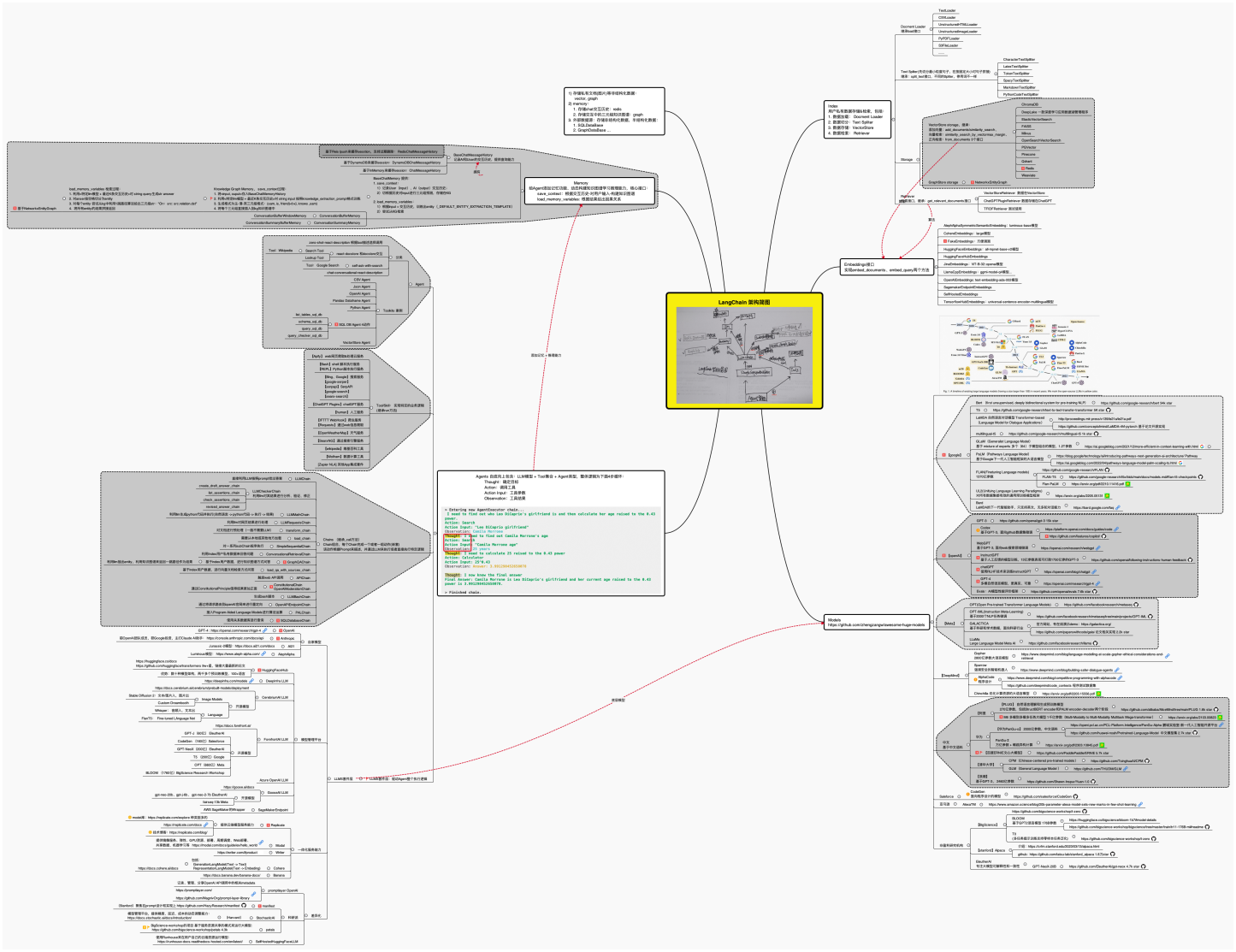


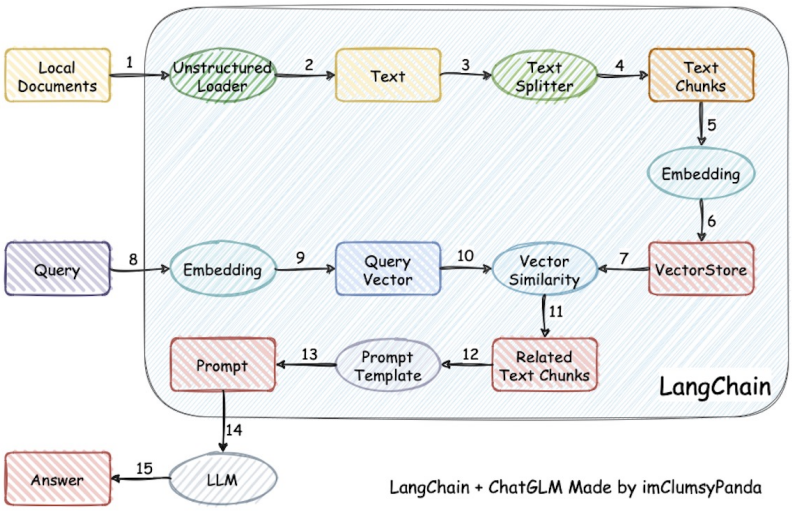
第二部分 基于LangChain + ChatGLM-6B(23年7月初版)的本地知识库问答

2.1 核心步骤：如何通过LangChain+LLM实现本地知识库问答

2023年7月，GitHub上有一个利用 langchain 思想实现的基于本地知识库的问答应用：langchain-ChatGLM (*这是其GitHub地址，当然还有和它类似的但现已支持Vicuna-13b的项目，比如LangChain-ChatGLM-Webui*)，目标期望建立一套对中文场景与开源模型支持友好、可离线运行的知识库问答解决方案

- 该项目受 GanymedeNil 的项目 document.ai，和 AlexZhangji 创建的 ChatGLM-6B Pull Request 启发，建立了全流程可使用开源模型实现的本地知识库问答应用。现已支持使用 ChatGLM-6B、 ClueAI/ChatYuan-large-v2 等大语言模型的接入
- 该项目中 Embedding 默认选用的是 GanymedeNil/text2vec-large-chinese，LLM 默认选用的是 ChatGLM-6B，依托上述模型，本项目可实现全部使用**开源**模型**离线私有部署**

本项目实现原理如下图所示 (*与基于文档的问答 大同小异，过程包括：1 加载文档 -> 2 读取文档 -> 3/4文档分割 -> 5/6 文本向量化 -> 8/9 问句向量化 -> 10 在文档向量中匹配出与问句向量最相似的top k个 -> 11/12/13 匹配出的文本作为上下文和问题一起添加到prompt中 -> 14/15提交给LLM生成回答*)



1. **第一阶段：加载文件-读取文件-文本分割(Text splitter)**
   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

2. **第二阶段：文本向量化(embedding)-存储到向量数据库**
   文本向量化(embedding)：这通常涉及到NLP的特征抽取，可以通过诸如TF-IDF、word2vec、BERT等方法将分割好的文本转化为数值向量
   存储到向量数据库：文本向量化之后存储到数据库vectorstore (*FAISS，下一节会详解FAISS*)

```python
def init_vector_store(self):
    persist_dir = os.path.join(VECTORE_PATH, ".vectordb")  # 持久化向量数据库的地址
    print("向量数据库持久化地址: ", persist_dir)              # 打印持久化地址

    # 如果持久化地址存在
    if os.path.exists(persist_dir):
        # 从本地持久化文件中加载
        print("从本地向量加载数据...")
        # 使用 Chroma 加载持久化的向量数据
        vector_store = Chroma(persist_directory=persist_dir, embedding_function=self.embeddings)

    # 如果持久化地址不存在
    else:
        # 加载知识库
        documents = self.load_knownlege()
        # 使用 Chroma 从文档中创建向量存储
        vector_store = Chroma.from_documents(documents=documents,
                                             embedding=self.embeddings,
                                             persist_directory=persist_dir)
        vector_store.persist()      # 持久化向量存储
    return vector_store             # 返回向量存储
```

