-

本地部署：Ollama

https://bailian.console.aliyun.com/cn-beijing/?tab=model#/api-key

阿里云百练平台 - apikey

- 代码调用云端大模型

创建 apikey

Pip install openai (国内慢可以用-i https://pypi.tuna.tsinghua.edu.cn/simple)

编写代码测试 - 找到 api 代码示例即可，openAI 兼容

- Ollama - 简化大型语言模型本地部署和运行过程的开源软件，自己提供算力

OpenAI 库

Python SDK

- 获取客户端对象（apikey，baseurl）
- 调用模型（client.chat.completions.create 创建 chatcompletion 对象）

client.chat.completions.create创建ChatCompletion对象

主要参数有2个：

- model：选择所用模型，如代码的qwen3-max
- messages：提供给模型的消息
  - 类型：list，可以包含多个字典消息
  - 每个字典消息包含2个key
    - role：角色
    - content：内容
- system角色：设定助手的整体行为、角色和规则，为对话提供上下文框架（如指定助手身份、回答风格、核心要求），是全局的背景设定，影响后续所有交互。
- assistant角色：代表 AI 助手的回答，可以在代码中认为设定
- user角色：代表用户，发送问题、指令或需求

```
from openai.types.chat.chat_completion import ChatCompletion

response: ChatCompletion = client.chat.completions.create(
    model="qwen3-max",
    messages=[
        {"role": "system", "content": "你是一个Python编程专家。"},
        {"role": "assistant", "content": "我是一个Python编程专家。请问有什么可以帮助您的吗？"},
        {"role": "user", "content": "for循环输出1到5的数字"}
    ]
)
```

- 处理结果（print（response.choices[0].message.content)

- stream 流式输出（体验好

1. Client.chat.completions.create()设定参数 stream=True
2. for 循环 response 对象，并在循环内输出内容

- 附带历史消息

调用模型传入参数 messages，要求是 list 对象
基于此填入历史消息，让模型知晓对话上下文，更好的回答

Tips:
当增加历史记录后，API 可能需要更长的时间处理上下文，或者返回了更多的初始化块，导致这些 None 变得非常明显。所以我们做一个修改：

```
for chunk in response:
content = chunk.choices[0].delta.content
if content is not None:
print(
content,
end=" ", # 每一段内容后添加一个空格，保持输出在同一行
flush=True # 确保每次输出都立即显示，而不是等待缓冲区满了才显示
)
print() # 输出完成后换行
```

这个历史消息是在 message 的 list 内，组织历史消息提供给模型，当前的历史消息是一次性的，如果是生产系统，可以将消息保存到文件、数据库等持久化工具内，需要的时候提取使用。Langchain 库，短期记忆和长期记忆。

- Prompting engineering （ in-context prompting）

1. 详细的描述
2. 当模型充当某个角色
3. 使用分隔符标明‘’’ ‘’’
4. 指定步骤
5. 提供例子 few-shot leaning
6. 使用参考文本（知识库用法）

Zero-shot 零样本学习，属性迁移。基于已训练的能力让模型直接生成结果
Few-shot 给出少量示例，让模型参考示例回答

- Json 数据格式

带有格式的字符串
Csv-结构化，抽取信息方便，但是数据不含 schema，有一定风险
json 是有 schema，空间占用大
“key”：value
json 对象-python 字典
json 数组-python 列表内含多个字典

