博学谷
www.boxuegu.com
黑马程序员线上品牌

LangChain基础知识入门

一样的教育，不一样的品质

目录

Contents

01

什么是LangChain

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自身并不开发LLMs，它的核心理念是为各种LLMs实现通用的接口，把LLMs相关的组件“链接”在一起，简化LLMs应用的开发难度,方便开发者快速地开发复杂的LLMs应用.

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

02

Models组件

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)

01

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。

02

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。

03

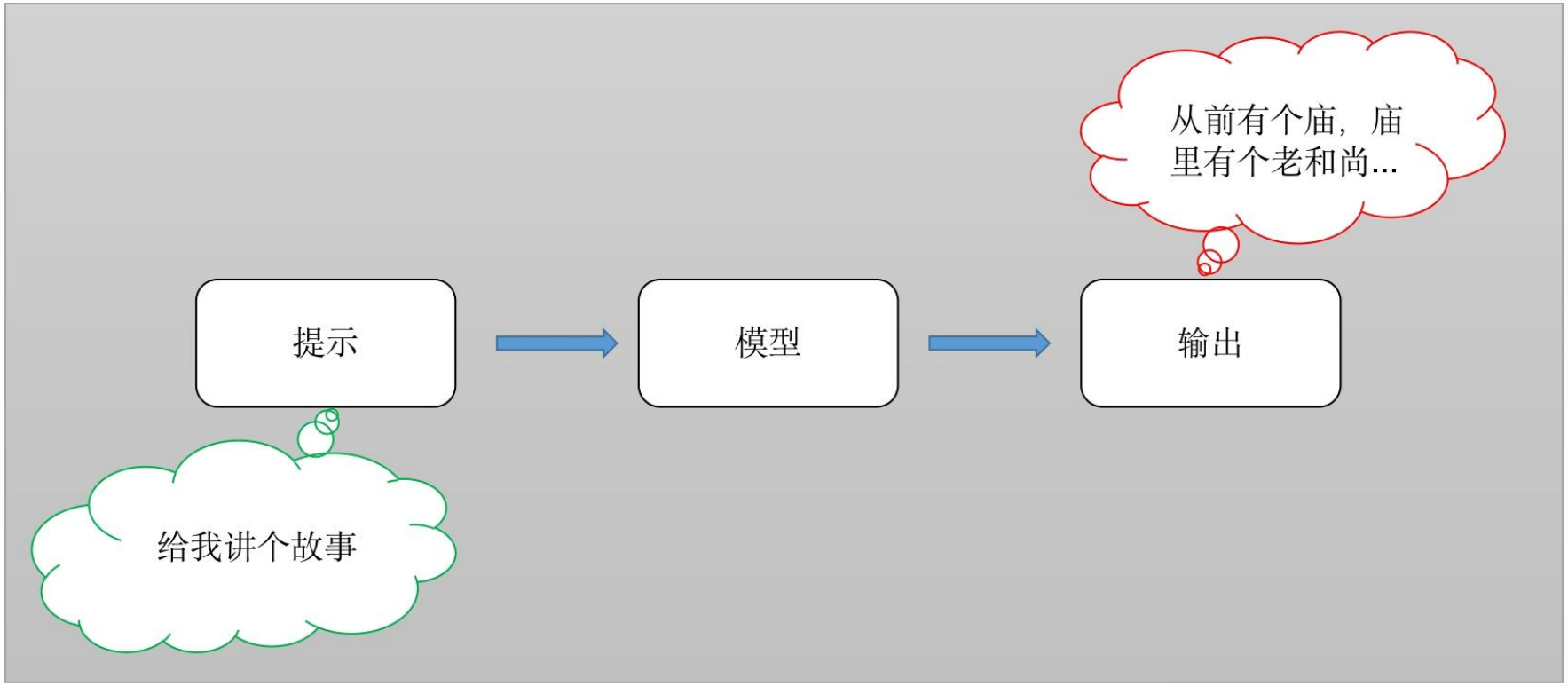
文本嵌入模型

文本嵌入模型接收文本作为输入，返回的是浮点数列表。

Models组件

◆ LLMs（大语言模型）

LLMs使用场景最多，常用大模型的下载库：https://huggingface.co/models：



Models组件

LLMs（大语言模型）

- 第一步：安装必备的工具包：langchain 和 ollama(属于第三方库)

```
pip install langchain
pip install ollama
```

