从LangChain+LLM的本地知识库问答到LLM与知识图谱、数据库的结合

前言

过去半年，随着ChatGPT的火爆，直接带火了整个LLM这个方向，然LLM毕竟更多是基于过去的经验数据 预训练　而来，没法获取最新的知识，以及各企业私有的知识

- 为了获取最新的知识，ChatGPT plus版集成了bing搜索的功能，有的模型则会调用一个定位于 “链接各种AI模型、工具的langchain”的bing功能
- 为了处理企业私有的知识，要么基于开源模型微调，要么也可以基于langchain的思想调取一个外挂的向量知识库(类似存在本地的数据库一样)

所以越来越多的人开始关注langchain并把它与LLM结合起来应用，更直接推动了 数据库　、知识图谱与LLM的结合应用

本文侧重讲解

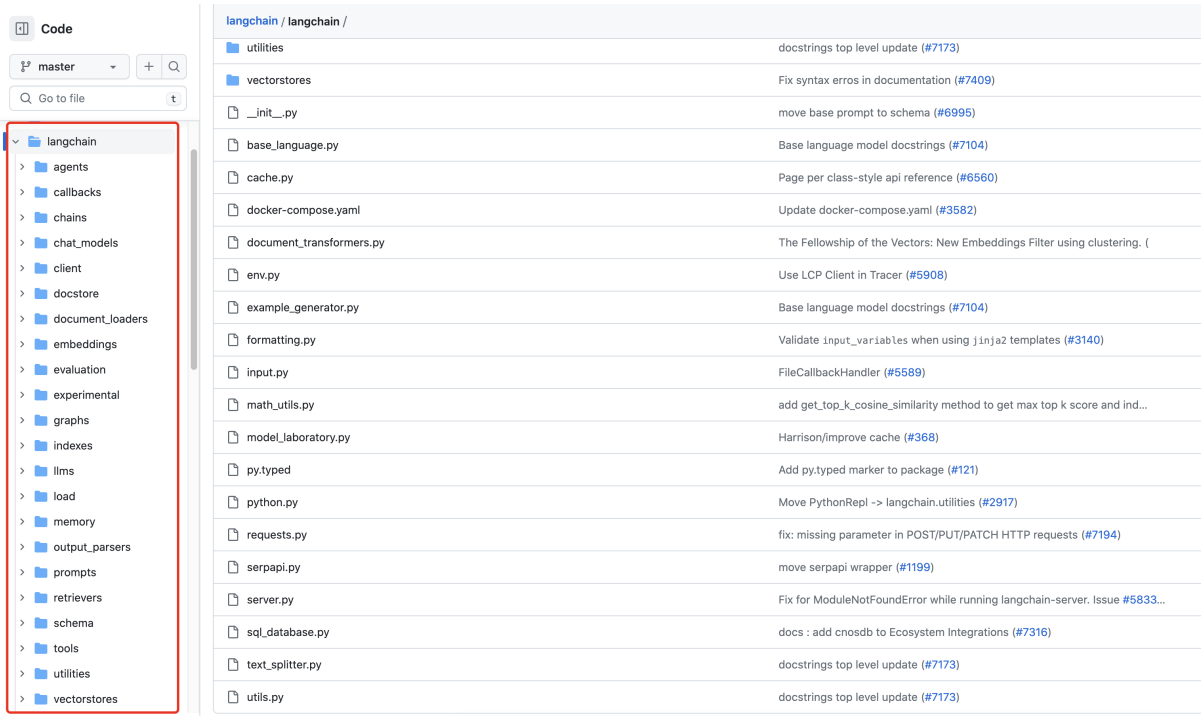
- LLM与langchain/数据库/知识图谱的结合应用
  比如，虽说基于知识图谱的问答 早在2019年之前就有很多研究了，但谁会想到今年KBQA因为LLM如此突飞猛进呢
  再比如，还会解读langchain-ChatGLM项目的关键源码，不只是把它当做一个工具使用，因为对工具的原理更了解，则对工具的使用更顺畅
- 其中，解读langchain-ChatGLM项目源码其实不易，因为涉及的项目、技术点不少，所以一开始容易绕晕，好在根据该项目的流程一步步抽丝剥茧之后，给大家呈现了清晰的代码架构
  过程中，**我从接触该langchain-ChatGLM项目到整体源码梳理清晰并写清楚历时了近一周，而大家有了本文之后，可能不到一天便可以理清了(提升近7倍效率)**，这便是本文的价值和意义之一

