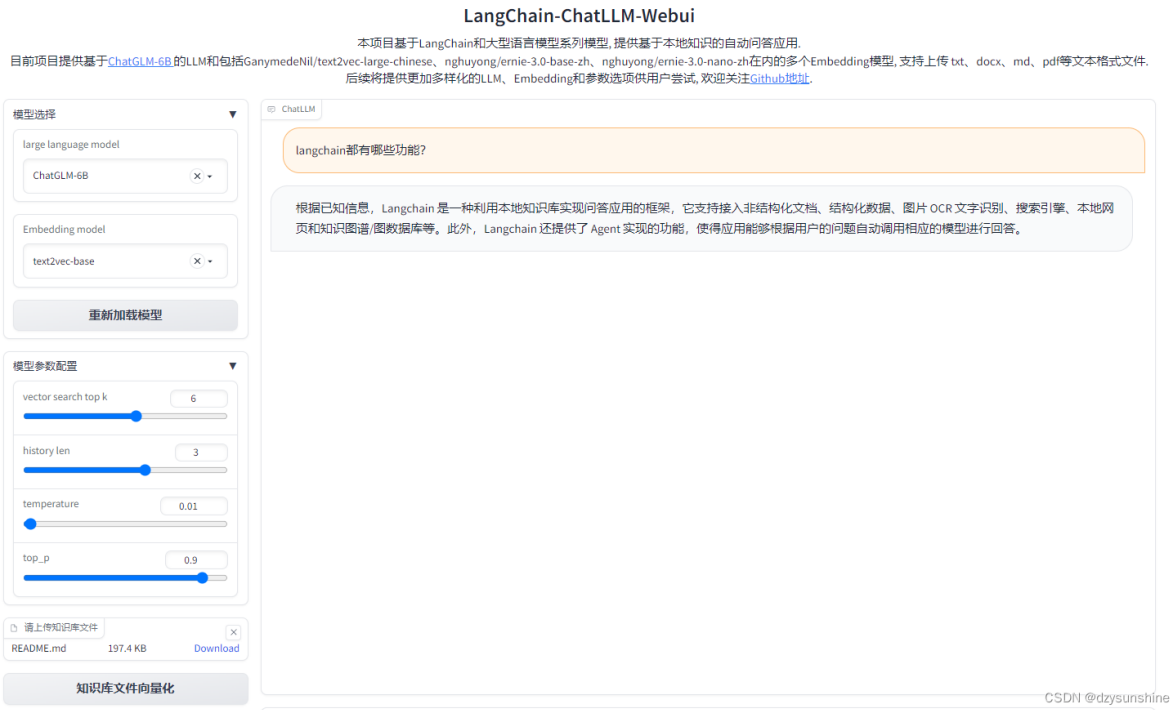
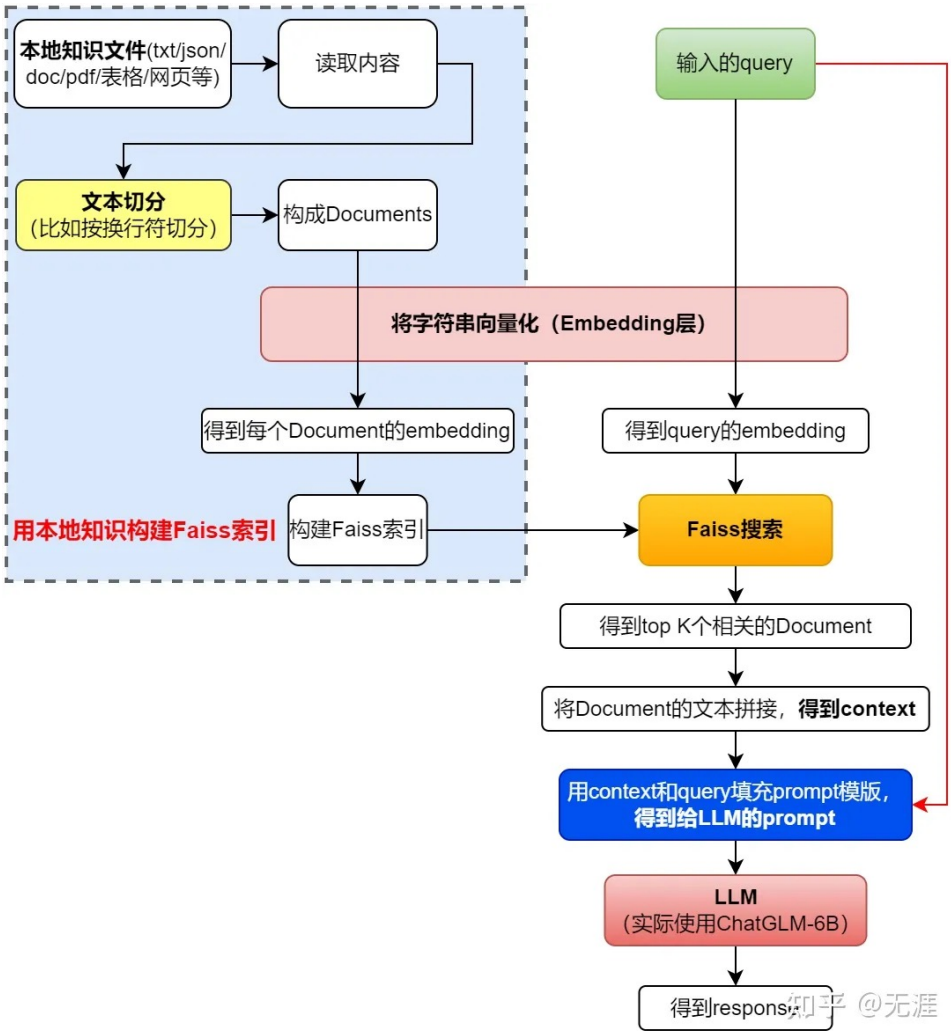


