ChatEase: Streamlined Server Chatbot Configuration

1. 项目简介

1.1 项目背景

- 大语言模型今年爆火，不论是工业界还是学术界，都在着急向此新热点进军，可以从包括但不限于如下现象中体会到这种炒热点的趋势：
  - 一个有关AI求职的公众号发布的有关科研招生/招聘的广告，十之六七都是关于LLM微调、LLM部署、ChatBot开发的。
  - 一位拥有谷歌学术主页的PhD，在2023年之前发表的papers一致都是有关Robotics，RL，CV的，在2023年突然发表了一篇LLM+Robotics的顶会paper，且被各大公众号转发，但其idea咋一看还似乎有点粗糙，就是微调了一个多模态LLM，给它输入图片，然后它输出控制机器人的元动作指令。
- 大语言模型在今年突然爆火，生态爆发式繁荣，各种部署LLM的工具链、制作聊天机器人的前端框架、减小LLM容量优化LLM推理速度的技术、LLM从NLP领域向其他领域的泛化，前仆后继接踵而至，意欲迅速占领各个生态位。但在野蛮生长的繁荣之下隐藏着大家急功近利着急占坑带来的混乱与粗糙。LLM不像有些传统的技术，比如CV领域的很多任务，特别是一些low-level的任务，例如目标跟踪、语义分割等，都已经相当成熟了，很多饱经考验的开源框架拿起来就能用。具体来说，在LLM周边生态中，很多时候会出现这样的情景：
  - 一种新的LLM的作者给出了该模型的权重、配置文件以及部署代码——但代码是错的，并且与正确的代码相去甚远，并且明明作者自己都没有用自己给出的代码。
  - 一种新的多模态LLM出现了，世界最大的AI开源社区Hugging Face的给出了该模型的权重、配置文件以及部署代码——但是只给出处理文本模态的代码，没给处理图片、音频、视频等模态的代码。
  - 一种部署LLM的新框架出现了，该框架集成了目前几乎所有主流LLM的实现，号称使用方便。但当你要使用某种LLM时，你必须把该LLM的名称传给该框架的接口，然后该框架从互联网上当即下载该模型的权重与配置文件。该框架的文档中未提及能否直接使用本地已经提前下载好的权重与配置文件。
  - 一种部署/微调LLM或辅助调试ChatBot的新框架出现了，该框架生成自己有多么好用，可当你点入该框架的主页并按要求注册完他们的账号后，却发现该框架目前还处于内测阶段，若想使用，只能在waitlist中排队等待。
- LLM非常吃算力，即便是部署本项目的服务器配备的RTX4090，也勉强只能让运用某些参数容量提炼技术的LLM在其上推理一些简单的任务，而对于复杂的任务，则RTX4090不论是浮点数算力还是显存，都无法承担。于是，会出现下述情景：
  - 一份长度1000字的文本输入给LLM，LLM还未推理完成，显存即爆掉。
  - 一张大小50*50的图片，结合文本“这张图片的内容是什么？”，LLM将推理10分钟才能给出答案。并且，若是该为输入两张50*50的图片，则LLM不会推理，因为显存会爆掉。
  - 后端的LLM推理太慢，时延超过了一定限制，则搭建前端页面的chainlit会直接报告请求超时，不会再继续等待后端LLM返回结果，导致ChatBot前端页面无响应。且拥有LLM混乱粗糙的周边生态，目前chainlit官方文档中没有提供修改该超时时延机制的方式。
- 项目的功能不是特别复杂，因为现在聊天机器人的质量除了用户交互的体验外，瓶颈主要仅在与其背后模型的训练、微调，也就是深度学习那部分。但是我们由于算力限制的客观原因，无法进行训练与微调，故而只能进行部署、前后端连接。在此过程中遇到的主要困难仍然是LLM周边生态的混乱与粗糙。
  - 但退一步说，即便是充钱的ChatGPT-Plus，它的功能也没有特别复杂。我们的项目不需要充钱就能享受到RAG和多模态的服务，这一点是免费的ChatGPT所没有的。
- 这里还可以额外说明的是：LLM和ChatBot是有区别的，虽然在生活中很多时候人们往往会把它们的概念弄混淆。于是，什么是LLM？LLM的工作原理的本质是你给他一串文本序列，它预测这串文本序列的尾巴上的下一个单词是什么。简单举个例子，一个模型，你给它输入“同济大学是一所”，它给你输出“研究型综合性大学。”，这就是LLM。一个WebApp，你给它输入“同济大学是什么？”，WebApp把你的问题打包成prompt“你是一个无所不知的专家，现在有一个人类向你问了一个问题，你要如实回答，不能有纰漏与错别字。\n 这个人类的问题是：[同济大学是什么？]\n 你的回答是：”，然后把打包后的prompt喂给后端LLM，后端的LLM预测出“你的回答是：”后面的文本序列，然后返回给前端，前端打印出来。这就是你所看到的ChatBot。

    - ChatBot与LLM的区别在开源大模型部署框架langchain的官方文档中就能明显感受到。
    - 如何把LLM转化成ChatBot呢？基本原理就是上述的打包prompt的过程。而prompt也是一个大有文章的技术，这一点将在下文中项目的具体实现部分详述。

1.2 项目功能

项目目前已发布的增量仍然是一个原型，主要目的还是为了进一步探索用户的需求，以及打通ChatBot开发的技术栈，故而我们的功能实现更多是是面向技术探索，同时征询用户更偏好哪种核心功能。