阅读过程中若有任何问题，欢迎随时留言，会一一及时回复/解答，共同探讨、共同深挖

第一部分 什么是LangChain：LLM的外挂/功能库

1.1 langchain的整体组成架构

通俗讲，所谓langchain (官网地址、GitHub地址)，即把AI中常用的很多功能都封装成库，且有调用各种商用模型API、开源模型的接口，支持以下各种组件



初次接触的朋友一看这么多组件可能直接晕了(封装的东西非常多，感觉它想把LLM所需要用到的功能/工具都封装起来)，为方便理解，我们可以先从大的层面把整个langchain库划分为三个大层：基础层、能力层、应用层

1.1.1 基础层：models、LLMs、index

Models：模型
各种类型的模型和模型集成，比如OpenAI的各个API/GPT-4等等，为各种不同基础模型提供统一接口
比如通过API完成一次问答

```
import os
os.environ["OPENAI_API_KEY"] = '你的api key'
from langchain.llms import OpenAI

llm = OpenAI(model_name="text-davinci-003",max_tokens=1024)
llm("怎么评价人工智能")
```

得到的回答如下图所示

```
>>> import os
>>> os.environ["OPENAI_API_KEY"] = 'sk-PFwMvMFcU7VXaL5cYM9vT[redacted]'
>>> from langchain.llms import OpenAI
>>>
>>> llm = OpenAI(model_name="text-davinci-003",max_tokens=1024)
>>> llm("怎么评价人工智能")
'\n\n人工智能是一门极具潜力的学科，可以帮助人类解决许多复杂的问题。近年来，人工智能的发展取得了长足的进步，已经成功应用于诸多领域，为人们的生活带来了极大的便利。未来，人工智能将发挥更大的作用，为人类的发展做出重大贡献。'
>>>
```

LLMS层
这一层主要强调对models层能力的封装以及服务化输出能力，主要有：

- 各类LLM模型管理平台：强调的模型的种类丰富度以及易用性
- 一体化服务能力产品：强调开箱即用
- 差异化能力：比如聚焦于Prompt管理(包括提示管理、提示优化和提示序列化)、基于共享资源的模型运行模式等等

比如Google's PaLM Text APIs，再比如 *llms/openai.py* 文件下

```python
        model_token_mapping = {
            "gpt-4": 8192,
            "gpt-4-0314": 8192,
            "gpt-4-0613": 8192,
            "gpt-4-32k": 32768,
            "gpt-4-32k-0314": 32768,
            "gpt-4-32k-0613": 32768,
            "gpt-3.5-turbo": 4096,
            "gpt-3.5-turbo-0301": 4096,
            "gpt-3.5-turbo-0613": 4096,
            "gpt-3.5-turbo-16k": 16385,
            "gpt-3.5-turbo-16k-0613": 16385,
            "text-ada-001": 2049,
            "ada": 2049,
            "text-babbage-001": 2040,
            "babbage": 2049,
            "text-curie-001": 2049,
            "curie": 2049,
            "davinci": 2049,
            "text-davinci-003": 4097,
            "text-davinci-002": 4097,
            "code-davinci-002": 8001,
            "code-davinci-001": 8001,
            "code-cushman-002": 2048,
            "code-cushman-001": 2048,
        }
```

Index(索引)：**Vector方案、KG方案**
对用户私域文本、图片、PDF等各类文档进行存储和检索(相当于结构化文档，以便让外部数据和模型交互)，具体实现上有两个方案：一个Vector方案、一个KG方案

对于Vector方案：即对文件先切分为Chunks，在按Chunks分别编码存储并检索，可参考此代码文件：langchain/libs/langchain/langchain/indexes /vectorstore.py
该代码文件依次实现

模块导入：导入了各种类型检查、数据结构、预定义类和函数
接下来，实现了**一个函数_get_default_text_splitter，两个类VectorStoreIndexWrapper、VectorstoreIndexCreator**

_get_default_text_splitter 函数：
这是一个私有函数，返回一个默认的文本分割器，它可以将文本递归地分割成大小为1000的块，且块与块之间有重叠

```python
# 默认的文本分割器函数
def _get_default_text_splitter() -> TextSplitter:
    return RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
```