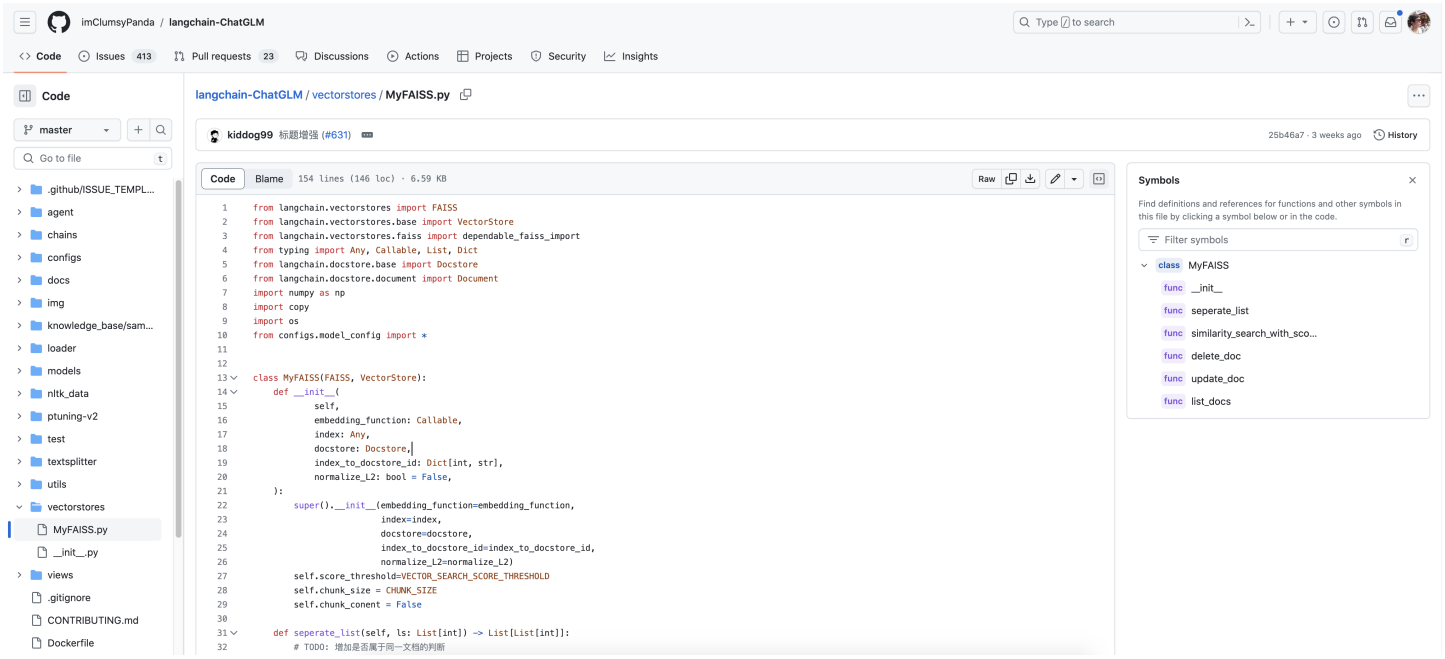
显存占用约13G

第三部分 逐行深入分析：langchain-ChatGLM(23年7月初版)项目的源码解读

再回顾一遍langchain-ChatGLM这个项目的架构图(图源)



你会发现该项目主要由以下各大模块组成(注意，该项目的最新版已经变化很大，本第三部分可以认为是针对v0.1.14左右的版本，往上最多到v0.1.16，新版对很多功能做了更高的封装，而从原理理解的角度来说，看老版 更好理解些)



1. chains： 工作链路实现，如 chains/local_doc_qa 实现了基于本地文档的问答实现
2. configs：配置文件存储
3. knowledge_base/content：用于存储上传的原始文件
4. loader： 文档加载器的实现类
5. models： llm的接口类与实现类，针对开源模型提供流式输出支持
6. textsplitter： 文本切分的实现类
7. vectorstores：用于存储向量库文件，即本地知识库本体
8. ..

接下来，为方便读者一目了然，更快理解

1. 我基本给“下面该项目中的每一行代码”都添加上了中文注释
2. 且为理解更顺畅，**我解读各个代码文件夹的顺序是根据项目流程逐一展开的** (而非上图GitHub上各个代码文件夹的呈现顺序)

