8167 words | 21 minutes

Personal Knowledge Base Assistant Project

I. Introduction

1. Project Background

In today's world where data is surging, effectively managing and retrieving information has become a key skill. In order to meet this challenge, this project came into being, aiming to build a personal knowledge base assistant based on Langchain. This assistant provides users with a reliable information acquisition platform through an efficient information management system and powerful retrieval functions.

The core goal of this project is to give full play to the advantages of large language models in processing natural language queries, while customizing them for user needs to achieve intelligent understanding and accurate response to complex information. During the project development process, the team deeply analyzed the potential and limitations of large language models, especially their tendency to generate hallucinatory information. To solve this problem, the project integrated RAG technology, a method that combines retrieval and generation, which can retrieve relevant information from a large amount of data before generating answers, thereby significantly improving the accuracy and reliability of answers.

By introducing RAG technology, this project not only improves the accuracy of information retrieval, but also effectively suppresses the misleading information that Langchain may generate. This combined retrieval and generation method ensures the accuracy and authority of the intelligent assistant when providing information, making it a powerful assistant for users when facing massive data.

2. Goals and Significance

This project is dedicated to developing an efficient and intelligent personal knowledge base system, aiming to optimize the user's knowledge acquisition process in the information torrent. By integrating Langchain and natural language processing technology, the system achieves rapid access and integration of decentralized data sources, enabling users to efficiently retrieve and utilize information through intuitive natural language interaction.

The core value of the project is reflected in the following aspects:

1. **Optimize information retrieval efficiency** : Using the Langchain-based framework, the system can retrieve relevant information from a wide range of data sets before generating answers, thereby accelerating the process of locating and extracting information.

2. **Strengthen knowledge organization and management** : Support users to build personalized knowledge bases, promote knowledge accumulation and effective management through structured storage and classification, and thus enhance users' mastery and application of professional knowledge.

3. **Assisted decision making** : Through accurate information provision and analysis, the system enhances the user's decision-making ability in complex situations, especially in situations where quick judgment and response are required.

4. **Personalized information services** : The system allows users to customize the knowledge base according to their specific needs, realize personalized information retrieval and services, and ensure that users can obtain the most relevant and valuable knowledge.

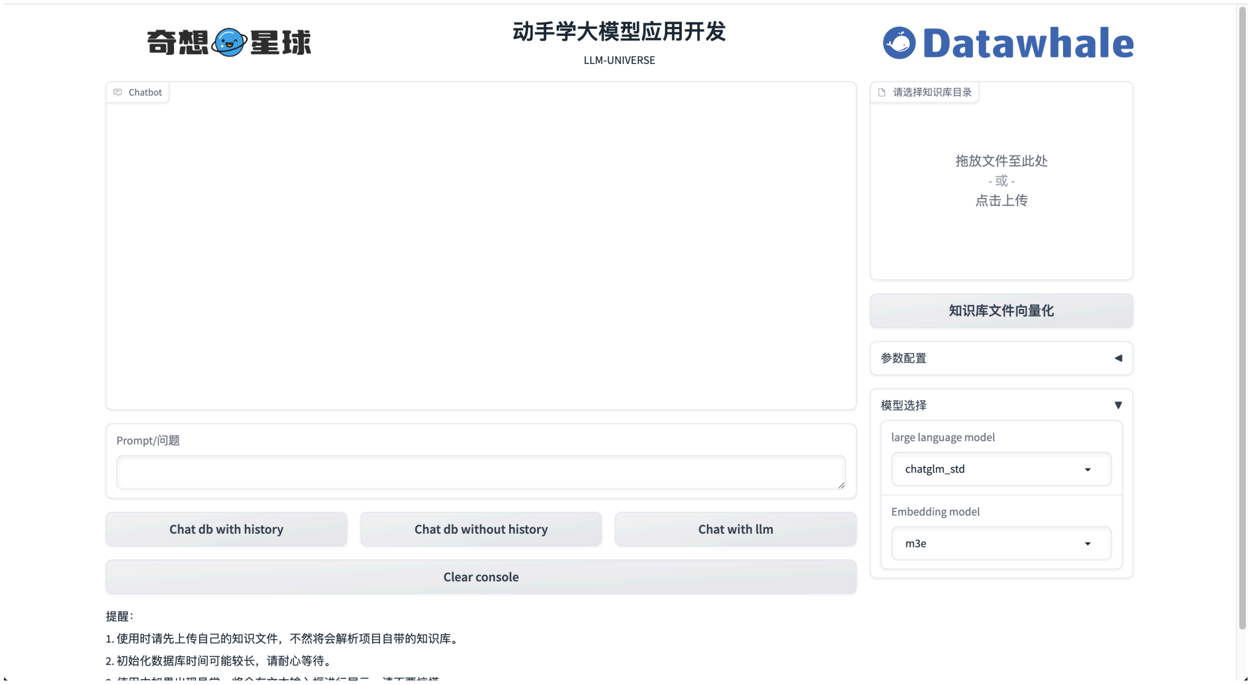
5. **Technological innovation demonstration** : The project demonstrates the advantages of RAG technology in solving the Langchain illusion problem. By combining retrieval and generation methods, it improves the accuracy and reliability of information and provides new ideas for technological innovation in the field of intelligent information management.

6. **Promote the application of intelligent assistants** : Through user-friendly interface design and convenient deployment options, the project makes intelligent assistant technology easier to understand and use, promoting the application and popularization of this technology in a wider range of fields

3. Main functions

This project can implement knowledge questions and answers based on Datawhale's existing project README, allowing users to quickly understand the status of Datawhale's existing projects.