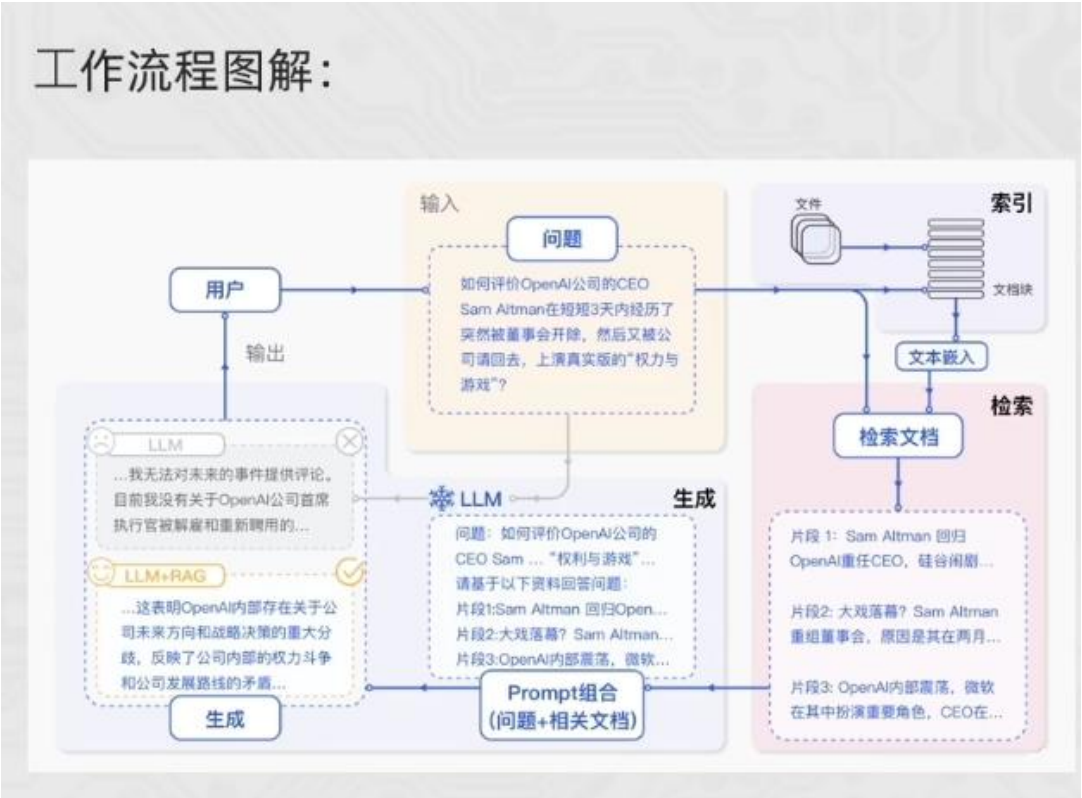
- LangChain

编程 SDK
Prompt，model，history，indexes，chains，agent

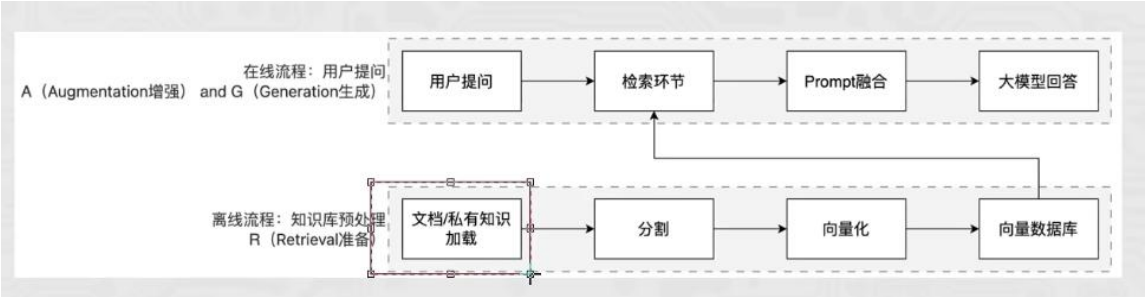
在 vscode 中的 venv 安装：
/Users/gena/Desktop/ai_agent_hm/**.venv**/bin/pip install langchain langchain-openai

- RAG

Retrieval- augmented generation



文本嵌入，基于自己的数据库，给模型之前自己先基于问题检索到了，然后再通过提示词优化，把自己检索到的信息和用户的提问一起打包提供给模型。



RAG 标准流程由索引（Indexing）、检索（Retriever）和生成（Generation）三个核心阶段组成。

- **索引阶段**，通过处理多种来源多种格式的文档提取其中文本，将其切分为标准长度的文本块（chunk），并进行嵌入向量化（embedding），向量存储在向量数据库（vector database）中。
  - 加载文件
  - 内容提取
  - 文本分割，形成chunk
  - 文本向量化
  - 存向量数据库
- **检索阶段**，用户输入的查询（query）被转化为向量表示，通过相似度匹配从向量数据库中检索出最相关的文本块。
  - query向量化
  - 在文本向量中匹配出与问句向量相似的top_k个
- **生成阶段**，检索到的相关文本与原始查询共同构成提示词（Prompt），输入大语言模型（LLM），生成精确且具备上下文关联的回答。
  - 匹配出的文本作为上下文和问题一起添加到prompt中
  - 提交给LLM生成答案：