如有问题，可以随时留言评论

3.1 agent：custom_agent/bing_search

3.1.1 agent/custom_agent.py

```python
from langchain.agents import Tool          # 导入工具模块
from langchain.tools import BaseTool       # 导入基础工具类
from langchain import PromptTemplate, LLMChain      # 导入提示模板和语言模型链
from agent.custom_search import DeepSearch          # 导入自定义搜索模块

# 导入基础单动作代理，输出解析器，语言模型单动作代理和代理执行器
from langchain.agents import BaseSingleActionAgent, AgentOutputParser, LLMSingleActionAgent, AgentExecutor
from typing import List, Tuple, Any, Union, Optional, Type      # 导入类型注释模块
from langchain.schema import AgentAction, AgentFinish           # 导入代理动作和代理完成模式
from langchain.prompts import StringPromptTemplate          # 导入字符串提示模板
from langchain.callbacks.manager import CallbackManagerForToolRun      # 导入工具运行回调管理器
from langchain.base_language import BaseLanguageModel       # 导入基础语言模型
import re                                                   # 导入正则表达式模块

# 定义一个代理模板字符串
agent_template = """
你现在是一个{role}。这里是一些已知信息：
{related_content}
{background_infomation}
{question_guide}：{input}

{answer_format}
"""

# 定义一个自定义提示模板类，继承自字符串提示模板
class CustomPromptTemplate(StringPromptTemplate):
    template: str           # 提示模板字符串
    tools: List[Tool]       # 工具列表
```

```
    # 定义一个格式化函数，根据提供的参数生成最终的提示模板
    def format(self, **kwargs) -> str:
        intermediate_steps = kwargs.pop("intermediate_steps")
        # 判断是否有互联网查询信息
        if len(intermediate_steps) == 0:
            # 如果没有，则给出默认的背景信息，角色，问题指导和回答格式
            background_infomation = "\n"
            role = "傻瓜机器人"
            question_guide = "我现在有一个问题"
            answer_format = "如果你知道答案，请直接给出你的回答！如果你不知道答案，请你只回答\"DeepSearch('搜索词')\"，并将'搜索词'替换为你认为需要搜索的

        else:
            # 否则，根据 intermediate_steps 中的 AgentAction 拼装 background_infomation
            background_infomation = "\n\n你还有这些已知信息作为参考：\n\n"
            action, observation = intermediate_steps[0]
            background_infomation += f"{observation}\n"
            role = "聪明的 AI 助手"
            question_guide = "请根据这些已知信息回答我的问题"
            answer_format = ""

        kwargs["background_infomation"] = background_infomation
        kwargs["role"] = role
        kwargs["question_guide"] = question_guide
        kwargs["answer_format"] = answer_format
        return self.template.format(**kwargs)  # 格式化模板并返回

# 定义一个自定义搜索工具类，继承自基础工具类
class CustomSearchTool(BaseTool):
    name: str = "DeepSearch"           # 工具名称
    description: str = ""              # 工具描述

    # 定义一个运行函数，接受一个查询字符串和一个可选的回调管理器作为参数，返回DeepSearch的搜索结果
    def _run(self, query: str, run_manager: Optional[CallbackManagerForToolRun] = None):
        return DeepSearch.search(query = query)

    # 定义一个异步运行函数，但由于DeepSearch不支持异步，所以直接抛出一个未实现错误
    async def _arun(self, query: str):
        raise NotImplementedError("DeepSearch does not support async")

# 定义一个自定义代理类，继承自基础单动作代理
class CustomAgent(BaseSingleActionAgent):
    # 定义一个输入键的属性
    @property
    def input_keys(self):
        return ["input"]

    # 定义一个计划函数，接受一组中间步骤和其他参数，返回一个代理动作或者代理完成
    def plan(self, intermedate_steps: List[Tuple[AgentAction, str]],
             **kwargs: Any) -> Union[AgentAction, AgentFinish]:
        return AgentAction(tool="DeepSearch", tool_input=kwargs["input"], log="")

# 定义一个自定义输出解析器，继承自代理输出解析器
class CustomOutputParser(AgentOutputParser):
    # 定义一个解析函数，接受一个语言模型的输出字符串，返回一个代理动作或者代理完成
    def parse(self, llm_output: str) -> Union[AgentAction, AgentFinish]:
        # 使用正则表达式匹配输出字符串，group1是调用函数名字，group2是传入参数
        match = re.match(r'^[\s\w]*(DeepSearch)\(([^\)]+)\)', llm_output, re.DOTALL)
        print(match)

        # 如果语言模型没有返回 DeepSearch() 则认为直接结束指令
        if not match:
            return AgentFinish(
                return_values={"output": llm_output.strip()},
                log=llm_output,
            )
        # 否则的话都认为需要调用 Tool
        else:
            action = match.group(1).strip()
            action_input = match.group(2).strip()
            return AgentAction(tool=action, tool_input=action_input.strip(" ").strip('"'), log=llm_output)


# 定义一个深度代理类
class DeepAgent:
    tool_name: str = "DeepSearch"       # 工具名称
    agent_executor: any                 # 代理执行器
    tools: List[Tool]                   # 工具列表
    llm_chain: any                      # 语言模型链

```

```
    # 定义一个查询函数，接受一个相关内容字符串和一个查询字符串，返回执行器的运行结果
        def query(self, related_content: str = "", query: str =
        tool_name =这段代码的主要目的是建立一个深度搜索的AI代理。AI代理首先通过接收一个问题输入，然后根据输入生成一个提示模板，然后通过该模板引导AI生成回答或

```python
        self.tool_name
        result = self.agent_executor.run(related_content=related_content, input=query ,tool_name=self.tool_name)
        return result       # 返回执行器的运行结果

    # 在初始化函数中，首先从DeepSearch工具创建一个工具实例，并添加到工具列表中
    def __init__(self, llm: BaseLanguageModel, **kwargs):
        tools = [
                    Tool.from_function(
                        func=DeepSearch.search,
                        name="DeepSearch",
                        description=""
                    )
                ]
        self.tools = tools      # 保存工具列表
        tool_names = [tool.name for tool in tools]    # 提取工具列表中的工具名称
        output_parser = CustomOutputParser()          # 创建一个自定义输出解析器实例
        # 创建一个自定义提示模板实例
        prompt = CustomPromptTemplate(template=agent_template,
                                      tools=tools,
                                      input_variables=["related_content","tool_name", "input", "intermediate_steps"])
        # 创建一个语言模型链实例
        llm_chain = LLMChain(llm=llm, prompt=prompt)
        self.llm_chain = llm_chain      # 保存语言模型链实例

        # 创建一个语言模型单动作代理实例
        agent = LLMSingleActionAgent(
            llm_chain=llm_chain,
            output_parser=output_parser,
            stop=["\nObservation:"],
            allowed_tools=tool_names
        )

