跟龙哥学真AI

企业RAG技术实战

RAG（Retrieval-Augmented Generation）介绍

Retrieval-Augmented Generation for Large Language Models: A Survey：https://arxiv.org/abs/2312.10997

github项目：https://github.com/Tongji-KGLLM/RAG-Survey

RAGFlow项目

ragflow项目地址：https://github.com/infiniflow/ragflow

环境配置

wsl安装

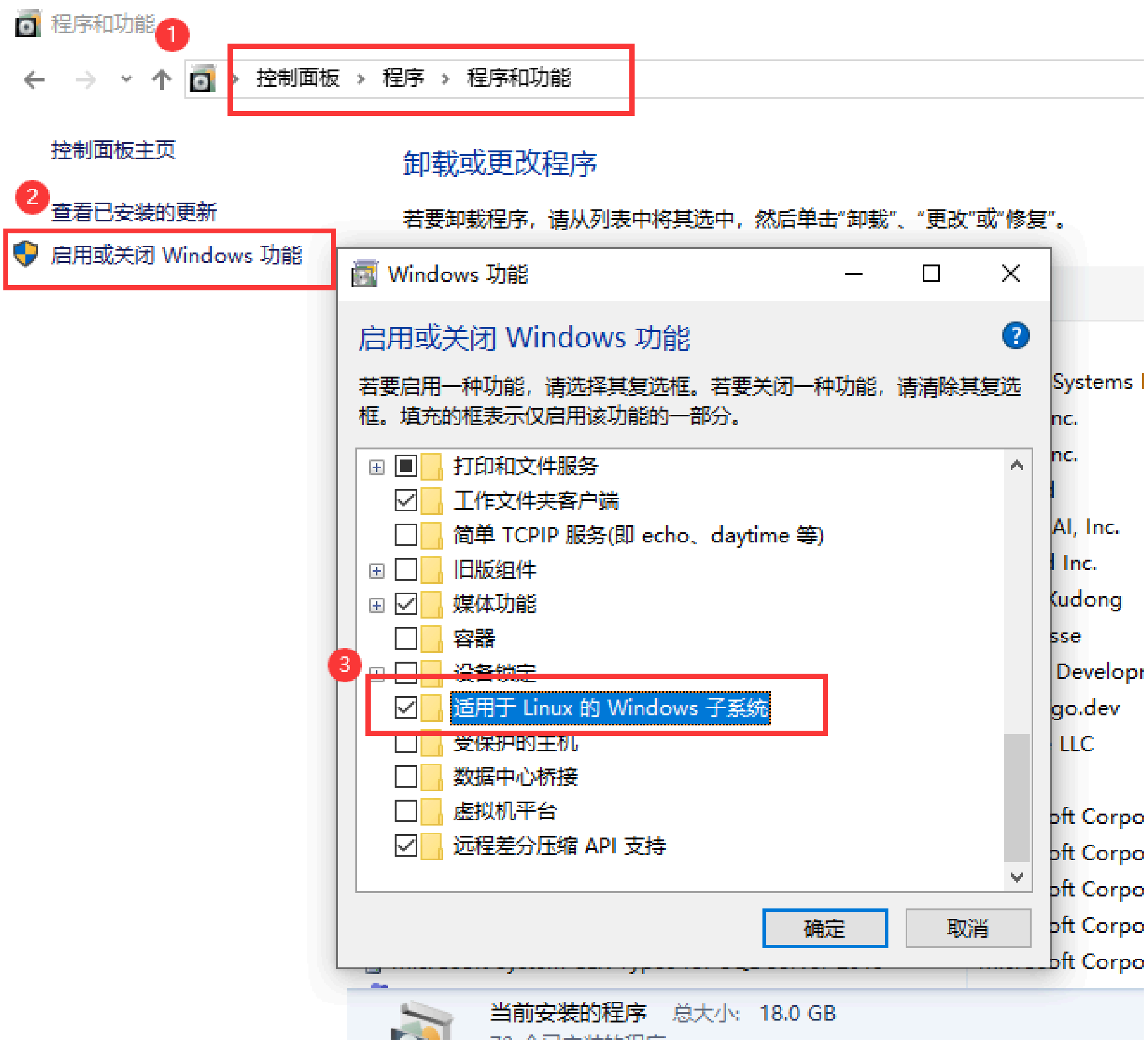
安装 WSL文档： https://learn.microsoft.com/zh-cn/windows/wsl/install

WSL基本命令：https://learn.microsoft.com/zh-cn/windows/wsl/basic-commands

1、管理员权限打开powershell命令

```
#启动wsl子系统
dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux
/all /norestart

#启用虚拟机平台支持
dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all
/norestart

#设置wsl2
wsl --set-default-version 2


#龙哥抖音号：龙哥紫貂智能
```

也可以手动选择：



程序和功能

控制面板 › 程序 › 程序和功能

控制面板主页

查看已安装的更新

启用或关闭 Windows 功能

卸载或更改程序

若要卸载程序，请从列表中将其选中，然后单击“卸载”、“更改”或“修复”。

Windows 功能

启用或关闭 Windows 功能

若要启用一种功能，请选择其复选框。若要关闭一种功能，请清除其复选框。填充的框表示仅启用该功能的一部分。

- ▪ 打印和文件服务
- ✓ 工作文件夹客户端
- ☐ 简单 TCPIP 服务(即 echo、daytime 等)
- ☐ 旧版组件
- ✓ 媒体功能
- ☐ 容器
- ☐ 设备锁定
- ✓ 适用于 Linux 的 Windows 子系统
- ☐ 受保护的主机
- ☐ 数据中心桥接
- ☐ 虚拟机平台
- ✓ 远程差分压缩 API 支持

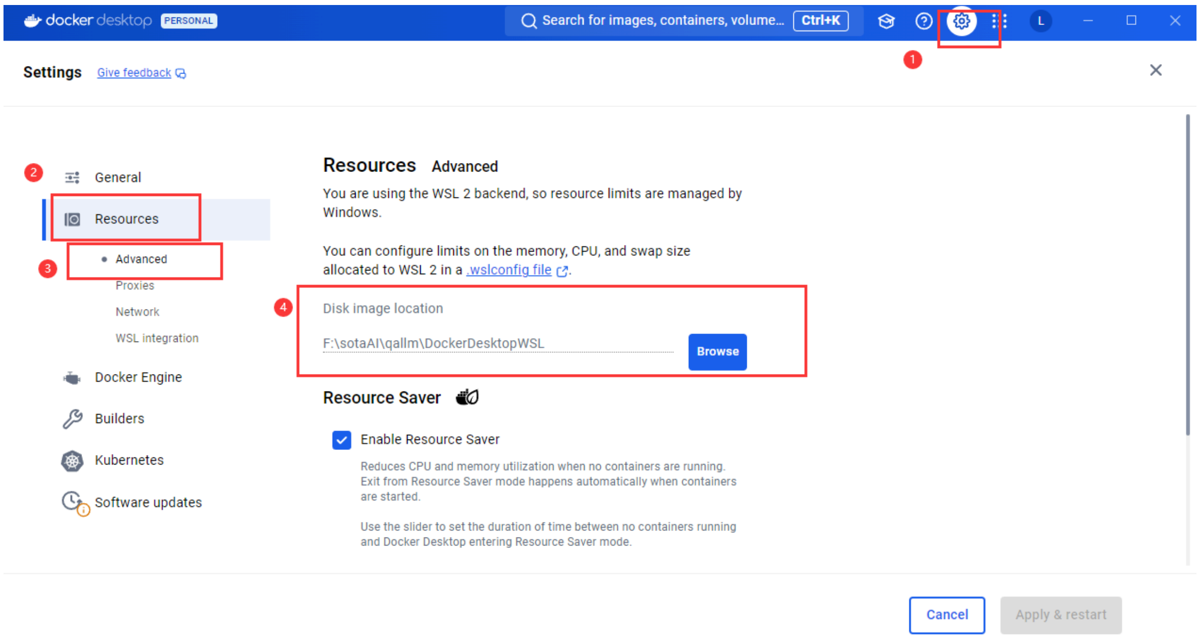
当前安装的程序 总大小: 18.0 GB

docker环境安装

下载安装路径：https://docs.docker.com/engine/install/

设置镜像本地路径

默认镜像拉取到本地后会保存在c:盘，我们设置到其它路径



Resources Advanced

You are using the WSL 2 backend, so resource limits are managed by Windows.

You can configure limits on the memory, CPU, and swap size allocated to WSL 2 in a .wslconfig file.

Disk image location

F:\sotaAI\qallm\DockerDesktopWSL Browse

Resource Saver

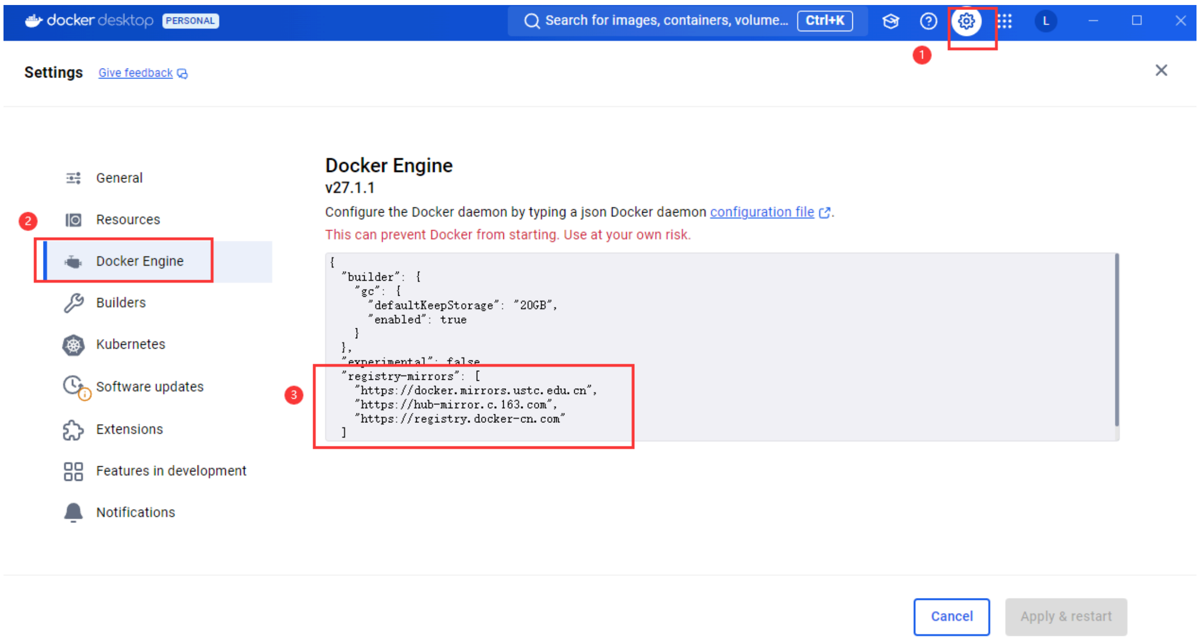
[✓] Enable Resource Saver

Reduces CPU and memory utilization when no containers are running. Exit from Resource Saver mode happens automatically when containers are started.

Use the slider to set the duration of time between no containers running and Docker Desktop entering Resource Saver mode.