RAG 的核心价值：

- 解决知识实效性问题（大模型的训练数据有截止时间，RAG 可以接入最新的文档，比如公司财报，政策文件），让模型输出“与时俱进”
- 降低幻觉：模型的回答是基于检索到的事实资料
- 无需重新训练模型：相比于 fine-tuning，RAG 只需更新知识库，成本更低，效率更高

- 向量

离线流程： 知识和信息--向量嵌入（向量化）--存入向量库
在线流程： 用户的提问--向量嵌入（向量化）--在向量库中匹配

向量（Vector）是文本的“数学身份证”。把文字的语义信息转换成一串固定长度的数字列表让计算机看懂文字的含义并做相似度计算

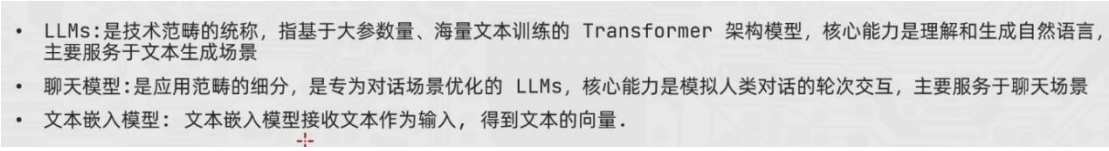
文本嵌入模型（text-embedding-v1 （1536 维的向量）），通过深度学习等技术，从文本提取语义特征并映射为固定长度的数字序列。
生成向量的维度是比较重要的指标。（维度越多，越精准，但是性能压力大）

- 余弦相似度算法

衡量两个向量方向相似程度的核心算法

- Langchain

LLMs
ChatModels
Embedding Models



1. LLMs

invoke 是一次性返回全部结果
stream 是逐段返回结果，流式输出

2. Chat Models

AIMessage - AI 输出信息，针对问题的回答
HumanMessage - 人类消息
SystemMessage - 用于指定模型具体所处的环境和背景

对于简写形式，可以在运行时填充具体值

3. Embedding Models

将字符串作为输入，返回一个浮点数的列表（向量）
Embed_query 单次
Embed_document 批量

4. PromptTemplate - 通用提示词模版，支持动态注入信息。

langchain 中提供了 promptTemplate 类，协助优化提示词
提示词模版是可以加入 chain 链的写法

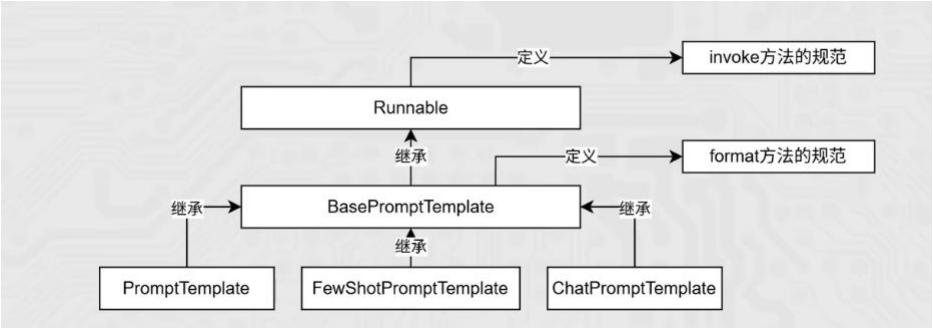
5. FewShotPromptTemplate - 支持基于模版注入任意数量的示例信息

```python
from langchain_core.prompts import FewShotPromptTemplate

FewShotPromptTemplate(
    examples=None,
    example_prompt=None,
    prefix=None,
    suffix=None,
    input_variables=None
)
```

参数:

- examples: 示例数据，list，内套字典
- example_prompt: 示例数据的提示词模板
- prefix: 组装提示词，示例数据前内容
- suffix: 组装提示词，示例数据后内容
- input_variables: 列表，注入的变量列表