        # 创建一个代理执行器实例
        agent_executor = AgentExecutor.from_agent_and_tools(agent=agent, tools=tools, verbose=True)
        self.agent_executor = agent_executor         # 保存代理执行器实例
```

3.1.2 agent/bing_search.py

```python
#coding=utf8
# 声明文件编码格式为 utf8

from langchain.utilities import BingSearchAPIWrapper
# 导入 BingSearchAPIWrapper 类，这个类用于与 Bing 搜索 API 进行交互

from configs.model_config import BING_SEARCH_URL, BING_SUBSCRIPTION_KEY
# 导入配置文件中的 Bing 搜索 URL 和 Bing 订阅密钥

def bing_search(text, result_len=3):
    # 定义一个名为 bing_search 的函数，该函数接收一个文本和结果长度的参数，默认结果长度为3

    if not (BING_SEARCH_URL and BING_SUBSCRIPTION_KEY):
        # 如果 Bing 搜索 URL 或 Bing 订阅密钥未设置，则返回一个错误信息的文档
        return [{"snippet": "please set BING_SUBSCRIPTION_KEY and BING_SEARCH_URL in os ENV",
                 "title": "env inof not fould",
                 "link": "https://python.langchain.com/en/latest/modules/agents/tools/examples/bing_search.html"}]

    search = BingSearchAPIWrapper(bing_subscription_key=BING_SUBSCRIPTION_KEY,
                                  bing_search_url=BING_SEARCH_URL)
    # 创建 BingSearchAPIWrapper 类的实例，该实例用于与 Bing 搜索 API 进行交互

    return search.results(text, result_len)
    # 返回搜索结果，结果的数量由 result_len 参数决定

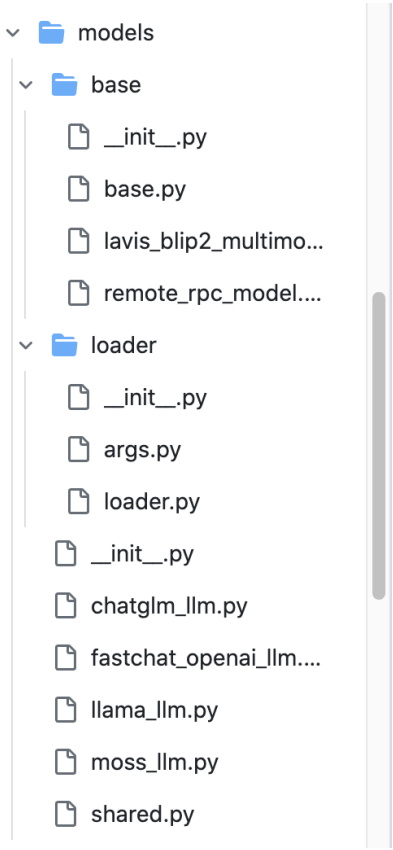
if __name__ == "__main__":
    # 如果这个文件被直接运行，而不是被导入作为模块，那么就执行以下代码

    r = bing_search('python')
    # 使用 Bing 搜索 API 来搜索 "python" 这个词，并将结果保存在变量 r 中

    print(r)
    # 打印出搜索结果
```
```

3.2 models：包含models和文档加载器loader

- models： llm的接口类与实现类，针对开源模型提供流式输出支持
- loader： 文档加载器的实现类



3.2.1 models/chatglm_llm.py

```python
from abc import ABC  # 导入抽象基类
from langchain.llms.base import LLM           # 导入语言学习模型基类
from typing import Optional, List             # 导入类型标注模块
from models.loader import LoaderCheckPoint    # 导入模型加载点
from models.base import (BaseAnswer,          # 导入基本回答模型
                         AnswerResult)        # 导入回答结果模型


class ChatGLM(BaseAnswer, LLM, ABC):  # 定义ChatGLM类，继承基础回答、语言学习模型和抽象基类
    max_token: int = 10000          # 最大的token数
    temperature: float = 0.01       # 温度参数，用于控制生成文本的随机性
    top_p = 0.9  # 排序前0.9的token会被保留
    checkPoint: LoaderCheckPoint = None  # 检查点模型
    # history = []          # 历史记录
    history_len: int = 10   # 历史记录长度

    def __init__(self, checkPoint: LoaderCheckPoint = None):  # 初始化方法
        super().__init__()  # 调用父类的初始化方法
        self.checkPoint = checkPoint  # 赋值检查点模型

    @property
    def _llm_type(self) -> str:  # 定义只读属性_llm_type，返回语言学习模型的类型
        return "ChatGLM"

    @property
    def _check_point(self) -> LoaderCheckPoint:  # 定义只读属性_check_point，返回检查点模型
        return self.checkPoint

    @property
    def _history_len(self) -> int:  # 定义只读属性_history_len，返回历史记录的长度
        return self.history_len

    def set_history_len(self, history_len: int = 10) -> None:  # 设置历史记录长度
        self.history_len = history_len

