openai import OpenAI

#准备模型

llm\_ollama = Ollama(model='qwen2')

embedded\_model\_openai = OpenAIEmbedding(model\_name="text-embedding-3-small", embed\_batch\_size=50)

Settings.llm=llm\_ollama

Settings.embed\_model=embedded\_model\_openai

P438

index = None

stored\_docs = {}

#确保线程安全

lock = Lock()

#索引持久化存储的位置

index\_name = "./saved\_index"

pkl\_name = "stored\_documents.pkl"

#索引服务端口

SERVER\_PORT = 5602

# 初始化索引

def initialize\_index():

global index, stored\_docs

#构造向量存储

chroma = chromadb.HttpClient(host="localhost", port=8000)

collection = chroma.get\_or\_create\_collection(name="chat\_docs\_collection")

vector\_store = ChromaVectorStore(chroma\_collection=collection)

#注意使用with lock进行互斥的索引访问

with lock:

#如果已经存在持久化存储的数据，那么加载

if os.path.exists(index\_name):

storage\_context = StorageContext.from\_defaults(

persist\_dir=index\_name,

vector\_store=vector\_store)

index = load\_index\_from\_storage(storage\_context= storage\_context)

else:

storage\_context = StorageContext.from\_defaults(

vector\_store=vector\_store)

#首次构造空的索引，后面再插入

index = VectorStoreIndex([],storage\_context= storage\_context)

index.storage\_context.persist(persist\_dir=index\_name)

#将已经上传的文档信息从存储的文档中读取到内存

if os.path.exists(pkl\_name):

with open(pkl\_name, "rb") as f:

stored\_docs = pickle.load(f)

P439

#定义查询索引的方法

def query\_index(query\_text):

"""Query the global index."""

global index

response = index.as\_query\_engine().query(query\_text)

return response

#在已有的索引中插入新的文档对象

def insert\_into\_index(doc\_file\_path, doc\_id=None):

"""在已有的索引中插入新的文档对象."""

global index, stored\_docs

document = SimpleDirectoryReader(input\_files=[doc\_file\_path]).load\_data()[0]

if doc\_id is not None:

document.doc\_id = doc\_id

#使用with lock实现共享对象的互斥（顺序）访问

with lock:

index.insert(document)

index.storage\_context.persist(persist\_dir=index\_name)

#这里简化使用，只读取前200个字符

stored\_docs[document.doc\_id] = document.text[0:200] # only take the first 200 chars

with open(pkl\_name, "wb") as f:

pickle.dump(stored\_docs, f)

return

P440

def get\_documents\_list():

"""Get the list of currently stored documents."""

global stored\_doc

documents\_list = []

for doc\_id, doc\_text in stored\_docs.items():

documents\_list.append({"id": doc\_id, "text": doc\_text})

return documents\_list

if \_\_name\_\_ == "\_\_main\_\_":

#初始化索引

print("initializing index...")

initialize\_index()

#构造manager

print(f'Create server on port {SERVER\_PROT}...')

manager = BaseManager(('', SERVER\_PROT), b'password')

#注册函数

print("registering functions...")

manager.register('query\_index', query\_index)

manager.register('insert\_into\_index', insert\_into\_index)

manager.register('get\_documents\_list', get\_documents\_list)

server = manager.get\_server()

#启动server

print("server started...")

server.serve\_forever()

P442

from multiprocessing.managers import BaseManager

def test\_query\_index():

manager = BaseManager(('', 5602), b'password')

manager.register('query\_index')

manager.connect()

response = manager.query\_index('你好！').\_getvalue()

print(response)

if \_\_name\_\_ == "\_\_main\_\_":

test\_query\_index()

P443

import os

from multiprocessing.managers import BaseManager

from werkzeug.utils import secure\_filename

from fastapi import FastAPI, Request, UploadFile, File

from fastapi.responses import JSONResponse

from fastapi.exceptions import RequestValidationError

from fastapi.middleware.cors import CORSMiddleware

from pydantic import BaseModel

from typing import List

import os

from multiprocessing.managers import BaseManager

import uvicorn

app = FastAPI()

# 启用CORS

app.add\_middleware(

CORSMiddleware,

allow\_origins=["\*"],

allow\_methods=["\*"],

allow\_headers=["\*"],

)

# 连接Index Server模块

manager = BaseManager(('', 5602), b'password')

manager.register('query\_index')

manager.register('insert\_into\_index')

manager.register('get\_documents\_list')

manager.connect()

@app.get("/")

def home():

return "Hello, World!"

