# 一、LangChain从入门到实战

## 1. 什么是LangChain

- LangChain 由 Harrison Chase 创建于2022年10月，它是围绕LLMs（大语言模型）建立的一个框架
- LangChain: 一个让你的LLM变得更强大的开源框架。LangChain 就是一个 LLM 编程框架，你想开发一个基于 LLM 应用，需要什么组件它都有，直接使用就行；甚至针对常规的应用流程，它利用链(LangChain中Chain的由来)这个概念已经内置标准化方案了。下面我们从新兴的大语言模型（LLM）技术栈的角度来看看为何它的理念这么受欢迎
- 对于缺乏算力,缺乏算法工程师,节省时间的我们来说,想要实现一个自己的 LLM 编程框架,LangChain是最理想的选择
- LangChain目前有两个语言的实现： python 和 Nodejs.
- LangChain主要组件
    - Models => 模型，各种类型的模型和模型集成，比如GPT-4
    - Prompts => 提示，包括提示管理、提示优化和提示序列化
    - Memory => 记忆，用来保存和模型交互时的上下文状态
    - Indexes => 索引，用来结构化文档，以便和模型交互
    - Chains => 链，一系列对各种组件的调用
    - Agents => 代理，决定模型采取哪些行动，执行并且观察流程，直到完成为止
- LangChain使用场景
    - 个人助手
    - 输入标题问答系统
    - 聊天机器人
    - 表格数据查询
    - 文档总结
    - API交互
    - 信息提取