```

```
    def _call(self, prompt: str, stop: Optional[List[str]] = None) -> str:  # 定义_call方法，实现模型的具体调用
        print(f"__call:{prompt}")  # 打印调用的提示信息
        response, _ = self.checkPoint.model.chat(  # 调用模型的chat方法，获取回答和其他信息
            self.checkPoint.tokenizer,  # 使用的分词器
            prompt,  # 提示信息
            history=[],  # 历史记录
            max_length=self.max_token,      # 最大长度
            temperature=self.temperature    # 温度参数
        )
        print(f"response:{response}")  # 打印回答信息
        print(f"+++++++++++++++++++++++++++++++++++")  # 打印分隔线
        return response  # 返回回答

    def generatorAnswer(self, prompt: str,
                        history: List[List[str]] = [],
                        streaming: bool = False):  # 定义生成回答的方法，可以处理流式输入

        if streaming:  # 如果是流式输入
            history += [[]]  # 在历史记录中添加新的空列表
            for inum, (stream_resp, _) in enumerate(self.checkPoint.model.stream_chat(  # 对模型的stream_chat方法返回的结果进行枚举
                    self.checkPoint.tokenizer,  # 使用的分词器
                    prompt,  # 提示信息
                    history=history[-self.history_len:-1] if self.history_len > 1 else [],  # 使用的历史记录
                    max_length=self.max_token,  # 最大长度
                    temperature=self.temperature  # 温度参数
            )):
                # self.checkPoint.clear_torch_cache()  # 清空缓存
                history[-1] = [prompt, stream_resp]  # 更新最后一个历史记录
                answer_result = AnswerResult()  # 创建回答结果对象
                answer_result.history = history  # 更新回答结果的历史记录
                answer_result.llm_output = {"answer": stream_resp}  # 更新回答结果的输出
                yield answer_result  # 生成回答结果
        else:  # 如果不是流式输入
            response, _ = self.checkPoint.model.chat(  # 调用模型的chat方法，获取回答和其他信息
                self.checkPoint.tokenizer,  # 使用的分词器
                prompt,  # 提示信息
                history=history[-self.history_len:] if self.history_len > 0 else [],  # 使用的历史记录
                max_length=self.max_token,  # 最大长度
                temperature=self.temperature  # 温度参数
            )
            self.checkPoint.clear_torch_cache()  # 清空缓存
            history += [[prompt, response]]  # 更新历史记录
            answer_result = AnswerResult()  # 创建回答结果对象
            answer_result.history = history  # 更新回答结果的历史记录
            answer_result.llm_output = {"answer": response}  # 更新回答结果的输出
            yield answer_result  # 生成回答结果
```

3.2.2 models/shared.py

这个文件的作用是远程调用LLM

```
import sys      # 导入sys模块，通常用于与Python解释器进行交互
from typing import Any      # 从typing模块导入Any，用于表示任何类型

# 从models.loader.args模块导入parser，可能是解析命令行参数用
from models.loader.args import parser
# 从models.loader模块导入LoaderCheckPoint，可能是模型加载点
from models.loader import LoaderCheckPoint

# 从configs.model_config模块导入llm_model_dict和LLM_MODEL
from configs.model_config import (llm_model_dict, LLM_MODEL)
# 从models.base模块导入BaseAnswer，即模型的基础类
from models.base import BaseAnswer

# 定义一个名为loaderCheckPoint的变量，类型为LoaderCheckPoint，并初始化为None
loaderCheckPoint: LoaderCheckPoint = None


def loaderLLM(llm_model: str = None, no_remote_model: bool = False, use_ptuning_v2: bool = False) -> Any:
    """
    初始化 llm_model_ins LLM
    :param llm_model: 模型名称
    :param no_remote_model: 是否使用远程模型，如果需要加载本地模型，则添加 `--no-remote-model
    :param use_ptuning_v2: 是否使用 p-tuning-v2 PrefixEncoder
    :return:
    """
    pre_model_name = loaderCheckPoint.model_name      # 获取loaderCheckPoint的模型名称
```

```
    llm_model_info = llm_model_dict[pre_model_name]   # 从模型字典中获取模型信息

    if no_remote_model:      # 如果不使用远程模型
        loaderCheckPoint.no_remote_model = no_remote_model  # 将loaderCheckPoint的no_remote_model设置为True
    if use_ptuning_v2:       # 如果使用p-tuning-v2
        loaderCheckPoint.use_ptuning_v2 = use_ptuning_v2    # 将loaderCheckPoint的use_ptuning_v2设置为True

    if llm_model:            # 如果指定了模型名称
        llm_model_info = llm_model_dict[llm_model]  # 从模型字典中获取指定的模型信息

    if loaderCheckPoint.no_remote_model:  # 如果不使用远程模型
        loaderCheckPoint.model_name = llm_model_info['name']  # 将loaderCheckPoint的模型名称设置为模型信息中的name
    else:  # 如果使用远程模型
        loaderCheckPoint.model_name = llm_model_info['pretrained_model_name']  # 将loaderCheckPoint的模型名称设置为模型信息中的pretrained_mod

    loaderCheckPoint.model_path = llm_model_info["local_model_path"]  # 设置模型的本地路径

    if 'FastChatOpenAILLM' in llm_model_info["provides"]:  # 如果模型信息中的provides包含'FastChatOpenAILLM'
        loaderCheckPoint.unload_model()  # 卸载模型
    else:  # 如果不包含
        loaderCheckPoint.reload_model()  # 重新加载模型

    provides_class = getattr(sys.modules['models'], llm_model_info['provides'])  # 获取模型类
    modelInsLLM = provides_class(checkPoint=loaderCheckPoint)  # 创建模型实例
    if 'FastChatOpenAILLM' in llm_model_info["provides"]:      # 如果模型信息中的provides包含'FastChatOpenAILLM'
        modelInsLLM.set_api_base_url(llm_model_info['api_base_url'])  # 设置API基础URL
        modelInsLLM.call_model_name(llm_model_info['name'])    # 设置模型名称
    return modelInsLLM  # 返回模型实例
```

// 待更..