具体来说，我们的项目功能有如下：

- 基本的纯文本的问答。
    - 这是最简单的，一个聊天机器人必须要有
- RAG(Retrievad Augmented Generation)。
    - RAG就是希望用户传给ChatBot一个文档，ChatBot能够基于这个文档的内容回答我提出的问题。RAG功能分为5个步骤：Load、Split、Embed、Store、Retrive。
    - 同时，我们用两种途径实现了RAG：
        - 我们使用Langchain提供的API实现了RAG的5步流程。
        - 我们不使用开源项目提供的API，而是基本靠自己手动使用未被封装的sentence_transformers实现了RAG。手动实现RAG主要是考虑到使用Langchain提供的API虽然方便，但是Langchain框架里集成了太多东西，不够轻量级，而实现RAG的基本功能，直接简单依靠sentence_transformers库(Langchain也有封装该哭)即可，不仅可以帮助更深入地理解RAG以及Chatbot工作的原理，而且使得整个项目更加精炼。
- 多模态问答。
    - 因为可能客户可能输入图片，故而需要此功能。
- 用户控制问答期间ChatBot是否配备记忆历史聊天记录的功能
- 用户控制问答期间ChatBot的创造力，即：产生的回答是趋于稳健还是趋于自由散漫
- 用户控制ChatBot页面的基本外观

2. 项目架构

本项目的基本架构是WebApp惯用的前后端范式。

- 前端
    - chainlit框架搭建ChatBot界面。
        - chainlit框架是专门面向LLM量身打造的，也是跟着LLM火的。此框架具备上文所述的LLM周边生态的通病：生态很繁荣，但火得太快，文档、代码、功能都做得有些许简陋，但好在它用来搭一个简单的聊天机器人界面是非常方便的，故而适合我们项目的原型设计。
    - RAG的实现
        - 本项目的核心功能是RAG。如1.2节选所述，本项目实现RAG的方式有两种，一种是使用现有的框架，一种是基本手动实现。
        - 靠现有开源框架API实现RAG五大步骤：
            - Langchain接口：Langchain提供了实现RAG五步核心流程(load, split, embed, store, retrieve)的接口API。但是其API所使用到的形参与返回值都是由Langchain框架所定义的，有很强的局限性。也就是说，若是想使用Langchain API来实现RAG，则比较保守的方法是使用Langchain框架自己集成的LLM(需要用另外的Langchain API网上下载)、embedding model、LLM推理工具链(加载、tokenizer、输出等)，这些都需要访问外网，且所使用的模型的权重与配置文件受限于Langchain自己的体系。但我们项目所使用的LLM、embedding model、LLM推理工具链都是需要在本地部署，模型的权重和配置文件都是我们从Hugging face上获取的，且LLM推理工具链在我们的项目中更适合使用hugging face官方开发的transformers，更灵活。我们不可能仅仅为了方便使用的Langchain RAG API而放弃hugging face transformers更完备的生态，更自由灵活的LLM推理工具链。

        - 针对上述问题，我们经过探究，发现问题的根源，仍然出在LLM周边野蛮生长而粗糙的生态。Langchain的低兼容性本质就是它的API做得还不够好。解决这个问题的方法是：尽力让其他框架的数据表示、工具链兼容进Langchain的API，即：通过靠探索源码寻找偏门用法，以及探索各种数据集成关系尽力做数据转换。具体实现可见于代码。
    - 靠sentence_transformers手动实现RAG五大步骤：
        - Load：最简单的步骤，将文档加载进内存来
        - Split：加载进来的文档太大，需要分成多块，方便后期做Store
        - Store：将每个分块的文档用Embedding model(往往使用Transformers架构，但网络并不复杂)映射到n维空间(n的取值由embedding model自己决定)，然后得到的n维向量存储进faiss库提供的映射空间里。
        - Retrive：此时，用户会提出一个问题question，embedding model会将用户的问题再次用Embedding model(往往与上述Embedding model相同)映射进n维空间里，然后与faiss库的映射空间中的每一个文档分块(此时也是n维向量)求余弦相似度，得到的余弦相似度越大，说明该文档分块与用户提出的问题越相近，于是将该文档分块从映射空间中提取出来，并从n维向量还原成文本序列
        - Prompt：将Retriev得到的文本序列与用户提出的问题question用一定方式合并在一起，喂给LLM，等待LLM的输出。
        - sentence_transformers的使用主要体现在Embedding model的加载与推理。
        - 详细的RAG过程将在第3节讲述。
- 后端
    - 大模型的部署：
        - 主要使用了transformers框架。transformers框架是Hugging Face官方开发的当下最流行的部署LLM的工具。在本项目中，transformers主要是用于让LLM在服务器上跑起来，且正常输入输出。

3. 项目构建与测试

3.1 核心步骤RAG

由于本项目的核心工作在于RAG的实现，在此基础上实现多模态的处理(本项目目前仅考虑图片的处理)。而多模态的处理无外乎选取一个经典好用的多模态LLM(本项目选用的是Llava)部署在服务器上，并编写代码特判前端传过来的数据类型，假如是图片则用Hugging Face transformers提供的Llava处理图片的示例代码喂给llava即可。

在于简单的不基于RAG，或者不基于多模态的问答，只要实现了RAG，那自然也能实现。

鉴于此，关于本项目的构建，在代码之外，本文档主要阐述RAG的详细过程。而由于本项目使用sentence_transformers手搓的RAG代码暂且比较粗糙，不便于讲解(虽然Langchain体量重，且功能不完备，但写出来的代码确实要简洁一些)，故而下述RAG过程是基于本项目Langchain使用来讲解的：