Format 返回字符串，纯字符串替换，解析占位符生成提示词，支持解析{}占位符

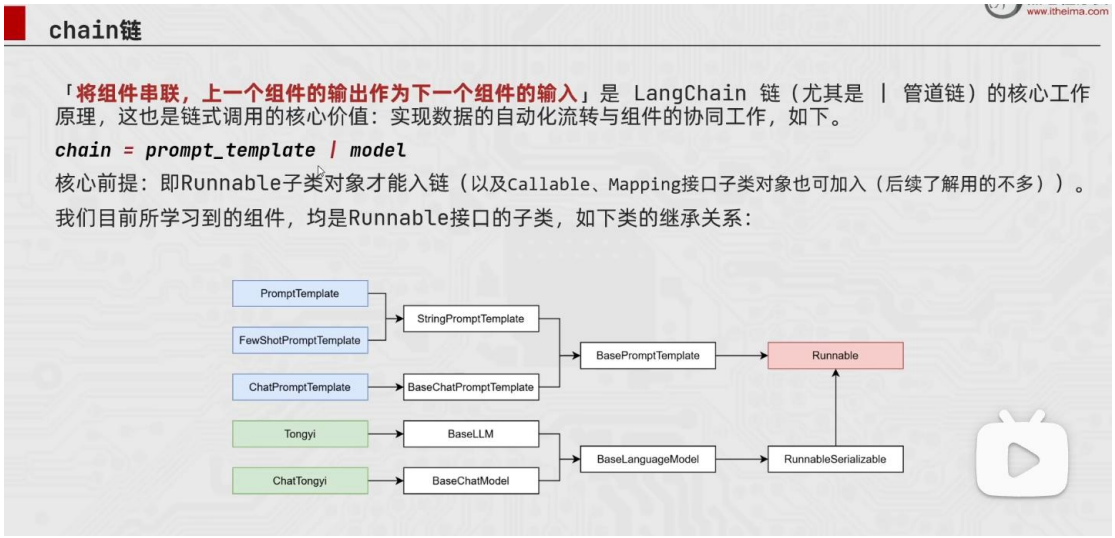
Invoke Runnable 接口标准方法，解析占位符生成提示词，promptValue 类对象，支持解析占位符和 MessagePlaceholder 结构化占位。传入字典。构建 chain