- 第二步：ollama工具的安装与使用

```
详情请见附件手册
```

Models组件

◆ LLMs（大语言模型）

- 第三步：代码实现

```
from langchain_community.llms import Ollama
# model = Ollama(model="qwen2.5:1.5b")
model = Ollama(model="qwen2.5:7b")
result = model.invoke("请给我讲个鬼故事")
print(result)
```

好的，那我给你讲一个短小精悍的鬼故事吧。

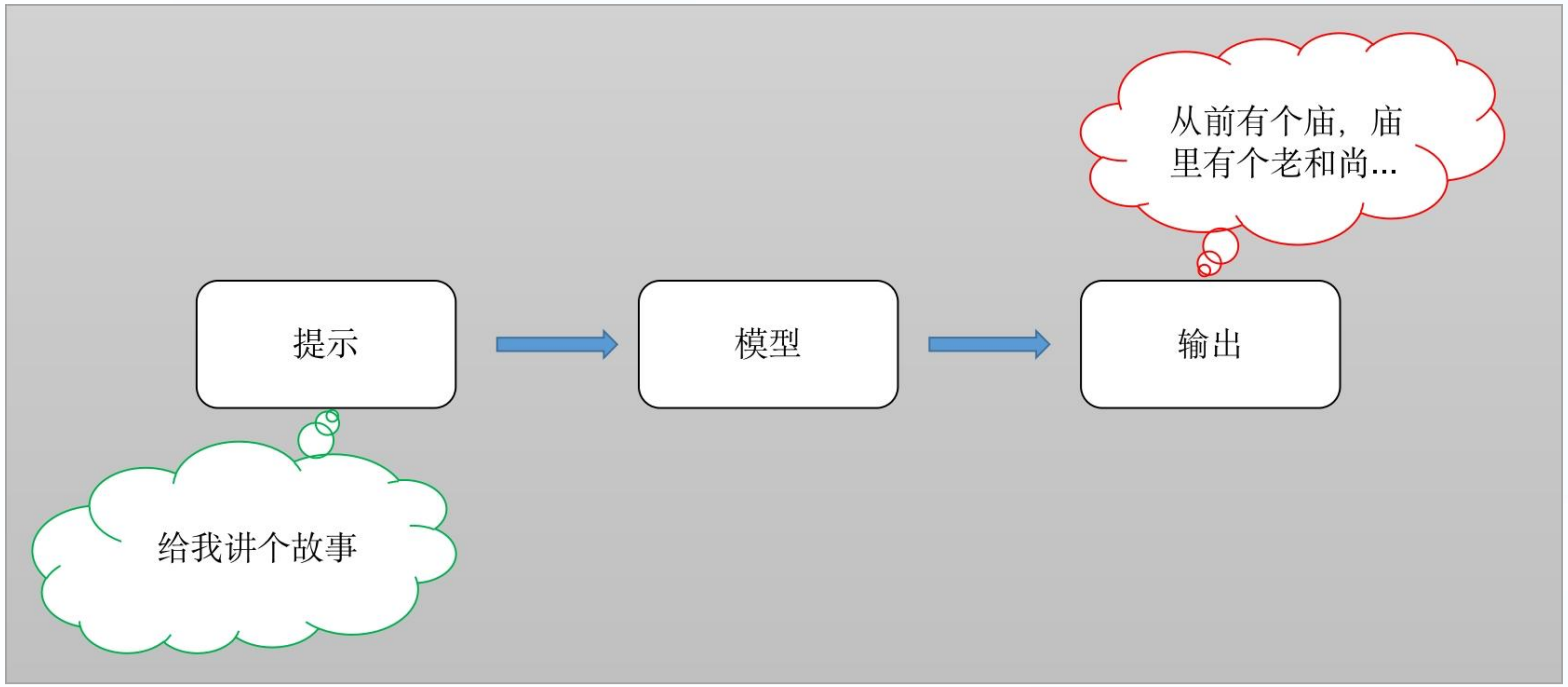
很久以前，在一个偏僻的小山村中，有一座荒废的老宅。这座老宅据说已经有几百年的历史了，但是因为主人在一次外出时突然失踪，因此逐渐被遗弃，成了人们口中的传说....