设置镜像源

国外镜像源拉取比较慢，更新成国内镜像源



Docker Engine

v27.1.1

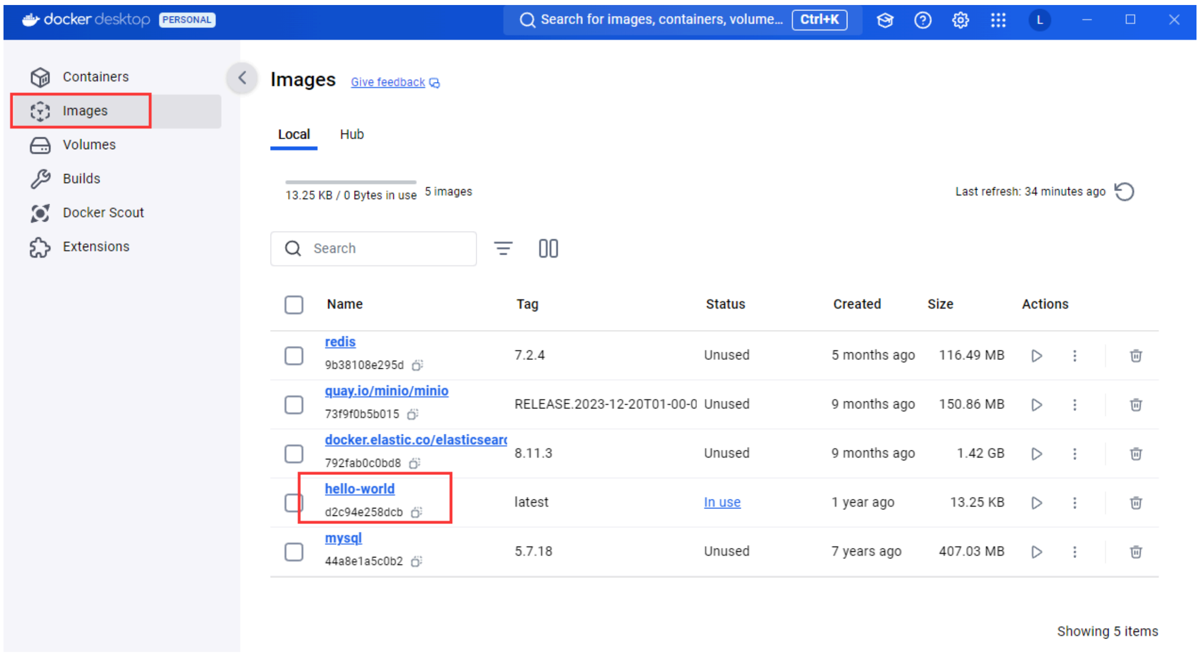
Configure the Docker daemon by typing a json Docker daemon configuration file.

This can prevent Docker from starting. Use at your own risk.

```
{
  "builder": {
    "gc": {
      "defaultKeepStorage": "20GB",
      "enabled": true
    }
  },
  "experimental": false
  "registry-mirrors": [
    "https://docker.mirrors.ustc.edu.cn",
    "https://hub-mirror.c.163.com",
    "https://registry.docker-cn.com"
  ]
```

测试docker是否正常

```
#cmd运行命令

docker run hello-world

#会拉取下来一个测试的hello-world镜像，在对应设置的镜像目录下也会看到，docker界面的
```



13.25 KB / 0 Bytes in use 5 images

Last refresh: 34 minutes ago

Name	Tag	Status	Created	Size	Actions
redis 9b38108e295d	7.2.4	Unused	5 months ago	116.49 MB	
quay.io/minio/minio 73f9f0b5b015	RELEASE.2023-12-20T01-00-0	Unused	9 months ago	150.86 MB	
docker.elastic.co/elasticsearc 792fab0c0bd8	8.11.3	Unused	9 months ago	1.42 GB	
hello-world d2c94e258dcb	latest	In use	1 year ago	13.25 KB	
mysql 44a8e1a5c0b2	5.7.18	Unused	7 years ago	407.03 MB	

Showing 5 items

安装运行RAGFlow

ragflow项目地址：https://github.com/infiniflow/ragflow

git把代码拉取下来

```
git clone https://github.com/infiniflow/ragflow.git
```

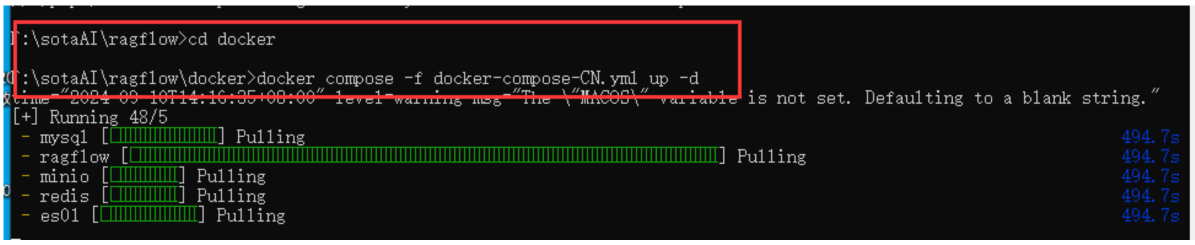
拉取代码后：

```
#cmd命令行切换到 ragflow\docker下,然后运行下面命令，一键自动下载项目的依赖镜像和环境

docker compose -f docker-compose-CN.yml up -d
```

> 请注意，运行上述命令会自动下载 RAGFlow 的开发版本 docker 镜像。如果你想下载并运行特定版本的 docker 镜像，请在 docker/.env 文件中找到 RAGFLOW_VERSION 变量，将其改为对应版本。例如 RAGFLOW_VERSION=v0.10.0，这个版本就是github的代码版本，然后运行上述命令。

> 核心镜像文件大约 9 GB，可能需要一定时间拉取。请耐心等待

这个过程会要一段时间，请耐心等待



```
:\sotaAI\ragflow>cd docker

:\sotaAI\ragflow\docker>docker compose -f docker-compose-CN.yml up -d
time="2024-09-10T14:16:35+08:00" level=warning msg="The \"MACOS\" variable is not set. Defaulting to a blank string."
[+] Running 48/5
 - mysql [||||||||||||||||] Pulling                                                          494.7s
 - ragflow [||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||] Pulling      494.7s
 - minio [||||||||||] Pulling                                                                494.7s
 - redis [||||||||||] Pulling                                                                494.7s
 - es01 [||||||||||||||] Pulling                                                             494.7s
```

启动

服务器启动成功后再次确认服务器状态：

```
docker logs -f ragflow-server
```

出现以下界面提示说明服务器启动成功：

```
    ____                 ______ __
   / __ \ ____ _ ____ _ / ____// /____  _      __
  / /_/ // __ `// __ `// /_   / // __ \| | /| / /
 / _, _// /_/ // /_/ // __/  / // /_/ /| |/ |/ /
/_/ |_| \__,_/ \__, //_/    /_/ \____/ |__/|__/
              /____/

 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:9380
 * Running on http://x.x.x.x:9380
 INFO:werkzeug:Press CTRL+C to quit
```

> 如果您跳过这一步系统确认步骤就登录 RAGFlow，你的浏览器有可能会提示 `network abnormal` 或 `网络异常`，因为 RAGFlow 可能并未完全启动成功。

登录

下面网站和端口即可打开系统

```
http://127.0.0.1:80
```

按照提示注册登录即可

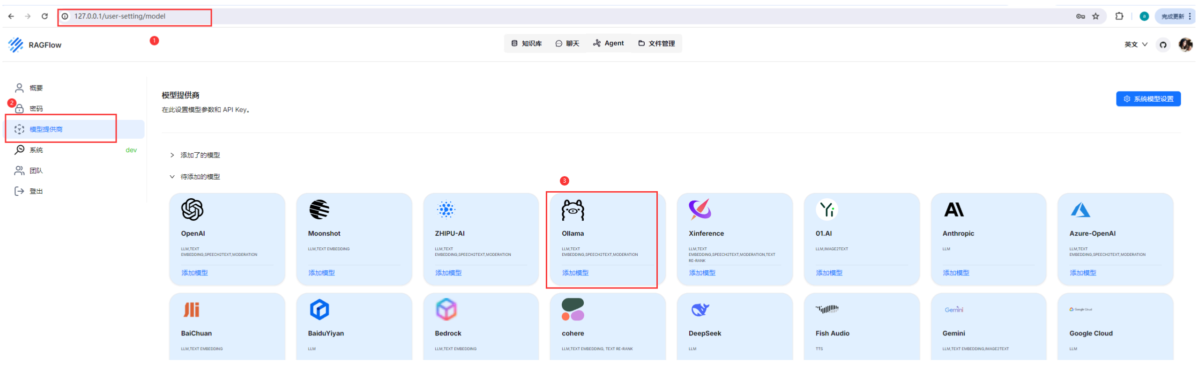
建立知识库与聊天

模型配置

正常我们选择本地ollama部署的大语言模型

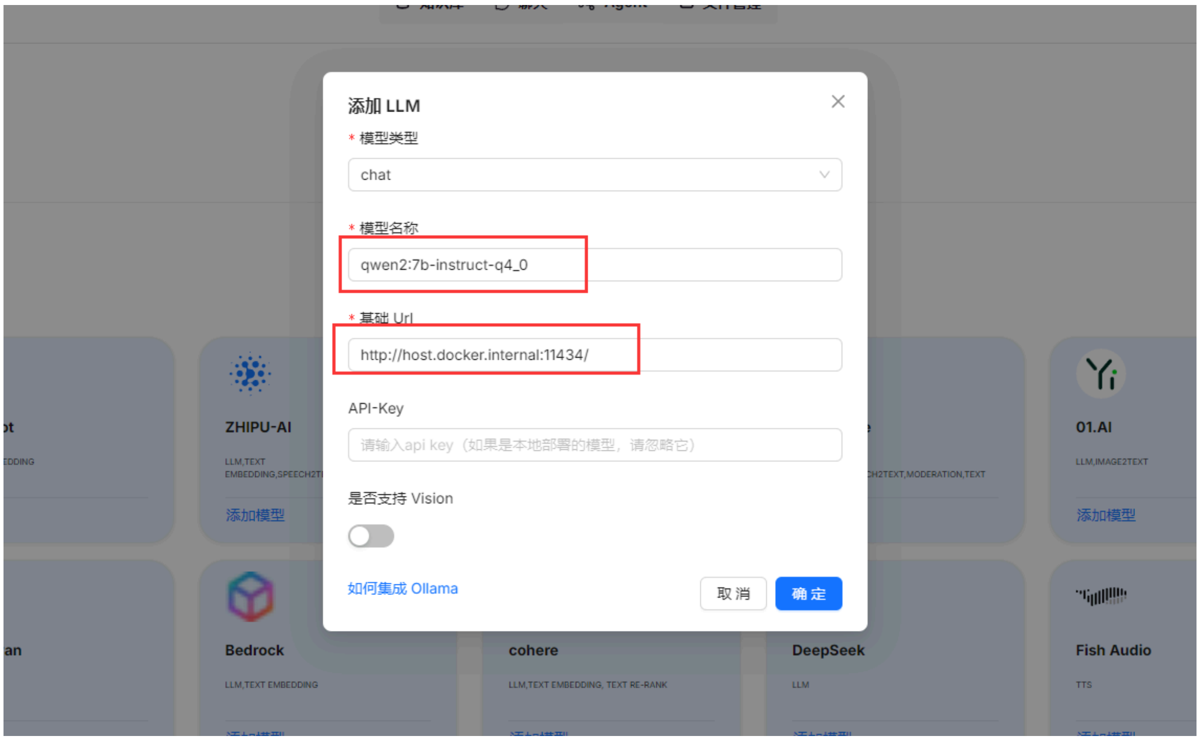
```
#默认的ollama是这个端口
http://127.0.0.1:11434

#项目是windows的docker启动，则使用
http://host.docker.internal:11434/

#模型
qwen2:7b-instruct-q4_0
```

