

3_Notes_3

参考：<https://github.com/InternLM/tutorial/tree/main/langchain>

基于 InternLM 和 LangChain 搭建你的知识库

目录

- 1 大模型开发范式
- 2 LangChain 简介
- 3 构建向量数据库
- 4 搭建知识库助手
- 5 Web Demo 部署
- 6 动手实战环节

1 大模型开发范式

RAG VS Finetune

RAG VS Finetune

- 低成本
- 可实时更新
- 受基座模型影响大
- 单次回答知识有限
- 可个性化微调
- 知识覆盖面广
- 成本高昂
- 无法实时更新

2 LangChain

LangChain 框架是一个开源工具，通过为各种 LLM 提供通用接口来简化应用程序的开发流程，帮助开发者自由构建 LLM 应用。

LangChain 的核心组成模块：

- 链 (Chains)：将组件组合实现端到端应用，通过一个对象封装实现一系列 LLM 操作
- Eg. 检索问答链，覆盖实现了 RAG (检索增强生成) 的全部流程

3 构建向量数据库

加载源文件 → 文档分块 → 文档向量化

Open

- 确定源文件类型，针对不同类型源文件选用不同的加载器
 - 核心在于将带格式文本转化为无格式字符串
- 由于单个文档往往超过模型上下文上限，我们需要对加载的文档进行切分
 - 一般按字符串长度进行分割
 - 可以手动控制分割块的长度和重叠区间长度
- 使用向量数据库来支持语义检索，需要将文档向量化存入向量数据库
 - 可以使用任一种 Embedding 模型来进行向量化
 - 可以使用多种支持语义检索的向量数据库，一般使用轻量级的 Chroma

4 搭建知识库助手

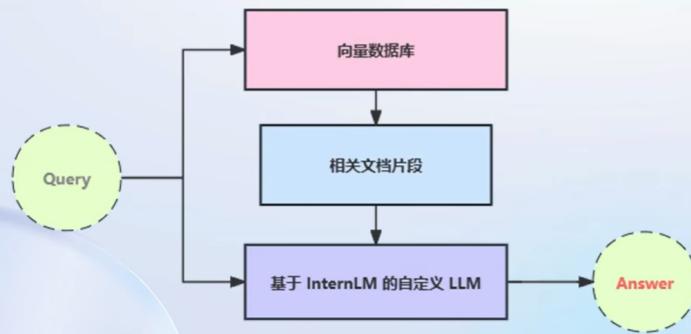
将 InternLM 接入 LangChain

- LangChain 支持自定义LLM，可以直接接入到框架中
- 我们只需将 InternLM 部署在本地，并封装一个自定义 LLM 类，调用本地 InternLM 即可

构建检索问答链



- LangChain 提供了检索问答链模版，可以自动实现知识检索、Prompt 嵌入、LLM 问答的全部流程
- 将基于 InternLM 的自定义 LLM 和已构建的向量数据库接入到检索问答链的上游
- 调用检索问答链，即可实现知识库助手的核心功能



RAG方案优化建议

- 基于RAG的问答系统性能核心受限于：
 - 检索精度
 - Prompt性能
- 一些可能的优化点：
 - 检索方面：
 - 基于语义进行分割，保证每一个chunk的语义完整
 - 给每一个chunk生成概括性索引，检索时匹配索引
 - Prompt方面：
 - 迭代优化Prompt策略

5 Web Demo 部署

OP

Web Demo 部署

- 有很多支持简易 Web 部署的框架，如 Gradio、Streamlit 等

The screenshot shows a dark-themed web application window titled "InternLM". At the top, there's a header bar with the title "InternLM" and a "书生源语" button. Below the header is a large text area containing a question and answer pair. The question is "什么是InternLM" and the answer is: "InternLM 是一个开源的轻量级训练框架，旨在支持大模型训练而无需大量的依赖。通过单一的代码库，它支持在拥有数千个 GPU 的大型集群上进行预训练，并在单个 GPU 上进行微调，同时实现了卓越的性能优化。" Below this text area is a "Prompt/问题" input field, followed by a "Chat" button and a "Clear console" button. At the bottom of the window, there are two links: "Use via API" and "使用Gradio构建" (Use Gradio to build).