Models组件

◆ Chat Models（聊天模型）

Chat Models和LLMs效果在某些场景表现基本类似，但是使用时需要按照约定传入合适的值。

常用下载库：https://huggingface.co/models：



Models组件

◆ Chat Models（聊天模型）

Chat Models（聊天模型）

AIMessage

用来保存LLM的响应，以便在下次请求时把这些信息传回给LLM

HumanMessage

发送给LLMs的提示信息，比如“实现一个快速排序方法”

SystemMessage

设置LLM模型的行为方式和目标。你可以在这里给出具体的指示，比如“作为一个代码专家”，或者“返回json格式”

ChatMessage

ChatMessage可以接收任意形式的值，但是在大多数时间，我们应该使用上面的三种类型

Models组件

◆ Chat Models（聊天模型）

```
from langchain_core.messages import HumanMessage, SystemMessage,
AIMessage
from langchain_community.chat_models import ChatOllama
model = ChatOllama(model="qwen2.5:7b")
messages = [
        SystemMessage(content="现在你是一个著名的诗人"),
        HumanMessage(content="给我写一首唐诗")
]
res = model.invoke(messages)
# print(res)
print(res.content)
```

答案：

春风拂绿江南岸，烟柳画桥水映天。

燕舞蝶飞花满地，一曲琴音入梦甜。

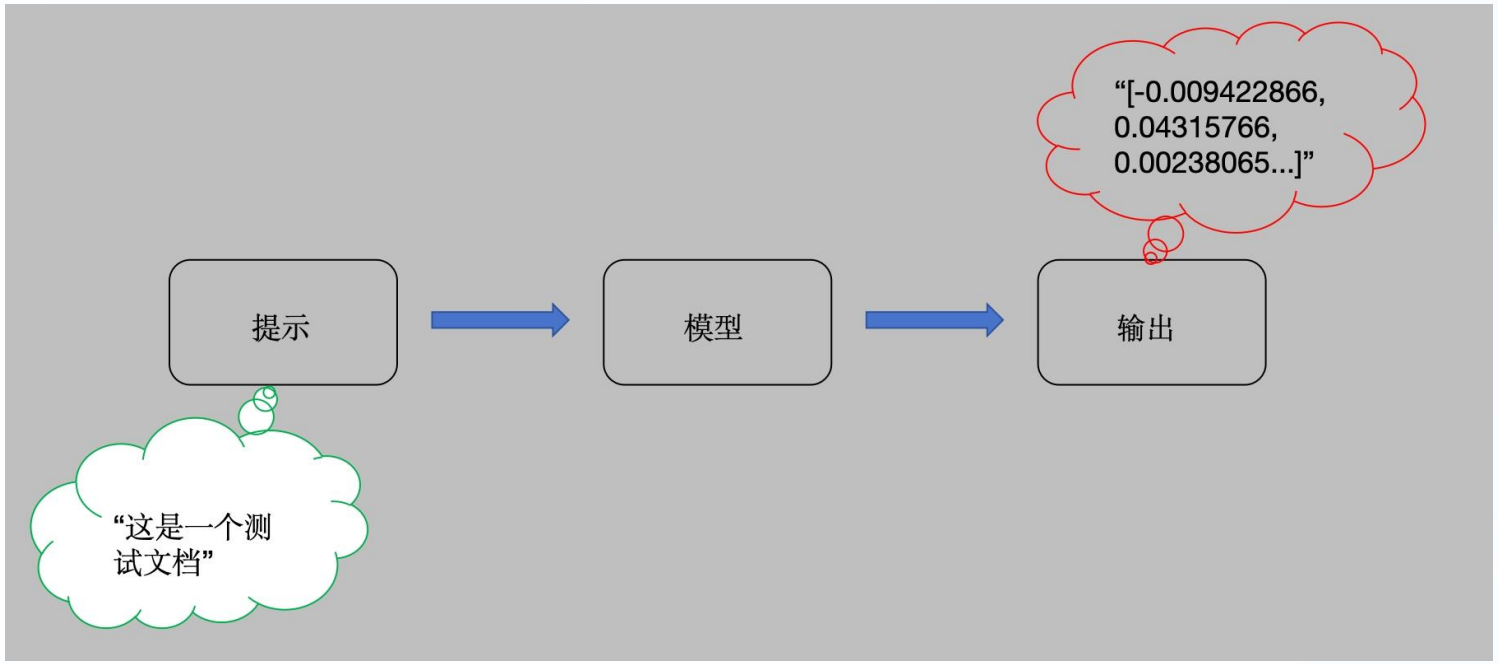
望月思君千里远，共赏明月初如练。

但愿人长久千里，天涯海角长相伴。

Models组件

◆ Embeddings Models(嵌入模型)