接下来是，**VectorStoreIndexWrapper** 类：
这是一个包装类，主要是为了方便地访问和查询向量存储（Vector Store）

1. vectorstore: 一个向量存储对象的属性

```python
    vectorstore: VectorStore  # 向量存储对象

    class Config:
        """Configuration for this pydantic object."""

        extra = Extra.forbid            # 额外配置项
```

```
        arbitrary_types_allowed = True  # 允许任意类型
```

2. query: 一个方法，它接受一个问题字符串并查询向量存储来获取答案

```
# 查询向量存储的函数
def query(
    self,
    question: str,                                          # 输入的问题字符串
    llm: Optional[BaseLanguageModel] = None,                # 可选的语言模型参数，默认为None
    retriever_kwargs: Optional[Dict[str, Any]] = None,      # 提取器的可选参数，默认为None
    **kwargs: Any                                           # 其他关键字参数
) -> str:
    """Query the vectorstore."""                            # 函数的文档字符串，描述函数的功能

    # 如果没有提供语言模型参数，则使用OpenAI作为默认语言模型，并设定温度参数为0
    llm = llm or OpenAI(temperature=0)

    # 如果没有提供提取器的参数，则初始化为空字典
    retriever_kwargs = retriever_kwargs or {}

    # 创建一个基于语言模型和向量存储提取器的检索QA链
    chain = RetrievalQA.from_chain_type(
        llm, retriever=self.vectorstore.as_retriever(**retriever_kwargs), **kwargs
    )

    # 使用创建的QA链运行提供的问题，并返回结果
    return chain.run(question)
```

解释一下上面出现的提取器
提取器首先从大型语料库中检索与问题相关的文档或片段，然后生成器根据这些检索到的文档生成答案。

提取器可以基于许多不同的技术，包括：

a.基于关键字的检索：使用关键字匹配来查找相关文档
b.向量空间模型：将文档和查询都表示为向量，并通过计算它们之间的相似度来检索相关文档
c.基于深度学习的方法：使用预训练的神经网络模型（如BERT、RoBERTa等）将文档和查询编码为向量，并进行相似度计算
d.索引方法：例如倒排索引，这是搜索引擎常用的技术，可以快速找到包含特定词或短语的文档
这些方法可以独立使用，也可以结合使用，以提高检索的准确性和速度

3. query_with_sources: 类似于query，但它还返回与查询结果相关的数据源

```
    # 查询向量存储并返回数据源的函数
    def query_with_sources(
        self,
        question: str,
        llm: Optional[BaseLanguageModel] = None,
        retriever_kwargs: Optional[Dict[str, Any]] = None,
        **kwargs: Any
    ) -> dict:
        """Query the vectorstore and get back sources."""
        llm = llm or OpenAI(temperature=0)              # 默认使用OpenAI作为语言模型
        retriever_kwargs = retriever_kwargs or {}       # 提取器参数
        chain = RetrievalQAWithSourcesChain.from_chain_type(
            llm, retriever=self.vectorstore.as_retriever(**retriever_kwargs), **kwargs
        )
        return chain({chain.question_key: question})
```

最后是**VectorstoreIndexCreator** 类：

这是一个创建向量存储索引的类

1. vectorstore_cls: 使用的向量存储类，默认为Chroma

```
    vectorstore_cls: Type[VectorStore] = Chroma             # 默认使用Chroma作为向量存储类
```

一个简化的向量存储可以看作是一个大型的表格或数据库，其中每行代表一个项目（如文档、图像、句子等），而每个项目则有一个与之关联的高维向量。向量的维度可以从几十到几千，取决于所使用的嵌入模型
例如：

2. embedding: 使用的嵌入类，默认为OpenAIEmbeddings

```
    embedding: Embeddings = Field(default_factory=OpenAIEmbeddings)  # 默认使用OpenAIEmbeddings作为嵌入类
```

3. text_splitter: 用于分割文本的文本分割器

```
text_splitter: TextSplitter = Field(default_factory=_get_default_text_splitter)  # 默认文本分割器
```