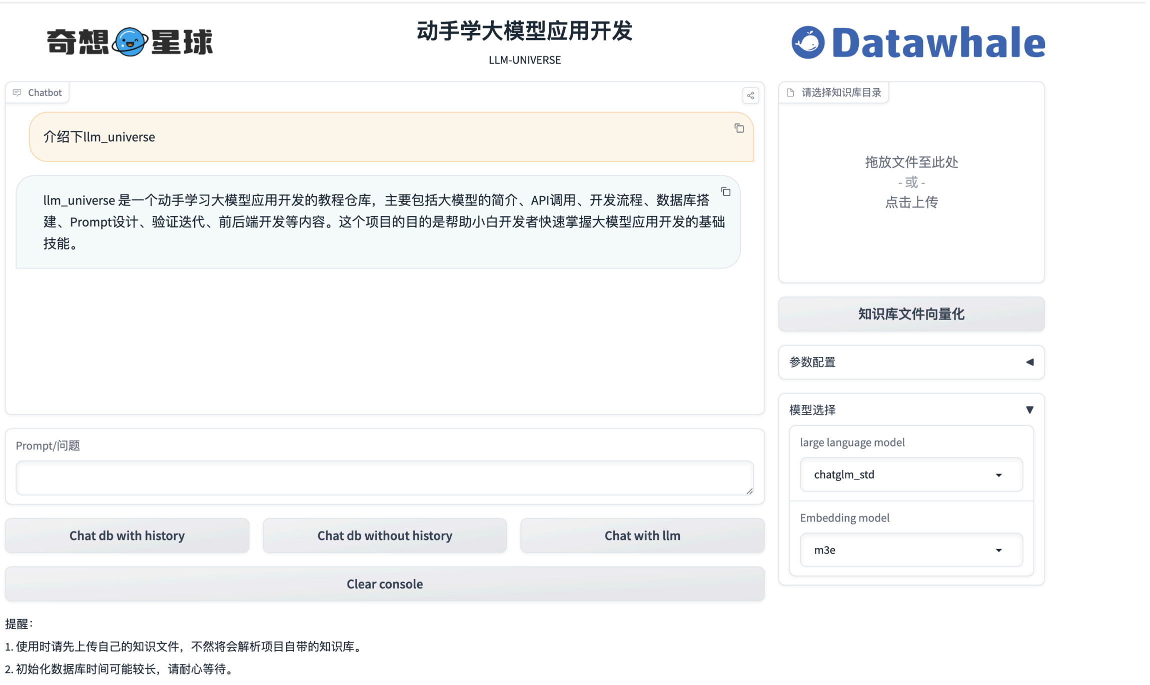
Project start interface



提醒：

1. 使用时请先上传自己的知识文件，不然将会解析项目自带的知识库。
2. 初始化数据库时间可能较长，请耐心等待。

Question and answer demonstration interface



介绍下llm_universe

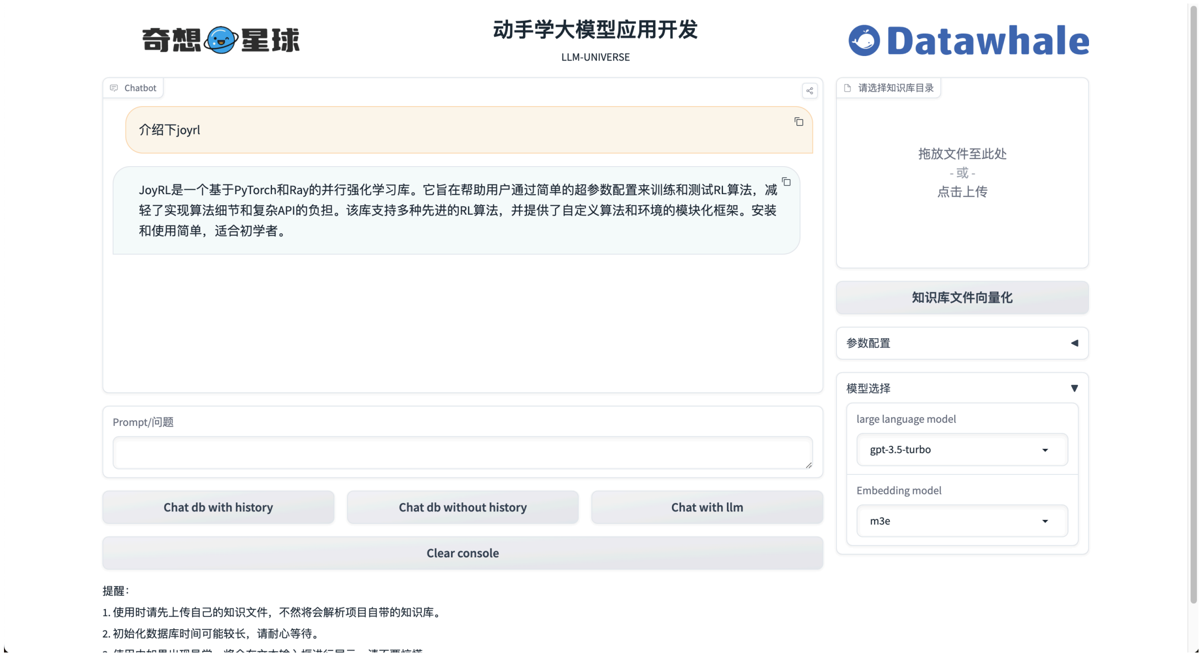
llm_universe 是一个动手学习大模型应用开发的教程仓库，主要包括大模型的简介、API调用、开发流程、数据库搭建、Prompt设计、验证迭代、前后端开发等内容。这个项目的目的是帮助小白开发者快速掌握大模型应用开发的基础技能。

提醒：

1. 使用时请先上传自己的知识文件，不然将会解析项目自带的知识库。
2. 初始化数据库时间可能较长，请耐心等待。

Example demonstration interface

1. *Introducing the joyrl demo*



介绍下joyrl

JoyRL是一个基于PyTorch和Ray的并行强化学习库。它旨在帮助用户通过简单的超参数配置来训练和测试RL算法，减轻了实现算法细节和复杂API的负担。该库支持多种先进的RL算法，并提供了自定义算法和环境的模块化框架。安装和使用简单，适合初学者。