    LangChain官网

## 2. 千帆大模型平台

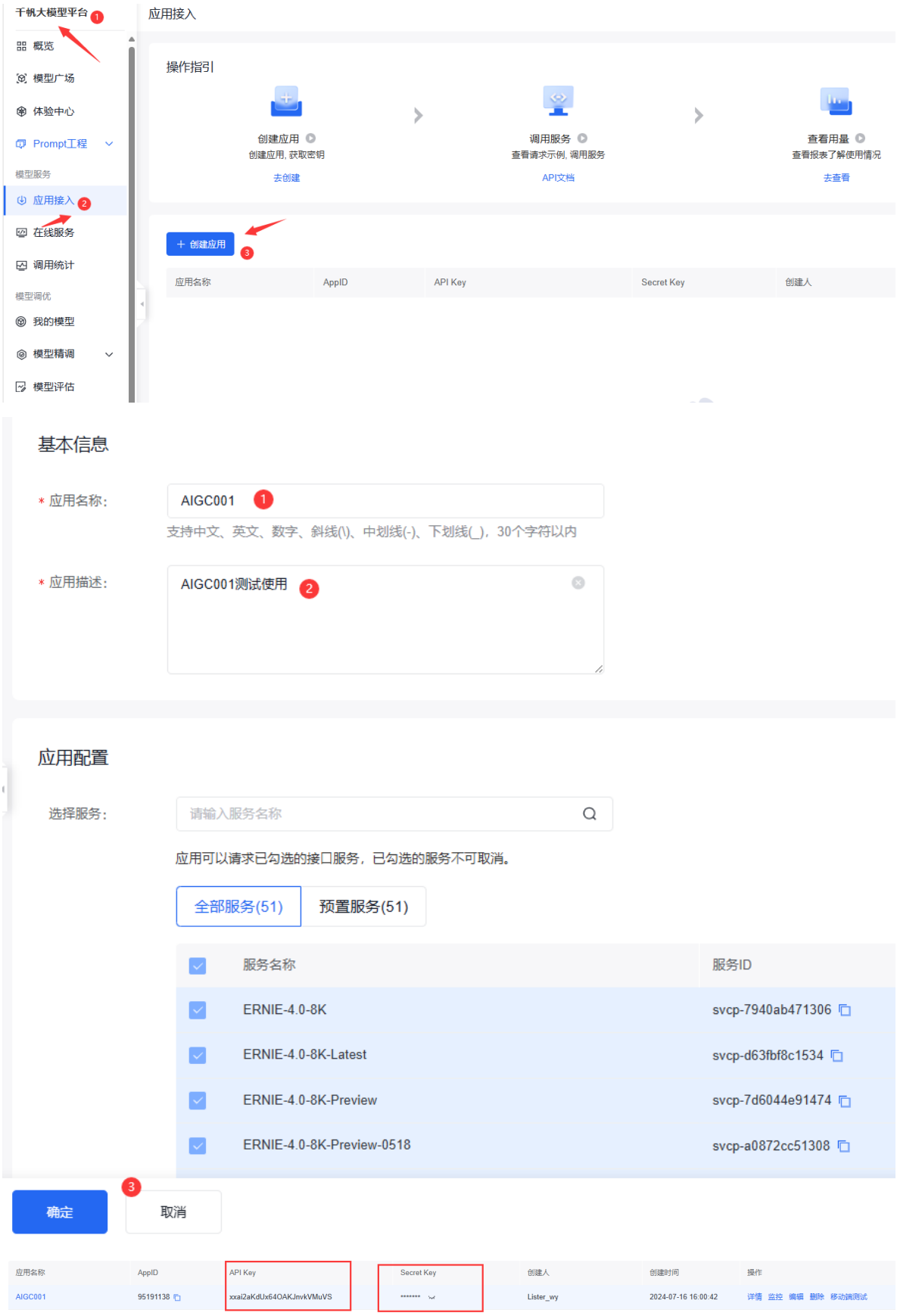
- 进入千帆大模型官网 千帆大模型官网
- 需要注册
- 实名认证

## 3. Models组件

- LangChain目前支持三种模型类型：LLMs、Chat Models(聊天模型)、Embeddings Models(嵌入模型)
    - LLMs => 大语言模型接收文本字符作为输入，返回的也是文本字符。
    - 聊天模型 => 基于LLMs, 不同的是它接收聊天消息(一种特定格式的数据)作为输入,返回的也是聊天消息。
    - 文本嵌入模型 => 文本嵌入模型接收文本作为输入, 返回的是浮点数列表。

      常用大模型的下载库

- 创建应用,得到两个KEY(**API Key,Secret Key**)




# 3.1 LLMs (大语言模型)

> 示例

```
import os
from langchain_community.llms import QianfanLLMEndpoint
os.environ['QIANFAN_AK'] = "换成你自己的API KEY"
os.environ['QIANFAN_SK'] = "换成你自己的SECRET KEY"
llm = QianfanLLMEndpoint(model="ERNIE-Bot-turbo")
res = llm("给我讲一下小马过河的故事")
print(res)
```



## 3.2 Chat Models(聊天模型)

> 示例

```
import  os
from langchain_community.chat_models import QianfanChatEndpoint
from langchain_core.messages import HumanMessage
os.environ['QIANFAN_AK'] = "换成你自己的API KEY"
os.environ['QIANFAN_SK'] = "换成你自己的SECRET KEY"
chat = QianfanChatEndpoint(model="ERNIE-Bot-turbo")
messages = [
    HumanMessage(content="给我讲一下小马过河的故事")
]
res = chat(messages)
print(res)
```



## 3.3 Embeddings Models(嵌入模型)

> 示例

```
import  os
from langchain_community.embeddings import QianfanEmbeddingsEndpoint
os.environ['QIANFAN_AK'] ="换成你自己的API KEY"
os.environ['QIANFAN_SK'] ="换成你自己的SECRET KEY"
embed = QianfanEmbeddingsEndpoint()
res1 = embed.embed_query('给我讲一下小马过河的故事')
print(res1)
# 打印结果：[0.039765920490026474, 0.022634351626038555, -0.018896507099270821,
....,  ]
# res2 = embed.embed_documents(['这是第一个测试文档', '这是第二个测试文档'])
# print(res2)
# 打印结果：[[0.03977284952998161, 0.022625437006354332, -0.018921626731753357,
....,  ]]
```



# 4. Prompts组件

- Prompt是指当用户输入信息给模型时加入的提示，这个提示的形式可以是zero-shot或者 few-shot等方式，目的是让模型理解更为复杂的业务场景以便更好的解决问题。
- 提示模板：如果你有了一个起作用的提示，你可能想把它作为一个模板用于解决其他问题，LangChain就提供了PromptTemplates组件，它可以帮助你更方便的构建提示。

## 4.1 zero-shot提示方式

> 示例

```
import  os
from langchain_core.prompts import PromptTemplate
from langchain_community.llms import QianfanLLMEndpoint
os.environ['QIANFAN_AK'] ="换成你自己的API KEY"
os.environ['QIANFAN_SK'] ="换成你自己的SECRET KEY"
# 定义模板
template = "给我出一道关于{subject}的题目"
prompt = PromptTemplate(input_variables=["subject"], template=template)
prompt_text = prompt.format(subject="语文")
llm = QianfanLLMEndpoint()
result = llm(prompt_text)
print(result)
```