Embeddings Models特点：将字符串作为输入，返回一个浮动数的列表。在NLP中，Embedding的作用就是将数据进行文本向量化。



Models组件

◆ Embedding Models（向量模型）

```
from langchain_community.embeddings import OllamaEmbeddings

model = OllamaEmbeddings(model="mxbai-embed-large", temperature=0)
res1 = model.embed_query('这是第一个测试文档')

print(res1)
# [-0.07429124414920807, -0.11384586989879608, 0.5147323608398438, 0.6581658720970154,.....]

print(len(res1)) # 1024

res2 = model.embed_documents(['这是第一个测试文档', '这是第二个测试文档'])

print(res2)
# [[0.3369153141975403, -0.22442954778671265,...],[0.602321207523346, 0.6731221675872803,...]]
```

03

Prompts组件

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。

Prompts组件

◆ zero-shot提示方式：

```
from langchain import PromptTemplate
from langchain_community.llms import Ollama
model = Ollama(model="qwen2.5:7b")
# 定义模板
template = "我的邻居姓{lastname}，他生了个儿子，给他儿子起个名字"

prompt = PromptTemplate(
    input_variables=["lastname"],
    template=template,
)

prompt_text = prompt.format(lastname="王")
print(prompt_text)
# result: 我的邻居姓王，他生了个儿子，给他儿子起个名字


result = model(prompt_text)
print(result)
```

Prompts组件

◆ few-shot提示方式：

```
from langchain import PromptTemplate, FewShotPromptTemplate
from langchain_community.llms import Ollama
model = Ollama(model="qwen2.5:7b")

examples = [
    {"word": "开心", "antonym": "难过"},
    {"word": "高", "antonym": "矮"},
]

example_template = """
单词: {word}
反义词: {antonym}\\n
"""

example_prompt = PromptTemplate(
    input_variables=["word", "antonym"],
    template=example_template,
)
```

```
few_shot_prompt = FewShotPromptTemplate(
    examples=examples,
    example_prompt=example_prompt,
    prefix="给出每个单词的反义词",
    suffix="单词: {input}\\n反义词:",
    input_variables=["input"],
    example_separator="\\n",
)

prompt_text = few_shot_prompt.format(input="粗")

print(model(prompt_text))
# 细
```

04

Chains组件

Chains组件

在LangChain中，Chains描述了将LLM与其他组件结合起来完成一个应用程序的过程.

针对上一小节的提示模版例子，zero-shot里面，我们可以用链来连接提示模版组件和模型，进而可以实现代码的更改：

```
from langchain import PromptTemplate
from langchain_community.llms import Ollama
from langchain.chains import LLMChain

# 定义模板
template = "我的邻居姓{lastname}，他生了个儿子，给他儿子起个名字"

prompt = PromptTemplate(
    input_variables=["lastname"],
    template=template,
)
llm = Ollama(model="qwen2.5:7b")

chain = LLMChain(llm=llm, prompt=prompt)
# 执行链
print(chain.run("王"))
```

Chains组件

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：

```
from langchain import PromptTemplate
from langchain_community.llms import Ollama
from langchain.chains import LLMChain, SimpleSequentialChain

# 创建第一条链
template = "我的邻居姓{lastname}，他生了个儿子，给他儿子起个名字"

first_prompt = PromptTemplate(
    input_variables=["lastname"],
    template=template,
)
llm = Ollama(model="qwen2.5:7b")


first_chain = LLMChain(llm=llm, prompt=first_prompt)
```

```
# 创建第二条链
second_prompt = PromptTemplate(
    input_variables=["child_name"],
    template="邻居的儿子名字叫{child_name}，给他起一个小名",
)

second_chain = LLMChain(llm=llm, prompt=second_prompt)


# 链接两条链
overall_chain = SimpleSequentialChain(chains=[first_chain, second_chain], verbose=True)

print(overall_chain)
print('*'*80)
# 执行链，只需要传入第一个参数
catchphrase = overall_chain.run("王")
print(catchphrase)
```