其中load_knownlege的实现为

```python
def load_knownlege(self):
    docments = []         # 初始化一个空列表来存储文档

    # 遍历 DATASETS_DIR 目录下的所有文件
    for root, _, files in os.walk(DATASETS_DIR, topdown=False):
        for file in files:
            filename = os.path.join(root, file)      # 获取文件的完整路径
            docs = self._load_file(filename)         # 加载文件中的文档

            # 更新 metadata 数据
            new_docs = []             # 初始化一个空列表来存储新文档
            for doc in docs:
                # 更新文档的 metadata，将 "source" 字段的值替换为不包含 DATASETS_DIR 的相对路径
                doc.metadata = {"source": doc.metadata["source"].replace(DATASETS_DIR, "")}
                print("文档2向量初始化中，请稍等...", doc.metadata)  # 打印正在初始化的文档的 metadata
                new_docs.append(doc)  # 将文档添加到新文档列表

            docments += new_docs      # 将新文档列表添加到总文档列表

    return docments      # 返回所有文档的列表
```

3. **第三阶段：问句向量化**
   这是将用户的查询或问题转化为向量，应使用与文本向量化相同的方法，以便在相同的空间中进行比较

4. **第四阶段：在文本向量中匹配出与问句向量最相似的top k个**
   这一步是信息检索的核心，通过计算余弦相似度、欧氏距离等方式，找出与问句向量最接近的文本向量

```
    def query(self, q):
        """在向量数据库中查找与问句向量相似的文本向量"""
        vector_store = self.init_vector_store()
        docs = vector_store.similarity_search_with_score(q, k=self.top_k)
        for doc in docs:
            dc, s = doc
            yield s, dc
```

5. **第五阶段：匹配出的文本作为上下文和问题一起添加到prompt中**
   这是利用匹配出的文本来形成与问题相关的上下文，用于输入给语言模型

6. **第六阶段：提交给LLM生成回答**
   最后，将这个问题和上下文一起提交给语言模型(例如GPT系列)，让它生成回答
   比如知识查询(代码来源)

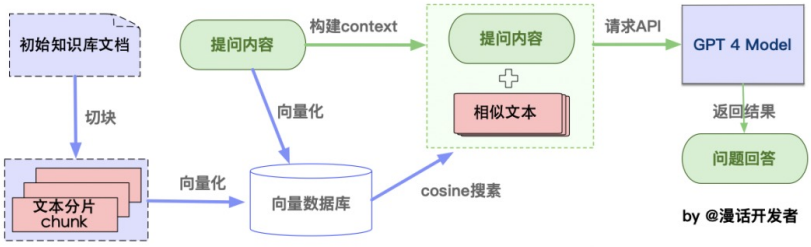
```
class KnownLedgeBaseQA:
    # 初始化
    def __init__(self) -> None:
        k2v = KnownLedge2Vector()      # 创建一个知识到向量的转换器
        self.vector_store = k2v.init_vector_store()     # 初始化向量存储
        self.llm = VicunaLLM()         # 创建一个 VicunaLLM 对象

    # 获得与查询相似的答案
    def get_similar_answer(self, query):
        # 创建一个提示模板
        prompt = PromptTemplate(
            template=conv_qa_prompt_template,
            input_variables=["context", "question"]  # 输入变量包括 "context"（上下文） 和 "question"（问题）
        )

        # 使用向量存储来检索文档
        retriever = self.vector_store.as_retriever(search_kwargs={"k": VECTOR_SEARCH_TOP_K})
        docs = retriever.get_relevant_documents(query=query)  # 获取与查询相关的文本

        context = [d.page_content for d in docs]     # 从文本中提取出内容
        result = prompt.format(context="\n".join(context), question=query) # 格式化模板，并用从文本中提取出的内容和问题填充
        return result                 # 返回结果
```