@app.exception\_handler(RequestValidationError)

async def validation\_exception\_handler(request, exc):

return JSONResponse(content="Invalid request parameters", status\_code=400)

P444

@app.get("/query/")

def query\_index(request: Request, query\_text: str):

global manager

if query\_text is None:

return JSONResponse(content="No text found, please include a ?text=blah parameter in the URL", status\_code=400)

response = manager.query\_index(query\_text).\_getvalue()

response\_json = {

"text": str(response),

"sources": [{"text": str(x.text),

"similarity": round(x.score, 2),

"doc\_id": str(x.id\_)

} for x in response.source\_nodes]

}

return JSONResponse(content=response\_json, status\_code=200)

P445

@app.post("/uploadFile")

async def upload\_file(request: Request, file: UploadFile = File(...), filename\_as\_doc\_id: bool = False):

global manager

try:

contents = await file.read()

filepath = os.path.join('documents', file.filename)

with open(filepath, "wb") as f:

f.write(contents)

if filename\_as\_doc\_id:

manager.insert\_into\_index(filepath, doc\_id=file.filename)

else:

manager.insert\_into\_index(filepath)

except Exception as e:

if os.path.exists(filepath):

os.remove(filepath)

return JSONResponse(content="Error: {}".format(str(e)), status\_code=500)

if os.path.exists(filepath):

os.remove(filepath)

return JSONResponse(content="File inserted!", status\_code=200)

P446

@app.get("/getDocuments")

def get\_documents(request: Request):

document\_list = manager.get\_documents\_list().\_getvalue()

return JSONResponse(content=document\_list, status\_code=200)

if \_\_name\_\_ == "\_\_main\_\_":

uvicorn.run(app, host="0.0.0.0", port=5601)

> python fast\_api.py

P447

http://localhost:5601/docs

P449

> npx create-react-app my-app --template typescript

P450

export type ResponseSources = {

text: string;

doc\_id: string;

similarity: number;

};

export type QueryResponse = {

text: string;

sources: ResponseSources[];

};

const queryIndex = async (query: string): Promise<QueryResponse> => {

const queryURL = new URL('http://localhost:5601/query?');

queryURL.searchParams.append('query\_text', query);

const response = await fetch(queryURL, { mode: 'cors' });

if (!response.ok) {

return { text: 'Error in query', sources: [] };

}

const queryResponse = (await response.json()) as QueryResponse;

return queryResponse;

};

export default queryIndex;

P451

import queryIndex, { ResponseSources } from '../apis/queryIndex';

const IndexQuery = () => {

......

const handleQuery = (e: React.KeyboardEvent<HTMLInputElement>) => {

if (e.key == 'Enter') {

setLoading(true);

#调用queryIndex函数

queryIndex(e.currentTarget.value).then((response) => {

setLoading(false);

#设置响应结果，同步前端UI状态

setResponseText(response.text);

setResponseSources(response.sources);

});

}

};

......

return (

<div className='query'>

<div className='query\_\_input'>

<label htmlFor='query-text'>输入你的问题</label>

<input

type='text'

name='query-text'

placeholder='你的问题'

onKeyDown={handleQuery}

></input>

</div>

}

P452

> npm start

P455

......

......省略模型与向量存储的准备代码......

#目录

HOME\_DIR='/Users/pingcy/src/multiagents/backend'

DATA\_DIR = f'{HOME\_DIR}/data'

STOR\_DIR = f'{HOME\_DIR}/app/storage'

#本应用的知识文档

city\_docs = {

"Beijing":f'{DATA\_DIR}/北京市.txt',

"Guangzhou":f'{DATA\_DIR}/广州市.txt',

"Nanjing":f'{DATA\_DIR}/南京市.txt',

"Shanghai":f'{DATA\_DIR}/上海市.txt'

}

#创建针对单个文档的Tool Agent

def \_create\_doc\_agent(name:str,callback\_manager: CallbackManager):

file = city\_docs[name]

......此处省略构造vector\_index对象的过程......

#vector\_index = ...

query\_engine = vector\_index.as\_query\_engine(similarity\_top\_k=5)

# 构造摘要索引与查询引擎

summary\_index = SummaryIndex(nodes)

summary\_engine = summary\_index.as\_query\_engine(

response\_mode="tree\_summarize")

# 把两个查询引擎工具化

query\_tool = QueryEngineTool.from\_defaults(query\_engine=query\_engine,

name=f'query\_tool',

description=f'用于回答关于城市{name}的具体问题，包括经济、旅游、文化、历史等方面')

summary\_tool = QueryEngineTool.from\_defaults(query\_engine=summary\_engine,

name=f'summary\_tool',

description=f'任何需要对城市{name}的各个方面进行全面总结的请求请使用本工具。如果您想查询有关 {name} 的某些详细信息，请使用query\_tool')

city\_tools = [query\_tool,summary\_tool]

# 使用两个工具创建单独的文档Agent

doc\_agent = ReActAgent.from\_tools(city\_tools,

verbose=True,

system\_prompt=f'你是一个专门设计用于回答有关城市{name}信息查询的助手。在回答问题时，你必须始终使用至少一个提供的工具；不要依赖先验知识。不要编造答案。',

callback\_manager=callback\_manager)

return doc\_agent

P457

#循环创建针对所有文档的Tool Agent，并将其保存到doc\_agents\_dict字典中

def \_create\_doc\_agents(callback\_manager: CallbackManager):

print('Creating document agents for all citys...\n')

doc\_agents\_dict = {}

for city in city\_docs.keys():

doc\_agents\_dict[city] = \_create\_doc\_agent(city,callback\_manager)

return doc\_agents\_dict

def \_create\_top\_agent(doc\_agents: Dict,callback\_manager: CallbackManager):

all\_tools = []

for city in doc\_agents.keys():

city\_summary = (

f" 这部分包含了有关{city}的城市信息. "

f" 如果需要回答有关{city}的任务问题，请使用这个工具.\n"

)