1. Load步骤即是后端代码接收到前端发来的message(此为chainlit定义的数据类型)，解析出其中的文件内容。但是，此时解析出的文件内容是String类型的，而文件内容在后面时要用于Split的，而Split操作是用的langchain提供的API，langchain用于Split的API只接收一种名为document的数据类型(由langchain定义，实际上也是封装的sentence_transformers里面的document数据类型)，于是我们需要用from langchain_core.documents.base import Document然后将String封装成稍微的document，才方便后序继续用langchain的API做Split。
2. Split步骤是使用langchain.text_splitter.RecursiveCharacterTextSplitter定义一个text_spliter对象，该对象在被构造时常常会传入两个简单的超参数：chunk_size以及chunk_overlap，功能如其名。

   继而用text_spliter去对Load得到的文件内容做Split即可了。

   为什么在RAG中要有Split这个步骤？我想，是为了方便后面做Retrive，毕竟split得到的文本基本单位chunk也是Retrive的基本单位。那么，有没有可能是Embedding model的输入文本长度不能过长呢？我觉得应该不是，因为此处chunk_size的超参数似乎是程序猿随意自选，与后序真正用到Embedding model的store以及retrieve步骤没有强关联。于是，可以进一步猜测Embedding model的工作原理：Embedding model面对太长的输入能截断，面对太短的输入能补全，这一点可以根据服务器上所用m3e-based Embedding model的结构是基于max _seq_length=512的Transformer得到验证。

3. Store步骤是使用的langchain.vectorstores.Chroma(其实也是langchain封装的Chroma库)，也就是langchain封装的Chroma，它起到一个在内存中划一块地存知识库，方便后序Retrieve步骤的作用。这里的Chroma除了要传入经过Split步骤得到的Splited Docs外，还要传入Embedding model。Embedding model是起什么作用的呢？Chroma划出来的存储空间存的不是Split后的文本，而是Split后的文本经过Embedding model推理得出的高维向量空间(本项目用到的m3d-based Embedding model输出是768维)。但是有一个问题，langchian提供的Chroma的API要求的Embedding model参数类型只能是langchain内置的embedding_function数据类型，该数据类型需要用langchain.embeddings.sentence_transformer。SentenceTransformerEmbeddings API来封装(其实也是langchain封装的sentence_transformers库)，但该API在langchain官方文档里显示出的用法是“要传入指定Embedding model的名称，然后该API自动给你从Hugging Face上将该model下载到本地，然后提取权重得到embedding_function变量，传入Chroma API，进行Store步骤”。但问题来了，我本地已经有Embedding model的权重与配置文件了，我还需要你来给我下？但是langchain目前只提供者一种得到embedding_function的API。于是，我怀着试试的心态去读了一下langchain的源码，发现好像我直接给SentenceTransformerEmbeddings API传入Embedding model在本地的路径也可以，我试了试，果然成功。

   看来，langchain的源码目前还不完善，文档也不完善，周边生态也不太完善，例如，衍生品LangSmith就还在内测中，我现在还在wait list中。

4. Retrive步骤，即是输入用户手打的提示文本，然后调用Embedding model也将用户输入的文本映射成高维向量，然后用此高维向量与Chroma中存储的文件被Split后映射成的高维向量计算余弦相似度，然后选择相似度最大的几个Splited docs作为被Retrieve得到的知识库中的文本，然后将该文本与用户手打的提示文本按照一定格式拼接组合在一起(langchain也提供现成的拼接模板，具体模板长什么样，和LLM训练时使用的输入数据有关，模板的格式与训练时输入数据越相近，则LLM推理准确度越高)，喂给LLM。

   上述过程也是由langchain框架实现的，其中有两个常用的可控超参数，一个是Retrieve出来的文本个数，一个是Retrive的方式，方式一般都是默认的上述提到的“余弦相似度”，但也可以是别的。

5. Prompt步骤，可以举个例子说明。首先，要有一个prompt_template，例如“你是一个无所不知的专家，现在有一个人类向你问了一个问题，你要如实回答，不能有纰漏与错别字。\n 这个人类的问题是：{} \n 可以参考的知识是:{} \n 你的回答是：”，第一个{}里填写用户提出的问题，第二个{}里填写Retriev步骤得到的文本序列。

   然后将填写完成后的结果喂给LLM即可。

   注意：prompt_template可以是用户自定义，也可以在网上(例如Langchain的官方文档)查找现有的实现。我猜测每一种LLM应该都有其对应的最合适的prompt_template，我猜测每一种LLM的作者都会在开源发行版中给出prompt_template，毕竟LLM在训练过程中假如就固定用某种prompt_template来实现，那在推理时，若继续沿用这种prompt_template，则推理效果应该是最好的。

3.2 核心代码

本项目的核心代码有3份，分别是：

- llm_server.py：后端代码。让LLM在服务器上跑起来，并处理前端发来的prompt。注意，此处的prompt已经是被RAG处理过的文本了。此代码同时能控制LLM处理多模态(带有图片)的prompt。
- chainlit_reasoner.py：前端代码。chainlit_chat.py发过来的是用户问题、文档、图片，此代码要针对文档做RAG，并将RAG的结果与图片与用户问题用prompt_template拼接在一起喂给运行中的LLM。
- chainlit_chat.py：使用chainlit框架实现的前端页面。