## 4.2 few-shot提示方式

> 示例

```
import  os
from langchain_core.prompts import PromptTemplate
from langchain_core.prompts import FewShotPromptTemplate
from langchain_community.llms import QianfanLLMEndpoint

os.environ['QIANFAN_AK'] = "换成你自己的API KEY"
os.environ['QIANFAN_SK'] = "换成你自己的SECRET KEY"

examples = [ {"word": "大", "antonym": "小"},
             {"word": "上", "antonym": "下"},
             {"word": "左", "antonym": "右"},
           ]
example_template = """
单词: {word}
反义词: {antonym}\\n
 """

# 实例化PromptTemplate对象
example_prompt = PromptTemplate(input_variables=["word",
"antonym"],template=example_template,)
# 实例化FewShotPromptTemplates
few_shot_prompt = FewShotPromptTemplate(
    examples=examples, # 模型训练的案例
    example_prompt=example_prompt, # 样例的模板
    prefix="给出每个单词的反义词", # 提示的前缀
    suffix="单词: {input}\\n反义词:", # 提示的后缀
    input_variables=["input"], # 在few-shot当中定义的变量
    example_separator="\\n", # 样例之间都使用换行进行隔开
)
# 格式化文本
prompt_text = few_shot_prompt.format(input="粗")
llm = QianfanLLMEndpoint()
print(llm(prompt_text))
```

# 5. Chains组件

- 基础用法

> 示例

```
import os
from langchain_community.llms import QianfanLLMEndpoint
from langchain_core.prompts import PromptTemplate
from langchain.chains import LLMChain

os.environ['QIANFAN_AK'] = "换成你自己的API KEY"
os.environ['QIANFAN_SK'] = "换成你自己的SECRET KEY"

# 1. 定义模板
template = "给我出一道关于小学三年级{subject}的题目"
prompt = PromptTemplate(input_variables=["subject"], template=template)

# 2. 链条
llm = QianfanLLMEndpoint()
chain = LLMChain(llm=llm, prompt=prompt)

# 3. 执行Chain
result = chain.run("体育")
print(f'result-->{result}')
```

- 如果你想将第一个模型输出的结果，直接作为第二个模型的输入，还可以使用LangChain的SimpleSequentialChain, 代码如下：

> 示例

```
import os
from langchain_core.prompts import PromptTemplate
from langchain_community.llms import QianfanLLMEndpoint
from langchain.chains import LLMChain, SimpleSequentialChain

os.environ['QIANFAN_AK'] = "换成你自己的API KEY"
os.environ['QIANFAN_SK'] = "换成你自己的SECRET KEY"

llm = QianfanLLMEndpoint()
# 创建第一条链
template = "给我出一道关于小学一年级{subject}的题目"
subject_prompt = PromptTemplate(input_variables=["subject"], template=template,
)
first_chain = LLMChain(llm=llm, prompt=subject_prompt)



# 创建第二条链
template = "给我出一道关于小学二年级{subject6}的题目"
second_prompt = PromptTemplate(input_variables=["subject6"], template=template)
second_chain = LLMChain(llm=llm, prompt=second_prompt)
# 链接两条链，verbose=True可以显示推理过程
overall_chain = SimpleSequentialChain(chains=[first_chain, second_chain],
verbose=True)  # verbose=True 可以显示链条的推理过程
# 执行链，只需要传入第一个参数
catchphrase = overall_chain.run("数学")
print(catchphrase)
```



# 6. Agents组件

- 在 LangChain 中 Agents 的作用就是根据用户的需求，来访问一些第三方工具(比如：搜索引擎 或者 数据库)，进而来解决相关需求问题。
- 为什么要借助第三方库?
  - 因为大模型虽然非常强大，但是也具备一定的局限性，比如不能回答实时信息、处理数学逻辑问题仍然非常的初级等等。因此，可以借助第三方工具来辅助大模型的应用。
  - Agent代理
    - 制定计划和思考下一步需要采取的行动
    - 负责控制整段代码的逻辑和执行，代理暴露了一个接口，用来接收用户输入， 并返回AgentAction或AgentFinish。
  - Toolkit工具包
    - 一些集成好了代理包，比如 `create_csv_agent` 可以使用模型解读 csv文件。
  - Tool工具
    - 解决问题的工具
    - 第三方服务的集成，比如计算器、网络 搜索（谷歌、bing）等等
  - AgentExecutor代理执行器
    - 它将代理和工具列表包装在一起,负责迭 代运行代理的循环，直到满足停止的标准。