界面演示如下


127.0.0.1/user-setting/model

模型提供商

在此设置模型参数和 API Key。


添加 LLM

* 模型类型

chat

* 模型名称

qwen2:7b-instruct-q4_0

* 基础 Url

http://host.docker.internal:11434/

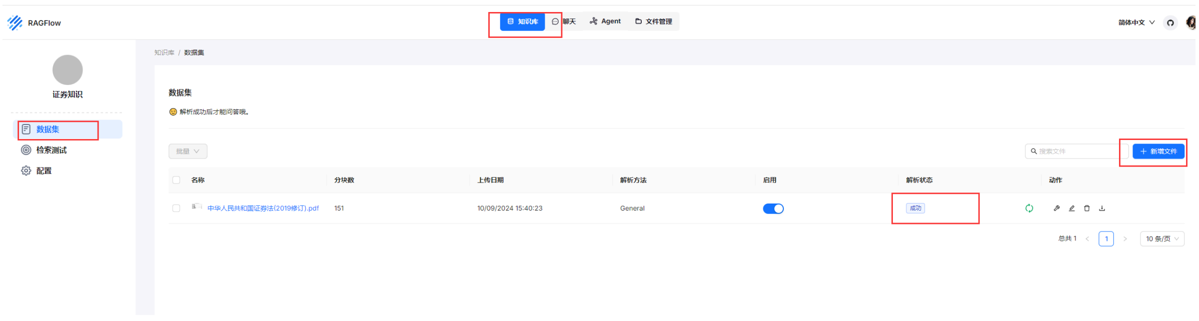
API-Key

请输入api key（如果是本地部署的模型，请忽略它）

是否支持 Vision

建立知识库

上传一个本地文件，一定要把文件解析好



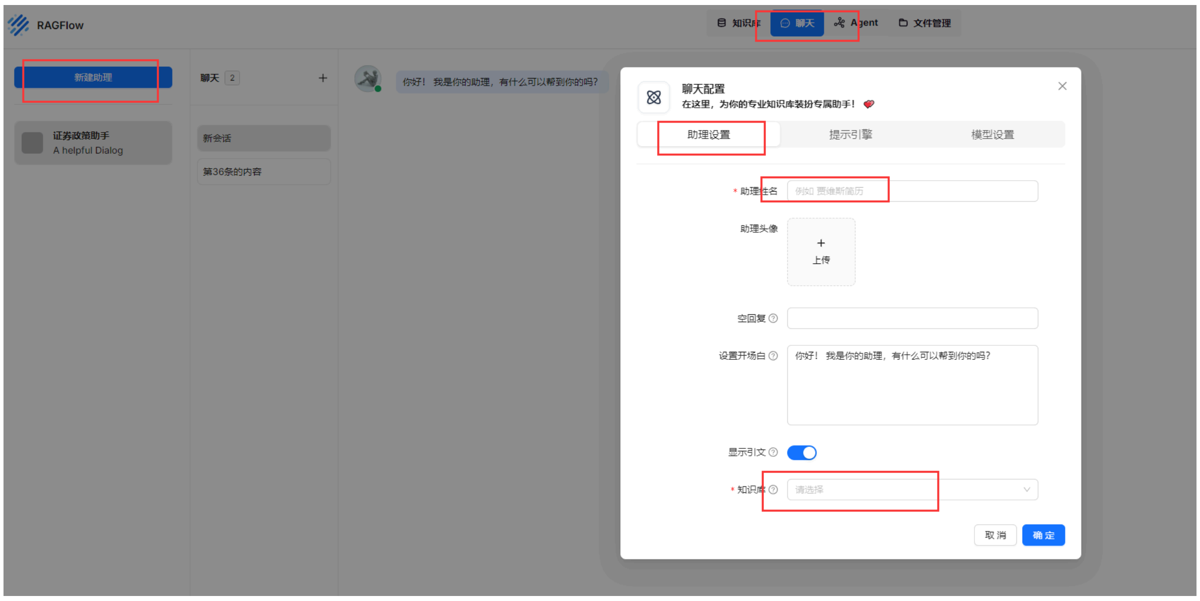
数据集

解析成功后才能问答哦。

名称	分块数	上传日期	解析方法	启用	解析状态	动作
中华人民共和国证券法(2019修订).pdf	151	10/09/2024 15:40:23	General	●	成功	

总共 1 ‹ 1 › 10 条/页

新建助理时，选择好对应的知识库



证券政策助手
A helpful Dialog

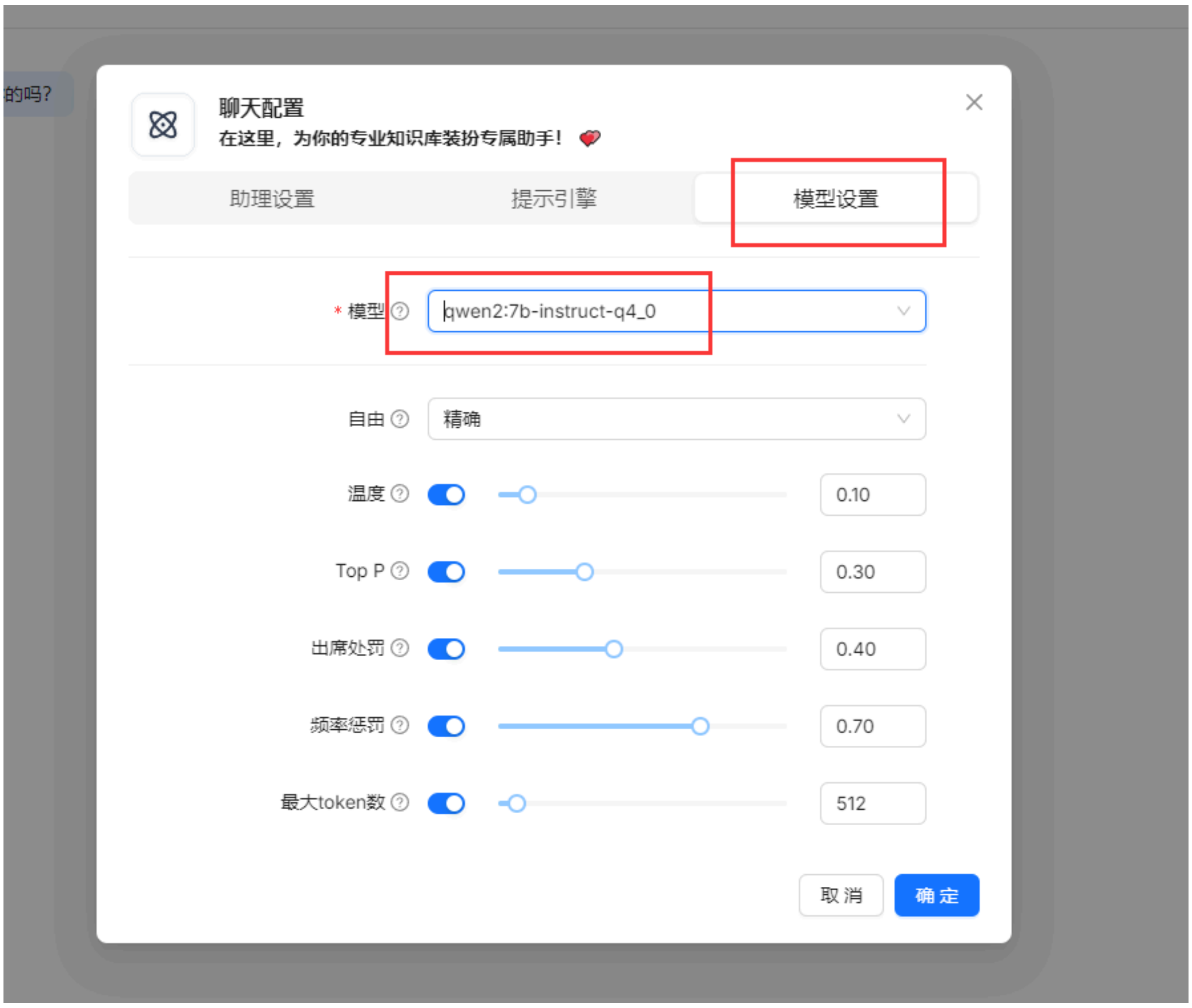
你好！ 我是你的助理，有什么可以帮到你的吗？

聊天配置
在这里，为你的专业知识库装扮专属助手！

助理设置 | 提示引擎 | 模型设置

* 助理姓名: 例如 贾维斯简历
助理头像: + 上传
空回复:
设置开场白: 你好！ 我是你的助理，有什么可以帮到你的吗？
显示引文: ●
* 知识库: 请选择

选择模型配置时的大模型


聊天配置

在这里，为你的专业知识库装扮专属助手！

* 模型 qwen2:7b-instruct-q4_0

自由 精确

温度 0.10

Top P 0.30

出席处罚 0.40

频率惩罚 0.70

最大token数 512

Naive RAG

使用langchain构建简单RAG

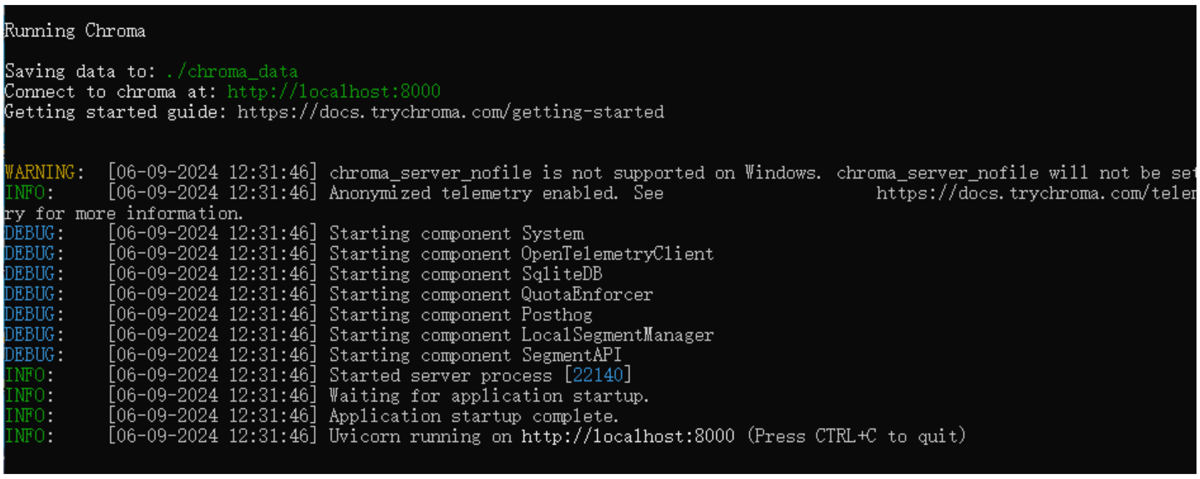
https://python.langchain.com/v0.2/docs/introduction/