具体代码如下：

llm_server.py

```
import sys
import os
# 测试修改
# 获取当前脚本文件的绝对路径
script_path = os.path.abspath(__file__)
# 获取当前脚本的父目录的父目录（即 config 和 llm 所在的目录）
project_dir = os.path.dirname(os.path.dirname(script_path))
# 将 project_dir 添加到 sys.path
if project_dir not in sys.path:
```

```
    sys.path.append(project_dir)

import uvicorn
import json
import datetime
import requests
from PIL import Image
from fastapi import FastAPI, Request
from fastapi.middleware.cors import CORSMiddleware
from transformers import LlavaForConditionalGeneration, AutoModelForCausalLM,AutoTokenizer, AutoProcessor
from config.model_config import *
from typing import List, Tuple
import io
import base64

DEVICE = "cuda"
DEVICE_ID = "0"
CUDA_DEVICE = f"{DEVICE}:{DEVICE_ID}" if DEVICE_ID else DEVICE

def torch_gc():
    if torch.cuda.is_available():
        with torch.cuda.device(CUDA_DEVICE):
            torch.cuda.empty_cache()
            torch.cuda.ipc_collect()

app = FastAPI()

origins = ["*"]

app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

@app.get("/")
async def root():
    return {"message": "Hello World"}

@app.get("/_stcore/health")
async def health_check():
    return {"message": "healthy"}

@app.get("/_stcore/allowed-message-origins")
async def allowed_message_origins_check():
    return {"message": "allowed"}

@app.post("/")
async def create_item(request: Request):
    global model, tokenizer
    json_post_raw = await request.json()
    json_post = json.dumps(json_post_raw)
    json_post_list = json.loads(json_post)
    prompt = json_post_list.get('prompt')
    history = json_post_list.get('history')
    max_length = json_post_list.get('max_length')
    top_p = json_post_list.get('top_p')
    # top_k = json_post_list.get('top_k')
    temperature = json_post_list.get('temperature')
    print(prompt)
```

```
    response, history = chat_with_history(query=prompt,
                                          history=history,
                                          max_length=max_length if max_length else 2048,
                                          top_p=top_p if top_p else 0.7,
                                          # top_k=top_k if top_k else 1,
                                          do_sample=True,
                                          temperature=temperature if temperature else 1e-8)
    now = datetime.datetime.now()
    time = now.strftime("%Y-%m-%d %H:%M:%S")
    answer = {
        "response": response,
        "history": history,
        "status": 200,
        "time": time
    }
    log = "[" + time + "] " + '", prompt:"' + prompt + '", response:"' + repr(response) + '"'
    print(log)
    torch_gc()
    return answer

@torch.inference_mode()
def chat_with_history(query: str, history: List[Tuple[str, str]] = None, **gen_kwargs):
    """
    带有历史信息的对话.
    :param query: 用户输入的对话语句.
    :param history: 历史对话信息, (query, response)组成的List.
    :param gen_kwargs: 生成参数, 参考
https://huggingface.co/docs/transformers/v4.33.2/en/main_classes/text_generation#transformers.GenerationConfig.
    :return: response: str, history: List[Tuple[str, str]]
    """
    global model, tokenizer
    if history is None:
        history = []
    prompt = "A chat between a curious user and an artificial intelligence assistant. The assistant gives helpful, " \
             "detailed, and polite answers to the user's questions.\n\n"
    for (old_query, response) in history:
        prompt += f"USER: {old_query}\nASSISTANT: {response}</s>\n"
    prompt += f"USER: {query}\nASSISTANT: "
    input_ids = tokenizer(prompt, return_tensors='pt').input_ids.cuda()
    output = model.generate(inputs=input_ids, **gen_kwargs)
    output = output.tolist()[0][len(input_ids[0]):]
    response = tokenizer.decode(output)
    history = history + [(query, response)]
    return response, history

if __name__ == '__main__':
    model_path = llm_model_dict['vicuna-13b-v1.5-GPTQ']['local_model_path']
    # model = LlavaForConditionalGeneration.from_pretrained(model_path, device_map="auto")
    model = AutoModelForCausalLM.from_pretrained(model_path, device_map="auto")
    # processor= AutoProcessor.from_pretrained(model_path)
    tokenizer=AutoTokenizer.from_pretrained(model_path)
    model.eval()
    uvicorn.run(app, host='0.0.0.0', port=25, workers=1)
```