6 动手实践

基础作业：复现课程知识库助手搭建过程 (截图)

1 LangChain 相关环境配置

在已完成 InternLM 的部署基础上，还需要安装以下依赖包：

```
pip install langchain==0.0.292
pip install gradio==4.4.0
pip install chromadb==0.4.15
```

```
pip install sentence-transformers==2.2.2
pip install unstructured==0.10.30
pip install markdown==3.3.7
```

同时，我们需要使用到开源词向量模型 Sentence Transformer: (我们也可以选用别的开源词向量模型来进行 Embedding，目前选用这个模型是相对轻量、支持中文且效果较好的，同学们可以自由尝试别的开源词向量模型)

首先需要使用 `huggingface` 官方提供的 `huggingface-cli` 命令行工具。安装依赖：

```
pip install -U huggingface_hub
```

然后在和 `/root/data` 目录下新建python文件 `download_hf.py`，填入以下代码：

- `resume-download`：断点续下
- `local-dir`：本地存储路径。 (linux环境下需要填写绝对路径)

```
import os

# 设置环境变量，使用 huggingface 镜像下载，不加的话下载失败了。
os.environ['HF_ENDPOINT'] = 'https://hf-mirror.com'

# 下载模型
os.system('huggingface-cli download --resume-download sentence-t
```

下载 NLTK 相关资源

我们在使用开源词向量模型构建开源词向量的时候，需要用到第三方库 `nltk` 的一些资源。正常情况下，其会自动从互联网上下载，但可能由于网络原因会导致下载中断，此处我们可以从国内仓库镜像地址下载相关资源，保存到服务器上。

我们用以下命令下载 `nltk` 资源并解压到服务器上：

```
cd /root/data
git clone https://gitee.com/yzy0612/nltk_data.git --branch gh-p
cd nltk_data
```

```
mv packages/* ./
cd tokenizers
unzip punkt.zip
cd ../taggers
unzip averaged_perceptron_tagger.zip
```

下载本项目代码

```
cd /root/code
git clone https://github.com/InternLM/tutorial
```

2 知识库搭建

2.1 数据收集

我们选择由上海人工智能实验室开源的一系列大模型工具开源仓库作为语料库来源，包括：

- OpenCompass：面向大模型评测的一站式平台
- IMDeploy：涵盖了 LLM 任务的全套轻量化、部署和服务解决方案的高效推理工具箱
- XTuner：轻量级微调大语言模型的工具库
- InternLM-XComposer：浦语·灵笔，基于书生·浦语大语言模型研发的视觉-语言大模型
- Lagent：一个轻量级、开源的基于大语言模型的智能体（agent）框架
- InternLM：一个开源的轻量级训练框架，旨在支持大模型训练而无需大量的依赖

首先我们需要将上述远程开源仓库 Clone 到本地，可以使用以下命令：

```
# 进入到数据库盘
cd /root/data
# clone 上述开源仓库
git clone https://gitee.com/open-compass/opencompass.git
git clone https://gitee.com/InternLM/lmdeploy.git
git clone https://gitee.com/InternLM/xtuner.git
```

```
git clone https://gitee.com/InternLM/InternLM-XComposer.git  
git clone https://gitee.com/InternLM/lagent.git  
git clone https://gitee.com/InternLM/InternLM.git
```

接着，为语料处理方便，我们将选用上述仓库中所有的 markdown、txt 文件作为示例语料库。

我们首先将上述仓库中所有满足条件的文件路径找出来，我们定义一个函数，该函数将递归指定文件夹路径，返回其中所有满足条件（即后缀名为 .md 或者 .txt 的文件）的文件路径：

```
import os  
def get_files(dir_path):  
    # args: dir_path, 目标文件夹路径  
    file_list = []  
    for filepath, dirnames, filenames in os.walk(dir_path):  
        # os.walk 函数将递归遍历指定文件夹  
        for filename in filenames:  
            # 通过后缀名判断文件类型是否满足要求  
            if filename.endswith(".md"):  
                # 如果满足要求，将其绝对路径加入到结果列表  
                file_list.append(os.path.join(filepath, filename))  
            elif filename.endswith(".txt"):  
                file_list.append(os.path.join(filepath, filename))  
    return file_list
```