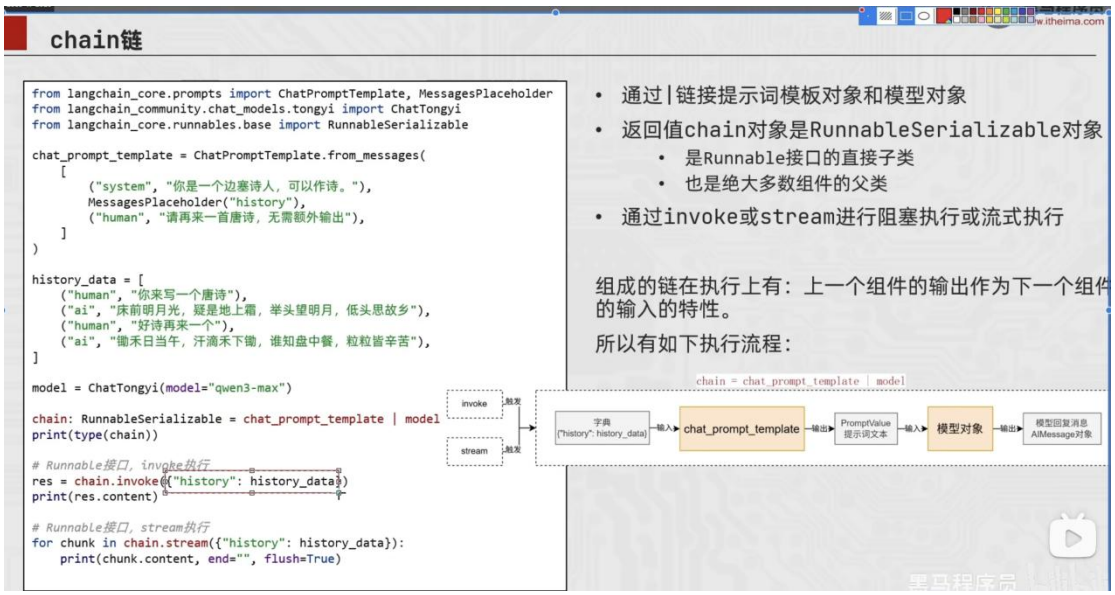
6. ChatPromptTemplate - 支持注入任意数量的历史会话信息

From_messages 可以接入一个 list 的消息

MessagePlaceholder (这是一个类) 作为占位，提供 history 作为占位的 key，基于 invoke 动态注入历史会话记录。

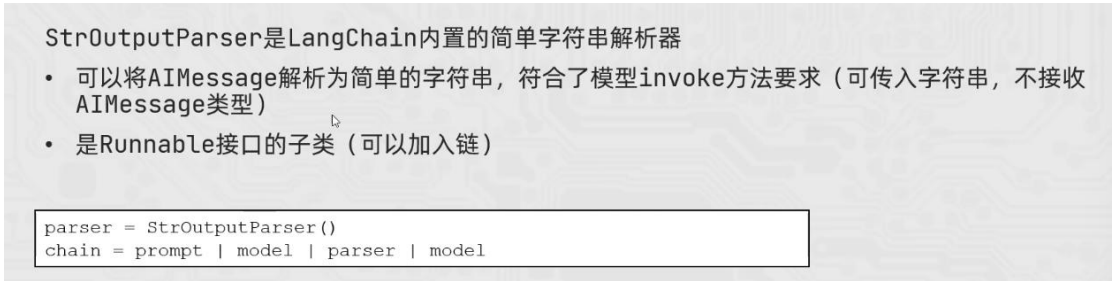
7. chain 链





Langchain 中绝大多数核心组件都继承了 Runnable 抽象基类
chain 变量是 RunnableSequence（RunnableSerializable 子类）类型

8. StrOutputParser 字符串输出解释器



9. JsonOutputParser

AIMessage 转换成 dict

10. RunnambeLambda（函数对象或 Lambda 匿名函数）

这样就不需要在 prompt 里面返回结果了，可以直接用 function

My_func = RunnableLambda(lambda ai_msg:{“name”:ai_msg.content})

如果跳过 RunnambeLambda 类，直接让函数加入链也是可以的，函数就是 Callable 接口的实例。

11. 临时 memory

完整代码：

```python
from langchain_community.chat_models.tongyi
import ChatTongyi
from langchain_core.chat_history import
InMemoryChatMessageHistory
from langchain_core.output_parsers import
StrOutputParser
from langchain_core.prompts import PromptTemplate
from langchain_core.runnables.history import
RunnableWithMessageHistory

def print_prompt(full_prompt):
    print("="*20, full_prompt.to_string(), "="*20)
    return full_prompt

model = ChatTongyi(model="qwen3-max")
prompt = PromptTemplate.from_template(
    "你需要根据对话历史回应用户问题。对话历史:
{chat_history}。用户当前输入：{input}， 请给出回应"
)

base_chain = prompt | print_prompt | model |
StrOutputParser()
```

```python
chat_history_store = {}     # 存放多个会话ID所对应的历史会话记录
def get_history(session_id):
    if session_id not in chat_history_store:
        # 存入新的实例
        chat_history_store[session_id] = InMemoryChatMessageHistory()
    return chat_history_store[session_id]


# 通过RunnableWithMessageHistory获取一个新的带有历史记录功能的chain
conversation_chain = RunnableWithMessageHistory(
    base_chain,    # 被附加历史消息的Runnable，通常是chain
    get_history,   # 获取历史会话的函数
    input_messages_key="input",          # 声明用户输入消息在模板中的占位符
    history_messages_key="chat_history" # 声明历史消息在模板中的占位符
)

if __name__ == '__main__':
    # 如下固定格式，配置当前会话的ID
    session_config = {"configurable": {"session_id": "user_001"}}

    print(conversation_chain.invoke({"input": "小明有一只猫"},
session_config))
    print(conversation_chain.invoke({"input": "小刚有两只狗"},
session_config))
    print(conversation_chain.invoke({"input": "共有几只宠物？"},
session_config))
```