chainlit_reasoner.py

```python
import sys
from config.prompt_config import *
from config.model_config import *
from dao.knowledge_dao import KnowledgeDAO
from util.llm_request import request_llm
from util.embedding import calculate_embedding
import faiss
import chainlit as cl
import asyncio

from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.vectorstores import Chroma
from langchain.embeddings import HuggingFaceEmbeddings
from langchain.embeddings.sentence_transformer import SentenceTransformerEmbeddings
from langchain import hub
from langchain_core.documents.base import Document
import docx
import io
from PIL import Image
import base64

# @singleton
class ChainlitReasoner:
    def __init__(self):
        self.knowledge_dao = KnowledgeDAO()
        self.qa_knowledge = self.knowledge_dao.load_qa_knowledge()
        self.faiss_index = faiss.IndexFlatL2(self.qa_knowledge['question_embeddings'].shape[1])
        self.faiss_index.add(self.qa_knowledge['question_embeddings'])
        self.response_history = []

    def test(self):
        ...

    async def simple_answer(self, user_question, response_history, **gen_kwargs):
        '''这里传给LLM的prompt是光秃秃的user_question，其实理应加一些prompt template的'''
        llm_response = request_llm(user_question.content, response_history, **gen_kwargs)
        await cl.Message(content=llm_response['response'].replace('</s>', '')).send()
        return llm_response['response'], llm_response['history']

    async def reason_answer(self, user_question, response_history, knowledge_top_k=3, **gen_kwargs):
        root_id = await cl.Message(content="Reasoning from knowledge...").send()
        await cl.Message(author="Question", content=f"{user_question}", parent_id=root_id).send()

        await cl.Message(author="Faiss", content=f'Searching related questions by calculating vector similarity.', parent_id=root_id).send()
        user_question_embedding = calculate_embedding([user_question.content])
        scores, indices = self.faiss_index.search(user_question_embedding, knowledge_top_k)
        # limits, distances, indices = self.faiss_index.range_search(user_question_embedding, threshold)
        question_knowledge_list = []
        related_question_list = [self.qa_knowledge['question_list'][i] for i in indices.tolist()[0]]
        # print(scores.tolist()[0])
        if scores.tolist()[0][0] > 120:
            related_question_list = []
        await cl.Message(author="Faiss", content=f'Searching result: {str(related_question_list)}', parent_id=root_id).send()

        await cl.Message(author="KG", content=f'Searching related knowledge.', parent_id=root_id).send()
        for knowledge in self.qa_knowledge['knowledge']:
            if knowledge[0] in related_question_list:
                question_knowledge_list.append(knowledge)
        for ontology in self.qa_knowledge['ontology']:
            if ontology[0] in related_question_list:
```

```
                question_knowledge_list.append(ontology)
        await cl.Message(author="KG", content=f'Searching result: {str(question_knowledge_list)}',
parent_id=root_id).send()

        final_prompt = get_ticket_prompt(question_knowledge_list, user_question.content)
        await cl.Message(author="Prompt", content=f'{str(final_prompt)}', parent_id=root_id).send()
        llm_response = request_llm(final_prompt, response_history, **gen_kwargs)

        # print(len(llm_response['response'].replace('</s>', '')))
        await cl.Message(content=llm_response['response'].replace('</s>', '')).send()

        return llm_response['response'], llm_response['history']


    async def RAG_answer(self, user_question, response_history, **gen_kwargs):
        '''该if判断本次QA是否是RAG-based,若否,则跳转至simple_answer'''
        if user_question.elements==[]:
            return await self.simple_answer(user_question, response_history, **gen_kwargs)

        question=user_question.content
        file=user_question.elements[0]

        images_str=None
        prompt=None
        if file.name.split(".")[-1] in {"docx"}:
            file_stream = io.BytesIO(file.content)
            doc = docx.Document(file_stream)
            # Extracting text
            full_text = []
            for para in doc.paragraphs:
                full_text.append(para.text)
            full_text = '\n'.join(full_text)
            page_content=full_text

            '''下面这段被注释掉的代码是从docx文件中读取图片'''
            # # Extracting images
            # images = []
            # for rel in doc.part.rels.values():
            #     if "image" in rel.reltype:
            #         img = rel.target_part.blob
            #         images.append(img)

            prompt=self.get_RAG_prompt(file,page_content,question)
        elif file.name.split(".")[-1] in {"png","jpg"}:
            images_str=base64.b64encode(file.content).decode('utf-8')
            prompt=question
        elif file.name.split(".")[-1] in {"txt"} :
            page_content=file.content
            prompt=self.get_RAG_prompt(file,page_content,question)

        print(prompt)
        #LLM
        llm_response = request_llm(prompt=prompt, history=response_history, images_str=images_str,
**gen_kwargs)

        await cl.Message(content=llm_response['response'].replace('</s>', '')).send()

        return llm_response['response'], llm_response['history']


    def get_RAG_prompt(self,file,page_content,question):
        metadata={"id":file.id, "name": file.name, "mime": file.mime}
```

```
        docs = [Document(page_content=page_content, metadata=metadata)]
        #document split
        text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
        splits = text_splitter.split_documents(docs)
        #document vectorstore
        embedding_function=SentenceTransformerEmbeddings(model_name=embedding_model_dict['m3e-base'])

        vectorstore = Chroma.from_documents(documents=splits, embedding=embedding_function)
        #document retrieve
        retriever = vectorstore.as_retriever(search_type="similarity", search_kwargs={"k": 6})
        #prompt
        # prompt_template = hub.pull("rlm/rag-prompt")
        def prompt_template(context:str,question:str):
            return f"My question is: {question}\n\n There's some related information: {context}"

        def format_docs(docs):
            return "\n\n".join(doc.page_content for doc in docs)

        retrieved_docs = retriever.get_relevant_documents(question)
        context=format_docs(retrieved_docs)

        prompt=prompt_template(context, question)
        return prompt
```

chainlit_chat.py

```
import sys
import os
# 获取当前脚本文件的绝对路径
script_path = os.path.abspath(__file__)
# 获取当前脚本的父目录的父目录（即 config 和 llm 所在的目录）
project_dir = os.path.dirname(os.path.dirname(script_path))
# 将 project_dir 添加到 sys.path
if project_dir not in sys.path:
    sys.path.append(project_dir)

import chainlit as cl
from chainlit.input_widget import Select, Slider
from reasoner.chainlit_reasoner import ChainlitReasoner

chainlit_reasoner = ChainlitReasoner()

@cl.on_chat_start
async def start():
    settings = await cl.ChatSettings(
        [
            Select(
                id="Model",
                label="Base Large Language Model",
                values=["Vicuna-13B-v1.5-GPTQ"],
                initial_index=0,
            ),
            Select(
                id="Context",
                label="Context Mode",
                values=["No Context", "Within the Historical Context"],
                initial_index=0,
            ),
```