2.2 加载数据

得到所有目标文件路径之后，我们可以使用 LangChain 提供的 FileLoader 对象来加载目标文件，得到由目标文件解析出的纯文本内容。由于不同类型的文件需要对应不同的 FileLoader，我们判断目标文件类型，并针对性调用对应类型的 FileLoader，同时，调用 FileLoader 对象的 load 方法来得到加载之后的纯文本对象：

```
from tqdm import tqdm  
from langchain.document_loaders import UnstructuredFileLoader  
from langchain.document_loaders import UnstructuredMarkdownLoader
```

```
def get_text(dir_path):
    # args : dir_path, 目标文件夹路径
    # 首先调用上文定义的函数得到目标文件路径列表
    file_lst = get_files(dir_path)
    # docs 存放加载之后的纯文本对象
    docs = []
    # 遍历所有目标文件
    for one_file in tqdm(file_lst):
        file_type = one_file.split('.')[-1]
        if file_type == 'md':
            loader = UnstructuredMarkdownLoader(one_file)
        elif file_type == 'txt':
            loader = UnstructuredFileLoader(one_file)
        else:
            # 如果是不符合条件的文件，直接跳过
            continue
        docs.extend(loader.load())
    return docs
```

使用上文函数，我们得到的 `docs` 为一个纯文本对象对应的列表。

2.3 构建向量数据库

得到该列表之后，我们就可以将它引入到 LangChain 框架中构建向量数据库。由纯文本对象构建向量数据库，我们需要先对文本进行分块，接着对文本块进行向量化。

LangChain 提供了多种文本分块工具，此处我们使用字符串递归分割器，并选择分块大小为 500，块重叠长度为 150（由于篇幅限制，此处没有展示切割效果，学习者可以自行尝试一下，想要深入学习 LangChain 文本分块可以参考教程 [《LangChain - Chat With Your Data》](#)：

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=500, chunk_overlap=150)
split_docs = text_splitter.split_documents(docs)
```

接着我们选用开源词向量模型 [Sentence Transformer](#) 来进行文本向量化。LangChain 提供了直接引入 HuggingFace 开源社区中的模型进行向量化的接口：

```
from langchain.embeddings.huggingface import HuggingFaceEmbedding
embeddings = HuggingFaceEmbeddings(model_name="/root/model/sente
```

同时，考虑到 Chroma 是目前最常用的入门数据库，我们选择 Chroma 作为向量数据库，基于上文分块后的文档以及加载的开源向量化模型，将语料加载到指定路径下的向量数据库：

```
from langchain.vectorstores import Chroma
# 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
)
# 将加载的向量数据库持久化到磁盘上
vectordb.persist()
```

3 InternLM 接入 LangChain

为便捷构建 LLM 应用，我们需要基于本地部署的 InternLM，继承 LangChain 的 LLM 类自定义一个 InternLM LLM 子类，从而实现将 InternLM 接入到 LangChain 框架中。完成 LangChain 的自定义 LLM 子类之后，可以以完全一致的方式调用 LangChain 的接口，而无需考虑底层模型调用的不一致。

基于本地部署的 InternLM 自定义 LLM 类并不复杂，我们只需从 LangChain.llms.base.LLM 类继承一个子类，并重写构造函数与 `_call` 函数即可：

```
from langchain.llms.base import LLM
from typing import Any, List, Optional
```

```
from langchain.callbacks.manager import CallbackManagerForLLMRun
from transformers import AutoTokenizer, AutoModelForCausalLM
import torch

class InternLM_llm(LLM):
    # 基于本地 InternLM 自定义 LLM 类
    tokenizer : AutoTokenizer = None
    model: AutoModelForCausalLM = None

    def __init__(self, model_path :str):
        # model_path: InternLM 模型路径
        # 从本地初始化模型
        super().__init__()
        print("正在从本地加载模型...")
        self.tokenizer = AutoTokenizer.from_pretrained(model_path)
        self.model = AutoModelForCausalLM.from_pretrained(model_path)
        self.model = self.model.eval()
        print("完成本地模型的加载")

    def _call(self, prompt : str, stop: Optional[List[str]] = None,
             run_manager: Optional[CallbackManagerForLLMRun] = None,
             **kwargs: Any):
        # 重写调用函数
        system_prompt = """You are an AI assistant whose name is
- InternLM (书生·浦语) is a conversational language model
- InternLM (书生·浦语) can understand and communicate fluently
"""
        messages = [(system_prompt, '')]
        response, history = self.model.chat(self.tokenizer, prompt)
        return response