12. 长期 memory

memory 长期会话记忆

FileChatMessageHistory类实现，核心思路：

- 基于文件存储会话记录，以session_id为文件名，不同session_id有不同文件存储消息

继承BaseChatMessageHistory实现如下3个方法：

- add_messages:同步模式，添加消息
- messages:同步模式，获取消息
- clear：同步模式，清除消息

如右侧代码，官方在BaseChatMessageHistory类的注释中提供了一个基于文件存储的示例代码。

```python
import json, os
from langchain_core.messages import messages_from_dict, message_to_dict

class FileChatMessageHistory(BaseChatMessageHistory):
    storage_path: str
    session_id: str
    @property
    def messages(self) -> list[BaseMessage]:
        try:
            with open(
                os.path.join(self.storage_path, self.session_id),
                "r",
                encoding="utf-8",
            ) as f:
                messages_data = json.load(f)
            return messages_from_dict(messages_data)
        except FileNotFoundError:
            return []

    def add_messages(self, messages: Sequence[BaseMessage]) -> None:
        all_messages = list(self.messages)  # Existing messages
        all_messages.extend(messages)  # Add new messages

        serialized = [message_to_dict(message) for message in all_messages]
        file_path = os.path.join(self.storage_path, self.session_id)
        os.makedirs(os.path.dirname(file_path), exist_ok=True)
        with open(file_path, "w", encoding="utf-8") as f:
            json.dump(serialized, f)

    def clear(self) -> None:
        file_path = os.path.join(self.storage_path, self.session_id)
        os.makedirs(os.path.dirname(file_path), exist_ok=True)
        with open(file_path, "w", encoding="utf-8") as f:
            json.dump([], f)
```

我们的核心思路啊其实是有两部分的

13. Document loaders 文档加载器

文档加载器提供了一套标准接口，用于将不同来源（如 CSV、PDF 或 JSON等）的数据读取为 LangChain 的文档格式。这确保了无论数据来源如何，都能对其进行一致性处理。

文档加载器（内置或自行实现）需实现BaseLoader接口。

Class Document，是LangChain内文档的统一载体，所有文档加载器最终返回此类的实例。

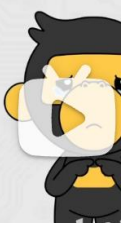
一个基础的Document类实例，基于如下代码创建：

```python
from langchain_core.documents import Document

document = Document(
    page_content="Hello, world!", metadata={"source": "https://example.com"}
)
```

可以看到，Document类其核心记录了：

- page_content：文档内容
- metadata：文档元数据（字典）



将这些来源的数据呢都统统读取为我们

Document 核心记录--
Page_content:文档内容
Metadata：文档元数据（字典）
不同的文档加载器定义不同的参数，但是都实现了统一的接口（方法）
Load（） 一次性加载全部文档 documents = loader.load()
Lazy_load() 延迟流式传输文档，对大型数据集有用，避免内存溢出。

```
For document in loader.lazy_load():
        Print(document)
```

Langchain_community.document_loaders import
CSVLoader
JSONLoader
PDFLoader

自定义CSV文件的解析和加载

```
loader = CSVLoader(
    file_path="./xxx.csv",
    csv_args={
        "delimiter": ",",    # 指定分隔符
        "quotechar": '"'     # 指定字符串的引号包裹
        # 字段列表（无表头使用，有表头勿用会读取首行做为数据）
        "fieldnames": ["name", "age", "gender"],
    },
)

data = loader.load()

print(data)
```

JSONLoader 需要额外安装 pip install jq

jq是一个跨平台的json解析工具，LangChain底层对JSON的解析就是基于jq工具实现的。
将JSON数据的信息抽取出来，封装为Document对象，抽取的时候依赖jq_schema语法。

```
{
    "name": "周杰轮",
    "age": 11,
    "hobby": ["唱", "跳", "RAP"],
    "other": {
        "addr": "深圳",
        "tel": "12332112321"
    }
}
```

- .表示整个JSON对象（根）
- []表示数组
- .name表示抽取周杰轮
- .hobby表示抽取爱好数组
- .hobby[1]或.hobby.[1]表示抽取跳
- .other.addr表示抽取地址深圳

```
[
    {"name": "周杰轮", "age": 11, "gender": "男"},
    {"name": "蔡依临", "age": 12, "gender": "女"},
    {"name": "王力鸿", "age": 11, "gender": "男"}
]
```