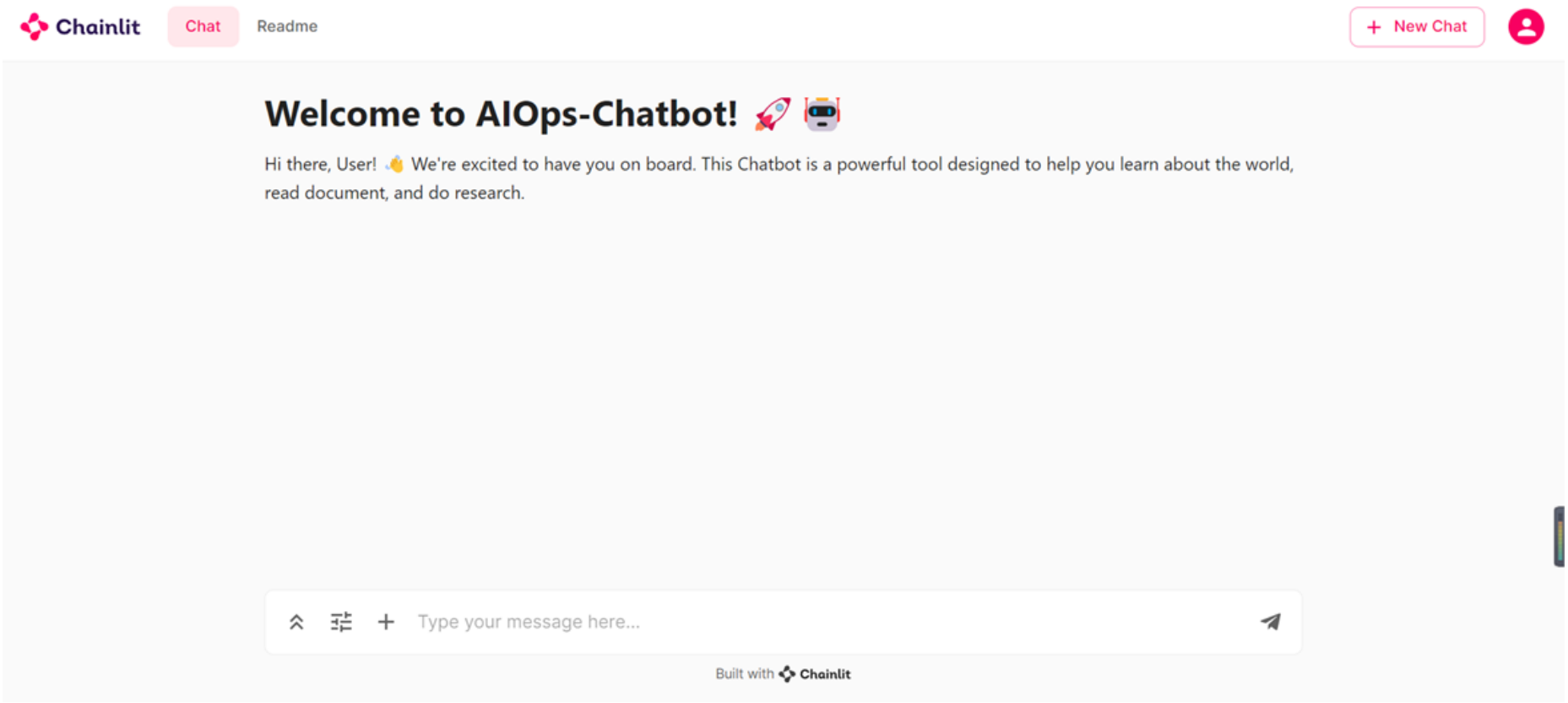
```
            Select(
                id="Mode",
                label="Knowledge Mode",
                values=["Free","KG-based"],
                initial_index=0,
            ),
            Slider(
                id="Temperature",
                label="Temperature",
                initial=0,
                min=0,
                max=1,
                step=0.1,
            )
        ]
    ).send()
    await setup_agent(settings)
    cl.user_session.set("response_history", [])

@cl.on_settings_update
async def setup_agent(settings):
    cl.user_session.set("settings", settings)

@cl.on_message
async def main(message: str):
    settings = cl.user_session.get("settings")
    context_mode, knowledge_mode, temperature = settings['Context'], settings['Mode'],
settings['Temperature']
    response_history = cl.user_session.get("response_history") if settings['Context'] == "Within the
Historical Context" else []

    if knowledge_mode == 'Free':
        _, response_history = await chainlit_reasoner.RAG_answer(message, response_history,
temperature=temperature)
    elif knowledge_mode == 'KG-based':
        _, response_history = await chainlit_reasoner.reason_answer(message, response_history,
knowledge_top_k=1, temperature=temperature)

    if settings['Context'] == "Within the Historical Context":
        cl.user_session.set("response_history", response_history)
```