4. from_loaders: 从给定的加载器列表中创建一个向量存储索引

```
    # 从加载器创建向量存储索引的函数
    def from_loaders(self, loaders: List[BaseLoader]) -> VectorStoreIndexWrapper:
        """Create a vectorstore index from loaders."""
        docs = []
        for loader in loaders:              # 遍历加载器
            docs.extend(loader.load())      # 加载文档
        return self.from_documents(docs)
```

5. from_documents: 从给定的文档列表中创建一个向量存储索引

```
    # 从文档创建向量存储索引的函数
    def from_documents(self, documents: List[Document]) -> VectorStoreIndexWrapper:
        """Create a vectorstore index from documents."""
        sub_docs = self.text_splitter.split_documents(documents)      # 分割文档
        vectorstore = self.vectorstore_cls.from_documents(
            sub_docs, self.embedding, **self.vectorstore_kwargs       # 从文档创建向量存储
        )
        return VectorStoreIndexWrapper(vectorstore=vectorstore)       # 返回向量存储的包装对象
```

对于KG方案：这部分利用LLM抽取文件中的三元组，将其存储为KG供后续检索，可参考此代码文件：langchain/libs/langchain/langchain/indexes /graph.py

```
"""Graph Index Creator."""                    # 定义"图索引创建器"的描述

# 导入相关的模块和类型定义
from typing import Optional, Type             # 导入可选类型和类型的基础类型
from langchain import BasePromptTemplate      # 导入基础提示模板
from langchain.chains.llm import LLMChain     # 导入LLM链
from langchain.graphs.networkx_graph import NetworkxEntityGraph, parse_triples  # 导入Networkx实体图和解析三元组的功能
from langchain.indexes.prompts.knowledge_triplet_extraction import (  # 从知识三元组提取模块导入对应的提示
    KNOWLEDGE_TRIPLE_EXTRACTION_PROMPT,
)
from langchain.pydantic_v1 import BaseModel                     # 导入基础模型
from langchain.schema.language_model import BaseLanguageModel    # 导入基础语言模型的定义

class GraphIndexCreator(BaseModel):  # 定义图索引创建器类，继承自BaseModel
    """Functionality to create graph index."""   # 描述该类的功能为"创建图索引"

    llm: Optional[BaseLanguageModel] = None      # 定义可选的语言模型属性，默认为None
    graph_type: Type[NetworkxEntityGraph] = NetworkxEntityGraph  # 定义图的类型，默认为NetworkxEntityGraph

    def from_text(
        self, text: str, prompt: BasePromptTemplate = KNOWLEDGE_TRIPLE_EXTRACTION_PROMPT
    ) -> NetworkxEntityGraph:                    # 定义一个方法，从文本中创建图索引
        """Create graph index from text."""      # 描述该方法的功能
        if self.llm is None:                     # 如果语言模型为None，则抛出异常
            raise ValueError("llm should not be None")
        graph = self.graph_type()  # 创建一个新的图
        chain = LLMChain(llm=self.llm, prompt=prompt)  # 使用当前的语言模型和提示创建一个LLM链
        output = chain.predict(text=text)        # 使用LLM链对文本进行预测
        knowledge = parse_triples(output)        # 解析预测输出得到的三元组
        for triple in knowledge:                 # 遍历所有的三元组
            graph.add_triple(triple)             # 将三元组添加到图中
        return graph  # 返回创建的图

    async def afrom_text(             # 定义一个异步版本的from_text方法
        self, text: str, prompt: BasePromptTemplate = KNOWLEDGE_TRIPLE_EXTRACTION_PROMPT
    ) -> NetworkxEntityGraph:
        """Create graph index from text asynchronously."""  # 描述该异步方法的功能
        if self.llm is None:          # 如果语言模型为None，则抛出异常
            raise ValueError("llm should not be None")
        graph = self.graph_type()     # 创建一个新的图
        chain = LLMChain(llm=self.llm, prompt=prompt)       # 使用当前的语言模型和提示创建一个LLM链
        output = await chain.apredict(text=text)   # 异步使用LLM链对文本进行预测
        knowledge = parse_triples(output)          # 解析预测输出得到的三元组
        for triple in knowledge:                   # 遍历所有的三元组
            graph.add_triple(triple)               # 将三元组添加到图中
        return graph                               # 返回创建的图
```

另外，为了索引，便不得不牵涉以下这些能力

- **Document Loaders**，文档加载的标准接口
  与各种格式的文档及数据源集成，比如Arxiv、Email、Excel、Markdown、PDF(所以可以做类似ChatPDF这样的应用)、Youtube …