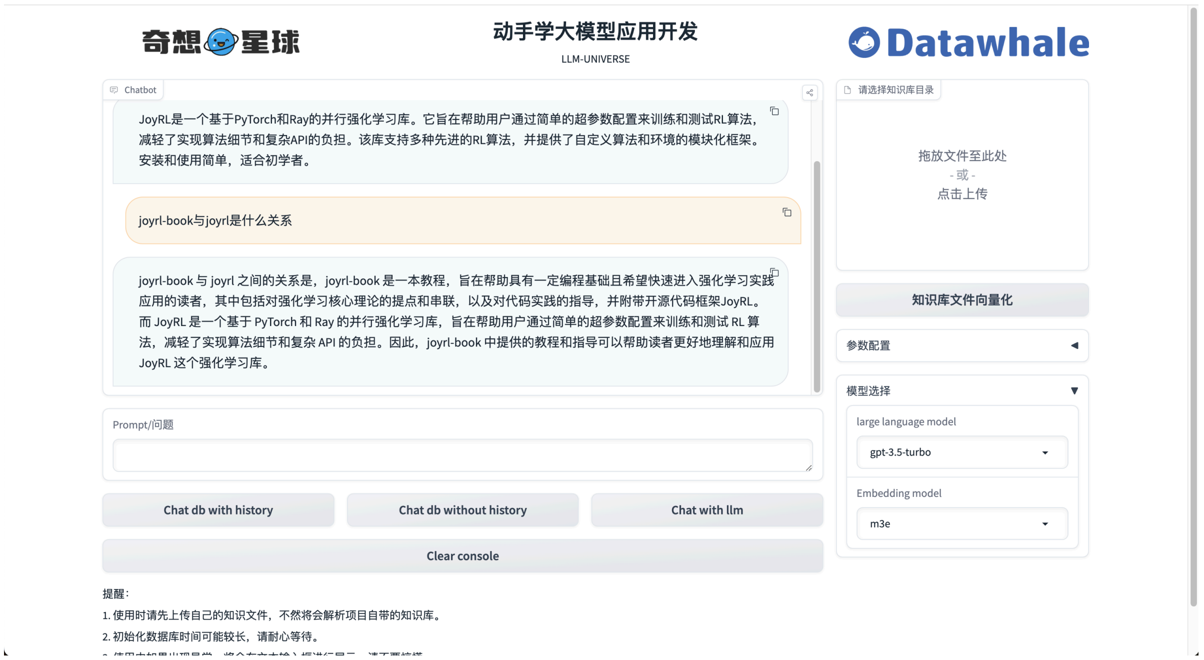
Prompt/问题

large language model
gpt-3.5-turbo

Embedding model
m3e

提醒：
1. 使用时请先上传自己的知识文件，不然将会解析项目自带的知识库。
2. 初始化数据库时间可能较长，请耐心等待。

2. *What is the relationship between joyrl-book and joyrl?*



JoyRL是一个基于PyTorch和Ray的并行强化学习库。它旨在帮助用户通过简单的超参数配置来训练和测试RL算法，减轻了实现算法细节和复杂API的负担。该库支持多种先进的RL算法，并提供了自定义算法和环境的模块化框架。安装和使用简单，适合初学者。

joyrl-book与joyrl是什么关系

joyrl-book 与 joyrl 之间的关系是，joyrl-book 是一本教程，旨在帮助具有一定编程基础且希望快速进入强化学习实践应用的读者，其中包括对强化学习核心理论的提点和串联，以及对代码实践的指导，并附带开源代码框架JoyRL。而 JoyRL 是一个基于 PyTorch 和 Ray 的并行强化学习库，旨在帮助用户通过简单的超参数配置来训练和测试 RL 算法，减轻了实现算法细节和复杂 API 的负担。因此，joyrl-book 中提供的教程和指导可以帮助读者更好地理解和应用 JoyRL 这个强化学习库。

Prompt/问题

large language model
gpt-3.5-turbo

Embedding model
m3e

提醒：
1. 使用时请先上传自己的知识文件，不然将会解析项目自带的知识库。
2. 初始化数据库时间可能较长，请耐心等待。

2. Technical Implementation

1. Environmental Dependence

1.1 Technical resource requirements

- **CPU** : Intel 5th generation processor (for cloud CPU, it is recommended to choose cloud CPU service with 2 cores or above)
- **Memory (RAM)** : At least 4 GB

- **Operating system** : Windows, macOS, Linux

1.2 Project Setup

Clone the repository

```shell
git clone https://github.com/logan-zou/Chat_with_Datawhale_langchain.git
cd Chat_with_Datawhale_langchain
```

Create a Conda environment and install dependencies

- Python>=3.9
- pytorch>=2.0.0

```shell
# 创建 Conda 环境
conda create -n llm-universe python==3.9.0
# 激活 Conda 环境
conda activate llm-universe
# 安装依赖项
pip install -r requirements.txt
```

1.3 Project Operation

- Start the service as a local API

```shell
   # Linux 系统
   cd project/serve
   uvicorn api:app --reload
```

```shell
# Windows 系统
cd project/serve
python api.py
```

- Run the project

```shell
cd llm-universe/project/serve
python run_gradio.py -model_name='chatglm_std' -embedding_model='m3e'
```

2. Brief description of development process

2.1 Current project version and future plans

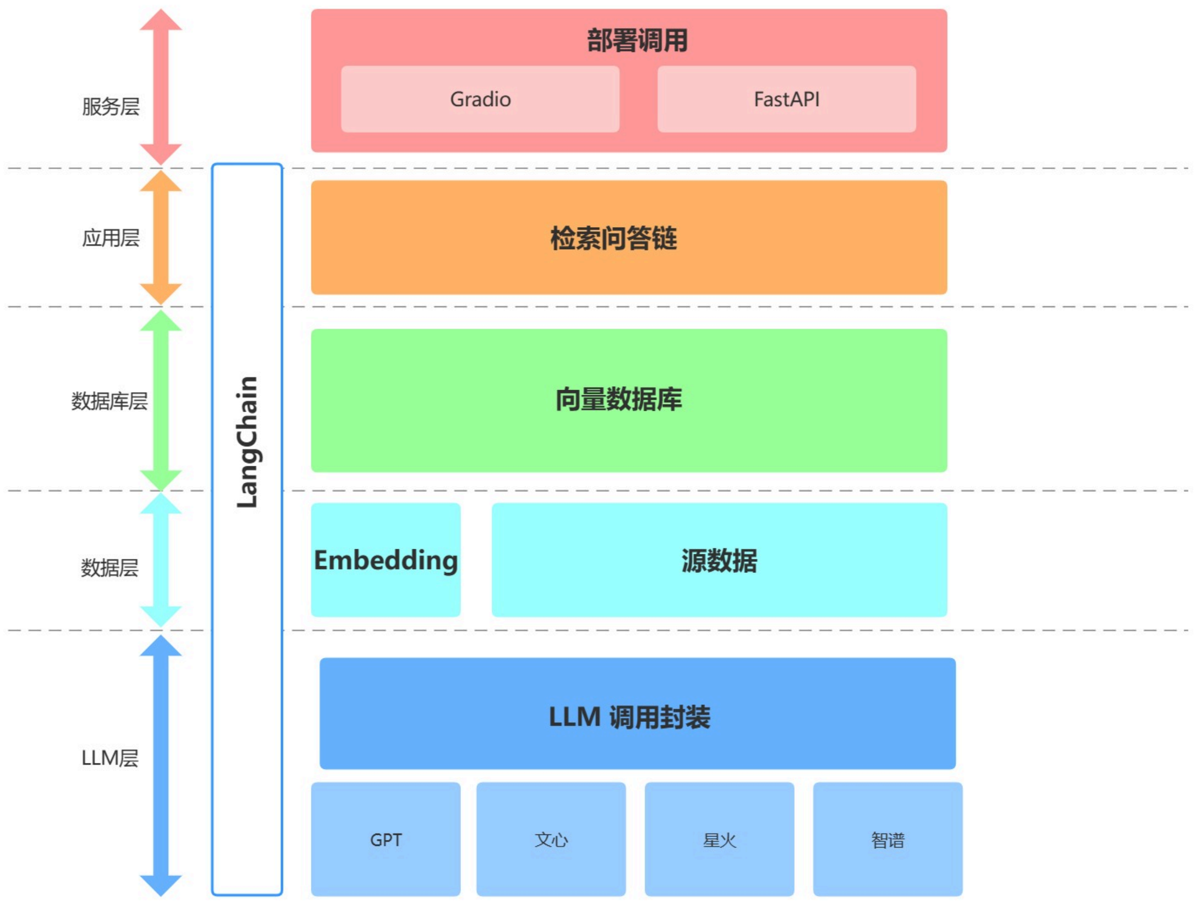
- **Current version** : 0.2.0 (updated on 2024.3.17)
  - **update content**
    - [√] Added m3e embedding
    - [√] Added new knowledge base content
    - [√] Added summary of all Mds of Datawhale
    - [√] Fix gradio display error
  - **Currently supported models**
    - OpenAi
      - [√] gpt-3.5-turbo
      - [√] gpt-3.5-turbo-16k-0613
      - [√] gpt-3.5-turbo-0613
      - [√] gpt-4
      - [√] gpt-4-32k
    - A Word from the Heart
      - [√] ERNIE-Bot
      - [√] ERNIE-Bot-4
      - [√] ERNIE-Bot-turbo
    - iFlytek Spark
      - [√] Spark-1.5
      - [√] Spark-2.0
    - Zhipu AI
      - [√] chatglm_pro
      - [√] chatglm_std
      - [√] chatglm_lite
- **future plan**
  - [ ] Update Zhipu Ai embedding