- .[].得到3个字典
- .[].name 表示抽取全部的name，即得到3个name信息

了解jq的基本抽取规则后，即可使用JSONLoader加载JSON文件了。

```
from langchain_community.document_loaders import JSONLoader

loader = JSONLoader(
    file_path="xxx.json",    # 文件路径
    jq_schema="",            # jq schema语法
    text_content=False,      # 抽取的是否是字符串，默认True
    json_lines=True,         # 是否是JsonLines文件（每一行都是JSON的文件）
)
```

如下是一个典型的JsonLines文件

```
{"name": "周杰轮", "age": 11, "gender": "男"}
{"name": "蔡依临", "age": 12, "gender": "女"}
{"name": "王力鸿", "age": 11, "gender": "男"}
```

Text_content = False 告知 jsonloader 我抽取的不是字符串是字典。
json_lines = True, Json lines 每一行都是独立的 json 文件

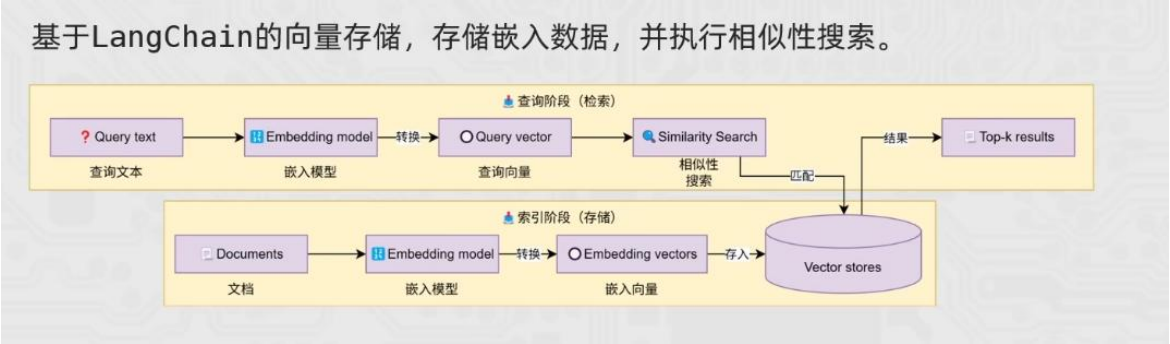
PyPDFLoader -- pip install pypdf
这些 pdf 可以后面用来转向量，转知识库

```
from langchain_community.document_loaders import PyPDFLoader

loader = PyPDFLoader(
    file_path="",    # 文件路径必填
    mode='page',     # 读取模式，可选page（按页面划分不同Document）和single（单个Document）
    password='password', # 文件密码
)
```

i = 0
For doc in loader.lazy_load() :
i += 1
Print(doc)
14. 文档加载器
RecursiveCharacterTextSplitter - 递归字符文本分割器

15. Vector stores 向量存储



- 文本转向量
- 创建向量存储，基于向量存储完成--

存入向量，删除向量，向量检索

Add_documents

Delete

Similarity search

内置向量存储的使用

```python
from langchain_core.vectorstores import InMemoryVectorStore
from langchain_community.embeddings import DashScopeEmbeddings

vector_store = InMemoryVectorStore(embedding=DashScopeEmbeddings())
# 添加文档到向量存储，并指定id
vector_store.add_documents(documents=[doc1, doc2], ids=["id1", "id2"])
# 删除文档（通过指定的id删除）
vector_store.delete(ids=["id1"])
# 相似性搜索
similar_docs = vector_store.similarity_search("your query here", 4)
```

外部（Chroma）向量存储的使用

```python
from langchain_community.embeddings import DashScopeEmbeddings
from langchain_chroma import Chroma

vector_store = Chroma(
    collection_name="example_collection",
    embedding_function=DashScopeEmbeddings(),
    persist_directory="./chroma_langchain_db",  # Where to save data locally, remove if not necessary
)
```

Chroma 数据库，可以完成持久化的向量存储

16. 向量检索构建提示词

用户提问+向量库中检索到的参考资料

Add_texts(list[str])

先通过向量存储检索匹配信息

将用户提问和匹配信息一同封装到提示词模版中提问模型

17. RunnablePassthrough 的使用

把向量检索也加入到链中