https://www.trychroma.com/

环境准备

安装chroma

```
pip install chromadb #安装
chroma run    #运行
```



```
Running Chroma

Saving data to: ./chroma_data
Connect to chroma at: http://localhost:8000
Getting started guide: https://docs.trychroma.com/getting-started


WARNING:  [06-09-2024 12:31:46] chroma_server_nofile is not supported on Windows. chroma_server_nofile will not be set
INFO:     [06-09-2024 12:31:46] Anonymized telemetry enabled. See                    https://docs.trychroma.com/teler
ry for more information.
DEBUG:    [06-09-2024 12:31:46] Starting component System
DEBUG:    [06-09-2024 12:31:46] Starting component OpenTelemetryClient
DEBUG:    [06-09-2024 12:31:46] Starting component SqliteDB
DEBUG:    [06-09-2024 12:31:46] Starting component QuotaEnforcer
DEBUG:    [06-09-2024 12:31:46] Starting component Posthog
DEBUG:    [06-09-2024 12:31:46] Starting component LocalSegmentManager
DEBUG:    [06-09-2024 12:31:46] Starting component SegmentAPI
INFO:     [06-09-2024 12:31:46] Started server process [22140]
INFO:     [06-09-2024 12:31:46] Waiting for application startup.
INFO:     [06-09-2024 12:31:46] Application startup complete.
INFO:     [06-09-2024 12:31:46] Uvicorn running on http://localhost:8000 (Press CTRL+C to quit)
```

```
#conda创建  python=3.10版本的虚环境
conda create -n llmrag python=3.10
#激活conda创建的名字叫llmrag的虚环境
conda activate llmrag


#torch安装
conda install pytorch==2.3.1 torchvision==0.18.1 torchaudio==2.3.1 pytorch-
cuda=12.1 -c pytorch -c nvidia


#安装依赖

#用openai的模型
pip install openai

#如果是本地部署ollama,langchain 对ollama的支持
pip install -U langchain-ollama

#通义千问线上版
pip install -U langchain_openai


#支持chroma
pip install langchain_chroma
pip install -U langchain-community
```

使用openai的模型

如果使用openai的模型来生成，你需要在项目的根目录下的 .env 文件中设置相关的环境变量。要获取 OpenAI 的 API 密钥，你需要注册 OpenAI 账户，并在https://platform.openai.com/account/api-keys 页面中选择"创建新的密钥"。

```
OPENAI_API_KEY="<YOUR_OPENAI_API_KEY>"
```

完成这些设置后，运行下面的命令来加载你所设置的环境变量。

```
import dotenv
dotenv.load_dotenv()
```

通义千问

注册地址：https://help.aliyun.com/zh/dashscope/developer-reference/acquisition-and-configuration-of-api-key

兼容openai文档地址：https://help.aliyun.com/zh/dashscope/developer-reference/compatibility-of-openai-with-dashscope

```
DASHSCOPE_API_KEY="YOUR_DASHSCOPE_API_KEY"
```

完成这些设置后，运行下面的命令来加载你所设置的环境变量。

```
import dotenv
dotenv.load_dotenv()
```

使用ollama

则需要安装前面模型微调训练中的ollama服务搭建方式，把ollama服务跑起来，并下载好推理用的模型

```
#https://ollama.com/library/qwen2:7b-instruct-q4_0
#龙哥在这儿下载了一个  qwn2  7b的模型
ollama run qwen2:7b-instruct-q4_0
```

准备数据

完整代码

```
import requests
from langchain.document_loaders import WebBaseLoader

from langchain.text_splitter import RecursiveCharacterTextSplitter

from langchain.prompts import ChatPromptTemplate

from langchain.schema.runnable import RunnablePassthrough
from langchain.schema.output_parser import StrOutputParser

from langchain_ollama.llms import OllamaLLM

from langchain.chat_models import ChatOpenAI
from langchain.embeddings import OpenAIEmbeddings
```

```python
from langchain_chroma import Chroma
from langchain_community.embeddings import HuggingFaceBgeEmbeddings



#准备知识库数据，建索引
def prepare_data():

    loader = WebBaseLoader("https://baike.baidu.com/item/AIGC?
fromModule=lemma_search-box")
    documents = loader.load()
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=500,
chunk_overlap=50)
    chunks = text_splitter.split_documents(documents)

    print(chunks[0].page_content)

    return chunks

#embedding 知识库，保存到向量数据库
def embedding_data(chunks):

    #openai embedding
    #rag_embeddings=OpenAIEmbeddings()

    #创建BAAI的embedding
    rag_embeddings = HuggingFaceBgeEmbeddings(model_name="BAAI/bge-small-zh-
v1.5")

    #embed保存知识到向量数据库
    vector_store = Chroma.from_documents(documents=chunks,
embedding=rag_embeddings,persist_directory="./chroma_langchain_db")
    retriever = vector_store.as_retriever()
    return vector_store,retriever



#使用ollama服务
llm = OllamaLLM(model="qwen2:7b-instruct-q4_0")
template = """您是问答任务的助理。
使用以下检索到的上下文来回答问题。
如果你不知道答案，就说你不知道。
最多使用三句话，不超过100字，保持答案简洁。
Question: {question}
Context: {context}
Answer:
"""
prompt = ChatPromptTemplate.from_template(template)

chunks = prepare_data()

vector_store,retriever = embedding_data(chunks)


#生成答案
def generate_answer(question):
    '''
    #使用通义千问
    llm = ChatOpenAI(
            api_key=os.getenv("DASHSCOPE_API_KEY"), # 如果您没有配置环境变量，请在此处
用您的API Key进行替换
            base_url="https://dashscope.aliyuncs.com/compatible-mode/v1", # 填写
DashScope base_url
            model="qwen-plus"
            )
    '''


    #使用openai
    #llm = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0)

    rag_chain = (
        {"context": retriever,  "question": RunnablePassthrough()}
        | prompt
        | llm
        | StrOutputParser()
    )


    resp=rag_chain.invoke(question)
    print(resp)


query = "艾伦•图灵的论文叫什么"

generate_answer(query)
```

Advanced RAG

数据提取

提取方式

llamaindex reader文档：https://docs.llamaindex.ai/en/stable/api_reference/readers/

添加元数据

llamaindex metadata提取文档：https://docs.llamaindex.ai/en/stable/api_reference/extractors/

例子：https://docs.llamaindex.ai/en/stable/examples/metadata_extraction/EntityExtractionClimate/

```
EntityExtractor      #提取每个节点内容中提到的实体（如地点、人物、事物的名称）
KeywordExtractor
MarvinMetadataExtractor
QuestionsAnsweredExtractor    #提取每个节点可以回答的一组问题
RelikPathExtractor    #实体链接和关系提取
SummaryExtractor    #自动提取一组节点的摘要
TitleExtractor    #提取每个节点上下文的标题
```

```
from llama_index.core.extractors import (
    TitleExtractor,
    QuestionsAnsweredExtractor,
)
from llama_index.core.node_parser import TokenTextSplitter

# 定义文本分割器
text_splitter = TokenTextSplitter(
    separator=" ", chunk_size=512, chunk_overlap=128
)

# 定义标题提取器和问题提取器
title_extractor = TitleExtractor(nodes=5)
qa_extractor = QuestionsAnsweredExtractor(questions=3)

# 假设文档已定义 -> 提取节点
from llama_index.core.ingestion import IngestionPipeline

pipeline = IngestionPipeline(
    transformations=[text_splitter, title_extractor, qa_extractor]
)

nodes = pipeline.run(
    documents=documents,
    in_place=True,
    show_progress=True,
)
```

知识图谱

使用KG来组织多个文档，可以参考这篇研究论文KGP：Knowledge Graph Prompting for Multi-Document Question Answering ： https://arxiv.org/abs/2308.11730

llamaindex 知识图谱文档：https://docs.llamaindex.ai/en/stable/api_reference/indices/knowledge_graph/

```
#使用Llama-Index来连接到Neo4j，以构建和查询知识图谱,将文档中的信息转化为知识图谱
from llama_index.graph_stores.neo4j import Neo4jGraphStore
from llama_index.core import KnowledgeGraphIndex
```

```
username = "neo4j-xxx"
password = "neo4j-password-xxx"
url = "neo4j-url-xxxx:7687"
database = "neo4j"


# 初始化Neo4j图存储
graph_store = Neo4jGraphStore(
    username=username,
    password=password,
    url=url,
    database=database,
)

# 创建存储上下文
storage_context = StorageContext.from_defaults(graph_store=graph_store)

# 构建知识图谱索引
index = KnowledgeGraphIndex.from_documents(
    documents,
    storage_context=storage_context,
    max_triplets_per_chunk=2,
)


# 初始化一个空的索引
index = KnowledgeGraphIndex.from_documents([], storage_context=storage_context)

# 手动添加三元组
node_0_tups = [
    ("author", "worked on", "writing"),
    ("author", "worked on", "programming"),
]
for tup in node_0_tups:
    index.upsert_triplet_and_node(tup, nodes[0])
```

层次化索引结构

总分层级索引（总→细，提高搜索的效率）

利用元数据进行多层次过滤：https://docs.llamaindex.ai/en/stable/examples/query_engine/multi_doc_auto_retrieval/multi_doc_auto_retrieval/

```
from llama_index.core import SummaryIndex
from llama_index.core.async_utils import run_jobs
from llama_index.llms.openai import OpenAI
from llama_index.core.schema import IndexNode
from llama_index.core.vector_stores import (
```

```python
    FilterOperator,
    MetadataFilter,
    MetadataFilters,
)


async def aprocess_doc(doc, include_summary: bool = True):
    """Process doc."""
    metadata = doc.metadata

    date_tokens = metadata["created_at"].split("T")[0].split("-")
    year = int(date_tokens[0])
    month = int(date_tokens[1])
    day = int(date_tokens[2])

    assignee = (
        "" if "assignee" not in doc.metadata else doc.metadata["assignee"]
    )
    size = ""
    if len(doc.metadata["labels"]) > 0:
        size_arr = [l for l in doc.metadata["labels"] if "size:" in l]
        size = size_arr[0].split(":")[1] if len(size_arr) > 0 else ""
    new_metadata = {
        "state": metadata["state"],
        "year": year,
        "month": month,
        "day": day,
        "assignee": assignee,
        "size": size,
    }

    # 提取文档总结摘要
    summary_index = SummaryIndex.from_documents([doc])
    query_str = "Give a one-sentence concise summary of this issue."
    query_engine = summary_index.as_query_engine(
        llm=OpenAI(model="gpt-3.5-turbo")
    )
    summary_txt = await query_engine.aquery(query_str)
    summary_txt = str(summary_txt)

    index_id = doc.metadata["index_id"]
    # 通过 doc id过滤出对应的文档
    filters = MetadataFilters(
        filters=[
            MetadataFilter(
                key="index_id", operator=FilterOperator.EQ, value=int(index_id)
            ),
        ]
    )
    #创建的一个索引节点，包括有元数据和摘要总结文本
    index_node = IndexNode(
        text=summary_txt,   #总结
        metadata=new_metadata,   #包括年月日，状态等
        obj=doc_index.as_retriever(filters=filters),
        index_id=doc.id_,   #文档id
    )
    return index_node


async def aprocess_docs(docs):
    """Process metadata on docs."""

    index_nodes = []
    tasks = []
    for doc in docs:   #遍历所有文档处理
        task = aprocess_doc(doc)
        tasks.append(task)

    index_nodes = await run_jobs(tasks, show_progress=True, workers=3)

    return index_nodes
```