2.2 Core Idea

The core is to implement the underlying encapsulation for the four large model APIs, build a retrieval question-answering chain with switchable models based on Langchain, and implement the API and personal lightweight large model application deployed by Gradio.

2.3 Technology Stack Used

This project is a personal knowledge base assistant based on a large model, built on the LangChain framework. The core technologies include LLM API calls, vector databases, retrieval question-answering chains, etc. The overall project architecture is as follows:



As mentioned above, this project is divided into LLM layer, data layer, database layer, application layer and service layer from bottom to top.

① The LLM layer mainly encapsulates LLM calls based on four popular LLM APIs, supports users to access different models with a unified entrance and method, and supports model switching at any time;

② The data layer mainly includes the source data of the personal knowledge base and the Embedding API. The source data can be used by the vector database after Embedding processing;

③ The database layer is mainly a vector database built based on the source data of the personal knowledge base. In this project, we chose Chroma;

④ The application layer is the top-level encapsulation of the core functions. We further encapsulate it based on the retrieval question-answering chain base class provided by LangChain to support switching between different models and conveniently implement database-based retrieval question-answering;

⑤ The top layer is the service layer. We implemented two methods: Gradio to build a Demo and FastAPI to build an API to support service access for this project.

3. Application Detail

1. Core architecture

llm-universe personal knowledge base assistant address:

https://github.com/datawhalechina/llm-universe/tree/main

This project is a typical RAG project. It implements local knowledge base question and answer through langchain+LLM and establishes a local knowledge base dialogue application that can be implemented using open source models throughout the entire process. Currently, it supports the access of large language models such as ***ChatGPT*** , ***Spark model*** , ***Wenxin large model*** , and ***Zhipu GLM*** . The implementation principle of this project is the same as that of general RAG projects, as shown in the previous text and the figure below:

The entire RAG process includes the following operations:

1. User asks a question

2.加载和读取知识库文档

3.对知识库文档进行分割

4.对分割后的知识库文本向量化并存入向量库建立索引

5.对问句 Query 向量化

6.在知识库文档向量中匹配出与问句 Query 向量最相似的 top k 个

7.匹配出的知识库文本文本作为上下文 Context 和问题一起添加到 prompt 中

8.提交给 LLM 生成回答 Answer

可以大致分为索引，检索和生成三个阶段，这三个阶段将在下面小节配合该 llm-universe 知识库助手项目进行拆解。

2、索引-indexing

本节讲述该项目 llm-universe 个人知识库助手：创建知识库并加载文件-读取文件-**文本分割**(Text splitter)，知识库**文本向量化**(embedding)以及存储到**向量数据库**的实现，

其中**加载文件**：这是读取存储在本地的知识库文件的步骤。**读取文件**：读取加载的文件内容，通常是将其转化为文本格式 。**文本分割(Text splitter)：按照一定的规则(例如段落、句子、词语等)将文本分割。文本向量化：**这通常涉及到 NLP 的特征抽取，该项目通过本地 m3e 文本嵌入模型，openai，zhipuai 开源 api 等方法将分割好的文本转化为数值向量并存储到向量数据库

2.1 知识库搭建-加载和读取

该项目llm-universe个人知识库助手选用 Datawhale 一些经典开源课程、视频（部分）作为示例，具体包括：

- **《机器学习公式详解》PDF版本**
- **《面向开发者的 LLM 入门教程 第一部分 Prompt Engineering》md版本**
- **《强化学习入门指南》MP4版本**
- 以及datawhale总仓库所有开源项目的readme https://github.com/datawhalechina