如你所见，这种通过组合langchain+LLM的方式，特别适合一些垂直领域或大型集团企业搭建通过LLM的智能对话能力搭建企业内部的私有问答系统，也适合个人专门针对一些英文paper进行问答，比如比较火的一个开源项目：ChatPDF，其从文档处理角度来看，实现流程如下(图源)：



2.2 Facebook AI Similarity Search(FAISS)：高效向量相似度检索

Faiss的全称是Facebook AI Similarity Search (官方介绍页、GitHub地址)，是FaceBook的AI团队针对大规模相似度检索问题开发的一个工具，使用C++编写，有python接口，对10亿量级的索引可以做到毫秒级检索的性能

简单来说，Faiss的工作，就是把我们自己的候选向量集封装成一个index数据库，它可以加速我们检索相似向量TopK的过程，其中有些索引还支持GPU构建

2.2.1 Faiss检索相似向量TopK的基本流程

Faiss检索相似向量TopK的工程基本都能分为三步：

1. 得到向量库

```
import numpy as np
d = 64                                           # 向量维度
```

```
nb = 100000                                    # index向量库的数据量
nq = 10000                                     # 待检索query的数目
np.random.seed(1234)
xb = np.random.random((nb, d)).astype('float32')
xb[:, 0] += np.arange(nb) / 1000.              # index向量库的向量
xq = np.random.random((nq, d)).astype('float32')
xq[:, 0] += np.arange(nq) / 1000.              # 待检索的query向量
```

2. 用faiss 构建index，并将向量添加到index中
   其中的构建索引选用暴力检索的方法FlatL2，L2代表构建的index采用的相似度度量方法为L2范数，即欧氏距离

```
import faiss
index = faiss.IndexFlatL2(d)
print(index.is_trained)          # 输出为True，代表该类index不需要训练，只需要add向量进去即可
index.add(xb)                    # 将向量库中的向量加入到index中
print(index.ntotal)              # 输出index中包含的向量总数，为100000
```

3. 用faiss index 检索，检索出TopK的相似query

```
k = 4                       # topK的K值
D, I = index.search(xq, k)# xq为待检索向量，返回的I为每个待检索query最相似TopK的索引list，D为其对应的距离
print(I[:5])
print(D[-5:])
```

打印输出为：

```
>>>
[[  0 393 363  78]
 [  1 555 277 364]
 [  2 304 101  13]
 [  3 173  18 182]
 [  4 288 370 531]]
[[ 0.       7.17517328  7.2076292   7.25116253]
 [ 0.       6.32356453  6.6845808   6.79994535]
 [ 0.       5.79640865  6.39173603  7.28151226]
 [ 0.       7.27790546  7.52798653  7.66284657]
 [ 0.       6.76380348  7.29512024  7.36881447]]
```

2.2.2 FAISS构建索引的多种方式

构建index方法和传参方法可以为

```
dim, measure = 64, faiss.METRIC_L2
param = 'Flat'
index = faiss.index_factory(dim, param, measure)
```

- dim为向量维数
- 最重要的是param参数，它是传入index的参数，代表需要构建什么类型的索引；
- measure为度量方法，目前支持两种，欧氏距离和inner product，即内积。因此，要计算余弦相似度，只需要将vecs归一化后，使用内积度量即可

此文，现在faiss官方支持八种度量方式，分别是：

1. METRIC_INNER_PRODUCT（内积）
2. METRIC_L1（曼哈顿距离）
3. METRIC_L2（欧氏距离）
4. METRIC_Linf（无穷范数）
5. METRIC_Lp（p范数）
6. METRIC_BrayCurtis（BC相异度）
7. METRIC_Canberra（兰氏距离/堪培拉距离）
8. METRIC_JensenShannon（JS散度）