父子层级索引（细→总，提高搜索精确问题的准确性）

```
'''
对于每个块大小为 1024 的文本块，我们创建更小的文本块：

8 个文本块，大小为 128
4 个大小为 256 的文本块
2 个大小为 512 的文本块
我们将大小为 1024 的原始文本块附加到文本块列表中
'''


sub_chunk_sizes = [ 128 , 256 , 512 ]
sub_node_parsers = [
    SimpleNodeParser.from_defaults(chunk_size=c) for c in sub_chunk_sizes
]

all_nodes = []
for base_node in base_nodes:
    for n in sub_node_parsers:
        sub_nodes = n.get_nodes_from_documents([base_node])
        sub_inodes = [
            IndexNode.from_text_node(sn, base_node.node_id) for sn in sub_nodes
        ]
        all_nodes.extend(sub_inodes)

    # 还将原始节点添加到节点
    original_node = IndexNode.from_text_node(base_node, base_node.node_id)
    all_nodes.append(original_node)
all_nodes_dict = {n.node_id: n for n in all_nodes}
```

TreeIndex

```
TreeIndex
TreeAllLeafRetriever   #从叶节点构建查询特定的树并返回响应。
TreeSelectLeafEmbeddingRetriever   #利用嵌入相似性在索引图中遍历节点并选择最佳叶节点。
TreeSelectLeafRetriever   #遍历索引图并搜索能够最好回答查询的叶节点。
TreeRootRetriever   #直接从根节点检索答案


#龙哥抖音号：龙哥紫貂智能
```

句子窗口索引

```
# 使用默认设置创建句子窗口节点解析器
node_parser = SentenceWindowNodeParser.from_defaults(
    window_size= 3 ,
    window_metadata_key= "window" ,
    original_text_metadata_key= "original_text" ,
)
sentence_nodes = node_parser.get_nodes_from_documents(docs)
sentence_index = VectorStoreIndex(sentence_nodes,
service_context=service_context)
```

多种切分方式并行查询

并行优化例子:

https://docs.llamaindex.ai/en/stable/examples/ingestion/parallel_execution_ingestion_pipeline/

```
from llama_index.core import Document
from llama_index.embeddings.openai import OpenAIEmbedding
from llama_index.core.node_parser import SentenceSplitter
from llama_index.core.extractors import TitleExtractor
from llama_index.core.ingestion import IngestionPipeline

# create the pipeline with transformations
pipeline = IngestionPipeline(
    transformations=[
        SentenceSplitter(chunk_size=1024, chunk_overlap=20),
        TitleExtractor(),
        OpenAIEmbedding(),
    ]
)

# since we'll be testing performance, using timeit and cProfile
# we're going to disable cache
pipeline.disable_cache = True

nodes = pipeline.run(documents=documents, num_workers=4)   #四个线程并行,这个
pipeline是做embedding，而不是做查询
```

多chunks大小索引和查询：

https://docs.llamaindex.ai/en/stable/examples/retrievers/ensemble_retrieval/

```
# initialize modules
llm = OpenAI(model="gpt-4")
chunk_sizes = [128, 256, 512, 1024]
nodes_list = []
vector_indices = []
for chunk_size in chunk_sizes:
    print(f"Chunk Size: {chunk_size}")
    splitter = SentenceSplitter(chunk_size=chunk_size)
    nodes = splitter.get_nodes_from_documents(docs)

    # add chunk size to nodes to track later
    for node in nodes:
        node.metadata["chunk_size"] = chunk_size
        node.excluded_embed_metadata_keys = ["chunk_size"]
        node.excluded_llm_metadata_keys = ["chunk_size"]

    nodes_list.append(nodes)

    # build vector index
    vector_index = VectorStoreIndex(nodes)
    vector_indices.append(vector_index)
```

预检索过程（Pre-Retrieval Process)

提示词优化

https://docs.llamaindex.ai/en/stable/examples/prompts/advanced_prompts/

提示词改写

这有一篇论文：Query Rewriting for Retrieval-Augmented Large Language Models https://arxiv.org/pdf/2305.14283

子查询

llamaindex 查询引擎例子：https://docs.llamaindex.ai/en/stable/examples/query_engine/multi_doc_auto_retrieval/multi_doc_auto_retrieval/

```
from llama_index.query_engine import SubQuestionQueryEngine
from llama_index.question_generation import BaseQuestionGenerator
from llama_index.response_synthesizers import ResponseSynthesizer
from llama_index.retrievers import BaseRetriever
from typing import Sequence

# 假设我们已经有了一个基础的 QuestionGenerator 和 ResponseSynthesizer
```

```
question_gen: BaseQuestionGenerator = ...
response_synthesizer: ResponseSynthesizer = ...
query_engine_tools: Sequence[BaseRetriever] = ...

# 创建子问题查询引擎
sub_question_query_engine = SubQuestionQueryEngine(
    question_gen=question_gen,
    response_synthesizer=response_synthesizer,
    query_engine_tools=query_engine_tools
)

# 使用子问题查询引擎进行查询
response = sub_question_query_engine.query("比较和对比苹果和橙子。")
print(response)
```

HyDE（假设答案）

有篇论文 Precise Zero-Shot Dense Retrieval without Relevance Labels： https://arxiv.org/pdf/2212.10496

```
from llama_index.core.indices.query.query_transform import HyDEQueryTransform
from llama_index.core.query_engine import TransformQueryEngine
from IPython.display import Markdown, display

index = VectorStoreIndex.from_documents(documents)
query_engine = index.as_query_engine()
hyde = HyDEQueryTransform(include_original=True)
hyde_query_engine = TransformQueryEngine(query_engine, hyde)
response = hyde_query_engine.query(query_str)
display(Markdown(f"<b>{response}</b>"))

query_bundle = hyde(query_str)
hyde_doc = query_bundle.embedding_strs[0]
```

Reverse HyDE（假设问题）

CoVe

使用Meta AI 提出的Chain-of-Verification：https://arxiv.org/abs/2309.11495，

检索过程（Retrieval）

在条件成本允许的情况下，可以适当地进行微调，从而提升其在垂直领域检索效果。以下是部分文档中介绍的微调方式，我们后面课程会详细讲

- LlamaIndex Embedding 微调方式：https://github.com/run-llama/finetune-embedding/blob/main/evaluate.ipynb
- Retrieval Augmentation 微调:https://docs.llamaindex.ai/en/stable/examples/finetuning/knowledge/finetune_retrieval_aug.html#fine-tuning-with-retrieval-augmentation

- cross-encoder 交叉编码微调：https://docs.llamaindex.ai/en/latest/examples/finetuning/cross_encoder_finetuning/cross_encoder_finetuning.html#

后检索过程（Post-Retrieval Process）

提示词压缩（Prompt Compression）

llamaindex关于提示词的文档：https://docs.llamaindex.ai/en/stable/examples/prompts/advanced_prompts/

微软的LLMLinggua项目：https://github.com/microsoft/LLMLingua

LLMLingua

论文：https://aclanthology.org/2023.emnlp-main.825.pdf

LongLLMLingua

论文：https://aclanthology.org/2024.acl-long.91.pdf

LongLLMLingua2

论文：https://aclanthology.org/2024.findings-acl.57.pdf

```
# Setup LLMLingua
from llama_index.query_engine import RetrieverQueryEngine
from llama_index.response_synthesizers import CompactAndRefine
from llama_index.indices.postprocessor import LongLLMLinguaPostprocessor

node_postprocessor = LongLLMLinguaPostprocessor(
    instruction_str="Given the context, please answer the final question",
    target_token=300,
    rank_method="longllmlingua",
    additional_compress_kwargs={
        "condition_compare": True,
        "condition_in_question": "after",
        "context_budget": "+100",
        "reorder_context": "sort",  # enable document reorder,
        "dynamic_context_compression_ratio": 0.3,
    },
)
```

使用llamaindex实例

https://docs.llamaindex.ai/en/stable/api_reference/

环境配置

我们继续使用前面langchain例子的python虚环境，不用新建，激活就行

不同LLM环境配置

```
#conda创建  python=3.10版本的虚环境
#conda create -n llmrag python=3.10
#激活conda创建的名字叫llmrag的虚环境
conda activate llmrag


#torch安装
conda install pytorch==2.3.1 torchvision==0.18.1 torchaudio==2.3.1 pytorch-
cuda=12.1 -c pytorch -c nvidia

#安装依赖
pip install llama_hub llama_index llama-index-readers-web trafilatura
pip install llama-index-vector-stores-chroma    #支持chroma向量数据库
pip install llama-index-embeddings-huggingface



#用openai的模型
pip install openai

#如果是本地部署ollama
pip install llama-index-llms-ollama

#通义千问线上版
pip install llama-index-llms-dashscope
```

api_key设置

api_key环境变量和langchain项目方法一样

```
DASHSCOPE_API_KEY="YOUR_DASHSCOPE_API_KEY"

import dotenv
dotenv.load_dotenv()
```

安装chroma

安装chroma轻量级向量数据库，因为它轻量并且支持windows，不需要wsl，不需要docker

```
pip install chromadb #安装
chroma run   #运行
```

准备数据

在这个例子以及后面的例子中，语料库我们都将使用百度百科关于aigc的知识：https://baike.baidu.com/item/AIGC?fromModule=lemma_search-box

完整代码

```
from llama_index.readers.web import TrafilaturaWebReader
from llama_index.core.node_parser import SimpleNodeParser
from llama_index.core.schema import IndexNode




from llama_index.embeddings.huggingface import HuggingFaceEmbedding
from llama_index.core import VectorStoreIndex, StorageContext
#from llama_index.llms.openai import OpenAI
#from llama_index.embeddings.openai import OpenAIEmbedding
from llama_index.llms.ollama import Ollama

from llama_index.core.node_parser import SentenceSplitter
from llama_index.core import Settings
import chromadb


from llama_index.vector_stores.chroma import ChromaVectorStore

def prepare_data():
    url="https://baike.baidu.com/item/AIGC?fromModule=lemma_search-box"
    docs = TrafilaturaWebReader().load_data([url])
    return docs



#embed保存知识到向量数据库

def embedding_data(docs):
    #向量数据库客户端
    chroma_client = chromadb.EphemeralClient()
    chroma_collection = chroma_client.create_collection("quickstart")

    #向量数据库，指定了存储位置
    vector_store =
ChromaVectorStore(chroma_collection=chroma_collection,persist_dir="./chroma_langc
hain_db")
    storage_context = StorageContext.from_defaults(vector_store=vector_store)

    #创建文档切割器
    node_parser = SimpleNodeParser.from_defaults(chunk_size=500,chunk_overlap=50)

    #创建BAAI的embedding
    embed_model = HuggingFaceEmbedding(model_name="BAAI/bge-small-zh-v1.5")
    #创建index
    base_index = VectorStoreIndex.from_documents(documents=docs,transformations=
[node_parser],storage_context=storage_context, embed_model=embed_model)
```

```
    return base_index,embed_model

#龙哥抖音号：龙哥紫貂智能


def get_llm():

    #创建OpenAI的llm
    #llm = OpenAI(model="gpt-3.5-turbo")

    #通义千问
    '''
    from llama_index.llms.dashscope import DashScope, DashScopeGenerationModels
    llm = DashScope(model_name=DashScopeGenerationModels.QWEN_MAX)
    '''

    #ollama本地模型
    llm = Ollama(model="qwen2:7b-instruct-q4_0", request_timeout=120.0)

    #创建谷歌gemini的llm
    # llm = Gemini()

    return llm

def retrieve_data(question):
    #创建检索器
    base_retriever = base_index.as_retriever(similarity_top_k=2)

    #检索相关文档
    retrievals = base_retriever.retrieve(
        question
    )

    #print(retrievals)


    #https://docs.llamaindex.ai/en/stable/examples/low_level/response_synthesis/


    from llama_index.core.response.notebook_utils import display_source_node

    for n in retrievals:
        display_source_node(n, source_length=1500)

    return retrievals


def generate_answer(question):

    query_engine = base_index.as_query_engine()

    #大语言模型的回答
    response = query_engine.query(
       question
    )
    print(str(response))


question="艾伦•图灵的论文叫什么"
docs=prepare_data()
llm=get_llm()
base_index,embed_model=embedding_data(docs)

#通过设置来配置  llm,embedding等等
Settings.llm = llm
Settings.embed_model = embed_model
#Settings.node_parser = SentenceSplitter(chunk_size=512, chunk_overlap=20)
Settings.num_output = 512
Settings.context_window = 3000


retrieve_data(question)
generate_answer(question)
```

