# LangChain 系列教程（八）：综合实战

## 介绍

该篇文章将给出一个利用LangChain实现RAG的示例程序。这段程序将会体现出LangChain六个核心模块的应用。

RAG（Retrieval-Augmented Generation，检索增强生成）是一种结合文档检索与语言生成的技术框架，能够让大模型在生成回答前先从外部知识中检索相关内容，从而提升准确性与上下文一致性。

## 代码

### 环境配置

# 设置环境变量，配置 API 地址与密钥（使用第三方 OpenAI 镜像）  
import os  
os.environ["OPENAI\_API\_BASE"] = "https://api.chatanywhere.tech"  
os.environ["OPENAI\_API\_KEY"] = 'sk-xxx'

### 加载文档并保存至向量数据库，以实现Retrieval

# -------------------- 文档加载与切分 --------------------  
from langchain\_community.document\_loaders import PyPDFLoader, Docx2txtLoader, TextLoader  
  
base\_dir = 'Docs' # 文档所在文件夹路径  
documents = [] # 存放加载后的 Document 对象  
  
# 遍历文件夹下所有文件，根据文件类型调用不同的加载器  
for file in os.listdir(base\_dir):  
 file\_path = os.path.join(base\_dir, file)  
 if file.endswith('.pdf'):  
 loader = PyPDFLoader(file\_path)  
 documents.extend(loader.load())  
 elif file.endswith('.docx'):  
 loader = Docx2txtLoader(file\_path)  
 documents.extend(loader.load())  
 elif file.endswith('.txt'):  
 loader = TextLoader(file\_path)  
 documents.extend(loader.load())  
  
# 文本切分器：将长文档切分为多个段落块，chunk\_size 为每段最大长度  
from langchain.text\_splitter import RecursiveCharacterTextSplitter  
text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=200, chunk\_overlap=10)  
chunked\_documents = text\_splitter.split\_documents(documents)

# -------------------- 嵌入 + 存储到向量数据库 --------------------  
from langchain\_community.vectorstores import Qdrant  
from langchain\_openai import OpenAIEmbeddings  
  
# 将切分后的文档嵌入为向量，并临时存储在 Qdrant 的内存数据库中  
vectorstore = Qdrant.from\_documents(  
 documents=chunked\_documents,  
 embedding=OpenAIEmbeddings(),  
 location=":memory:", # 仅存于内存，适合教学/测试  
 collection\_name="my\_documents"  
)

### 构建Chains

这边构建了两条链，一条RAG问答链、一条摘要链。

# -------------------- 构建 RAG 问答链 --------------------  
import logging  
from langchain\_openai import ChatOpenAI  
from langchain.retrievers.multi\_query import MultiQueryRetriever  
from langchain.memory import ConversationBufferMemory  
from langchain.chains import ConversationalRetrievalChain  
  
# 打开多查询检索日志，方便调试  
logging.basicConfig()  
logging.getLogger('langchain.retrievers.multi\_query').setLevel(logging.INFO)  
  
# 初始化语言模型（OpenAI GPT-3.5）  
llm = ChatOpenAI(model\_name="gpt-3.5-turbo", temperature=0)  
  
# 使用 MultiQueryRetriever，让模型自动生成多个查询，提高召回率  
retriever\_from\_llm = MultiQueryRetriever.from\_llm(retriever=vectorstore.as\_retriever(), llm=llm)  
  
# 初始化对话记忆  
memory = ConversationBufferMemory(memory\_key="chat\_history", return\_messages=True)  
memory.clear()  
  
# 创建 ConversationalRetrievalChain，实现 RAG 问答（带上下文记忆）  
qa\_chain = ConversationalRetrievalChain.from\_llm(llm=llm, retriever=retriever\_from\_llm, memory=memory, verbose=True)  
  
# 提问示例：首轮提问  
result = qa\_chain({"question": "什么是MCP？"})  
print(result["answer"])  
  
# 追问上下文问题：测试记忆能力  
print(qa\_chain({"question": "它的使用方法是什么？"})["answer"])

# -------------------- 文档摘要链（Map-Reduce） --------------------  
from langchain.chains.summarize import load\_summarize\_chain  
  
summ\_chain = load\_summarize\_chain(llm=llm, chain\_type="map\_reduce", verbose=True)  
  
def qa\_with\_summary(question: str):  
 docs = retriever\_from\_llm.get\_relevant\_documents(question)  
 summary = summ\_chain.run(docs)  
 return summary  
  
print(qa\_with\_summary("请用 100 字总结 MCP 的核心概念"))

### 定义Agent

agent能调用三个工具，除了上述的两条链之外还能调用python解释器。

# -------------------- Agent 多工具组合 --------------------  
from langchain.tools import Tool  
from langchain\_experimental.tools.python.tool import PythonREPLTool  
from langchain.agents import initialize\_agent, AgentType  
  
# 定义一个函数，用于通过Retrieval QA链回答用户的问题  
def retriever\_tool\_func(q):  
 return qa\_chain({"question": q})["answer"] # 调用qa\_chain处理问题，并提取答案部分  
  
# 创建一个用于文档检索问答的工具  
retrieval\_tool = Tool(name="RAG\_QA", func=retriever\_tool\_func, description="对一般知识类问题，先检索文档再回答")  
  
# 创建一个用于文档摘要的工具  
summary\_tool = Tool(name="RAG\_Summary", func=qa\_with\_summary, description="当用户要求概括/总结时使用")  
  
# 创建一个用于执行Python代码的工具  
python\_tool = PythonREPLTool()  
  
# 将上述工具组合成一个工具列表，供Agent使用  
tools = [retrieval\_tool, summary\_tool, python\_tool]  
  
# 初始化一个智能体（Agent），配置如下：  
# - 使用OpenAI的GPT-3.5模型（llm）  
# - 提供的工具列表（tools）  
# - 使用ZERO\_SHOT\_REACT\_DESCRIPTION类型的Agent（基于ReAct框架）  
# - 启用详细日志输出（verbose=True）  
# - 使用会话记忆（memory）以保持上下文  
agent = initialize\_agent(  
 tools=tools, # 提供的工具列表  
 llm=llm, # 使用的语言模型  
 agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION, # Agent的类型  
 verbose=True, # 是否输出详细的日志信息  
 memory=memory # 会话记忆，用于保持对话的上下文  
)  
  
# 示例：触发总结工具  
print(agent.run("MCP 的作用能一步总结给我吗？"))  
  
# 示例：触发 Python 计算工具  
print(agent.run("计算 (27\*13)+9 等于多少？"))  
  
# 示例：触发文档检索问答工具  
print(agent.run("MCP 和 LangChain Output Parser 有何区别？"))

### 增加Callbacks

# -------------------- 回调打印输出事件（可用于调试、打字机效果） --------------------  
from langchain\_core.callbacks import BaseCallbackHandler  
  
class SimpleCallbackHandler(BaseCallbackHandler):  
 def on\_llm\_start(self, serialized, prompts, \*\*kwargs):  
 print("\n🔵 LLM started")  
  
 def on\_llm\_new\_token(self, token, \*\*kwargs):  
 print(token, end="", flush=True)  
  
 def on\_llm\_end(self, response, \*\*kwargs):  
 print("\n🟢 LLM finished")  
  
 def on\_chain\_end(self, outputs, \*\*kwargs):  
 print(f"\n🟡 Chain outputs: {list(outputs.keys())}")  
  
callback\_handler = SimpleCallbackHandler()  
  
# Agent 执行带回调（打字机效果）  
agent.run("LangChain 的作用能一步总结给我吗？", callbacks=[callback\_handler])