3.2 效果测试

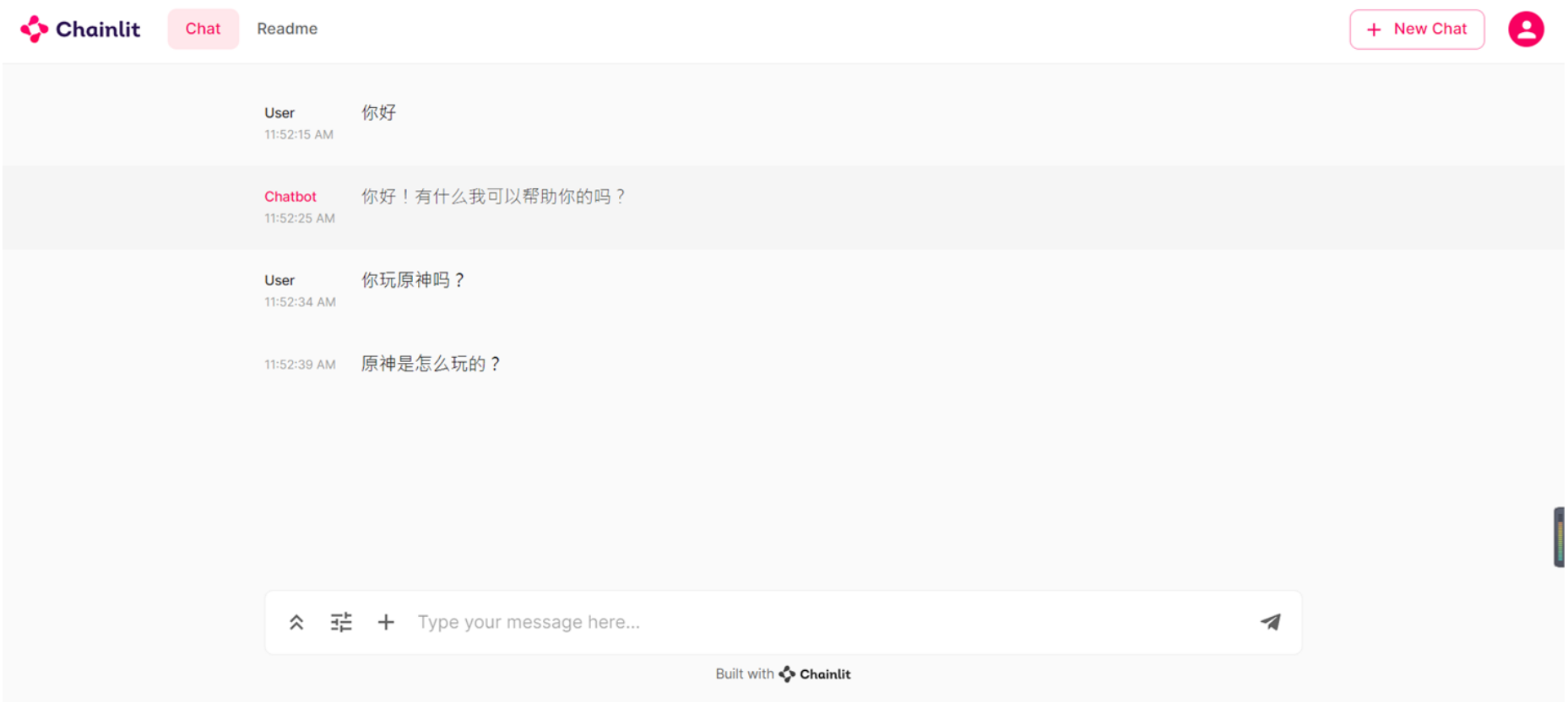
本项目目前使用的实验环境是一张RTX4090，由于算力资源的限制，故而每次用户提出问题，服务器上的LLM都会推理很久。而由于chainlit框架在长时间得不到后端响应后就会停止向后端请求，所以ChatBot用户界面上无法显示LLM的推理结果(此问题尚待后期解决)。但是后端服务器上的命令行是可以打印出LLM的推理结果的，故表明问题仅仅出现在chainlit框架的时延设置问题——当然，根本原因还是在于算力资源的问题。

故而，下述很多实验测试结果将不会以ChatBot用户界面的形式呈现，而是以后端服务器命令行打印结果的形式呈现。

- 问答页面



- 用户询问普通文本：
  - 用户界面结果(关于原神的问句太长，LLM推理时间过长，chainlit不予请求推理结果，即不显示与该界面中)



  - 服务器命令行打印结果

```
/workspace/LLMenv/lib/python3.8/site-packages/transformers/generation/configuration_utils.py:394: UserWarning: `do_sam
ple` is set to `False`. However, `top_p` is set to `0.6` -- this flag is only used in sample-based generation modes. Y
ou should set `do_sample=True` or unset `top_p`. This was detected when initializing the generation config instance, w
hich means the corresponding file may hold incorrect parameterization and should be fixed.
  warnings.warn(
INFO:     Started server process [941]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:25 (Press CTRL+C to quit)
你好
/workspace/LLMenv/lib/python3.8/site-packages/transformers/generation/utils.py:1518: UserWarning: You have modified th
e pretrained model configuration to control generation. This is a deprecated strategy to control generation and will b
e removed soon, in a future version. Please use and modify the model generation configuration (see https://huggingface
.co/docs/transformers/generation_strategies#default-text-generation-configuration )
  warnings.warn(
[2024-01-13 03:40:58] ", prompt:"你好", response:"'你好！有什么我可以帮助你的吗？</s>'"
INFO:     127.0.0.1:41444 - "POST / HTTP/1.1" 200 OK
你玩原神吗？
[2024-01-13 03:41:57] ", prompt:"你玩原神吗？", response:"'作为一个人工智能，我没有实际的感官，所以我不能玩游戏。但是
，我可以回答关于原神的问题，帮助您更好地了解这款游戏。请问有什么问题我可以帮您解答吗？</s>'"
INFO:     127.0.0.1:60952 - "POST / HTTP/1.1" 200 OK
原神是怎么玩的？
```

```
^C^C[2024-01-13 03:47:01] ", prompt:"原神是怎么玩的？", response:"'原神是一款由中国游戏公司miHoYo制作的开放世界动作角
色扮演游戏。游戏的主要玩法包括探索世界、完成任务、挖掘资源、升级角色、与其他玩家互动等。\n\n在游戏中，玩家扮演一名神秘
的旅行者，探索一个充满神秘和奇幻元素的世界。游戏中有许多不同的地区和环境，每个地区都有自己的怪物、资源和难度等级。玩家
需要在这些地区中探索，完成任务和挖掘资源，以升级角色和获得更强大的装备。\n\n除了探索和完成任务之外，玩家还可以与其他玩
家互动，加入团队或创建自己的团队，一起完成更具挑战性的任务和冒险。游戏中还有许多其他的玩法，如游戏内购买、角色交换等，
使游戏更加丰富多彩。\n\n总的来说，原神是一款非常丰富多彩的游戏，需要玩家在探索、完成任务、升级角色和与其他玩家互动等方
面进行不断的尝试和探索。</s>'"
INFO:     127.0.0.1:49294 - "POST / HTTP/1.1" 200 OK
INFO:     Shutting down
INFO:     Finished server process [941]
```