这些知识库源数据放置在 **../../data_base/knowledge_db** 目录下，用户也可以自己存放自己其他的文件。

1.下面讲一下如何获取 DataWhale 总仓库的所有开源项目的 readme ，用户可以通过先运行 **project/database/test_get_all_repo.py** 文件，用来获取 Datawhale 总仓库所有开源项目的 readme，代码如下：

```python
import json
import requests
import os
import base64
import loguru
from dotenv import load_dotenv
# 加载环境变量
load_dotenv()
# 从环境变量中获取TOKEN
TOKEN = os.getenv('TOKEN')
# 定义获取组织仓库的函数
```

```
def get_repos(org_name, token, export_dir):
    headers = {
        'Authorization': f'token {token}',
    }
    url = f'https://api.github.com/orgs/{org_name}/repos'
    response = requests.get(url, headers=headers, params={'per_page': 200
    if response.status_code == 200:
        repos = response.json()
        loguru.logger.info(f'Fetched {len(repos)} repositories for {org_n
        # 使用 export_dir 确定保存仓库名的文件路径
        repositories_path = os.path.join(export_dir, 'repositories.txt')
        with open(repositories_path, 'w', encoding='utf-8') as file:
            for repo in repos:
                file.write(repo['name'] + '\n')
        return repos
    else:
        loguru.logger.error(f"Error fetching repositories: {response.stat
        loguru.logger.error(response.text)
        return []
# 定义拉取仓库README文件的函数
def fetch_repo_readme(org_name, repo_name, token, export_dir):
    headers = {
        'Authorization': f'token {token}',
    }
    url = f'https://api.github.com/repos/{org_name}/{repo_name}/readme'
    response = requests.get(url, headers=headers)
    if response.status_code == 200:
        readme_content = response.json()['content']
        # 解码base64内容
        readme_content = base64.b64decode(readme_content).decode('utf-8')
        # 使用 export_dir 确定保存 README 的文件路径
        repo_dir = os.path.join(export_dir, repo_name)
        if not os.path.exists(repo_dir):
            os.makedirs(repo_dir)
        readme_path = os.path.join(repo_dir, 'README.md')
        with open(readme_path, 'w', encoding='utf-8') as file:
            file.write(readme_content)
    else:
        loguru.logger.error(f"Error fetching README for {repo_name}: {res
        loguru.logger.error(response.text)
# 主函数
if __name__ == '__main__':
    # 配置组织名称
    org_name = 'datawhalechina'
    # 配置 export_dir
    export_dir = "../../database/readme_db"  # 请替换为实际的目录路径
    # 获取仓库列表
    repos = get_repos(org_name, TOKEN, export_dir)
    # 打印仓库名称
    if repos:
        for repo in repos:
            repo_name = repo['name']
            # 拉取每个仓库的README
            fetch_repo_readme(org_name, repo_name, TOKEN, export_dir)
    # 清理临时文件夹
    # if os.path.exists('temp'):
    #     shutil.rmtree('temp')
```

默认会把这些readme文件放在同目录database下的readme_db文件。其中这些readme文件含有不少无关信息，即再运行**project/database/text_summary_readme.py文件**可以调用大模型生成每个readme文件的摘要并保存到上述知识库目录../../data_base/knowledge_db/readme_summary文件夹中，****。代码如下：

```python
import os
from dotenv import load_dotenv
import openai
from test_get_all_repo import get_repos
from bs4 import BeautifulSoup
import markdown
import re
import time
# Load environment variables
load_dotenv()
TOKEN = os.getenv('TOKEN')
# Set up the OpenAI API client
openai_api_key = os.environ["OPENAI_API_KEY"]

# 过滤文本中链接防止大语言模型风控
def remove_urls(text):
    # 正则表达式模式，用于匹配URL
    url_pattern = re.compile(r'https?://[^\s]*')
    # 替换所有匹配的URL为空字符串
    text = re.sub(url_pattern, '', text)
    # 正则表达式模式，用于匹配特定的文本
    specific_text_pattern = re.compile(r'扫描下方二维码关注公众号|提取码|关注|科
                                       r'|组队打卡|任务打卡|组队学习的那些事|学
    # 替换所有匹配的特定文本为空字符串
    text = re.sub(specific_text_pattern, '', text)
    return text

# 抽取md中的文本
def extract_text_from_md(md_content):
    # Convert Markdown to HTML
    html = markdown.markdown(md_content)
    # Use BeautifulSoup to extract text
    soup = BeautifulSoup(html, 'html.parser')

    return remove_urls(soup.get_text())

def generate_llm_summary(repo_name, readme_content,model):
    prompt = f"1：这个仓库名是 {repo_name}. 此仓库的readme全部内容是: {readme_c
           2:请用约200以内的中文概括这个仓库readme的内容,返回的概括格式要求：这
    openai.api_key = openai_api_key
    # 具体调用
    messages = [{"role": "system", "content": "你是一个人工智能助手"},
                {"role": "user", "content": prompt}]
    response = openai.ChatCompletion.create(
        model=model,
        messages=messages,
    )
    return response.choices[0].message["content"]

def main(org_name,export_dir,summary_dir,model):
    repos = get_repos(org_name, TOKEN, export_dir)

    # Create a directory to save summaries
    os.makedirs(summary_dir, exist_ok=True)

    for id, repo in enumerate(repos):
        repo_name = repo['name']
        readme_path = os.path.join(export_dir, repo_name, 'README.md')
        print(repo_name)
        if os.path.exists(readme_path):
            with open(readme_path, 'r', encoding='utf-8') as file:
                readme_content = file.read()
            # Extract text from the README
            readme_text = extract_text_from_md(readme_content)
```

```
                # Generate a summary for the README
                # 访问受限，每min一次
                time.sleep(60)
                print('第' + str(id) + '条' + 'summary开始')
                try:
                    summary = generate_llm_summary(repo_name, readme_text,mod
                    print(summary)
                    # Write summary to a Markdown file in the summary directo
                    summary_file_path = os.path.join(summary_dir, f"{repo_nam
                    with open(summary_file_path, 'w', encoding='utf-8') as su
                        summary_file.write(f"# {repo_name} Summary\n\n")
                        summary_file.write(summary)
                except openai.OpenAIError as e:
                    summary_file_path = os.path.join(summary_dir, f"{repo_nam
                    with open(summary_file_path, 'w', encoding='utf-8') as su
                        summary_file.write(f"# {repo_name} Summary风控\n\n")
                        summary_file.write("README内容风控。\n")
                    print(f"Error generating summary for {repo_name}: {e}")
                    # print(readme_text)
            else:
                print(f"文件不存在: {readme_path}")
                # If README doesn't exist, create an empty Markdown file
                summary_file_path = os.path.join(summary_dir, f"{repo_name}_s
                with open(summary_file_path, 'w', encoding='utf-8') as summar
                    summary_file.write(f"# {repo_name} Summary不存在\n\n")
                    summary_file.write("README文件不存在。\n")
    if __name__ == '__main__':
        # 配置组织名称
        org_name = 'datawhalechina'
        # 配置 export_dir
        export_dir = "../database/readme_db"  # 请替换为实际readme的目录路径
        summary_dir="../../data_base/knowledge_db/readme_summary"# 请替换为实际
        model="gpt-3.5-turbo"  #deepseek-chat,gpt-3.5-turbo,moonshot-v1-8k
        main(org_name,export_dir,summary_dir,model)
```