2.2.2.1 Flat： 暴力检索

- 优点：该方法是Faiss所有index中最准确的，召回率最高的方法，没有之一；
- 缺点：速度慢，占内存大。
- 使用情况：向量候选集很少，在50万以内，并且内存不紧张。
- 注：虽然都是暴力检索，**faiss的暴力检索速度比一般程序猿自己写的暴力检索要快上不少**，所以并不代表其无用武之地，建议有暴力检索需求的同学还是用下faiss。
- 构建方法：

```
dim, measure = 64, faiss.METRIC_L2
param = 'Flat'
index = faiss.index_factory(dim, param, measure)
index.is_trained                                    # 输出为True
index.add(xb)                                       # 向index中添加向量
```

2.2.2.2 IVFx Flat： 倒排暴力检索

- 优点：IVF主要利用倒排的思想，在文档检索场景下的倒排技术是指，一个kw后面挂上很多个包含该词的doc，由于kw数量远远小于doc，因此会大大减少了检索的时间。在向量中如何使用倒排呢？可以拿出每个聚类中心下的向量ID，每个中心ID后面挂上一堆非中心向量，每次查询向量的时候找到最近的几个中心ID，分别搜索这几个中心下的非中心向量。通过减小搜索范围，提升搜索效率。
- 缺点：速度也还不是很快。
- 使用情况：相比Flat会大大增加检索的速度，建议百万级别向量可以使用。
- 参数：IVFx中的x是k-means聚类中心的个数
- 构建方法：

```
dim, measure = 64, faiss.METRIC_L2
param = 'IVF100,Flat'                           # 代表k-means聚类中心为100,
index = faiss.index_factory(dim, param, measure)
print(index.is_trained)                          # 此时输出为False，因为倒排索引需要训练k-means，
index.train(xb)                                  # 因此需要先训练index，再add向量
index.add(xb)
```

2.2.2.3 PQx： 乘积量化

- 优点：利用乘积量化的方法，改进了普通检索，将一个向量的维度切成x段，每段分别进行检索，每段向量的检索结果取交集后得出最后的TopK。因此速度很快，而且占用内存较小，召回率也相对较高。
- 缺点：召回率相较于暴力检索，下降较多。
- 使用情况：内存及其稀缺，并且需要较快的检索速度，不那么在意召回率
- 参数：PQx中的x为将向量切分的段数，因此，**x需要能被向量维度整除**，且x越大，切分越细致，时间复杂度越高
- 构建方法：

```
dim, measure = 64, faiss.METRIC_L2
param =  'PQ16'
index = faiss.index_factory(dim, param, measure)
print(index.is_trained)                          # 此时输出为False，因为倒排索引需要训练k-means，
index.train(xb)                                  # 因此需要先训练index，再add向量
index.add(xb)
```

2.2.2.4 IVFxPQy 倒排乘积量化

- 优点：工业界大量使用此方法，各项指标都均可以接受，利用乘积量化的方法，改进了IVF的k-means，将一个向量的维度切成x段，每段分别进行k-means再检索。
- 缺点：集百家之长，自然也集百家之短
- 使用情况：一般来说，各方面没啥特殊的极端要求的话，**最推荐使用该方法！**
- 参数：IVFx，PQy，其中的x和y同上
- 构建方法：

```
dim, measure = 64, faiss.METRIC_L2
param =  'IVF100,PQ16'
index = faiss.index_factory(dim, param, measure)
print(index.is_trained)                          # 此时输出为False，因为倒排索引需要训练k-means，
index.train(xb)                                  # 因此需要先训练index，再add向量 index.add(xb)
```