Modular RAG

论文Modular RAG: Transforming RAG Systems into LEGO-like Reconfigurable Frameworks：https://arxiv.org/pdf/2407.21059

推理阶段

网易开源的QAnything

比如网易开源的QAnything：https://github.com/netease-youdao/QAnything

QAnything使用的检索组件BCEmbedding：https://github.com/netease-youdao/BCEmbedding有非常强悍的双语和跨语种能力

bce-embedding-base_v1和bce-reranker-base_v1的组合是SOTA

重写-检索-阅读（RRR）

重写-检索-阅读（RRR）也是典型的顺序结构（https://arxiv.org/pdf/2305.14283.pdf）。

条件模式

条件 RAG 的经典实现是semantic-router这个项目：https://github.com/aurelio-labs/semantic-router

llamaindex实现例子代码

```
#定义查询引擎和工具

from llama_index.core.tools import QueryEngineTool

list_query_engine = summary_index.as_query_engine(
    response_mode="tree_summarize", use_async=True
)
vector_query_engine = vector_index.as_query_engine(
    response_mode="tree_summarize", use_async=True
)


#description，是QueryEngineTool元数据，用途描述。这有助于路由器查询引擎根据
ToolRetrieverRouterQueryEngine的查询路由那个tool
list_tool = QueryEngineTool.from_defaults(
    query_engine=list_query_engine,
    description="Useful for questions asking for a biography of the author.",
)
vector_tool = QueryEngineTool.from_defaults(
    query_engine=vector_query_engine,
    description=(
        "Useful for retrieving specific snippets from the author's life, like"
        " his time in college, his time in YC, or more."
    ),
)


#定义路由查询引擎
from llama_index.core import VectorStoreIndex
from llama_index.core.objects import ObjectIndex

#从这2个中路由
obj_index = ObjectIndex.from_objects(
    [list_tool, vector_tool],
    index_cls=VectorStoreIndex,
)

from llama_index.core.query_engine import ToolRetrieverRouterQueryEngine

query_engine = ToolRetrieverRouterQueryEngine(obj_index.as_retriever())

#根据query，和description进行相似度比较
response = query_engine.query("What is a biography of the author's life?")
print(str(response))
```

迭代检索

迭代检索的一个典型案例是ITER-RETGEN ：https://arxiv.org/abs/2305.15294，它迭代检索增强和生成增强。

递归检索

1、澄清树（TOC)

论文Tree of Clarifications: Answering Ambiguous Questions with Retrieval-Augmented Large Language Models：https://arxiv.org/pdf/2310.14696

llamaindex文档：https://docs.llamaindex.ai/en/stable/examples/query_engine/pdf_tables/recursive_retriever/

```
from llama_index.core.retrievers import RecursiveRetriever
from llama_index.core.query_engine import RetrieverQueryEngine
from llama_index.core import get_response_synthesizer

recursive_retriever = RecursiveRetriever(
    "vector",
    retriever_dict={"vector": vector_retriever},
    query_engine_dict=df_id_query_engine_mapping,
    verbose=True,
)

response_synthesizer = get_response_synthesizer(response_mode="compact")

query_engine = RetrieverQueryEngine.from_args(
    recursive_retriever, response_synthesizer=response_synthesizer
)

response = query_engine.query(
    "What's the net worth of the second richest billionaire in 2023?"
)
```

自适应（主动）检索

1、基于提示词方法：

一个典型的实现示例是　前瞻式主动检索增强生成 FLARE（Forward-Looking Active Retrieval Augmented Generation）：

https://blog.lancedb.com/better-rag-with-active-retrieval-augmented-generation-flare-3b66646e2a9f/

论文Active Retrieval Augmented Generation：https://arxiv.org/pdf/2305.06983

llamaindex实现了FLARE instruct

```
from llama_index.llms import OpenAI
from llama_index.query_engine import FLAREInstructQueryEngine
from llama_index import (
    VectorStoreIndex,
    SimpleDirectoryReader,
    ServiceContext,
)
### Recipe
### Build a FLAREInstructQueryEngine which has the generator LLM play
### a more active role in retrieval by prompting it to elicit retrieval
### instructions on what it needs to answer the user query.

# Build FLAREInstructQueryEngine
documents = SimpleDirectoryReader("./data/paul_graham").load_data()
index = VectorStoreIndex.from_documents(documents)
index_query_engine = index.as_query_engine(similarity_top_k=2)
```

```
service_context = ServiceContext.from_defaults(llm=OpenAI(model="gpt-4"))
flare_query_engine = FLAREInstructQueryEngine(
    query_engine=index_query_engine,
    service_context=service_context,
    max_iterations=7,    #迭代搜索7次
    verbose=True,
)

# Use your advanced RAG
response = flare_query_engine.query(
    "Can you tell me about the author's trajectory in the startup world?"
)
```

langchain中实现了 FlareChain

下面是FLARE Direct 例子代码

```
from langchain import PromptTemplate, LLMChain
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.chains import RetrievalQA
from langchain.embeddings import HuggingFaceBgeEmbeddings
from langchain.document_loaders import PyPDFLoader
from langchain.vectorstores import LanceDB
from langchain.document_loaders import ArxivLoader
from langchain.chains import FlareChain
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain
import os
import gradio as gr
import lancedb
from io import BytesIO
from langchain.llms import OpenAI
import getpass

# pass your api key
os.environ["OPENAI_API_KEY"] = "sk-yourapikeyforopenai"

llm = OpenAI()

os.environ["OPENAI_API_KEY"] = "sk-yourapikeyforopenai"
llm = OpenAI()
model_name = "BAAI/bge-large-en"
model_kwargs = {'device': 'cpu'}
encode_kwargs = {'normalize_embeddings': False}
embeddings = HuggingFaceBgeEmbeddings(
    model_name=model_name,
    model_kwargs=model_kwargs,
    encode_kwargs=encode_kwargs
)
# here is example https://arxiv.org/pdf/2305.06983.pdf
# you need to pass this number to query 2305.06983
# fetch docs from arxiv, in this case it's the FLARE paper
```

```
docs = ArxivLoader(query="2305.06983", load_max_docs=2).load()
# instantiate text splitter
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1500,
chunk_overlap=150)
# split the document into chunks
doc_chunks = text_splitter.split_documents(docs)
# lancedb vectordb
db = lancedb.connect('/tmp/lancedb')
table = db.create_table("documentsai", data=[
    {"vector": embeddings.embed_query("Hello World"), "text": "Hello World",
"id": "1"}
], mode="overwrite")
vector_store = LanceDB.from_documents(doc_chunks, embeddings, connection=table)
vector_store_retriever = vector_store.as_retriever()
flare = FlareChain.from_llm(
    llm=llm,
    retriever=vector_store_retriever,
    max_generation_len=300,
    min_prob=0.45     #任何以低于此概率生成的token都将被视为不确定
)
# Define a function to generate FLARE output based on user input
def generate_flare_output(input_text):
    output = flare.run(input_text)
    return output
input = gr.Text(
                label="Prompt",
                show_label=False,
                max_lines=1,
                placeholder="Enter your prompt",
                container=False,
            )
iface = gr.Interface(fn=generate_flare_output,
             inputs=input,
             outputs="text",
             title="My AI bot",
             description="FLARE implementation with lancedb & bge embedding.",
             allow_screenshot=False,
             allow_flagging=False
             )
iface.launch(debug=True)
```

2、Tuning-base：

一个典型案例是 Self-RAG（SELF-RAG: LEARNING TO RETRIEVE, GENERATE, AND CRITIQUE THROUGH SELF-REFLECTION）

论文：https://arxiv.org/pdf/2310.11511

github：https://selfrag.github.io/

微调后的模型

https://huggingface.co/selfrag/selfrag_llama2_7b

llamaindex self_rag 例子代码：首次执行时需要下载 SelfRAGPack

```
import os
os.environ["OPENAI_API_KEY"] = "YOUR_OPENAI_API_KEY"

from llama_index.core import Document, VectorStoreIndex
from llama_index.core.retrievers import VectorIndexRetriever
from llama_index.core.readers import SimpleDirectoryReader
from pathlib import Path


# Option: download SelfRAGPack
# The first execution requires the download of SelfRAGPack.
# Subsequent executions can comment this out.
from llama_index.core.llama_pack import download_llama_pack
download_llama_pack(
 "SelfRAGPack",
 "./self_rag_pack")

from llama_index.packs.self_rag import SelfRAGQueryEngine

documents = SimpleDirectoryReader("./data").load_data()
index = VectorStoreIndex.from_documents(documents)

# Setup a simple retriever
retriever = VectorIndexRetriever(
    index=index,
    similarity_top_k=10,
)

model_path = "/mypath/selfrag_llama2_7b.gguf"
query_engine = SelfRAGQueryEngine(str(model_path), retriever, verbose=True)
response = query_engine.query("历史上奥运金牌最多的国家?")
```

langchain例子深入

让LLM在用户的查询语句的基础上再生成多个查询语句，这些LLM生成的查询语句是从不同角度，不同视角对用户查询语句的补充：

```
# Multi Query: Different Perspectives
template = """You are an AI language model assistant. Your task is to generate
five
different versions of the given user question to retrieve relevant documents from
a vector
database. By generating multiple perspectives on the user question, your goal is
to help
the user overcome some of the limitations of the distance-based similarity
search.
```

```
Provide these alternative questions separated by newlines. Original question:
{question}"""

#
#generate five different versions
#

prompt_perspectives = ChatPromptTemplate.from_template(template)
generate_queries = (
    prompt_perspectives
    | llm
    | StrOutputParser()
    | (lambda x: x.split("\n"))
)

response = generate_queries.invoke({"question":'艾伦•图灵的论文叫什么'})
print(response)
```

LLM在此基础上又生成了5个问题：

```
['1. 请问艾伦·图灵的学术文章标题是什么？', '2. 想了解艾伦·图灵发表过的主要研究作品，能告诉我名称吗？', '3. 能提供一下艾伦·图灵的研究论文的具体题名吗？', '4. 我在找关于艾伦·图灵的一篇重要学术文章，它的题目是怎样的呢？', '5. 求告知艾伦·图灵的著名论
文的名称。']
```