3.3 configs：配置文件存储model_config.py

```
import torch.cuda
import torch.backends
import os
import logging
import uuid

LOG_FORMAT = "%(levelname) -5s %(asctime)s" "-1d: %(message)s"
logger = logging.getLogger()
logger.setLevel(logging.INFO)
logging.basicConfig(format=LOG_FORMAT)

# 在以下字典中修改属性值，以指定本地embedding模型存储位置
# 如将 "text2vec": "GanymedeNil/text2vec-large-chinese" 修改为 "text2vec": "User/Downloads/text2vec-large-chinese"
# 此处请写绝对路径
embedding_model_dict = {
    "ernie-tiny": "nghuyong/ernie-3.0-nano-zh",
    "ernie-base": "nghuyong/ernie-3.0-base-zh",
    "text2vec-base": "shibing624/text2vec-base-chinese",
    "text2vec": "GanymedeNil/text2vec-large-chinese",
    "m3e-small": "moka-ai/m3e-small",
    "m3e-base": "moka-ai/m3e-base",
}

# Embedding model name
EMBEDDING_MODEL = "text2vec"

# Embedding running device
EMBEDDING_DEVICE = "cuda" if torch.cuda.is_available() else "mps" if torch.backends.mps.is_available() else "cpu"


# supported LLM models
# llm_model_dict 处理了loader的一些预设行为，如加载位置，模型名称，模型处理器实例
# 在以下字典中修改属性值，以指定本地 LLM 模型存储位置
# 如将 "chatglm-6b" 的 "local_model_path" 由 None 修改为 "User/Downloads/chatglm-6b"
# 此处请写绝对路径
llm_model_dict = {
    "chatglm-6b-int4-qe": {
        "name": "chatglm-6b-int4-qe",
        "pretrained_model_name": "THUDM/chatglm-6b-int4-qe",
        "local_model_path": None,
        "provides": "ChatGLM"
    },
    "chatglm-6b-int4": {
```

```
        "name": "chatglm-6b-int4",
        "pretrained_model_name": "THUDM/chatglm-6b-int4",
        "local_model_path": None,
        "provides": "ChatGLM"
    },
    "chatglm-6b-int8": {
        "name": "chatglm-6b-int8",
        "pretrained_model_name": "THUDM/chatglm-6b-int8",
        "local_model_path": None,
        "provides": "ChatGLM"
    },
    "chatglm-6b": {
        "name": "chatglm-6b",
        "pretrained_model_name": "THUDM/chatglm-6b",
        "local_model_path": None,
        "provides": "ChatGLM"
    },
    "chatglm2-6b": {
        "name": "chatglm2-6b",
        "pretrained_model_name": "THUDM/chatglm2-6b",
        "local_model_path": None,
        "provides": "ChatGLM"
    },
    "chatglm2-6b-int4": {
        "name": "chatglm2-6b-int4",
        "pretrained_model_name": "THUDM/chatglm2-6b-int4",
        "local_model_path": None,
        "provides": "ChatGLM"
    },
    "chatglm2-6b-int8": {
        "name": "chatglm2-6b-int8",
        "pretrained_model_name": "THUDM/chatglm2-6b-int8",
        "local_model_path": None,
        "provides": "ChatGLM"
    },
    "chatyuan": {
        "name": "chatyuan",
        "pretrained_model_name": "ClueAI/ChatYuan-large-v2",
        "local_model_path": None,
        "provides": None
    },
    "moss": {
        "name": "moss",
        "pretrained_model_name": "fnlp/moss-moon-003-sft",
        "local_model_path": None,
        "provides": "MOSSLLM"
    },
    "vicuna-13b-hf": {
        "name": "vicuna-13b-hf",
        "pretrained_model_name": "vicuna-13b-hf",
        "local_model_path": None,
        "provides": "LLamaLLM"
    },

    # 通过 fastchat 调用的模型请参考如下格式
    "fastchat-chatglm-6b": {
        "name": "chatglm-6b",              # "name"修改为fastchat服务中的"model_name"
        "pretrained_model_name": "chatglm-6b",
        "local_model_path": None,
        "provides": "FastChatOpenAILLM",  # 使用fastchat api时，需保证"provides"为"FastChatOpenAILLM"
        "api_base_url": "http://localhost:8000/v1"  # "name"修改为fastchat服务中的"api_base_url"
    },
    "fastchat-chatglm2-6b": {
        "name": "chatglm2-6b",               # "name"修改为fastchat服务中的"model_name"
        "pretrained_model_name": "chatglm2-6b",
        "local_model_path": None,
        "provides": "FastChatOpenAILLM",    # 使用fastchat api时，需保证"provides"为"FastChatOpenAILLM"
        "api_base_url": "http://localhost:8000/v1"  # "name"修改为fastchat服务中的"api_base_url"
    },

    # 通过 fastchat 调用的模型请参考如下格式
    "fastchat-vicuna-13b-hf": {
        "name": "vicuna-13b-hf",           # "name"修改为fastchat服务中的"model_name"
        "pretrained_model_name": "vicuna-13b-hf",
        "local_model_path": None,
        "provides": "FastChatOpenAILLM",  # 使用fastchat api时，需保证"provides"为"FastChatOpenAILLM"
        "api_base_url": "http://localhost:8000/v1"  # "name"修改为fastchat服务中的"api_base_url"
    },
}
```

```
# LLM 名称
LLM_MODEL = "chatglm-6b"
# 量化加载8bit 模型
LOAD_IN_8BIT = False
# Load the model with bfloat16 precision. Requires NVIDIA Ampere GPU.
BF16 = False
# 本地lora存放的位置
LORA_DIR = "loras/"

# LLM lora path，默认为空，如果有请直接指定文件夹路径
LLM_LORA_PATH = ""
USE_LORA = True if LLM_LORA_PATH else False

# LLM streaming reponse
STREAMING = True

# Use p-tuning-v2 PrefixEncoder
USE_PTUNING_V2 = False

# LLM running device
LLM_DEVICE = "cuda" if torch.cuda.is_available() else "mps" if torch.backends.mps.is_available() else "cpu"

# 知识库默认存储路径
KB_ROOT_PATH = os.path.join(os.path.dirname(os.path.dirname(__file__)), "knowledge_base")

# 基于上下文的prompt模版，请务必保留"{question}"和"{context}"
PROMPT_TEMPLATE = """已知信息：
{context}

根据上述已知信息，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 “没有提供足够的相关信息”，不允许在答案中添加编造成分，

# 缓存知识库数量,如果是ChatGLM2,ChatGLM2-int4,ChatGLM2-int8模型若检索效果不好可以调成’10’
CACHED_VS_NUM = 1

# 文本分句长度
SENTENCE_SIZE = 100

# 匹配后单段上下文长度
CHUNK_SIZE = 250

# 传入LLM的历史记录长度
LLM_HISTORY_LEN = 3

# 知识库检索时返回的匹配内容条数
VECTOR_SEARCH_TOP_K = 5

# 知识检索内容相关度 Score，数值范围约为0-1100，如果为0，则不生效，经测试设置为小于500时，匹配结果更精准
VECTOR_SEARCH_SCORE_THRESHOLD = 0

NLTK_DATA_PATH = os.path.join(os.path.dirname(os.path.dirname(__file__)), "nltk_data")

FLAG_USER_NAME = uuid.uuid4().hex

logger.info(f"""
loading model config
llm device: {LLM_DEVICE}
embedding device: {EMBEDDING_DEVICE}
dir: {os.path.dirname(os.path.dirname(__file__))}
flagging username: {FLAG_USER_NAME}
""")

# 是否开启跨域，默认为False，如果需要开启，请设置为True
# is open cross domain
OPEN_CROSS_DOMAIN = False

# Bing 搜索必备变量
# 使用 Bing 搜索需要使用 Bing Subscription Key,需要在azure port中申请试用bing search
# 具体申请方式请见
# https://learn.microsoft.com/en-us/bing/search-apis/bing-web-search/create-bing-search-service-resource
# 使用python创建bing api 搜索实例详见:
# https://learn.microsoft.com/en-us/bing/search-apis/bing-web-search/quickstarts/rest/python
BING_SEARCH_URL = "https://api.bing.microsoft.com/v7.0/search"
# 注意不是bing Webmaster Tools的api key,

# 此外，如果是在服务器上，报Failed to establish a new connection: [Errno 110] Connection timed out
# 是因为服务器加了防火墙，需要联系管理员加白名单，如果公司的服务器的话，就别想了GG
BING_SUBSCRIPTION_KEY = ""

# 是否开启中文标题加强，以及标题增强的相关配置
```

```
# 通过增加标题判断，判断哪些文本为标题，并在metadata中进行标记；
# 然后将文本与往上一级的标题进行拼合，实现文本信息的增强。
ZH_TITLE_ENHANCE = False
```