> 示例

```
# 现在我们实现一个使用代理的例子：假设我们想查询一下中国目前有多少人口？我们可以使用多个代理工具，让Agents选择执行。
# 需要安装依赖库 pip install wikipedia
# 加载内置工具 llm-math 和 wikipedia

import os
from langchain_community.llms import QianfanLLMEndpoint
from langchain.agents import load_tools, initialize_agent, AgentType
from langchain_core.prompts import PromptTemplate

os.environ['QIANFAN_AK'] = "换成你自己的API KEY"
os.environ['QIANFAN_SK'] = "换成你自己的SECRET KEY"

llm = QianfanLLMEndpoint(model="ChatGLM2-6B-32K")

tools = load_tools(["llm-math", "wikipedia"], llm=llm)

agent = initialize_agent(tools=tools,
                         llm=llm,
                         agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
                         verbose=True)
prompt_template = "那个国家面积最大?"
prompt = PromptTemplate.from_template(prompt_template)
result = agent.run(prompt)
print(result)
```

# 7. Memory组件

- 大模型本身不具备上下文的概念,它并不保存上次交互的内容，ChatGPT之所以能够和人正常沟通对话，因为它进行了一层封装，将历史记录回传给了模型。
- 因此 LangChain 也提供了Memory组件, Memory分为两种类型：短期记忆和长期记忆。
- 短期记忆一般指单一会话时传递数据，长期记忆则是处理多个会话时获取和更新信息
- 目前的Memory组件只需要考虑ChatMessageHistory。举例分析：

> 示例

```
from langchain.memory import ChatMessageHistory
history = ChatMessageHistory()
history.add_user_message("吃了吗?")
history.add_ai_message("吃了...")
print(history.messages)
```

- 和qianfan结合，直接使用 ConversationChain：

> 示例

```
import os
from langchain.chains import ConversationChain
from langchain_community.chat_models import QianfanChatEndpoint

os.environ['QIANFAN_AK'] = "换成你自己的API KEY"
os.environ['QIANFAN_SK'] = "换成你自己的SECRET KEY"
# 实例化大模型
llm = QianfanChatEndpoint()
# 只要我们使用Conversationchain进行记录的时候,那么相当于大模型就拥有记忆
conversation = ConversationChain(llm=llm)
resut1 = conversation.predict(input="语文82")
print(resut1)
resut2 = conversation.predict(input="数学88")
print(resut2)
resut3 = conversation.predict(input="语文和数学一共多少分?")
print(resut3)
```

- 如果要像chatGPT一样，长期保存历史消息，可以使用`messages_to_dict`方法

> 示例

```
from langchain.memory import ChatMessageHistory
from langchain.schema import messages_from_dict, messages_to_dict
history = ChatMessageHistory()
history.add_user_message("吃了吗?")
history.add_ai_message("吃了")
dicts = messages_to_dict(history.messages)
# 查看存储的信息
print(dicts)
# 读取历史消息
new_messages = messages_from_dict(dicts)
print(new_messages)
```

# 8. Indexes组件

- Indexes组件的目的是让LangChain具备处理文档处理的能力，包括：文档加载、检索等。注意，这里的文档不局限于txt、pdf等文本类内容，还涵盖email、区块链、视频等内容
  - 文档加载器
  - 文本分割器
  - VectorStores
  - 检索器

## 8.1 文档加载器

- 文档加载器可以基于TextLoader包
- 文档加载器使用起来很简单，只需要引入相应的loader工具：

> 示例

```
from langchain_community.document_loaders import TextLoader

loader1 = TextLoader("./data.txt", encoding='utf8')
doc1 = loader1.load()
print(f'docs1-->{doc1}')
print(len(doc1))
print(doc1[0].page_content[:10])
```



- langchain里面还支持的文档加载器

| 文档加载器 | 描述 |
| --- | --- |
| CSV | CSV问价 |
| JSON Files | 加载JSON文件 |
| Jupyter Notebook | 加载notebook文件 |
| Markdown | 加载markdown文件 |
| Microsoft PowerPoint | 加载ppt文件 |
| PDF | 加载pdf文件 |
| Images | 加载图片 |
| File Directory | 加载目录下所有文件 |
| HTML | 网页 |

## 8.2 文档分割器

- 由于模型对输入的字符长度有限制，我们在碰到很长的文本时，需要把文本分割成多个小的文本片段。
- 文本分割最简单的方式是按照字符长度进行分割，但是这会带来很多问题，比如说如果文本是一段代码，一个函数被分割到两段之后就成了没有意义的字符，所以整体的原则是把语义相关的文本片段尽可能的放在一起。
- LangChain中最基本的文本分割器是CharacterTextSplitter，它按照指定的分隔符（默认“\n\n”）进行分割，并且考虑 文本片段的最大长度。