#把创建好的每个Tool Agent都工具化

doc\_tool = QueryEngineTool(

query\_engine=doc\_agents[city],

metadata=ToolMetadata(

name=f"tool\_{city}",

description=city\_summary,

),

)

all\_tools.append(doc\_tool)

#实现一个对象索引，用于检索工具

tool\_mapping = SimpleToolNodeMapping.from\_objects(all\_tools)

if not os.path.exists(f"{STOR\_DIR}/top"):

storage\_context = StorageContext.from\_defaults()

obj\_index = ObjectIndex.from\_objects(

all\_tools,

tool\_mapping,

VectorStoreIndex,

storage\_context=storage\_context

)

storage\_context.persist(persist\_dir=f"{STOR\_DIR}/top")

else:

storage\_context = StorageContext.from\_defaults(

persist\_dir=f"{STOR\_DIR}/top"

)

index = load\_index\_from\_storage(storage\_context)

obj\_index = ObjectIndex(index, tool\_mapping)

print('Creating top agent...\n')

#创建Top Agent

top\_agent = ReActAgent.from\_tools(

tool\_retriever=obj\_index.as\_retriever(similarity\_top\_k=3),

verbose=True,

system\_prompt="你是一个被设计来回答关于一组给定城市查询的助手。请始终使用提供的工具来回答一个问题。不要依赖先验知识。不要编造答案",

callback\_manager=callback\_manager)

return top\_agent

P459

def get\_agent():

#创建Agent，此处暂时忽略callback\_manager

callback\_manager = CallbackManager()

doc\_agents = \_create\_doc\_agents(callback\_manager)

top\_agent = \_create\_top\_agent(doc\_agents,callback\_manager)

return top\_agent

if \_\_name\_\_ == '\_\_main\_\_':

top\_agent = get\_agent()

print('Starting to stream chat...\n')

streaming\_response = top\_agent.streaming\_chat\_repl()

P460

#单个消息：角色与内容

class \_Message(BaseModel):

role: MessageRole

content: str

#接口数据：消息的顺序列表

class \_ChatData(BaseModel):

messages: List[\_Message]

P461

......

chat\_router = r = APIRouter()

@r.post("/nostream")

async def chat\_nostream(

data: \_ChatData

):

#获得Agent

agent = get\_agent()

if len(data.messages) == 0:

raise HTTPException(

status\_code=status.HTTP\_400\_BAD\_REQUEST,

detail="No messages provided",

)

#最后一个产生消息的角色必须是user

lastMessage = data.messages.pop()

if lastMessage.role != MessageRole.USER:

raise HTTPException(

status\_code=status.HTTP\_400\_BAD\_REQUEST,

detail="Last message must be from user",

)

# 创建消息历史

messages = [

ChatMessage(

role=m.role,

content=m.content,

)

for m in data.messages

]

#调用Agent获得响应结果并返回

chat\_result = agent.chat(lastMessage.content, messages)

return JSONResponse(content={"text":str(chat\_result)}, status\_code=200)

P462

import logging

import os

import uvicorn

from backend.api.chat import chat\_router

from fastapi import FastAPI

from fastapi.middleware.cors import CORSMiddleware

from dotenv import load\_dotenv

app = FastAPI()

app.add\_middleware(

CORSMiddleware,

allow\_origins=["\*"],

allow\_credentials=True,

allow\_methods=["\*"],

allow\_headers=["\*"])

#将chat\_router路由包含进来，有利于路由的模块化组织管理

app.include\_router(chat\_router, prefix="/api/chat")

if \_\_name\_\_ == "\_\_main\_\_":

uvicorn.run(app="main:app", host="0.0.0.0",port=8090,reload=True)

P464

#流式响应chat接口

@r.post("")

async def chat(

data: \_ChatData

):

......

#创建消息历史

messages = [

ChatMessage(

role=m.role,

content=decode\_sse\_messages(m.content),

)

for m in data.messages

]

#调用stream\_chat方法获取Agent的响应流

chat\_result = agent.stream\_chat(lastMessage.content, messages)

#构造一个生成器，用于迭代处理输出的token

def event\_generator():

for token in chat\_result.response\_gen:

yield convert\_sse(token)

#客户端流式响应

return StreamingResponse(event\_generator(),

media\_type="text/event-stream")

P465

def convert\_sse(obj: str | dict):

return "data: {}\n\n".format(json.dumps(obj,ensure\_ascii=False))

P467

......

#自定义一个事件类型，包含类型与事件信息

class EventObject(BaseModel):

type: str

payload: dict

#自定义流跟踪的事件处理器

class StreamingCallbackHandler(BaseCallbackHandler):

def \_\_init\_\_(self, queue: Queue) -> None:

super().\_\_init\_\_([], [])

self.\_queue = queue

#自定义on\_event\_start接口

def on\_event\_start(

self,

event\_type: CBEventType,

payload: Optional[Dict[str, Any]] = None,

event\_id: str = "",

parent\_id: str = "",

\*\*kwargs: Any,

) -> str:

#跟踪Agent的工具使用

if event\_type == CBEventType.FUNCTION\_CALL:

self.\_queue.put(

EventObject(

type="function\_call",

payload={

"arguments\_str": str(payload["function\_call"]),

"tool\_str": str(payload["tool"].name),

},

)

)