3.4 loader：文档加载与text转换

3.4.1 loader/pdf_loader.py

```
# 导入类型提示模块，用于强化代码的可读性和健壮性
from typing import List

# 导入UnstructuredFileLoader，这是一个从非结构化文件中加载文档的类
from langchain.document_loaders.unstructured import UnstructuredFileLoader

# 导入PaddleOCR，这是一个开源的OCR工具，用于从图片中识别和读取文字
from paddleocr import PaddleOCR

# 导入os模块，用于处理文件和目录
import os

# 导入fitz模块，用于处理PDF文件
import fitz

# 导入nltk模块，用于处理文本数据
import nltk

# 导入模型配置文件中的NLTK_DATA_PATH，这是nltk数据的路径
from configs.model_config import NLTK_DATA_PATH

# 设置nltk数据的路径，将模型配置中的路径添加到nltk的数据路径中
nltk.data.path = [NLTK_DATA_PATH] + nltk.data.path

# 定义一个类，UnstructuredPaddlePDFLoader，该类继承自UnstructuredFileLoader
class UnstructuredPaddlePDFLoader(UnstructuredFileLoader):

    # 定义一个内部方法_get_elements，返回一个列表
    def _get_elements(self) -> List:

        # 定义一个内部函数pdf_ocr_txt，用于从pdf中进行OCR并输出文本文件
        def pdf_ocr_txt(filepath, dir_path="tmp_files"):
            # 将dir_path与filepath的目录部分合并成一个新的路径
            full_dir_path = os.path.join(os.path.dirname(filepath), dir_path)

            # 如果full_dir_path对应的目录不存在，则创建这个目录
            if not os.path.exists(full_dir_path):
                os.makedirs(full_dir_path)

            # 创建一个PaddleOCR实例，设置一些参数
            ocr = PaddleOCR(use_angle_cls=True, lang="ch", use_gpu=False, show_log=False)

            # 打开pdf文件
            doc = fitz.open(filepath)

            # 创建一个txt文件的路径
            txt_file_path = os.path.join(full_dir_path, f"{os.path.split(filepath)[-1]}.txt")

            # 创建一个临时的图片文件路径
            img_name = os.path.join(full_dir_path, 'tmp.png')

            # 打开txt_file_path对应的文件，并以写模式打开
            with open(txt_file_path, 'w', encoding='utf-8') as fout:
                # 遍历pdf的所有页面
                for i in range(doc.page_count):
                    # 获取当前页面
                    page = doc[i]

                    # 获取当前页面的文本内容，并写入txt文件
                    text = page.get_text("")
                    fout.write(text)
                    fout.write("\n")

                    # 获取当前页面的所有图片
                    img_list = page.get_images()

                    # 遍历所有图片
                    for img in img_list:
```

```
                        # 将图片转换为Pixmap对象
                        pix = fitz.Pixmap(doc, img[0])

                        # 如果图片有颜色信息，则将其转换为RGB格式
                        if pix.n - pix.alpha >= 4:
                            pix = fitz.Pixmap(fitz.csRGB, pix)

                        # 保存图片
                        pix.save(img_name)

                        # 对图片进行OCR识别
                        result = ocr.ocr(img_name)

                        # 从OCR结果中提取文本，并写入txt文件
                        ocr_result = [i[1][0] for line in result for i in line]
                        fout.write("\n".join(ocr_result))

            # 如果图片文件存在，则删除它
            if os.path.exists(img_name):
                os.remove(img_name)

            # 返回txt文件的路径
            return txt_file_path

        # 调用上面定义的函数，获取txt文件的路径
        txt_file_path = pdf_ocr_txt(self.file_path)

        # 导入partition_text函数，该函数用于将文本文件分块
        from unstructured.partition.text import partition_text

        # 对txt文件进行分块，并返回分块结果
        return partition_text(filename=txt_file_path, **self.unstructured_kwargs)

# 运行入口
if __name__ == "__main__":
    # 导入sys模块，用于操作Python的运行环境
    import sys

    # 将当前文件的上一级目录添加到Python的搜索路径中
    sys.path.append(os.path.dirname(os.path.dirname(__file__)))

    # 定义一个pdf文件的路径
    filepath = os.path.join(os.path.dirname(os.path.dirname(__file__)), "knowledge_base", "samples", "content", "test.pdf")

    # 创建一个UnstructuredPaddlePDFLoader的实例
    loader = UnstructuredPaddlePDFLoader(filepath, mode="elements")

    # 加载文档
    docs = loader.load()

    # 遍历并打印所有文档
    for doc in docs:
        print(doc)
```