  相近的还有
  docstore，其中包含wikipedia.py等
  document_transformers
- **embeddings**(*langchain/libs/langchain/langchain/embeddings*)，则涉及到各种embeddings算法，分别体现在各种代码文件中：
  elasticsearch.py、*google_palm.py*、*gpt4all.py*、*huggingface.py*、*huggingface_hub.py*
  llamacpp.py、*minimax.py*、*modelscope_hub.py*、*mosaicml.py*
  openai.py
  sentence_transformer.py、*spacy_embeddings.py*、*tensorflow_hub.py*、*vertexai.py*

1.1.2 能力层：Chains、Memory、Tools

如果基础层提供了最核心的能力，能力层则给这些能力安装上手、脚、脑，让其具有记忆和触发万物的能力，包括：Chains、Memory、Tool三部分

- Chains：**链接**
  简言之，相当于包括一系列对各种组件的调用，可能是一个 Prompt 模板，一个语言模型，一个输出解析器，一起工作处理用户的输入，生成响应，并处理输出

  具体而言，则相当于按照不同的需求抽象并定制化不同的执行逻辑，Chain可以相互嵌套并串行执行，通过这一层，让LLM的能力链接到各行各业
  比如与Elasticsearch数据库交互的：elasticsearch_database
  比如基于知识图谱问答的：graph_qa
  其中的代码文件：*chains/graph_qa/base.py* 便实现了一个基于知识图谱实现的问答系统，具体步骤为
  首先，根据提取到的实体在知识图谱中查找相关的信息「这是通过 *self.graph.get_entity_knowledge(entity)* 实现的，它返回的是与实体相关的所有信息，形式为三元组」
  然后，将所有的三元组组合起来，形成上下文
  最后，将问题和上下文一起输入到qa_chain，得到最后的答案

```
   entities = get_entities(entity_string)  # 获取实体列表。
        context = ""               # 初始化上下文。
        all_triplets = []          # 初始化三元组列表。
        for entity in entities:    # 遍历每个实体
            all_triplets.extend(self.graph.get_entity_knowledge(entity))  # 获取实体的所有知识并加入到三元组列表中。
        context = "\n".join(all_triplets)          # 用换行符连接所有的三元组作为上下文。

        # 打印完整的上下文。
        _run_manager.on_text("Full Context:", end="\n", verbose=self.verbose)
        _run_manager.on_text(context, color="green", end="\n", verbose=self.verbose)

        # 使用上下文和问题获取答案。
        result = self.qa_chain(
            {"question": question, "context": context},
            callbacks=_run_manager.get_child(),
        )
        return {self.output_key: result[self.qa_chain.output_key]}  # 返回答案
```

  比如能自动生成代码并执行的：llm_math等等
  比如面向私域数据的：qa_with_sources，其中的这份代码文件 *chains/qa_with_sources/vector_db.py* 则是使用向量数据库的问题回答，核心在于以下两个函数
  reduce_tokens_below_limit

```
# 定义基于向量数据库的问题回答类
class VectorDBQAWithSourcesChain(BaseQAWithSourcesChain):
    """Question-answering with sources over a vector database."""

    # 定义向量数据库的字段
    vectorstore: VectorStore = Field(exclude=True)

    """Vector Database to connect to."""
    # 定义返回结果的数量
    k: int = 4

    # 是否基于令牌限制来减少返回结果的数量
    reduce_k_below_max_tokens: bool = False

    # 定义返回的文档基于令牌的最大限制
    max_tokens_limit: int = 3375

    # 定义额外的搜索参数
    search_kwargs: Dict[str, Any] = Field(default_factory=dict)

    # 定义函数来根据最大令牌限制来减少文档
    def _reduce_tokens_below_limit(self, docs: List[Document]) -> List[Document]:
```

```
        num_docs = len(docs)
                                24
        # 检查是否需要根据令牌减少文档数量
        if self.reduce_k_below_max_tokens and isinstance(
            self.combine_documents_chain, StuffDocumentsChain
        ):
            tokens = [
                self.combine_documents_chain.llm_chain.llm.get_num_tokens(
                    doc.page_content
                )
                for doc in docs
            ]
            token_count = sum(tokens[:num_docs])

            # 减少文档数量直到满足令牌限制
            while token_count > self.max_tokens_limit:
                num_docs -= 1
                token_count -= tokens[num_docs]

        return docs[:num_docs]
```