#自定义on\_event\_end接口

def on\_event\_end(

self,

event\_type: CBEventType,

payload: Optional[Dict[str, Any]] = None,

event\_id: str = "",

\*\*kwargs: Any,

) -> None:

#跟踪Agent的工具调用

if event\_type == CBEventType.FUNCTION\_CALL:

self.\_queue.put(

EventObject(

type="function\_call\_response",

payload={"response": payload["function\_call\_response"]},

)

)

#跟踪Agent每个步骤的大模型响应

elif event\_type == CBEventType.AGENT\_STEP:

self.\_queue.put(payload["response"])

@property

def queue(self) -> Queue:

"""Get the queue of events."""

return self.\_queue

def start\_trace(self, trace\_id: Optional[str] = None) -> None:

"""Run when an overall trace is launched."""

pass

def end\_trace(

self,

trace\_id: Optional[str] = None,

trace\_map: Optional[Dict[str, List[str]]] = None,

) -> None:

"""Run when an overall trace is exited."""

pass

P469

def get\_agent():

# 构造并加入自定义的事件处理器

queue = Queue()

handler = StreamingCallbackHandler(queue)

callback\_manager = CallbackManager([handler])

#使用自定义的事件处理器

doc\_agents = \_create\_doc\_agents(callback\_manager)

top\_agent = \_create\_top\_agent(doc\_agents,callback\_manager)

return top\_agent

P470

......

@r.post("")

async def chat(

data: \_ChatData

):

agent = get\_cached\_agent(user\_id)

......省略部分代码......

# 转换为ChatMessage列表格式

messages = [

ChatMessage(

role=m.role,

content=decode\_sse\_messages(m.content),

)

for m in data.messages

]

thread = Thread(target=agent.stream\_chat,

args=(lastMessage.content, messages))

thread.start()

#生成器，用于构造StreamingResponse对象

def event\_generator():

#从队列中读取对象

queue = agent.callback\_manager.handlers[0].queue

while True:

next\_item = queue.get(True, 60.0)

#判断next\_item

#如果是EventObject类型的，则是FUNCTION\_CALL事件

if isinstance(next\_item, EventObject):

yield convert\_sse(dict(next\_item))

#如果是StreamingAgentChatResponse类型的，则是AGENT\_STEP事件

elif isinstance(next\_item, StreamingAgentChatResponse):

response = cast(StreamingAgentChatResponse, next\_item)

#通过response\_gen方法迭代处理流式响应

for token in response.response\_gen:

yield convert\_sse(token)

break

return StreamingResponse(event\_generator(), media\_type= "text/event-stream")

P472

> pnpm create next-app@latest my-ai-app

> npm install ai @ai-sdk/openai @ai-sdk/react zod

P474

class \_ChatData(BaseModel):

messages: List[\_Message]

P475

@lru\_cache(maxsize=50)

def get\_cached\_agent(user\_id: str) -> OpenAIAgent:

return get\_agent()

P478

......

#生成答案

DEFAULT\_TEXT\_QA\_PROMPT\_TMPL = (

"以下是上下文\n"

"---------------------\n"

"{context\_str}\n"

"---------------------\n"

"请仅根据上面的上下文，回答以下问题，不要编造其他内容。\n"

"如果上下文中不存在相关信息，请拒绝回答。\n"

"问题: {query\_str}\n"

"答案: "

)

text\_qa\_prompt = PromptTemplate(DEFAULT\_TEXT\_QA\_PROMPT\_TMPL)

#评估相关性

EVALUATE\_PROMPT\_TEMPLATE="""您是一个评分人员，评估检索出的文档与用户问题的相关性。

以下是检索出的文档：

----------------

{context}

----------------

以下是用户问题：

----------------

{query\_str}

----------------

如果文档中包含与用户问题相关的关键词或语义，且有助于解答用户问题，请将其评为相关。

请给出yes或no来表明文档是否与问题相关。

注意只需要输出yes或no，不要有多余解释。

"""

evaluate\_prompt = PromptTemplate(EVALUATE\_PROMPT\_TEMPLATE)

#重写输入问题

REWRITE\_PROMPT\_TEMPLATE= """你需要生成对检索进行优化的问题。请根据输入内容，尝试推理其中的语义意图/含义。

这是初始问题：

----------------

{query\_str}

----------------

请提出一个改进的问题："""

rewrite\_prompt = PromptTemplate(REWRITE\_PROMPT\_TEMPLATE)

P479

#构造检索器

def create\_retriever(file):

docs = SimpleDirectoryReader(input\_files=[file]).load\_data()

index = VectorStoreIndex.from\_documents(docs)

return index.as\_retriever(similarity\_top\_k=3),index.as\_query\_engine()

#评估检索结果

def evaluate\_nodes(query\_str:str,retrieved\_nodes: List[Document]):

#构造一个用于评估的简单查询管道，直接使用大模型也一样

evaluate\_pipeline = QueryPipeline(chain=[evaluate\_prompt, llm\_openai])

filtered\_nodes = []

need\_search = False

for node in retrieved\_nodes:

#对Node中的内容与输入问题评估相关性

relevancy = evaluate\_pipeline.run(

context=node.text, query\_str=query\_str

)