多重查询

完整代码

```

import requests
from langchain.document_loaders import WebBaseLoader

from langchain.text_splitter import RecursiveCharacterTextSplitter

from langchain.prompts import ChatPromptTemplate

from langchain.schema.runnable import RunnablePassthrough
from langchain.schema.output_parser import StrOutputParser

from langchain_ollama.llms import OllamaLLM

from langchain.chat_models import ChatOpenAI
from langchain.embeddings import OpenAIEmbeddings


from langchain_chroma import Chroma
from langchain_community.embeddings import HuggingFaceBgeEmbeddings

from langchain.load import dumps, loads

import operator
```

```python
#准备知识库数据，建索引
def prepare_data():

    loader = WebBaseLoader("https://baike.baidu.com/item/AIGC?
fromModule=lemma_search-box")
    documents = loader.load()
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=500,
chunk_overlap=50)
    chunks = text_splitter.split_documents(documents)

    print(chunks[0].page_content)

    return chunks




#embedding 知识库，保存到向量数据库
def embedding_data(chunks):

    #openai embedding
    #rag_embeddings=OpenAIEmbeddings()

    #创建BAAI的embedding
    rag_embeddings = HuggingFaceBgeEmbeddings(model_name="BAAI/bge-small-zh-
v1.5")

    #embed保存知识到向量数据库
    vector_store = Chroma.from_documents(documents=chunks,
embedding=rag_embeddings,persist_directory="./chroma_langchain_db")
    retriever = vector_store.as_retriever()
    return vector_store,retriever

#龙 哥抖音号：龙 哥紫 貂智能

#获取多查询，检索出知识，然后RRF排序融合
def get_multiple_queries(question):
    # Multi Query: Different Perspectives
    template = """You are an AI language model assistant. Your task is to
generate five
    different versions of the given user question to retrieve relevant documents
from a vector
    database. By generating multiple perspectives on the user question, your goal
is to help
    the user overcome some of the limitations of the distance-based similarity
search.
    Provide these alternative questions separated by newlines. Original question:
{question}"""

    prompt_perspectives = ChatPromptTemplate.from_template(template)
    generate_queries = (
        prompt_perspectives
        | llm
        | StrOutputParser()
        | (lambda x: x.split("\n"))
    )
    #生成多个查询
    response = generate_queries.invoke({"question":question})
    print(response)

    #检索出知识
    all_results = retrieval_and_rank(response)

    #RRF融合
    reranked_results = reciprocal_rank_fusion(all_results)

    return generate_queries,reranked_results




def get_unique_union(documents: list[list]):
    """ Unique union of retrieved docs """
    # Flatten list of lists, and convert each Document to string
    flattened_docs = [dumps(doc) for sublist in documents for doc in sublist]
    # Get unique documents
    unique_docs = list(set(flattened_docs))
    # Return
    return [loads(doc) for doc in unique_docs]




#生成答案
def generate_answer(question):
    '''
    #使用通义千问
    llm = ChatOpenAI(
            api_key=os.getenv("DASHSCOPE_API_KEY"), # 如果您没有配置环境变量，请在此处
用您的API Key进行替换
            base_url="https://dashscope.aliyuncs.com/compatible-mode/v1", # 填写
DashScope base_url
            model="qwen-plus"
            )
    '''


    #使用openai
    #llm = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0)

    rag_chain = (
        {"context": retriever,  "question": RunnablePassthrough()}
        | prompt
        | llm
        | StrOutputParser()
    )


    resp=rag_chain.invoke(question)
    print(resp)
```

```
#生成答案
def multi_query_generate_answer(question):
    # RAG
    template = """Answer the following question based on this context:
    {context}
    Question: {question}
    """

    prompt = ChatPromptTemplate.from_template(template)

    retrieval_chain = generate_queries | retriever.map() | get_unique_union

    final_rag_chain = (
        {"context": retrieval_chain,
        "question": operator.itemgetter("question")}
        | prompt
        | llm
        | StrOutputParser()
    )

    response = final_rag_chain.invoke({"question":question})
    print(response)




#检索出每一个查询的相关知识，每一个查询对应的相关知识又相似得分，对每一个查询内部进行排名
def retrieval_and_rank(queries):
    all_results = {}
    for query in queries:
        if query:
            search_results = vector_store.similarity_search_with_score(query)
            results = []
            for res in search_results:
                content = res[0].page_content
                score = res[1]
                results.append((content, score))
            all_results[query] = results

    document_ranks = []
    for query, doc_score_list in all_results.items():
        #每一个查询内部的list排名
        ranking_list = [doc for doc, _ in sorted(doc_score_list, key=lambda x:
x[1], reverse=True)]
        document_ranks.append(ranking_list)
    return document_ranks



##对所有查询结果 按照rrf倒排融合
def reciprocal_rank_fusion(document_ranks, k=60):
    fused_scores = {}
    for docs in document_ranks:
        for rank, doc in enumerate(docs):
            doc_str = dumps(doc)
            if doc_str not in fused_scores:
                fused_scores[doc_str] = 0
            fused_scores[doc_str] += 1 / (rank + k)
    reranked_results = [
        (loads(doc), score)
        for doc, score in sorted(fused_scores.items(), key=lambda x: x[1],
reverse=True)
    ]
    return reranked_results




#使用ollama服务
llm = OllamaLLM(model="qwen2:7b-instruct-q4_0")
template = """您是问答任务的助理。
使用以下检索到的上下文来回答问题。
如果你不知道答案，就说你不知道。
最多使用三句话，不超过100字，保持答案简洁。
Question: {question}
Context: {context}
Answer:
"""
prompt = ChatPromptTemplate.from_template(template)

chunks = prepare_data()

vector_store,retriever = embedding_data(chunks)


query = "艾伦•图灵的论文叫什么"


generate_queries,querys= get_multiple_queries(query)


multi_query_generate_answer(query)

#generate_answer(query)






#生成提示词模版
'''
template = """你是一名智能助手，根据上下文回答用户的问题，不需要回答额外的信息或捏造事实。

    已知内容:
    {context}

    问题:
    {question}
    """
```

```
prompt = PromptTemplate(template=template, input_variables=["context",
"question"])
# ----------------- 验证效果 ----------------- #
chain = LLMChain(llm=llm, prompt=prompt)
result = chain.run(context=reranked_results, question=query)

'''
```

RAG效果评估

RAG评价指标

langchain 的 **Criteria Evaluation**

https://python.langchain.com/v0.1/docs/guides/productionization/evaluation/string/criteria_eval_chain/

```
conciseness, relevance, correctness, coherence, harmfulness, maliciousness,
helpfulness, controversiality, misogyny, criminality, insensitivity
```

Ragas 中的 **Aspect Critique**

https://docs.ragas.io/en/latest/concepts/metrics/critique.html

```
harmfulness, maliciousness, coherence, correctness, conciseness
```

常用评估工具

Ragas

https://docs.ragas.io/en/latest/getstarted/index.html

也可以通过 langsmith（https://www.langchain.com/langsmith）来监控每次评估的过程，帮助分析每次评估的原因和观察 API key 的消耗。

TruLens

Llama-Index

https://docs.llamaindex.ai/en/stable/optimizing/evaluation/evaluation.html

phoenix

github开源项目：https://github.com/Arize-ai/phoenix

文档地址：https://docs.arize.com/phoenix

deepeval

https://github.com/confident-ai/deepeval

OpenAI Evals

https://github.com/openai/evals

LangSmith 和 Langfuse：

https://docs.smith.langchain.com/old/evaluation

使用 RAGAs 评估

github项目地址：https://github.com/explodinggradients/ragas

文档地址：https://docs.ragas.io/en/latest/index.html

环境准备

```
conda activate llmrag


pip install ragas==0.1.12
#这儿要注意，如果安装最新的有可能会出现Failed to parse output. Returning None错误
#具体看最新版本是否修复
```

准备评估数据

在这个例子以及后面的例子中，语料库我们都将使用百度百科关于aigc的知识：https://baike.baidu.com/item/AIGC?fromModule=lemma_search-box

操作如下：

```
from datasets import Dataset

#第3个问题可以换一个，不然评估工具有兼容问题

questions = ["艾伦•图灵的论文叫什么?",
             "人工智能生成的画作在佳士得拍卖行卖了什么价格?",
             "目前企业/机构端在使用相关的AIGC能力时，主要有哪五种方式?",
            ]

ground_truths = [["计算机器与智能（Computing Machinery and Intelligence ）"],
                 ["43.25万美元"],
                 ["直接使用、Prompt、LoRA、Finetune、Train"]]
answers = []
contexts = []

#答案和 相关文档都是langchain通过检索生成
for query in questions:
  answers.append(rag_chain.invoke(query))
  contexts.append([docs.page_content for docs in
retriever.get_relevant_documents(query)])

#前面讲到的四个数据
data = {
    "question": questions,
    "answer": answers,
    "contexts": contexts,
    "ground_truths": ground_truths
}


# 将字典转换为数据集
dataset = Dataset.from_dict(data)
```

评估 RAG

评估指标，可以参考文档：https://docs.ragas.io/en/stable/concepts/metrics/index.html#ragas-metrics

```
from ragas import evaluate
from ragas.metrics import (
    faithfulness,
    answer_relevancy,
    context_recall,
    context_precision,
)
```

```
result = evaluate(
    dataset = dataset,
    llm=llm,
    embeddings=embedding,
    metrics=[
        context_precision, #准确率
        context_recall,    #召回率
        faithfulness,      #忠实度
        answer_relevancy,  #相关性
    ],
)

df = result.to_pandas()
```

完整代码

```
import requests
from langchain.document_loaders import WebBaseLoader

from langchain.text_splitter import RecursiveCharacterTextSplitter

from langchain.prompts import ChatPromptTemplate

from langchain.schema.runnable import RunnablePassthrough
from langchain.schema.output_parser import StrOutputParser

from langchain_ollama.llms import OllamaLLM

from langchain.chat_models import ChatOpenAI
from langchain.embeddings import OpenAIEmbeddings


from langchain_chroma import Chroma
from langchain_community.embeddings import HuggingFaceBgeEmbeddings

from datasets import Dataset

from ragas.run_config import RunConfig



#准备知识库数据，建索引
def prepare_data():

    loader = WebBaseLoader("https://baike.baidu.com/item/AIGC?
fromModule=lemma_search-box")
    documents = loader.load()
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=256,
chunk_overlap=50)
    chunks = text_splitter.split_documents(documents)
    print(chunks[0].page_content)

    return chunks




#embedding 知识库，保存到向量数据库
def embedding_data(chunks):

    #openai embedding
    #rag_embeddings=OpenAIEmbeddings()

    #创建BAAI的embedding
    rag_embeddings = HuggingFaceBgeEmbeddings(model_name="BAAI/bge-small-zh-
v1.5")

    #embed保存知识到向量数据库
    vector_store = Chroma.from_documents(documents=chunks,
embedding=rag_embeddings,persist_directory="./chroma_langchain_db")
    retriever = vector_store.as_retriever()
    return vector_store,retriever,rag_embeddings




#使用ollama服务
llm = OllamaLLM(model="qwen2:7b-instruct-q4_0")
template = """您是问答任务的助理。
使用以下检索到的上下文来回答问题。
如果你不知道答案，就说你不知道。
最多使用三句话，不超过100字，保持答案简洁。
Question: {question}
Context: {context}
Answer:
"""
prompt = ChatPromptTemplate.from_template(template)

chunks = prepare_data()

vector_store,retriever,embedding= embedding_data(chunks)


#生成答案
def ragas_eval():
    '''
    #使用通义千问
    llm = ChatOpenAI(
            api_key=os.getenv("DASHSCOPE_API_KEY"), # 如果您没有配置环境变量，请在此处
用您的API Key进行替换
            base_url="https://dashscope.aliyuncs.com/compatible-mode/v1", # 填写
DashScope base_url
            model="qwen-plus"
            )
    '''
    #使用openai
    #llm = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0)

    rag_chain = (
        {"context": retriever,  "question": RunnablePassthrough()}
        | prompt
        | llm
        | StrOutputParser()
    )


    questions = ["艾伦•图灵的论文叫什么?",
                "人工智能生成的画作在佳士得拍卖行卖了什么价格?",
                "目前企业在使用相关的AIGC能力时，主要有哪五种方式?",
                ]

    ground_truths = ["计算机器与智能（Computing Machinery and Intelligence ）",
                    "2018年，人工智能生成的一幅画作在佳士得拍卖行以43.25万美元的价格成交",
                    "企业在使用AIGC能力时的五种主要方式包括：直接使用、Prompt、LoRA、
Finetune、Train"]

    answers = []
    contexts = []

    # Inference
    for query in questions:
        answers.append(rag_chain.invoke(query))
        contexts.append([docs.page_content for docs in
retriever.get_relevant_documents(query)])

    # To dict
    data = {
        "question": questions,
        "answer": answers,
        "contexts": contexts,
        "ground_truth": ground_truths
    }

    # Convert dict to dataset
    dataset = Dataset.from_dict(data)
    return dataset




from ragas import evaluate
from ragas.metrics import (
    faithfulness,
    answer_relevancy,
    context_recall,
    context_precision,
)


run_config = RunConfig(
    max_retries=10,
    max_wait=60,
    log_tenacity=True
)


dataset=ragas_eval()

result = evaluate(
    dataset = dataset,
    llm=llm,
    embeddings=embedding,
    run_config=run_config,
    metrics=[
        faithfulness,
    ],
)

print(result)
df = result.to_pandas()

print(df)


#龙 哥抖音号：龙 哥紫 貂智能
```