_get_docs

```
    # 获取相关文档的函数
    def _get_docs(
        self, inputs: Dict[str, Any], *, run_manager: CallbackManagerForChainRun
    ) -> List[Document]:
        question = inputs[self.question_key]

        # 从向量存储中搜索相似的文档
        docs = self.vectorstore.similarity_search(
            question, k=self.k, **self.search_kwargs
        )
        return self._reduce_tokens_below_limit(docs)
```

比如面向SQL数据源的：sql_database，可以重点关注这份代码文件：*chains/sql_database/query.py*

比如面向模型对话的：chat_models，包括这些代码文件：__init__.py、anthropic.py、azure_openai.py、base.py、fake.py、google_palm.py、human.py、jinachat.py、openai.py、promptlayer_openai.py、vertexai.py

另外，还有比较让人眼前一亮的：
constitutional_ai：对最终结果进行偏见、合规问题处理的逻辑，保证最终的结果符合价值观
llm_checker：能让LLM自动检测自己的输出是否有没有问题的逻辑

- **Memory：记忆**
简言之，用来保存和模型交互时的上下文状态，处理长期记忆

具体而言，这层主要有两个核心点：
→ 对Chains的执行过程中的输入、输出进行记忆并结构化存储，为下一步的交互提供上下文，这部分简单存储在Redis即可
→ 根据交互历史构建知识图谱，根据关联信息给出准确结果，对应的代码文件为：memory/kg.py

```
# 定义知识图谱对话记忆类
class ConversationKGMemory(BaseChatMemory):
    """知识图谱对话记忆类

    在对话中与外部知识图谱集成，存储和检索对话中的知识三元组信息。
    """

    k: int = 2  # 考虑的上下文对话数量
    human_prefix: str = "Human"  # 人类前缀
    ai_prefix: str = "AI"  # AI前缀
    kg: NetworkxEntityGraph = Field(default_factory=NetworkxEntityGraph)  # 知识图谱实例
    knowledge_extraction_prompt: BasePromptTemplate = KNOWLEDGE_TRIPLE_EXTRACTION_PROMPT          # 知识提取提示
    entity_extraction_prompt: BasePromptTemplate = ENTITY_EXTRACTION_PROMPT  # 实体提取提示
    llm: BaseLanguageModel                  # 基础语言模型
    summary_message_cls: Type[BaseMessage] = SystemMessage  # 总结消息类
    memory_key: str = "history"             # 历史记忆键

    def load_memory_variables(self, inputs: Dict[str, Any]) -> Dict[str, Any]:
        """返回历史缓冲区。"""
        entities = self._get_current_entities(inputs)  # 获取当前实体

        summary_strings = []
        for entity in entities:  # 对于每个实体
            knowledge = self.kg.get_entity_knowledge(entity)      # 获取与实体相关的知识
```

```
            if knowledge:
                summary = f"On {entity}: {'. '.join(knowledge)}."  # 构建总结字符串
                summary_strings.append(summary)
        context: Union[str, List]
        if not summary_strings:
            context = [] if self.return_messages else ""
        elif self.return_messages:
            context = [
                self.summary_message_cls(content=text) for text in summary_strings
            ]
        else:
            context = "\n".join(summary_strings)

        return {self.memory_key: context}

    @property
    def memory_variables(self) -> List[str]:
        """始终返回记忆变量列表。"""
        return [self.memory_key]

    def _get_prompt_input_key(self, inputs: Dict[str, Any]) -> str:
        """获取提示的输入键。"""
        if self.input_key is None:
            return get_prompt_input_key(inputs, self.memory_variables)
        return self.input_key

    def _get_prompt_output_key(self, outputs: Dict[str, Any]) -> str:
        """获取提示的输出键。"""
        if self.output_key is None:
            if len(outputs) != 1:
                raise ValueError(f"One output key expected, got {outputs.keys()}")
            return list(outputs.keys())[0]
        return self.output_key

    def get_current_entities(self, input_string: str) -> List[str]:
        """从输入字符串中获取当前实体。"""
        chain = LLMChain(llm=self.llm, prompt=self.entity_extraction_prompt)
        buffer_string = get_buffer_string(
            self.chat_memory.messages[-self.k * 2 :],
            human_prefix=self.human_prefix,
            ai_prefix=self.ai_prefix,
        )
        output = chain.predict(
            history=buffer_string,
            input=input_string,
        )
        return get_entities(output)

    def _get_current_entities(self, inputs: Dict[str, Any]) -> List[str]:
        """获取对话中的当前实体。"""
        prompt_input_key = self._get_prompt_input_key(inputs)
        return self.get_current_entities(inputs[prompt_input_key])

    def get_knowledge_triplets(self, input_string: str) -> List[KnowledgeTriple]:
        """从输入字符串中获取知识三元组。"""
        chain = LLMChain(llm=self.llm, prompt=self.knowledge_extraction_prompt)
        buffer_string = get_buffer_string(
            self.chat_memory.messages[-self.k * 2 :],
            human_prefix=self.human_prefix,
            ai_prefix=self.ai_prefix,
        )
        output = chain.predict(
            history=buffer_string,
            input=input_string,
            verbose=True,
        )
        knowledge = parse_triples(output)  # 解析三元组
        return knowledge

    def _get_and_update_kg(self, inputs: Dict[str, Any]) -> None:
        """从对话历史中获取并更新知识图谱。"""
        prompt_input_key = self._get_prompt_input_key(inputs)
        knowledge = self.get_knowledge_triplets(inputs[prompt_input_key])
        for triple in knowledge:
            self.kg.add_triple(triple)  # 向知识图谱中添加三元组

    def save_context(self, inputs: Dict[str, Any], outputs: Dict[str, str]) -> None:
        """将此对话的上下文保存到缓冲区。"""
        super().save_context(inputs, outputs)
```

```
        self._get_and_update_kg(inputs)105
    def clear(self) -> None:
        """清除记忆内容。"""
        super().clear()
        self.kg.clear()  # 清除知识图谱内容
```