    @property
    def _llm_type(self) -> str:
        return "InternLM"
```

在上述类定义中，我们分别重写了构造函数和 `_call` 函数：对于构造函数，我们在对象实例化的一开始加载本地部署的 InternLM 模型，从而避免每一次调用都需要重新加载模型带来的时间过长；`_call` 函数是 LLM 类的核心函数，LangChain 会调用该函数来调用 LLM，在该函数中，我们调用已实例化模型的 `chat` 方法，从而实现对模型的调用并返回调用结果。

4 构建检索问答链

LangChain 通过提供检索问答链对象来实现对于 RAG 全流程的封装。所谓检索问答链，即通过一个对象完成检索增强问答（即RAG）的全流程，针对 RAG 的更多概念，我们会在视频内容中讲解，也欢迎读者查阅该教程来进一步了解：[《LLM Universe》](#)。我们可以调用一个 LangChain 提供的 `RetrievalQA` 对象，通过初始化时填入已构建的数据库和自定义 LLM 作为参数，来简便地完成检索增强问答的全流程，LangChain 会自动完成基于用户提问进行检索、获取相关文档、拼接为合适的 Prompt 并交给 LLM 问答的全部流程。

4.1 加载向量数据库

首先我们需要将上文构建的向量数据库导入进来，我们可以直接通过 Chroma 以及上文定义的词向量模型来加载已构建的数据库：

```
from langchain.vectorstores import Chroma
from langchain.embeddings.huggingface import HuggingFaceEmbedder
import os

# 定义 Embeddings
embeddings = HuggingFaceEmbeddings(model_name="/root/model/sente

# 向量数据库持久化路径
persist_directory = 'data_base/vector_db/chroma'

# 加载数据库
vectordb = Chroma(
    persist_directory=persist_directory,
```

```
embedding_function=embeddings  
)
```

上述代码得到的 `vectordb` 对象即为我们已构建的向量数据库对象，该对象可以针对用户的 `query` 进行语义向量检索，得到与用户提问相关的知识片段。

4.2 实例化自定义 LLM 与 Prompt Template

我们实例化一个基于 InternLM 自定义的 LLM 对象：

```
from LLM import InternLM_LLM  
llm = InternLM_LLM(model_path = "/root/model/Shanghai_AI_Laborat  
llm.predict("你是谁")
```

构建检索问答链，还需要构建一个 Prompt Template，该 Template 其实基于一个带变量的字符串，在检索之后，LangChain 会将检索到的相关文档片段填入到 Template 的变量中，从而实现带知识的 Prompt 构建。我们可以基于 LangChain 的 Template 基类来实例化这样一个 Template 对象：

```
from langchain.prompts import PromptTemplate  
  
# 我们所构造的 Prompt 模板  
template = """使用以下上下文来回答用户的问题。如果你不知道答案，就说你不知  
总是使用中文回答。  
问题: {question}  
可参考的上下文：  
...  
{context}  
...  
如果给定的上下文无法让你做出回答，请回答你不知道。  
有用的回答: """  
  
# 调用 LangChain 的方法来实例化一个 Template 对象，该对象包含了 contex  
# question 两个变量，在实际调用时，这两个变量会被检索到的文档片段和用户提  
QA_CHAIN_PROMPT = PromptTemplate(input_variables=["context", "que
```

4.3 构建检索问答链

最后，可以调用 LangChain 提供的检索问答链构造函数，基于我们的自定义 LLM、Prompt Template 和向量知识库来构建一个基于 InternLM 的检索问答链：

```
from langchain.chains import RetrievalQA

qa_chain = RetrievalQA.from_chain_type(llm,retriever=vectoradb.as
```

得到的 `qa_chain` 对象即可以实现我们的核心功能，即基于 InternLM 模型的专业知识库助手。我们可以对比该检索问答链和纯 LLM 的问答效果：

```
# 检索问答链回答效果
question = "什么是InternLM"
result = qa_chain({"query": question})
print("检索问答链回答 question 的结果:")
print(result["result"])