#如果相关，则返回；否则，需要搜索

if(relevancy.message.content.lower()=='yes'):

filtered\_nodes.append(node)

else:

need\_search = True

return filtered\_nodes,need\_search

P480

#重写输入问题

def rewrite(query\_str: str):

new\_query\_str = llm\_openai.predict(

rewrite\_prompt, query\_str = query\_str

)

return new\_query\_str

P481

#搜索

def web\_search(query\_str:str):

tavily\_tool = TavilyToolSpec(api\_key="tvly-\*\*\*")

search\_results = tavily\_tool.search(query\_str,max\_results=5)

return "\n".join([result.text for result in search\_results])

#使用大模型直接生成答案

def query(query\_str,context\_str):

response = llm\_openai.predict(

text\_qa\_prompt, context\_str=context\_str, query\_str=query\_str

)

return response

......

file\_name = "../../data/citys/南京市.txt"

#构造检索器与查询引擎

retriever,query\_engine = create\_retriever(file\_name)

query\_str = '南京市的人口数量是多少与分布情况如何？参加2024年中考的学生数量是多少？'

#先测试直接生成答案

response = query\_engine.query(query\_str)

print(f'-----------------Response from query engine-------------------')

pprint\_response(response,show\_source=True)

#测试C-RAG流程

#C-RAG：检索

print(f'-----------------Response from CRAG-------------------')

retrieved\_nodes = retriever.retrieve(query\_str)

print(f'{len(retrieved\_nodes)} nodes retrieved.\n')

#C-RAG：评估检索结果，仅保留相关的上下文

filtered\_nodes,need\_search = evaluate\_nodes(query\_str,retrieved\_nodes)

print(f'{len(filtered\_nodes)} nodes relevant.\n')

filtered\_texts = [node.text for node in filtered\_nodes]

filtered\_text = "\n".join(filtered\_texts)

#C-RAG：如果存在不相关知识，那么重写输入问题并借助网络搜索

if need\_search:

new\_query\_str = rewrite(query\_str)

search\_text = web\_search(new\_query\_str)

#组合成新的上下文，并进行生成

context\_str = filtered\_text + "\n" + search\_text

response = query(query\_str,context\_str)

print(f'Final Response from crag: \n{response}')

P488

Response：[Relevant] 字节调动的Coze是一个大模型的应用开发平台，提供了一站式开发大模型应用的相关工具、插件与编码环境. [Partially supported] [Utility:5]

Response: 当然![Retrieval]<paragraph>

P490

......

#IsSUP的3种自省token

\_IS\_SUPPORTED\_TOKENS = [

"[Fully supported]",

"[Partially supported]",

"[No support / Contradictory]",

]

#计算IsSUP得分

def \_is\_supported\_score(

pred\_tokens: List[int], pred\_log\_probs\_dict: List[Dict[str, float]]

) -> float:

#最终的得分

is\_supported\_score = 0

#首先找到输出的自省token的位置，然后退出，这个类型的指标的token只有一个

token\_appear\_id = -1

for tok\_idx, token in enumerate(pred\_tokens):

if token in \_IS\_SUPPORTED\_TOKENS:

token\_appear\_id = tok\_idx

break

#如果找到了自省token的位置，比如为[Fully supported]

if token\_appear\_id > -1:

#在这个位置上查找所有该类型的指标的3种自省token的输出概率

#保存到issup\_score\_dict这个字典中

issup\_score\_dict = {}

for token in \_IS\_SUPPORTED\_TOKENS:

prob = pred\_log\_probs\_dict[token\_appear\_id][token]

issup\_score\_dict[token] = np.exp(float(prob))

#用上面的计算公式计算最终得分

is\_supported\_score = (

issup\_score\_dict["[Fully supported]"]

+ 0.5 \* issup\_score\_dict["[Partially supported]"]

) / np.sum(list(issup\_score\_dict.values()))

return is\_supported\_score

P492

> pip install llama\_cpp\_python

> pip install huggingface-hub

> huggingface-cli \

download m4r1/selfrag\_llama2\_7b-GGUF \

selfrag\_llama2\_7b.q4\_k\_m.gguf \

--local-dir ./model \

--local-dir-use-symlinks False

......

from llama\_cpp import Llama

\_MODEL\_KWARGS = {"logits\_all": True, "n\_ctx": 2048, "n\_gpu\_layers":200}

\_GENERATE\_KWARGS = {"temperature": 0.0,"top\_p": 1.0,"max\_tokens": 1024,"logprobs": 1000}

#大模型

llm=Llama(model\_path="./model/selfrag\_llama2\_7b.q4\_k\_m.gguf",\*\*\_MODEL\_KWARGS)

#格式化Prompt，注意按照此格式输入问题和关联知识

def format\_prompt(input, paragraph=None):

prompt = "### Instruction:\n{0}\n\n### Response:\n".format(input)

if paragraph is not None:

prompt += "[Retrieval]<paragraph>{0}</paragraph>".format(paragraph)

return prompt

#测试两个问题，一个无须检索知识，另一个需要检索知识

query\_1 = "写一首歌颂母爱的小诗"

query\_2 = "能否介绍一下字节跳动的AI平台Coze？"

queries = [query\_1, query\_2]

#分别测试，并打印出结果(response)以及更详细的token输出细节

for query in queries:

pred = llm(format\_prompt(query),\*\*\_GENERATE\_KWARGS)

print("\nResponse: {0}".format(pred["choices"][0]["text"]))

print('\nDetails:\n')

print(pred["choices"][0])

P493

Response: Mother love, so pure and true,

A bond that's stronger than any tie.[No Retrieval]You give your all, without a thought,

Your love is the light in our lives.[No Retrieval]In you we find strength and courage,

......follow its owners everywhere.[Utility:5]

Response: Certainly![Retrieval]<paragraph>

Coze is a platform that uses AI to help businesses automate customer service.[Utility:5]

......