- **Tools层，工具**
  其实Chains层可以根据LLM + Prompt执行一些特定的逻辑，但是如果要用Chain实现所有的逻辑不现实，可以通过Tools层也可以实现，Tools层理解为技能比较合理，典型的比如搜索、Wikipedia、天气预报、ChatGPT服务等等

1.1.3 应用层：Agents

- Agents：**代理**
  简言之，有了基础层和能力层，我们可以构建各种各样好玩的，有价值的服务，这里就是Agent

  具体而言，Agent 作为代理人去向 LLM 发出请求，然后采取行动，且检查结果直到工作完成，包括LLM无法处理的任务的代理 (例如搜索或计算，类似ChatGPT plus的插件有调用bing和计算器的功能)
  比如，Agent 可以使用维基百科查找 Barack Obama 的出生日期，然后使用计算器计算他在 2023 年的年龄

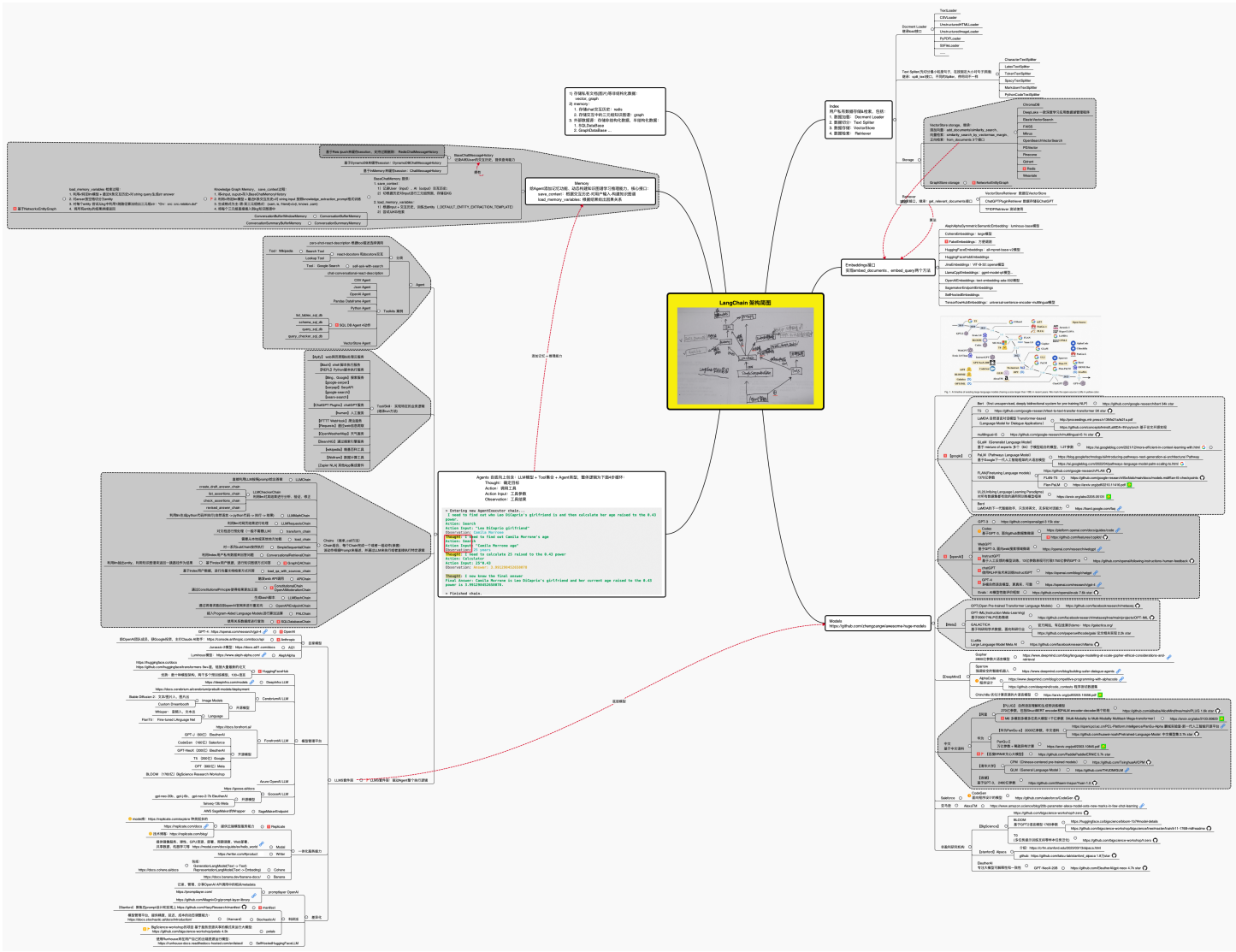
```
# pip install wikipedia
from langchain.agents import load_tools
from langchain.agents import initialize_agent
from langchain.agents import AgentType

tools = load_tools(["wikipedia", "llm-math"], llm=llm)
agent = initialize_agent(tools,
                         llm,
                         agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
                         verbose=True)


agent.run("奥巴马的生日是哪天？到2023年他多少岁了?")
```

  此外，关于Wikipedia可以关注下这个代码文件：*langchain/docstore/wikipedia.py* ...