其中 **extract_text_from_md()** 函数用来抽取 md 文件中的文本， **remove_urls()** 函数过滤了 readme 文本中的一些网页链接以及过滤了可能引起大模型风控一些词汇。接着调用 generate_llm_summary() 让大模型生成每个 readme 的概括。

2.在上述知识库构建完毕之后，**../../data_base/knowledge_db** 目录下就有了 Datawhale 开源的所有项目的 readme 概括的 md 文件，以及**《机器学习公式详解》PDF版本**，**《面向开发**

者的 LLM 入门教程 第一部分 Prompt Engineering》md版本，**《强化学习入门指南》MP4版本**等文件。

其中有 mp4 格式，md 格式，以及 pdf 格式，对这些文件的加载方式，该项目将代码放在了 **project/database/create_db.py文件** 下，部分代码如下。其中 pdf 格式文件用 PyMuPDFLoader 加载器，md格式文件用UnstructuredMarkdownLoader加载器。要注意的是其实数据处理是一件非常复杂和业务个性化的事，如pdf文件中包含图表，图片和文字以及不同层次标题，这些都需要根据业务进行精细化处理。具体操作可以关注**第二部分的高阶RAG教程技术**进行自行摸索：

```python
from langchain.document_loaders import UnstructuredFileLoader
from langchain.document_loaders import UnstructuredMarkdownLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.document_loaders import PyMuPDFLoader
from langchain.vectorstores import Chroma
# 首先实现基本配置

DEFAULT_DB_PATH = "../../data_base/knowledge_db"
DEFAULT_PERSIST_PATH = "../../data_base/vector_db"
...
...
...
def file_loader(file, loaders):
    if isinstance(file, tempfile._TemporaryFileWrapper):
        file = file.name
    if not os.path.isfile(file):
        [file_loader(os.path.join(file, f), loaders) for f in  os.listdir
        return
    file_type = file.split('.')[-1]
    if file_type == 'pdf':
        loaders.append(PyMuPDFLoader(file))
    elif file_type == 'md':
        pattern = r"不存在|风控"
        match = re.search(pattern, file)
        if not match:
            loaders.append(UnstructuredMarkdownLoader(file))
    elif file_type == 'txt':
        loaders.append(UnstructuredFileLoader(file))
    return
    ...
    ...
```

2.2 文本分割和向量化

文本分割和向量化操作，在整个 RAG 流程中是必不可少的。需要将上述载入的知识库分本或进行 token 长度进行分割，或者进行语义模型进行分割。该项目利用 Langchain 中的文本分割器根据 chunk_size (块大小)和 chunk_overlap (块与块之间的重叠大小)进行分割。

- chunk_size 指每个块包含的字符或 Token（如单词、句子等）的数量
- chunk_overlap 指两个块之间共享的字符数量，用于保持上下文的连贯性，避免分割丢失上下文信息

1. 可以设置一个最大的 Token 长度，然后根据这个最大的 Token 长度来切分文档。这样切分出来的文档片段是一个一个均匀长度的文档片段。而片段与片段之间的一些重叠的内容，能保证检索的时候能够检索到相关的文档片段。这部分文本分割代码也在 **project/database/create_db.py** 文件，该项目采用了 langchain 中 RecursiveCharacterTextSplitter 文本分割器进行分割。代码如下：

```python
......
def create_db(files=DEFAULT_DB_PATH, persist_directory=DEFAULT_PERSIST_PA
    """
    该函数用于加载 PDF 文件，切分文档，生成文档的嵌入向量，创建向量数据库。

    参数:
    file: 存放文件的路径。
    embeddings: 用于生产 Embedding 的模型

    返回:
    vectordb: 创建的数据库。
    """
    if files == None:
        return "can't load empty file"
    if type(files) != list:
        files = [files]
    loaders = []
    [file_loader(file, loaders) for file in files]
    docs = []
    for loader in loaders:
        if loader is not None:
            docs.extend(loader.load())
    # 切分文档
    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=500, chunk_overlap=150)
    split_docs = text_splitter.split_documents(docs)
    ....
    ....
    ....此处省略了其他代码
    ....
    return vectordb
...........
```

2. 而在切分好知识库文本之后，需要对文本进行 **向量化** 。该项目在 **project/embedding/call_embedding.py** ， 文本嵌入方式可选本地 m3e 模型，以及调用 openai 和 zhipuai 的 api 的方式进行文本嵌入。代码如下：

python

```python
import os
import sys

sys.path.append(os.path.dirname(os.path.dirname(__file__)))
sys.path.append(r"../../")
from embedding.zhipuai_embedding import ZhipuAIEmbeddings
from langchain.embeddings.huggingface import HuggingFaceEmbeddings
from langchain.embeddings.openai import OpenAIEmbeddings
from llm.call_llm import parse_llm_api_key


def get_embedding(embedding: str, embedding_key: str = None, env_file: st
   if embedding == 'm3e':
      return HuggingFaceEmbeddings(model_name="moka-ai/m3e-base")
   if embedding_key == None:
      embedding_key = parse_llm_api_key(embedding)
   if embedding == "openai":
      return OpenAIEmbeddings(openai_api_key=embedding_key)
   elif embedding == "zhipuai":
      return ZhipuAIEmbeddings(zhipuai_api_key=embedding_key)
   else:
      raise ValueError(f"embedding {embedding} not support ")
```

2.3 向量数据库

After segmenting and vectorizing the knowledge base text, you need to define a vector database to store document fragments and corresponding vector representations. In the vector database, data is represented in the form of vectors, and each vector represents a data item. These vectors can be numbers, text, images, or other types of data.