// 待更..

3.5 textsplitter：文档切分

3.5.1 textsplitter/ali_text_splitter.py

ali_text_splitter.py的代码如下所示

```
# 导入CharacterTextSplitter模块，用于文本切分
from langchain.text_splitter import CharacterTextSplitter
import re                   # 导入正则表达式模块，用于文本匹配和替换
from typing import List     # 导入List类型，用于指定返回的数据类型

# 定义一个新的类AliTextSplitter，继承自CharacterTextSplitter
class AliTextSplitter(CharacterTextSplitter):
    # 类的初始化函数，如果参数pdf为True，那么使用pdf文本切分规则，否则使用默认规则
    def __init__(self, pdf: bool = False, **kwargs):
        # 调用父类的初始化函数，接收传入的其他参数
        super().__init__(**kwargs)
        self.pdf = pdf          # 将pdf参数保存为类的成员变量

    # 定义文本切分方法，输入参数为一个字符串，返回值为字符串列表
```

```
    def split_text(self, text: str) -> List[str]:
        if self.pdf:            # 如果pdf参数为True，那么对文本进行预处理
            # 替换掉连续的3个及以上的换行符为一个换行符
            text = re.sub(r"\n{3,}", r"\n", text)
            # 将所有的空白字符（包括空格、制表符、换页符等）替换为一个空格
            text = re.sub('\s', " ", text)
            # 将连续的两个换行符替换为一个空字符
            text = re.sub("\n\n", "", text)

        # 导入pipeline模块，用于创建一个处理流程
        from modelscope.pipelines import pipeline

        # 创建一个document-segmentation任务的处理流程
        # 用的模型为damo/nlp_bert_document-segmentation_chinese-base，计算设备为cpu
        p = pipeline(
            task="document-segmentation",
            model='damo/nlp_bert_document-segmentation_chinese-base',
            device="cpu")
        result = p(documents=text)    # 对输入的文本进行处理，返回处理结果
        sent_list = [i for i in result["text"].split("\n\t") if i]  # 将处理结果按照换行符和制表符进行切分，得到句子列表
        return sent_list              # 返回句子列表
```

其中，有三点值得注意下

- 参数use_document_segmentation指定是否用语义切分文档
  此处采取的文档语义分割模型为达摩院开源的：***nlp_bert_document-segmentation_chinese-base***（这是其论文）
- 另，如果使用模型进行文档语义切分，那么需要安装：

  ```
  modelscope[nlp]：pip install "modelscope[nlp]" -f https://modelscope.oss-cn-beijing.aliyuncs.com/releases/repo.html
  ```

- 且考虑到使用了三个模型，可能对于低配置gpu不太友好，因此这里将模型load进cpu计算，有需要的话可以替换device为自己的显卡id

3.6 knowledge_base：存储用户上传的文件并向量化

knowledge_bas下面有两个文件，一个content 即用户上传的原始文件，vector_store则用于存储向量库文件，即本地知识库本体，因为content因人而异 谁上传啥就是啥 所以没啥好分析，而vector_store下面则有两个文件，一个index.faiss，一个index.pkl

3.7 chains：向量搜索/匹配

如之前所述，本节开头图中“FAISS索引、FAISS搜索”中的“FAISS”是Facebook AI推出的一种用于有效搜索大规模高维向量空间中相似度的库，在大规模数据集中快速找到与给定向量最相似的向量是很多AI应用的重要组成部分，例如在推荐系统、自然语言处理、图像检索等领域

3.7.1 chains/modules /vectorstores.py文件：根据查询向量query在向量数据库中查找与query相似的文本向量

主要是关于

1. FAISS (Facebook AI Similarity Search)的使用，具体体现在*max_marginal_relevance_search_by_vector* 中(**如下图的最上面部分**)
2. 以及一个FAISS向量存储类(FAISSVS，FAISSVS类继承自FAISS类)的定义，包含两个方法
   一个 max_marginal_relevance_search (**如下图的中间部分**，其最后会调用上面的max_marginal_relevance_search_by_vector)
   一个 __from (**如下图的最下面部分**)