最终langchain的整体技术架构可以如下图所示 (查看高清大图，此外，这里还有另一个架构图)



1.2 langchain的部分应用示例：联网搜索 + 文档问答

但看理论介绍，你可能没法理解langchain到底有什么用，为方便大家理解，特举几个langchain的应用示例

1.2.1 通过 Google 搜索并返回答案

由于需要借助 Serpapi 来进行实现，而Serpapi 提供了 Google 搜索的API 接口

故先到 Serpapi 官网(https://serpapi.com/)上注册一个用户，并复制他给我们生成 API key，然后设置到环境变量里面去

```
import os
os.environ["OPENAI_API_KEY"] = '你的api key'
os.environ["SERPAPI_API_KEY"] = '你的api key'
```

然后，开始编写代码

```
from langchain.agents import load_tools
from langchain.agents import initialize_agent
from langchain.llms import OpenAI
from langchain.agents import AgentType

# 加载 OpenAI 模型
llm = OpenAI(temperature=0,max_tokens=2048)

 # 加载 serpapi 工具
tools = load_tools(["serpapi"])

# 如果搜索完想再计算一下可以这么写
# tools = load_tools(['serpapi', 'llm-math'], llm=llm)

# 如果搜索完想再让他再用python的print做点简单的计算，可以这样写
# tools=load_tools(["serpapi","python_repl"])

# 工具加载后都需要初始化，verbose 参数为 True，会打印全部的执行详情
```