05

Agents组件

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。

Agents组件

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。

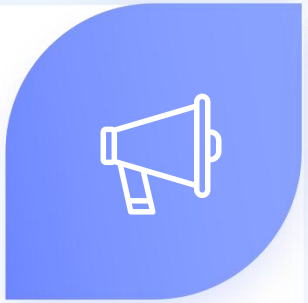
Toolkit工具包

- 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。

```
from langchain.agents import create_csv_agent
from langchain.llms import OpenAI
agent = create_csv_agent(OpenAI(temperature=0),
'data.csv', verbose=True)
agent.run("一共有多少行数据?")
```

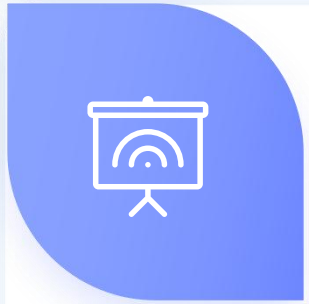
Tool工具

- 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。



Agents组件

现在我们实现一个使用代理的例子：我们可以使用多个代理工具，让Agents选择执行。代码如下：

```
from langchain.agents import load_tools
from langchain.agents import initialize_agent
from langchain.agents import AgentType
from langchain_community.llms import Ollama
from langchain_core.prompts import PromptTemplate

#  实例化大模型
llm = Ollama(model="qwen2.5:7b")

#  设置工具
# "serpapi"实时联网搜素工具、"math": 数学计算的工具
# tools = load_tools(["serpapi", "llm-math"], llm=llm)
tools = load_tools(["llm-math"], llm=llm)

# 实例化代理Agent:返回 AgentExecutor 类型的实例
agent = initialize_agent(tools, llm,
agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbose=True)
```

```
# 准备提示词
prompt_template = """解以下方程：3x + 4(x + 2) - 84
= y; 其中x为3，请问y是多少？"""
prompt =
PromptTemplate.from_template(prompt_template)
print('prompt-->', prompt)

# 代理Agent工作
result = agent.run(prompt)
print(result)
```

Agents组件

查询langchain支持的工具。代码如下:

```
from langchain.agents import get_all_tool_names
results = get_all_tool_names()
print(results)
```

结果展示:

```
['python_repl', 'requests', 'requests_get', 'requests_post', 'requests_patch', 'requests_put',
'requests_delete', 'terminal', 'sleep', 'wolfram-alpha', 'google-search', 'google-search-results-json', 'searx-
search-results-json', 'bing-search', 'metaphor-search', 'ddg-search', 'google-serper', 'google-scholar',
'google-serper-results-json', 'searchapi', 'searchapi-results-json', 'serpapi', 'dalle-image-generator',
'twilio', 'searx-search', 'wikipedia', 'arxiv', 'golden-query', 'pubmed', 'human', 'awslambda', 'sceneXplain',
'graphql', 'openweathermap-api', 'dataforseo-api-search', 'dataforseo-api-search-json',
'eleven_labs_text2speech', 'google_cloud_texttospeech', 'news-api', 'tmdb-api', 'podcast-api', 'memorize',
'llm-math', 'open-meteo-api']
```

Agents组件

LangChain支持的工具如下：

工具	描述
Bing Search	Bing搜索
Google Search	Google搜索
Google Serper API	一个从google搜索提取数据的API
Python REPL	执行python代码

06

Memory组件

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：

```
from langchain.memory import ChatMessageHistory

history = ChatMessageHistory()
history.add_user_message("在吗？")
history.add_ai_message("有什么事?")

print(history.messages)

# [HumanMessage(content='在吗？', additional_kwargs={}), AIMessage(content='有什么事?', additional_kwargs={})]
```

Memory组件

和qwen结合，直接使用 ConversationChain：

```
from langchain import ConversationChain
from langchain_community.llms import Ollama

#  实例化大模型
llm = Ollama(model="qwen2.5:7b")
conversation = ConversationChain(llm=llm)
resut1 = conversation.predict(input="小明有1只猫")
print(resut1)
print('*'*80)
resut2 = conversation.predict(input="小刚有2只狗")
print(resut2)
print('*'*80)
resut3 = conversation.predict(input="小明和小刚一共有几只宠物?")
print(resut3)
print('*'*80)
```