- 用户输入文档(使用RAG功能)

  - 服务器命令行打印结果(对于文档的推理太费算力，前端用户界面已经无法显示了)

```
/workspace/LLMenv/lib/python3.8/site-packages/transformers/generation/utils.py:1518: UserWarning: You have modified th
e pretrained model configuration to control generation. This is a deprecated strategy to control generation and will b
e removed soon, in a future version. Please use and modify the model generation configuration (see https://huggingface
.co/docs/transformers/generation_strategies#default-text-generation-configuration )
  warnings.warn(
[2024-01-13 05:05:50] ", prompt:"My question is: 这篇文档的内容是什么?

 There's some related information: 对于 SOAP 来讲，比如我创建一个订单，用 POST，在 XML 里面写明动作是 CreateOrder；删
除一个订单，还是用 POST，在 XML 里面写明了动作是 DeleteOrder。其实创建订单完全可以使用 POST 动作，然后在 XML 里面放一
个订单的信息就可以了，而删除用 DELETE 动作，然后在 XML 里面放一个订单的 ID 就可以了。
    于是上面的那个 SOAP 就变成下面这个简单的模样。
POST /purchaseOrder HTTP/1.1
Host: www.cnblog.com
Content-Type: application/xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
 <order>
     <date>2018-07-01</date>
      <className> 板栗焖鸡 </className>
       <price>58</price>
  </order>", response:"'这篇文档主要是讨论 SOAP 协议中的动作（action）和其在 HTTP 请求中的表示方式。根据文档中提到的信
息，对于 SOAP 协议，创建订单和删除订单都可以使用 POST 动作，并在 XML 中明确指定动作名称。例如，在上述示例中，创建订单
的动作名称为 CreateOrder，删除订单的动作名称为 DeleteOrder。\n\n在这个简化的示例中，创建订单的 HTTP 请求使用 POST 动作
，并在请求正文中包含订单信息的 XML 数据。删除订单的 HTTP 请求使用 DELETE 动作，并在请求正文中包含要删除的订单 ID 的 XM
L 数据。这样的请求模式简化了 SOAP 协议的表现形式，使其更加简洁易懂。</s>'"
INFO:     127.0.0.1:46340 - "POST / HTTP/1.1" 200 OK
```

- 用户输入图片，并询问“请问该图片的内容是什么？”

  - 后端服务器命令行打印结果(喂给Llava-13B-1.5-GPTQ 50*50的图片，在RTX4090上已经会推理长达10分钟左右了，若是喂两张，则显存爆掉)

```
INFO:     Started server process [1245802]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:25 (Press CTRL+C to quit)
<image>
A chat between a curious user and an artificial intelligence assistant. The assistant gives helpful, detailed, and poli
te answers to the user's questions.

USER: 描述此图片内容
ASSISTANT:

A chat between a curious user and an artificial intelligence assistant. The assistant gives helpful, detailed, and poli
te answers to the user's questions.

USER: 描述此图片内容
ASSISTANT: 这是一个笑谈的表情符号，它看起来像一个笑谈的人的脸。它有眼睛和嘴巴，并且它的眼睛看起来像它在笑。这个表情符号
可以用来表达快乐和友好的情绪。
[2023-12-26 19:24:18] ", prompt:"描述此图片内容", response:""\nA chat between a curious user and an artificial intellig
ence assistant. The assistant gives helpful, detailed, and polite answers to the user's questions.\n\nUSER: 描述此图片
内容\nASSISTANT: 这是一个笑谈的表情符号，它看起来像一个笑谈的人的脸。它有眼睛和嘴巴，并且它的眼睛看起来像它在笑。这个表
情符号可以用来表达快乐和友好的情绪。 ""
INFO:     127.0.0.1:48536 - "POST / HTTP/1.1" 200 OK
```

4. 收获与展望

本项目相当于满足前述用户需求的一个聊天机器人的原型设计，相当于第一个增量，只是实现了基本的功能，且功能演示收到算力资源的显示并不能呈现出一个良好的效果。除了算力资源带来的限制外，本项目当前采用了虽然方方便但功能尚不足够强大的chainlit框架搭建前端，用户交互体验也尚不够好。同时，本项目对于多模态的支持也仅限于让用户直接输入文本+图片，但不能做到图片粘贴在文档里然后喂给LLM。这些缺陷都有待后期增量逐步完善。

但是，本项目却让我们有了一份实操LLM部署与ChatBot制作的经验，让我们打通了ChatBot的基本技术栈，并浅尝了LLM的生态，这在近几年LLM爆火成为新热点的情势下，对我们充实相关项目经验是非常有利的。