# 仅 LLM 回答效果
result_2 = llm(question)
print("大模型回答 question 的结果:")
print(result_2)
```

5 部署 Web Demo

在完成上述核心功能后，我们可以基于 `Gradio` 框架将其部署到 Web 网页，从而搭建一个小型 Demo，便于测试与使用。

我们首先将上文的代码内容封装为一个返回构建的检索问答链对象的函数，并在启动 `Gradio` 的第一时间调用该函数得到检索问答链对象，后续直接使用该对象进行问答对话，从而避免重复加载模型：

```
from langchain.vectorstores import Chroma
from langchain.embeddings.huggingface import HuggingFaceEmbedding
import os
from LLM import InternLM_LLM
from langchain.prompts import PromptTemplate
```

```
from langchain.chains import RetrievalQA

def load_chain():
    # 加载问答链
    # 定义 Embeddings
    embeddings = HuggingFaceEmbeddings(model_name="/root/model/some_model")

    # 向量数据库持久化路径
    persist_directory = 'data_base/vector_db/chroma'

    # 加载数据库
    vectordb = Chroma(
        persist_directory=persist_directory,  # 允许我们将persist_directory设为None
        embedding_function=embeddings
    )

    # 加载自定义 LLM
    llm = InternLM_LLM(model_path = "/root/model/Shanghai_AI_Lab/InternLM-7B")

    # 定义一个 Prompt Template
    template = """使用以下上下文来回答最后的问题。如果你不知道答案，就说你知道不了。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"])

    # 运行 chain
    qa_chain = RetrievalQA.from_chain_type(llm,retriever=vectordb,
                                            chain_type="stuff",
                                            prompt=QA_CHAIN_PROMPT)

    return qa_chain
```

接着我们定义一个类，该类负责加载并存储检索问答链，并响应 Web 界面里调用检索问答链进行回答的动作：

```
class Model_center():
    """
    存储检索问答链的对象
    """

    def __init__(self):
        # 构造函数，加载检索问答链
        self.chain = load_chain()

    def qa_chain_self_answer(self, question: str, chat_history):
        """
        调用问答链进行回答
        """

        if question == None or len(question) < 1:
            return "", chat_history
        try:
            chat_history.append(
                (question, self.chain({"query": question})["result"]))
            # 将问答结果直接附加到问答历史中，Gradio 会将其展示出来
            return "", chat_history
        except Exception as e:
            return e, chat_history
```

然后我们只需按照 Gradio 的框架使用方法，实例化一个 Web 界面并将点击动作绑定到上述类的回答方法即可：

```
import gradio as gr

# 实例化核心功能对象
model_center = Model_center()
# 创建一个 Web 界面
block = gr.Blocks()
with block as demo:
    with gr.Row(equal_height=True):
        with gr.Column(scale=15):
            # 展示的页面标题
```

```

gr.Markdown(""""<h1><center>InternLM</center></h1>
             <center>书生浦语</center>
             """")

with gr.Row():
    with gr.Column(scale=4):
        # 创建一个聊天机器人对象
        chatbot = gr.Chatbot(height=450, show_copy_button=True)
        # 创建一个文本框组件，用于输入 prompt。
        msg = gr.Textbox(label="Prompt/问题")

    with gr.Row():
        # 创建提交按钮。
        db_wo_his_btn = gr.Button("Chat")
    with gr.Row():
        # 创建一个清除按钮，用于清除聊天机器人组件的内容。
        clear = gr.ClearButton(
            components=[chatbot], value="Clear console")

    # 设置按钮的点击事件。当点击时，调用上面定义的 qa_chain_self_answer 方法。
    db_wo_his_btn.click(model_center.qa_chain_self_answer, inputs=[msg, chatbot], outputs=[msg, chatbot])

gr.Markdown(""""提醒：  

1. 初始化数据库时间可能较长，请耐心等待。  

2. 使用中如果出现异常，将会在文本输入框进行展示，请不要惊慌。<br>
""")
```

gr.close_all()
直接启动
demo.launch()

通过将上述代码封装为 run_gradio.py 脚本，直接通过 python 命令运行，即可在本地启动知识库助手的 Web Demo，默认会在 7860 端口运行，接下来将服务器端口映射到本地端口即可访问：



Homework_3