The vector database uses efficient indexing and query algorithms to speed up the storage and retrieval of vector data. This project uses the chromadb vector database (similar vector databases include faiss, etc.). The code for defining the vector database is also in **the project/database/create_db.py** file. The persist_directory is the local persistence address. The vectordb.persist() operation can persist the vector database to the local, and the local existing vector database can be loaded again later. The complete text segmentation, vectorization, and vector database definition code are as follows:

python

```python
def create_db(files=DEFAULT_DB_PATH, persist_directory=DEFAULT_PERSIST_PA
    """
    该函数用于加载 PDF 文件，切分文档，生成文档的嵌入向量，创建向量数据库。

    参数:
    file: 存放文件的路径。
    embeddings: 用于生产 Embedding 的模型

    返回:
    vectordb: 创建的数据库。
    """
    if files == None:
        return "can't load empty file"
    if type(files) != list:
        files = [files]
    loaders = []
    [file_loader(file, loaders) for file in files]
    docs = []
    for loader in loaders:
        if loader is not None:
            docs.extend(loader.load())
    # 切分文档
    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=500, chunk_overlap=150)
    split_docs = text_splitter.split_documents(docs)
    if type(embeddings) == str:
```

```python
    embeddings = get_embedding(embedding=embeddings)
# 定义持久化路径
persist_directory = '../../data_base/vector_db/chroma'
# 加载数据库
vectordb = Chroma.from_documents(
documents=split_docs,
embedding=embeddings,
persist_directory=persist_directory  # 允许我们将persist_directory目录保
)

vectordb.persist()
return vectordb
```

3. Retriver and Generator

This section enters the retrieval and generation phase of RAG, which is to vectorize the query and match the top k fragments in the knowledge base document vector that are most similar to the query vector. The matched knowledge base text is added to the prompt as the context and then submitted to LLM to generate the answer. The following will be explained based on the llm_universe personal knowledge base assistant.

3.1 Vector database search

After segmenting and vectorizing the text in the previous section and building a vector database index, we can now use the vector database for efficient retrieval. A vector database is a library used to effectively search for similarities in large-scale high-dimensional vector spaces. It can quickly find the vector that is most similar to a given query vector in a large-scale data set. As shown in the following example:

python

```python
question="什么是机器学习"
Copy to clipboardErrorCopied
sim_docs = vectordb.similarity_search(question,k=3)
print(f"检索到的内容数: {len(sim_docs)}")
```

```
检索到的内容数: 3
```

```
for i, sim_doc in enumerate(sim_docs):
    print(f"检索到的第{i}个内容: \n{sim_doc.page_content[:200]}", end="\n---
```

```
检索到的第0个内容:
导，同时也能体会到这三门数学课在机器学习上碰撞产生的“数学之美”。
1.1
引言
本节以概念理解为主，在此对“算法”和“模型”作补充说明。“算法”是指从数据中学得“模型”的具
体方法，例如后续章节中将会讲述的线性回归、对数几率回归、决策树等。“算法”产出的结果称为“
通常是具体的函数或者可抽象地看作为函数，例如一元线性回归算法产出的模型即为形如 f(x) = 

的一元一次函数。
--------------

检索到的第1个内容:
模型: 机器学习的一般流程如下: 首先收集若干样本（假设此时有 100 个），然后将其分为训练样
（80 个）和测试样本（20 个），其中 80 个训练样本构成的集合称为“训练集”，20 个测试样本
称为“测试集”，接着选用某个机器学习算法，让其在训练集上进行“学习”（或称为“训练”），然后

得到“模型”（或称为“学习器”），最后用测试集来测试模型的效果。执行以上流程时，表示我们已
--------------

检索到的第2个内容:
→_→
欢迎去各大电商平台选购纸质版南瓜书《机器学习公式详解》
←_←
第 1 章
绪论
本章作为“西瓜书”的开篇，主要讲解什么是机器学习以及机器学习的相关数学符号，为后续内容作
铺垫，并未涉及复杂的算法理论，因此阅读本章时只需耐心梳理清楚所有概念和数学符号即可。此外
阅读本章前建议先阅读西瓜书目录前页的《主要符号表》，它能解答在阅读“西瓜书”过程中产生的大
分对数学符号的疑惑。
本章也作为
```

3.2 Calling the large model llm

Here, we take the **project/qa_chain/model_to_llm.py** code as an example. In the **project/llm/** directory folder, we define the encapsulation of open source model API calls such as ***Spark*** , ***Zhipu GLM*** , and ***Wenxin LLM*** **, and import these modules in the project/qa_chain/model_to_llm.py** file. You can call LLM according to the model name passed in by the user. The code is as follows:

python

```python
def model_to_llm(model:str=None, temperature:float=0.0, appid:str=None, a
        """
        星火：model,temperature,appid,api_key,api_secret
        百度问心：model,temperature,api_key,api_secret
        智谱：model,temperature,api_key
        OpenAI：model,temperature,api_key
        """
        if model in ["gpt-3.5-turbo", "gpt-3.5-turbo-16k-0613", "gpt-3.5-
            if api_key == None:
                api_key = parse_llm_api_key("openai")
            llm = ChatOpenAI(model_name = model, temperature = temperatur
        elif model in ["ERNIE-Bot", "ERNIE-Bot-4", "ERNIE-Bot-turbo"]:
            if api_key == None or Wenxin_secret_key == None:
                api_key, Wenxin_secret_key = parse_llm_api_key("wenxin")
            llm = Wenxin_LLM(model=model, temperature = temperature, api_
        elif model in ["Spark-1.5", "Spark-2.0"]:
            if api_key == None or appid == None and Spark_api_secret == N
                api_key, appid, Spark_api_secret = parse_llm_api_key("spa
            llm = Spark_LLM(model=model, temperature = temperature, appid
        elif model in ["chatglm_pro", "chatglm_std", "chatglm_lite"]:
            if api_key == None:
                api_key = parse_llm_api_key("zhipuai")
            llm = ZhipuAILLM(model=model, zhipuai_api_key=api_key, temper
        else:
            raise ValueError(f"model{model} not support!!!")
        return llm
```

3.3 Prompt and build question-answer chain

Next, we come to the last step. After designing the prompt based on the knowledge base question and answer, we can combine the above retrieval and large model call to generate the answer. The format of constructing the prompt is as follows, which can be modified according to your business needs:

python

```python
from langchain.prompts import PromptTemplate

# template = """基于以下已知信息，简洁和专业的来回答用户的问题。
#             如果无法从中得到答案，请说 "根据已知信息无法回答该问题" 或 "没有提供足
#             答案请使用中文。
#             总是在回答的最后说“谢谢你的提问！”。
# 已知信息：{context}
# 问题：{question}"""
template = """使用以下上下文来回答最后的问题。如果你不知道答案，就说你不知道，不要试图
案。最多使用三句话。尽量使答案简明扼要。总是在回答的最后说“谢谢你的提问！”。
{context}
问题：{question}
有用的回答："""

QA_CHAIN_PROMPT = PromptTemplate(input_variables=["context","question"],
                                 template=template)

# 运行 chain
```

And build a Q&A chain: The method RetrievalQA.from_chain_type() that creates a retrieval QA chain has the following parameters:

- llm: specifies the LLM to be used
- Specify chain type: RetrievalQA.from_chain_type(chain_type="map_reduce"), or use load_qa_chain() method to specify chain type.
- Custom prompt: By specifying the chain_type_kwargs parameter in the RetrievalQA.from_chain_type() method, the parameter: chain_type_kwargs = {"prompt": PROMPT}
- Return source documents: by specifying the return_source_documents=True parameter in the RetrievalQA.from_chain_type() method; you can also use the RetrievalQAWithSourceChain() method to return a reference to the source document (coordinates or primary keys, indexes)

python

```python
# 自定义 QA 链
self.qa_chain = RetrievalQA.from_chain_type(llm=self.llm,
                                            retriever=self.retriever,
                                            return_source_documents=True,
                                            chain_type_kwargs={"prompt":self.
```

The effect of the question-answer chain is as follows: The prompt effect is constructed based on the combination of the recall result and the query

python

```
question_1 = "什么是南瓜书? "
question_2 = "王阳明是谁? "Copy to clipboardErrorCopied
```

```
result = qa_chain({"query": question_1})
print("大模型+知识库后回答 question_1 的结果: ")
print(result["result"])
```

```
大模型+知识库后回答 question_1 的结果:
南瓜书是对《机器学习》(西瓜书)中难以理解的公式进行解析和补充推导细节的一本书。谢谢你的
```

```
result = qa_chain({"query": question_2})
print("大模型+知识库后回答 question_2 的结果: ")
print(result["result"])
```

```
大模型+知识库后回答 question_2 的结果:
我不知道王阳明是谁, 谢谢你的提问!
```

The above detailed code of the retrieval question-answering chain without memory is in the project: **project/qa_chain/QA_chain_self.py** . In addition, the project also implements a retrieval question-answering chain with memory. The internal implementation details of the two custom retrieval question-answering chains are similar, except that different LangChain chains are called. The complete code of the retrieval question-answering chain with memory **is project/qa_chain/Chat_QA_chain_self.py** as follows:

python

```python
from langchain.prompts import PromptTemplate
from langchain.chains import RetrievalQA
from langchain.vectorstores import Chroma
from langchain.chains import ConversationalRetrievalChain
from langchain.memory import ConversationBufferMemory
from langchain.chat_models import ChatOpenAI

from qa_chain.model_to_llm import model_to_llm
from qa_chain.get_vectordb import get_vectordb



class Chat_QA_chain_self:
    """"
    带历史记录的问答链
    - model：调用的模型名称
    - temperature：温度系数，控制生成的随机性
    - top_k：返回检索的前k个相似文档
    - chat_history：历史记录，输入一个列表，默认是一个空列表
    - history_len：控制保留的最近 history_len 次对话
    - file_path：建库文件所在路径
    - persist_path：向量数据库持久化路径
    - appid：星火
    - api_key：星火、百度文心、OpenAI、智谱都需要传递的参数
    - Spark_api_secret：星火秘钥
    - Wenxin_secret_key：文心秘钥
    - embeddings：使用的embedding模型
    - embedding_key：使用的embedding模型的秘钥（智谱或者OpenAI）
    """
    def __init__(self,model:str, temperature:float=0.0, top_k:int=4, chat
        self.model = model
        self.temperature = temperature
        self.top_k = top_k
        self.chat_history = chat_history
        #self.history_len = history_len
        self.file_path = file_path
        self.persist_path = persist_path
        self.appid = appid
        self.api_key = api_key
        self.Spark_api_secret = Spark_api_secret
        self.Wenxin_secret_key = Wenxin_secret_key
        self.embedding = embedding
        self.embedding_key = embedding_key
        self.vectordb = get_vectordb(self.file_path, self.persist_path, s


    def clear_history(self):
        "清空历史记录"
        return self.chat_history.clear()


    def change_history_length(self,history_len:int=1):
        """
        保存指定对话轮次的历史记录
        输入参数:
        - history_len : 控制保留的最近 history_len 次对话
        - chat_history: 当前的历史对话记录
        输出: 返回最近 history_len 次对话
        """
        n = len(self.chat_history)
        return self.chat_history[n-history_len:]


    def answer(self, question:str=None,temperature = None, top_k = 4):
        """"
        核心方法, 调用问答链
        arguments:
        - question: 用户提问
        """

        if len(question) == 0:
            return "", self.chat_history

        if len(question) == 0:
            return ""

        if temperature == None:
            temperature = self.temperature

        llm = model_to_llm(self.model, temperature, self.appid, self.api_

        #self.memory = ConversationBufferMemory(memory_key="chat_history"

        retriever = self.vectordb.as_retriever(search_type="similarity",
                                            search_kwargs={'k': top_k})  #默认

        qa = ConversationalRetrievalChain.from_llm(
            llm = llm,
            retriever = retriever
        )

        #print(self.llm)
        result = qa({"question": question,"chat_history": self.chat_histo
        answer =  result['answer']
        self.chat_history.append((question,answer)) #更新历史记录

        return self.chat_history  #返回本次回答和更新后的历史记录
```

3. Summary and Outlook

3.1 Summary of key points of personal knowledge base

This example is a personal knowledge base assistant project based on a large language model (LLM), which helps users quickly locate and obtain knowledge related to DATa whale through intelligent retrieval and question-answering systems. The following are the key points of the project:

Key point 1

1. The project uses a variety of methods to complete the extraction and summarization of all md files in Datawhale and generate the corresponding knowledge base. While completing the extraction and summarization of md files, corresponding methods are also used to complete the filtering of web links in the readme text and words that may cause risk control of large models;

2. The project uses the text cutter in Langchain to complete the text segmentation before the knowledge base vectorization operation. The vector database uses efficient indexing and query algorithms to accelerate the storage and retrieval process of vector data, and quickly complete the establishment and use of personal knowledge base data.

Key point 2

The project provides underlying encapsulation of different APIs, so users can avoid complex encapsulation details and directly call the corresponding large language model.

3.2 Future Development Direction

1. User experience upgrade: Support users to upload and establish their own personal knowledge base, and build their own exclusive personal knowledge base assistant;
2. Model architecture upgrade: from the general architecture of REG to the multi-agent framework of Multi-Agent;
3. Functional optimization and upgrade: optimize the search functions within the existing structure to improve the search accuracy of the personal knowledge base.

4. Acknowledgements

I would like to thank Master San for the crawler and summary part of **the project .**

Next Chapter >

Case 2: The Great Model of Human Relationships and Worldly Wisdom