#这是模拟的参考知识，提供给大模型

paragraph="""Coze是字节跳动的大模型应用一站式开发平台。"""

from llama\_cpp import Llama

\_MODEL\_KWARGS = {"logits\_all": True, "n\_ctx": 2048, "n\_gpu\_layers":200}

\_GENERATE\_KWARGS = {"temperature": 0.0,"top\_p": 1.0,"max\_tokens": 1024,"logprobs": 1000}

llm=Llama(model\_path="./model/selfrag\_llama2\_7b.q4\_k\_m.gguf",\*\*\_MODEL\_KWARGS)

#此处默认输入参数paragraph为上面的知识

def format\_prompt(input, paragraph=paragraph):

prompt = "### Instruction:\n{0}\n\n### Response:\n".format(input)

if paragraph is not None:

prompt += "[Retrieval]<paragraph>{0}</paragraph>".format(paragraph)

return prompt

query = "能否介绍一下字节跳动的AI平台Coze？"

pred = llm(format\_prompt(query),\*\*\_GENERATE\_KWARGS)

print("\nResponse: {0}".format(pred["choices"][0]["text"]))

print('\nDetails:\n')

print(pred["choices"][0])

P494

Response: [Relevant]Coze is a platform developed by ByteDance, the parent company of TikTok, for building and deploying large-scale AI models.[Fully supported][Continue to Use Evidence]It provides an all-in-one development platform that includes tools for training, testing, and deploying AI models.[Utility:5]

P495

from dataclasses import dataclass

from typing import Any, Dict, List,Tuple

import numpy as np

from llama\_index.core.query\_engine import CustomQueryEngine

from llama\_index.llms.llama\_cpp import LlamaCPP

from llama\_index.core.base.base\_retriever import BaseRetriever

from llama\_index.core.response import Response

from llama\_index.core.bridge.pydantic import Field

from llama\_index.core.utils import print\_text

#定义所有的自省token

\_TOKENS = {

"retrieval": ["[No Retrieval]", "[Retrieval]", "[Continue to Use Evidence]"],

"relevance": ["[Irrelevant]", "[Relevant]"],

"support": ["[Fully supported]", "[Partially supported]", "[No support / Contradictory]"],

"utility": ["[Utility:1]", "[Utility:2]", "[Utility:3]", "[Utility:4]", "[Utility:5]"],

"ctrl": [

"[No Retrieval]","[Retrieval]","[Continue to Use Evidence]",

"[Irrelevant]","[Relevant]",

"[Fully supported]","[Partially supported]","[No support / Contradictory]",

"<paragraph>","</paragraph>",

"[Utility:1]","[Utility:2]","[Utility:3]","[Utility:4]","[Utility:5]",

],

}

#用CustomQueryEngine生成新的查询引擎

class SelfRAGQueryEngine(CustomQueryEngine):

P496

......

def \_\_init\_\_(

self,

llm: LlamaCPP,

retriever: BaseRetriever,

) -> None:

"""初始化查询引擎"""

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

self.llm = llm

self.retriever = retriever

......

def query(self, query\_str: str) -> str:

"""

自定义查询函数。

参数：

query\_str (str): 查询字符串。

返回：

Response: 查询的响应结果。

"""

#调用大模型获得响应结果

response = self.llm.complete(\_format\_prompt(query\_str))

answer = response.text

if "[Retrieval]" in answer:

print\_text("需要检索知识，开始检索...\n", color="blue")

documents = self.retriever.retrieve(query\_str)

print\_text(f"共检索到 {len(documents)} 个相关知识\n", color="blue")

paragraphs = [

\_format\_prompt(query\_str, document.node.text) for document in documents

]

#使用检索内容重新生成结果并评估

print\_text("=====开始：重新生成并评估====\n", color="blue")

llm\_response\_per\_paragraph,paragraphs\_final\_score = \

self.\_regen\_then\_eval(paragraphs)

print\_text("===结束：重新生成并评估====\n", color="blue")

best\_paragraph\_id = max(

paragraphs\_final\_score, key=paragraphs\_final\_score.get

)

answer = llm\_response\_per\_paragraph[best\_paragraph\_id]

print\_text(f"已选择最佳答案: {answer}\n", color="blue")

else:

print\_text("无须检索知识，直接输出答案\n",color="green")

answer = \_postprocess\_answer(answer)

print\_text(f"最终答案: {answer}\n", color="green")

return str(answer)

P498

def \_postprocess\_answer(answer: str) -> str:

for token in \_TOKENS["ctrl"]:

answer = answer.replace(token, "")

if "</s>" in answer:

answer = answer.replace("</s>", "")

if "\n" in answer:

answer = answer.replace("\n", "")

if "<|endoftext|>" in answer:

answer = answer.replace("<|endoftext|>", "")

return answer

......