2.2.2.5 LSH 局部敏感哈希

- 原理：哈希对大家再熟悉不过，向量也可以采用哈希来加速查找，我们这里说的哈希指的是局部敏感哈希（Locality Sensitive Hashing，LSH），不同于传统哈希尽量不产生碰撞，局部敏感哈希依赖碰撞来查找近邻。高维空间的两点若距离很近，那么设计一种哈希函数对这两点进行哈希计算后分桶，使得他们哈希分桶值有很大的概率是一样的，若两点之间的距离较远，则他们哈希分桶值相同的概率会很小。
- 优点：训练非常快，支持分批导入，index占内存很小，检索也比较快
- 缺点：召回率非常拉垮。
- 使用情况：候选向量库非常大，离线检索，内存资源比较稀缺的情况
- 构建方法：

```
dim, measure = 64, faiss.METRIC_L2
param =  'LSH'
index = faiss.index_factory(dim, param, measure)
```

```
print(index.is_trained)                          # 此时输出为True
index.add(xb)
```

2.2.2.6 HNSWx

- 优点：该方法为基于图检索的改进方法，检索速度极快，10亿级别秒出检索结果，而且召回率几乎可以媲美Flat，最高能达到惊人的97%。检索的时间复杂度为loglogn，几乎可以无视候选向量的量级了。并且支持分批导入，极其适合线上任务，毫秒级别体验。
- 缺点：构建索引极慢，占用内存极大（是Faiss中最大的，大于原向量占用的内存大小）
- 参数：HNSWx中的x为构建图时每个点最多连接多少个节点，x越大，构图越复杂，查询越精确，当然构建index时间也就越慢，x取4~64中的任何一个整数。
- 使用情况：不在乎内存，并且有充裕的时间来构建index
- 构建方法：

```
dim, measure = 64, faiss.METRIC_L2
param =  'HNSW64'
index = faiss.index_factory(dim, param, measure)
print(index.is_trained)                          # 此时输出为True
index.add(xb)
```

2.3 项目部署：langchain + ChatGLM-6B搭建本地知识库问答

2.3.1 部署过程一：支持多种使用模式

其中的LLM模型可以根据实际业务的需求选定，本项目中用的ChatGLM-6B，其GitHub地址为：https://github.com/THUDM/ChatGLM-6B
ChatGLM-6B 是一个开源的、支持中英双语的对话语言模型，基于 General LanguageModel (GLM) 架构，具有 62 亿参数。结合模型量化技术，用户可以在消费级的显卡上进行本地部署（INT4 量化级别下最低只需 6GB 显存）

ChatGLM-6B 使用了和 ChatGPT 相似的技术，针对中文问答和对话进行了优化。经过约 1T 标识符的中英双语训练，辅以监督微调、反馈自助、人类反馈强化学习等技术的加持，62 亿参数的 ChatGLM-6B 已经能生成相当符合人类偏好的回答

1. 新建一个python3.8.13的环境（模型文件还是可以用的）

   ```
   conda create -n langchain python==3.8.13
   ```

2. 拉取项目

   ```
   git clone https://github.com/imClumsyPanda/langchain-ChatGLM.git
   ```

3. 进入目录

   ```
   cd langchain-ChatGLM
   ```

4. 安装requirements.txt

   ```
   conda activate langchain
   pip install -r requirements.txt
   ```

5. 当前环境支持装langchain的最高版本是0.0.166，无法安装0.0.174，就先装下0.0.166试下
   修改配置文件路径：

   ```
   vi configs/model_config.py
   ```

6. 将chatglm-6b的路径设置成自己的

   ```
   “chatglm-6b”: {
   “name”: “chatglm-6b”,
   “pretrained_model_name”: “/data/sim_chatgpt/chatglm-6b”,
   “local_model_path”: None,
   “provides”: “ChatGLM”
   ```

7. 修改要运行的代码文件：webui.py

   ```
   vi webui.py
   ```

8. 将最后launch函数中的share设置为True，inbrowser设置为True
9. 执行webui.py文件

   ```
   python webui.py
   ```

   可能是网络问题，无法创建一个公用链接。可以进行云服务器和本地端口的映射，参考：https://www.cnblogs.com/monologuesmw/p/14465117.html