使用TruLens 评估

项目官网：https://www.trulens.org

github地址：https://github.com/truera/trulens/

环境准备

```
pip install trulens-eval
pip install trulens  trulens-apps-llamaindex trulens-providers-litellm litellm
```

ollama本地模型支持

trulens支持的provider：https://www.trulens.org/reference/trulens/providers/huggingface/provider/

对ollama的本地模型支持方法

https://github.com/truera/trulens/blob/main/examples/expositional/models/local_and_OSS_models/ollama_quickstart.ipynb

使用litellm支持ollama本地部署的大模型

```
import litellm
from trulens.providers.litellm import LiteLLM

litellm.set_verbose = False

provider = LiteLLM(
    model_engine="ollama/qwen2:7b-instruct-q4_0",
api_base="http://localhost:11434"
)
```

开始

在这个例子以及后面的例子中，语料库我们都将使用百度百科关于aigc的知识：https://baike.baidu.com/item/AIGC?fromModule=lemma_search-box

完整代码

```
from llama_index.readers.web import TrafilaturaWebReader
from llama_index.core.node_parser import SimpleNodeParser
from llama_index.core.schema import IndexNode




from llama_index.embeddings.huggingface import HuggingFaceEmbedding
from llama_index.core import VectorStoreIndex, StorageContext
#from llama_index.llms.openai import OpenAI
#from llama_index.embeddings.openai import OpenAIEmbedding
from llama_index.llms.ollama import Ollama

from llama_index.core.node_parser import SentenceSplitter
from llama_index.core import Settings
import chromadb


from llama_index.vector_stores.chroma import ChromaVectorStore

def prepare_data():
    url="https://baike.baidu.com/item/AIGC?fromModule=lemma_search-box"
    docs = TrafilaturaWebReader().load_data([url])
    return docs



#embed保存知识到向量数据库

def embedding_data(docs):
    #向量数据库客户端
    chroma_client = chromadb.EphemeralClient()
    chroma_collection = chroma_client.create_collection("quickstart")

    #向量数据库，指定了存储位置
    vector_store =
ChromaVectorStore(chroma_collection=chroma_collection,persist_dir="./chroma_langc
hain_db")
    storage_context = StorageContext.from_defaults(vector_store=vector_store)

    #创建文档切割器
    node_parser = SimpleNodeParser.from_defaults(chunk_size=500,chunk_overlap=50)

    #创建BAAI的embedding
    embed_model = HuggingFaceEmbedding(model_name="BAAI/bge-small-zh-v1.5")
    #创建index
    base_index = VectorStoreIndex.from_documents(documents=docs,transformations=
[node_parser],storage_context=storage_context, embed_model=embed_model)

    return base_index,embed_model




def get_llm():

    #创建OpenAI的llm
    #llm = OpenAI(model="gpt-3.5-turbo")

    #通义千问
    '''
    from llama_index.llms.dashscope import DashScope, DashScopeGenerationModels
    llm = DashScope(model_name=DashScopeGenerationModels.QWEN_MAX)
    '''

    #ollama本地模型
    llm = Ollama(model="qwen2:7b-instruct-q4_0", request_timeout=120.0)

    #创建谷歌gemini的llm
    # llm = Gemini()

    return llm

def retrieve_data(question):
    #创建检索器
    base_retriever = base_index.as_retriever(similarity_top_k=2)

    #检索相关文档
    retrievals = base_retriever.retrieve(
        question
    )

    #print(retrievals)


    #https://docs.llamaindex.ai/en/stable/examples/low_level/response_synthesis/


    from llama_index.core.response.notebook_utils import display_source_node

    for n in retrievals:
```

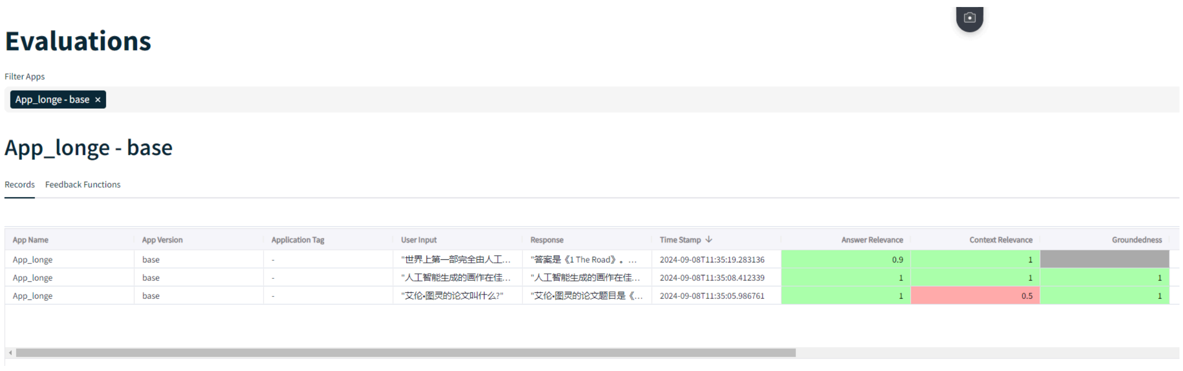
```python
        display_source_node(n, source_length=1500)

    return retrievals


def generate_answer(question):

    query_engine = base_index.as_query_engine()

    #大语言模型的回答
    response = query_engine.query(
       question
    )
    print(str(response))

    return query_engine,response


question="艾伦•图灵的论文叫什么"
docs=prepare_data()
llm=get_llm()
base_index,embed_model=embedding_data(docs)

#通过设置来配置  llm,embedding等等
Settings.llm = llm
Settings.embed_model = embed_model
#Settings.node_parser = SentenceSplitter(chunk_size=512, chunk_overlap=20)
Settings.num_output = 512
Settings.context_window = 3000


query_engine = base_index.as_query_engine()




from trulens.core import TruSession
#from trulens_eval import OpenAI as fOpenAI
import nest_asyncio


import numpy as np
from trulens.apps.llamaindex import TruLlama
from trulens.core import Feedback

import litellm
from trulens.providers.litellm import LiteLLM

from trulens.dashboard import run_dashboard

def prepare_tru():
    #设置线程的并发执行
    nest_asyncio.apply()


    #初始化数据库，它用来存储prompt、reponse、中间结果等信息。
    session = TruSession()
    session.reset_database()

    return session

#定义一个provider用来执行反馈
#provider = fOpenAI()


def prepare_feedback():

    litellm.set_verbose = False

    provider = LiteLLM(
        model_engine="ollama/qwen2:7b-instruct-q4_0",
api_base="http://localhost:11434"
    )

    f_answer_relevance = Feedback(
        provider.relevance_with_cot_reasons,    #反馈函数
        name="Answer Relevance"    #面板标识
    ).on_input_output()



    context_selection = TruLlama.select_context(query_engine)

    f_context_relevance = (
        Feedback(provider.context_relevance_with_cot_reasons, name="Context
Relevance")
        .on_input()    #用户查询
        .on(context_selection)    #检索结果
        .aggregate(np.mean)    #合计所有检索结果
    )


    f_groundedness = (
        Feedback(
            provider.groundedness_measure_with_cot_reasons, name="Groundedness"
        )
        .on(context_selection.collect())  # collect context chunks into a list
        .on_output()
    )


    #
    tru_recorder = TruLlama(
        app=query_engine,
        app_id="App_longe",
        feedbacks=[
            f_context_relevance,
            f_answer_relevance,
            f_groundedness
        ]
    )
    return tru_recorder

#定义问题
questions = ["艾伦•图灵的论文叫什么?",
             "人工智能生成的画作在佳士得拍卖行卖了什么价格?",
             "世界上第一部完全由人工智能创作的小说?",
            ]


session=prepare_tru()
tru_recorder=prepare_feedback()

def tru_eval():
    #执行评估，LLM回答所有的问题
    for question in questions:
        with tru_recorder as recording:
            query_engine.query(question)

    session.get_leaderboard()
    run_dashboard(session)
tru_eval()
```

面板查看评估结果

运行完整代码后默认下面地址：

```
Local URL: http://localhost:52699
```

可以本地打开dashboard页面查看评估结果：



Evaluations

Filter Apps

App_longe - base ×

App_longe - base

Records Feedback Functions

App Name	App Version	Application Tag	User Input	Response	Time Stamp ↓	Answer Relevance	Context Relevance	Groundedness
App_longe	base	-	"世界上第一部完全由人工...	"答案是《1 The Road》。...	2024-09-08T11:35:19.283136	0.9	1	
App_longe	base	-	"人工智能生成的画作在佳...	"人工智能生成的画作在佳...	2024-09-08T11:35:08.412339	1	1	1
App_longe	base	-	"艾伦•图灵的论文叫什么?"	"艾伦•图灵的论文题目是《...	2024-09-08T11:35:05.986761	1	0.5	1

Llama-Index评估

文档地址：https://docs.llamaindex.ai/en/stable/optimizing/evaluation/evaluation.html

```
from llama_index.evaluation import BatchEvalRunner
from llama_index.core.evaluation import (
    FaithfulnessEvaluator,
    RelevancyEvaluator,
    CorrectnessEvaluator,
)
```

```python
from llama_index.llms.openai import OpenAI


# gpt-4
gpt4 = OpenAI(temperature=0, model="gpt-4")

faithfulness_gpt4 = FaithfulnessEvaluator(llm=gpt4)   #忠实性
relevancy_gpt4 = RelevancyEvaluator(llm=gpt4)     #相关性
correctness_gpt4 = CorrectnessEvaluator(llm=gpt4)     #正确性


runner = BatchEvalRunner(
    {"faithfulness": faithfulness_gpt4, "relevancy": relevancy_gpt4},
    workers=8,
)

eval_results = await runner.aevaluate_queries(
    vector_index.as_query_engine(llm=llm), queries=qas.questions
)
```