def \_regen\_then\_eval(self, paragraphs: List[str]) ->Tuple[Dict[int,str],Dict[int,float]]:

"""

运行评估模块，调用大模型对给定的段落进行评估。

参数：

paragraphs (List[str]): 包含要评估的段落的列表。

返回：

Tuple[Dict[int,str],Dict[int,float]]: 包含生成的结果索引和评估字典。

"""

paragraphs\_final\_score = {}

llm\_response\_text = {}

for p\_idx, paragraph in enumerate(paragraphs):

#生成结果

response = self.llm.complete(paragraph)

pred = response.raw

llm\_response\_text[p\_idx] = response.text

#从raw字段中取得token输出概率相关的信息

#top\_logprobs字段保存每个位置上每个token的输出概率

logprobs = pred["choices"][0]["logprobs"]

pred\_log\_probs = logprobs["top\_logprobs"]

# 计算IsREL得分，相关性为第一个token，直接传入0

isrel\_score = \_relevance\_score(pred\_log\_probs[0])

# 计算IsSUP得分

issup\_score = \_is\_supported\_score(logprobs["tokens"], pred\_log\_probs)

# 计算IsUSE得分

isuse\_score = \_is\_useful\_score(logprobs["tokens"], pred\_log\_probs)

#最终得分

paragraphs\_final\_score[p\_idx] = (

isrel\_score + issup\_score + 0.5 \* isuse\_score

)

print\_text(

f"输入: {paragraph}\n响应: {llm\_response\_text[p\_idx]}\n评估: {paragraphs\_final\_score[p\_idx]}\n",

color="blue",

)

print\_text(

f"已完成 {p\_idx + 1}/{len(paragraphs)} 段落\n\n", color="blue"

)

return llm\_response\_text, paragraphs\_final\_score

P500

......

def \_relevance\_score(pred\_log\_probs: Dict[str, float]) -> float:

rel\_prob = np.exp(float(pred\_log\_probs["[Relevant]"]))

irel\_prob = np.exp(float(pred\_log\_probs["[Irrelevant]"]))

return rel\_prob / (rel\_prob + irel\_prob)

def \_is\_useful\_score(

pred\_tokens: List[int], pred\_log\_probs\_dict: List[Dict[str, float]]

) -> float:

isuse\_score = 0

utility\_token\_appear\_id = -1

#先找到位置

for tok\_idx, tok in enumerate(pred\_tokens):

if tok in \_TOKENS["utility"]:

utility\_token\_appear\_id = tok\_idx

#在这个位置上获取不同token的输出概率

if utility\_token\_appear\_id > -1:

ut\_score\_dict = {}

for token in \_TOKENS["utility"]:

prob = pred\_log\_probs\_dict[utility\_token\_appear\_id] [token]

ut\_score\_dict[token] = np.exp(float(prob))

#IsUSE的得分需要加权计算

ut\_sum = np.sum(list(ut\_score\_dict.values()))

ut\_weights = [-1, -0.5, 0, 0.5, 1]

isuse\_score = np.sum(

[

ut\_weights[i] \* (ut\_score\_dict[f"[Utility:{i + 1}]"] / ut\_sum)

for i in range(len(ut\_weights))

]

)

return isuse\_score

P501

import os

from llama\_index.llms.llama\_cpp import LlamaCPP

from llama\_index.core import Document, VectorStoreIndex

from llama\_index.core.retrievers import VectorIndexRetriever

from pathlib import Path

#导入已经构造的自定义Self-RAG查询引擎

from selfrag\_queryengine import SelfRAGQueryEngine

#注意打开logits\_all参数选项

\_MODEL\_KWARGS = {"logits\_all": True, "n\_ctx": 2048, "n\_gpu\_layers": -1}

\_GENERATE\_KWARGS = {

"temperature": 0.0,

"top\_p": 1.0,

"max\_tokens": 1000,

"logprobs": 32016,

}

# 之前下载并保存selfrag\_llama2\_7b模型的目录。

download\_dir = "../../model"

# 构造简单的测试文档，此处直接构造Document对象，方便观察检索结果

documents = [

Document(

text="Xiaomi 14 is the latest smartphone released by Xiaomi. It adopts a new design concept, the body is lighter and thinner, equipped with the latest processor, and the performance is more powerful."

),

Document(

text="Xiaomi 14 phone uses a 6.7-inch ultra-clear large screen, with a resolution of up to 2400x1080, whether watching videos or playing games, it can bring the ultimate visual experience."

),

Document(

text="Xiaomi 14 phone is equipped with the latest Snapdragon 888 processor, equipped with 8GB of running memory and 128GB of storage space, whether it is running large games or multitasking, it can easily cope."

),

Document(

text="Xiaomi 14 phone is equipped with a 5000mAh large-capacity battery, supports fast charging, even if you are traveling or using it for a long time, you don't have to worry about power issues."

),

Document(

text="Xiaomi 14 phone has a rear camera of 64 million pixels and a front camera of 20 million pixels. Whether it is taking pictures or recording videos, it can capture every wonderful moment in life."

),

Document(

text="Xiaomi 14 phone runs the latest MIUI 12 operating system. This operating system has a beautiful interface, smooth operation, and provides a wealth of functions and applications."