> 示例

```
from langchain.text_splitter import CharacterTextSplitter
# 实例化一个文本分割对象
text_splitter = CharacterTextSplitter(
    separator = " ", # 空格分割，但是空格也属于字符
    chunk_size = 5, # 指明每个分割文本块的大小
    chunk_overlap  = 1,#每块之间重叠的字符,有重复的内容才可以更好的衔接上下文
)
# 一句分割
a = text_splitter.split_text("a b c d e f") # 将a,b,c,d,e,f按空格进行分割
# 多句话分割（文档分割）
texts = text_splitter.create_documents(["a b c d e f", "e f g h"], )
```

- 除了CharacterTextSplitter分割器，LangChain还支持其他文档分割器 (部分):

| 文档分割器 | 描述 |
| --- | --- |
| LatexTextSplitter | 沿着Latex标题、标题、枚举等分割文本 |
| MarkdownTextSplitter | 沿着Markdown的标题、代码块或水平规则来分割文本 |
| TokenTextSplitter | 根据openAI的token数进行分割 |
| PythonCodeTextSplitter | 沿着Python类和方法的定义分割文本 |

## 8.3 VectorStores

- VectorStores是一种特殊类型的数据库，它的作用是存储由嵌入创建的向量，提供相似查询等功能。 我们使用其中一个Chroma 组件作为例子（pip install chromadb）

> 示例

```
import os
from langchain_community.embeddings import QianfanEmbeddingsEndpoint
from langchain_community.vectorstores import Chroma
from langchain.text_splitter import CharacterTextSplitter

os.environ['QIANFAN_AK'] = "换成你自己的API KEY"
os.environ['QIANFAN_SK'] = "换成你自己的SECRET KEY"

# 1.读取文档里面的内容：data.txt
with open('./data.txt',encoding='utf-8') as f:
    str = f.read()
# print(str)

# 2.要切分文档
text_spliter = CharacterTextSplitter(chunk_size=100, chunk_overlap=5)
texts = text_spliter.split_text(str)
print(len(texts))
print('*'*80)

# 3.将切分后的文档向量化并保存
embedd = QianfanEmbeddingsEndpoint()
docsearch = Chroma.from_texts(texts, embedd)

query = "说一下布朗尼本场的数据？"
result = docsearch.similarity_search(query)
```

```
print(result)
print(len(result))
```

- LangChain支持的VectorStore如下

| VectorStore | 描述 |
| --- | --- |
| Chroma | 一个开源嵌入式数据库 |
| ElasticSearch | ElasticSearch |
| Milvus | 用于存储、索引和管理由深度神经网络和其他机器学习（ML）模型产生的大量嵌入向量的数据库 |
| Redis | 基于redis的检索器 |
| FAISS | Facebook AI相似性搜索服务 |
| Pinecone | 一个具有广泛功能的向量数据库 |

## 8.3 检索器

- 检索器是一种便于模型查询的存储数据的方式，LangChain约定检索器组件至少有一个方法`get_relevant_texts`，这个方法接收查询字符串，返回一组文档。（pip install faiss-cpu）

> 示例

```
import os
from langchain_community.embeddings import QianfanEmbeddingsEndpoint
from langchain_community.document_loaders import TextLoader
from langchain_community.vectorstores import FAISS
from langchain.text_splitter import CharacterTextSplitter

os.environ['QIANFAN_AK'] = "换成你自己的API KEY"
os.environ['QIANFAN_SK'] = "换成你自己的SECRET KEY"

# 1.加载文档
loader = TextLoader('./data.txt', encoding='utf8')
docments = loader.load()
# print(f'docments-->{docments}')
# print(type(docments))

# 2.切分文档
text_spliter = CharacterTextSplitter(chunk_size=100,
                                     chunk_overlap=5)
texts = text_spliter.split_documents(docments)
# print(texts)
# print(len(texts))
# 3. 实例化embedding模型
embed = QianfanEmbeddingsEndpoint() #这一行代码创建了一个名为embed的实例
db = FAISS.from_documents(texts, embed) #使用FAISS库创建了一个数据库
retriever = db.as_retriever(search_kwargs={"k": 1}) #FAISS数据库转换为一个检索器，
{"k": 1}表示每次检索返回最相关的1个文档
result = retriever.get_relevant_documents("说一下布朗尼本场的数据？") #使用检索器来获取
与查询
print(result)
```

- LangChain支持的检索器组件如下：

| 检索器 | 介绍 |
| --- | --- |
| ChatGPT Plugin Retriever | ChatGPT检索插件 |
| VectorStore Retriever | VectorStore检索器 |
| Vespa retriever | 一个支持结构化文本和向量搜索的平台 |
| Weaviate Hybrid Search | 一个开源的向量搜索引擎 |
| Wikipedia | 支持wikipedia内容检索 |