Memory组件

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

```
from langchain.memory import ChatMessageHistory
from langchain.schema import messages_from_dict, messages_to_dict

history = ChatMessageHistory()
history.add_user_message("hi!")
history.add_ai_message("whats up?")

dicts = messages_to_dict(history.messages)

print(dicts)
# [{'type': 'human', 'data': {'content': 'hi!', 'additional_kwargs': {}}},{'type': 'ai', 'data': {'content': 'whats up?',
'additional_kwargs': {}}}]

# 读取历史消息
new_messages = messages_from_dict(dicts)

print(new_messages)
#[HumanMessage(content='hi!', additional_kwargs={}),AIMessage(content='whats up?', additional_kwargs={})]
```

07

Indexes组件

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

Indexes组件

文档加载器

文档加载器主要基于`Unstructured` 包，`Unstructured` 是一个python包，可以把各种类型的文件转换成文本。

文档加载器使用起来很简单，只需要引入相应的loader工具：

```
from langchain_community.document_loaders import UnstructuredFileLoader
loader = UnstructuredFileLoader('衣服属性.txt', encoding='utf8')
docs = loader.load()
first_01 = docs[0].page_content[:4]
print(first_01)

from langchain_community.document_loaders import TextLoader
loader = TextLoader('衣服属性.txt', encoding='utf8')
docs = loader.load()
first_01 = docs[0].page_content[:4]
print(first_01)
```

Indexes组件

文档加载器

LangChain支持的文档加载器（部分）：

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

Indexes组件

◆ 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放在一起。

Indexes组件

◆ 文档分割器

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。

我们看个例子：

```
from langchain.text_splitter import CharacterTextSplitter


text_splitter = CharacterTextSplitter(
    separator = " ", # 空格分割，但是空格也属于字符
    chunk_size = 5,
    chunk_overlap  = 0,
)

# 一句分割
a = text_splitter.split_text("a b c d e f")

# 多句话分割（文档分割）
texts = text_splitter.create_documents(["a b c d e f", "e f g h"], )
```

Indexes组件

文档分割器

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

Indexes组件

◆ 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）：

```
from langchain_community.embeddings import OllamaEmbeddings
from langchain.text_splitter import CharacterTextSplitter
from langchain_community.vectorstores import Chroma

# pku.txt内容：<https://www.pku.edu.cn/about.html>
with open('./pku.txt') as f:
    state_of_the_union = f.read()
text_splitter = CharacterTextSplitter(chunk_size=100, chunk_overlap=0)
texts = text_splitter.split_text(state_of_the_union)
print(texts)
embeddings = OllamaEmbeddings(model="mxbai-embed-large")

docsearch = Chroma.from_texts(texts, embeddings)

query = "1937年北京大学发生了什么？"
docs = docsearch.similarity_search(query)
print(docs)
```

LangChain主要组件 — Indexes（索引）

◆ VectorStores

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

Indexes组件

◆ 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）

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

loader = TextLoader('./pku.txt')
documents = loader.load()
text_splitter = CharacterTextSplitter(chunk_size=100, chunk_overlap=0)
texts = text_splitter.split_documents(documents)

embeddings = OllamaEmbeddings(model="mxbai-embed-large")

db = FAISS.from_documents(texts, embeddings)
retriever = db.as_retriever(search_kwargs={'k': 1})
docs = retriever.get_relevant_documents("北京大学什么时候成立的")
print(docs)

#打印结果:
```

LangChain主要组件 — Indexes（索引）

◆ 检索器

LangChain支持的检索器组件如下：

检索器	介绍
Azure Cognitive Search Retriever	Amazon ACS检索服务
ChatGPT Plugin Retriever	ChatGPT检索插件
Databerry	Databerry检索
ElasticSearch BM25	ElasticSearch检索器
Metal	Metal检索器
Pinecone Hybrid Search	Pinecone检索服务
SVM Retriever	SVM检索器
TF-IDF Retriever	TF-IDF检索器
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

08

LangChain使用场景

LangChain使用场景

01 个人助手

02 聊天机器人

03 API交互

04 输入标题问答系统

05 Tabular数据查询

06 信息提取

07 文档总结

博学谷
www.boxuegu.com
黑马程序员线上品牌

扫码关注博学谷微信公众号