),

Document(

text="Xiaomi 14 phone supports 5G network, fast download speed, low latency, whether watching high-definition videos or playing online games, you can enjoy the ultimate network experience."

),

Document(

text="Xiaomi 14 phone supports facial recognition and fingerprint unlocking, protects user privacy, and provides a more convenient unlocking method."

),

Document(

text="Xiaomi 14 phone supports wireless charging and reverse charging functions. Wireless charging can free you from the shackles of data cables, and reverse charging can charge your other devices."

),

Document(

text="Xiaomi 14 phone is equipped with a 90Hz high refresh rate screen, whether scrolling pages or playing games, it can bring a smooth visual experience."

),

]

# 嵌入与索引

index = VectorStoreIndex.from\_documents(documents)

# 构造一个检索器

retriever = VectorIndexRetriever(index=index,similarity\_top\_k=5)

# 构造一个大模型：使用Llama\_cpp作为推理工具

model\_path = Path(download\_dir) / "selfrag\_llama2\_7b.q4\_k\_m.gguf"

llm = LlamaCPP(model\_path=str(model\_path), model\_kwargs=\_MODEL\_KWARGS, generate\_kwargs=\_GENERATE\_KWARGS)

# 构造自定义的查询引擎

query\_engine = SelfRAGQueryEngine(llm, retriever)

# 查询一：无须检索的创作问题

print("\nQuery 1: write a poem about beautiful sunset")

response = query\_engine.query("write a poem about beautiful sunset")

# 查询二：需要检索的事实性问题

print("\nQuery 2: Tell me some truth about xiaomi 14 phone, especially about its battery and camera?")

response = query\_engine.query("Tell me some truth about xiaomi 14 phone, especially about its battery and camera?")

P509

......RaptorRetriever的部分实现......

#基于文档列表构造多级树状的索引结构

async def insert(self, documents: List[BaseNode]) -> None:

#嵌入模型/转换器

embed\_model = self.index.\_embed\_model

transformations = self.index.\_transformations

#对传入的文档做Node分割，这是底层叶子Node的基础

cur\_nodes = run\_transformations(documents, transformations, in\_place=False)

#根据设置的树层次循环构造。在每一次循环后都将本轮生成的父Node作为当前Node

#继续循环处理

for level in range(self.tree\_depth):

#给当前Node生成向量并暂存到id\_to\_embedding变量中

embeddings = await embed\_model.aget\_text\_embedding\_batch(

[node.get\_content(metadata\_mode="embed") for node in cur\_nodes]

)

id\_to\_embedding = {

node.id\_: embedding

for node, embedding in zip(cur\_nodes, embeddings)

}

# 聚类，将语义相近的Node聚类到一个聚簇中

nodes\_per\_cluster = get\_clusters(cur\_nodes, id\_to\_embedding)

#给每个聚簇都生成摘要

summaries\_per\_cluster = await \

self.summary\_module.generate\_summaries(nodes\_per\_cluster)

# 把生成的摘要构造成新的Node,即当前Node的父Node

new\_nodes = [

TextNode(

text=summary,

metadata={"level": level},

excluded\_embed\_metadata\_keys=["level"],

excluded\_llm\_metadata\_keys=["level"],

)

for summary in summaries\_per\_cluster

]

# 处理当前Node，设置其parent\_id为生成的父Node的id

# 根据生成的向量信息设置embedding字段

# 然后，把当前Node插入索引中，这样就完成了本层的索引构造

# 同时生成了这一层的父Node

nodes\_with\_embeddings = []

for cluster, summary\_doc in zip(nodes\_per\_cluster, new\_nodes):

for node in cluster:

node.metadata["parent\_id"] = summary\_doc.id\_

node.excluded\_embed\_metadata\_keys.append("parent\_id")

node.excluded\_llm\_metadata\_keys.append("parent\_id")

node.embedding = id\_to\_embedding[node.id\_]

nodes\_with\_embeddings.append(node)

self.index.insert\_nodes(nodes\_with\_embeddings)

# 以父Node作为新的当前Node，进入下一次循环

# 注意：此时父Node还没有插入索引中

cur\_nodes = new\_nodes

#在达到迭代次数后，把最后一次的父Node插入索引中

self.index.insert\_nodes(cur\_nodes)

P510

......生成每个聚簇的摘要Node，可增加并行处理......

async def generate\_summaries(

self, documents\_per\_cluster: List[List[BaseNode]]

) -> List[str]:

#构造一个tree\_summarize类型的响应生成器

responses = []

response\_synthesizer = get\_response\_synthesizer(

response\_mode="tree\_summarize", use\_async=True, llm=llm

)

#对输入的多个聚簇循环：给每个聚簇中的Node都生成摘要

jobs = []

for documents in documents\_per\_cluster:

with\_scores = [NodeWithScore(node=doc, score=1.0)

for doc in documents]

response = response\_synthesizer.asynthesize(

self.summary\_prompt, with\_scores)

responses.append(response )

return [str(response) for response in responses]

P511

#实现简单的全量检索

async def collapsed\_retrieval(self, query\_str: str) -> Response:

#直接对索引构造检索器后检索即可

return await self.index.as\_retriever(

similarity\_top\_k=3

).aretrieve(query\_str)