其他 Embedding 嵌入模型的配置与使用

其他Embedding嵌入模型的配置与使用

01. Hugging Face 文本嵌入模型

1.1 Hugging Face 本地模型

在某些对数据保密要求极高的场合下，数据不允许传递到外网，这个时候就可以考虑使用本地的文本嵌入模型——Hugging Face 本地嵌入模型，配置的技巧也非常简单，安装 `langchain-huggingface` 与 `sentence-transformers` 包，命令如下：

```
pip install -U langchain-huggingface sentence-transformers
```

其中 `langchain-huggingface` 是 langchain 团队基于 huggingface 封装的第三方社区包，`sentence-transformers` 是一个用于生成和使用预训练的文本嵌入，基于 `transformer` 架构，也是目前使用量最大的本地文本嵌入模型。

配置好后，就可以像正常的文本嵌入模型一样使用了，示例：

```
from langchain_huggingface import HuggingFaceEmbeddings

embeddings = HuggingFaceEmbeddings()

query_vector = embeddings.embed_query("你好，我是慕小课，我喜欢打篮球")

print(query_vector)
print(len(query_vector))
```

`sentence-transformers` 除此加载的时候，会将模型从本地应用加载到内容中，首次加载相对会慢，后续调用速度即可恢复正常。

除此之外，使用 Hugging Face 文本嵌入还可以加载任意本地的文本嵌入模型，传递 `model_name` 与 `cache_folder` 参数即可，示例：

```
from langchain_huggingface import HuggingFaceEmbeddings

embeddings = HuggingFaceEmbeddings(
    model_name="neuml/pubmedbert-base-embeddings",
    cache_folder="./embeddings/"
)

query_vector = embeddings.embed_query("你好，我是慕小课，我喜欢打篮球")

print(query_vector)
print(len(query_vector))
```

1.2 HuggingFace远程嵌入模型

部分模型的文件比较大，如果只是短期内调试，可以考虑使用 HuggingFace 提供的远程嵌入模型，首先安装对应的依赖：

```
pip install huggingface_hub
```

然后在 Hugging Face 官网（https://huggingface.co/）的 `setting` 中添加对应的访问秘钥，并配置到 `.env` 文件中：

```
HUGGINGFACEHUB_API_TOKEN=xxx
```

接下来就可以使用 Hugging Face 提供的推理服务，这样在本地服务器上就无需配置对应的文本嵌入模型了。

```
import dotenv
from langchain_huggingface import HuggingFaceEndpointEmbeddings

dotenv.load_dotenv()

embeddings = HuggingFaceEndpointEmbeddings(model="sentence-transformers/all-MiniLM-L12-v2")

query_vector = embeddings.embed_query("你好，我是慕小课，我喜欢打篮球")

print(query_vector)
print(len(query_vector))
```

相关资料信息：

1. Hugging Face 官网：https://huggingface.co/
2. HuggingFace 嵌入文档：https://python.langchain.com/v0.2/docs/integrations/text_embedding/sentence_transformers/
3. HuggingFace 嵌入翻译文档：http://imooc-langchain.shortvar.com/docs/integrations/text_embedding/sentence_transformers/

02. 百度千帆文本嵌入模型

如果想对接国内的文本嵌入模型提供商，可以考虑百度千帆，是目前国内生态最好，支持的模型最多（Embedding-V1、bge-large-zh、bge-large-en、tao-8k），速度最快的 AI 应用开发平台。

由于目前百度千帆并没有单独封装到独立的包，可以直接从 `langchain_community` 中导入，使用示例如下：

```
import dotenv
from langchain_community.embeddings.baidu_qianfan_endpoint import QianfanEmbeddingsEndpoint

dotenv.load_dotenv()

embeddings = QianfanEmbeddingsEndpoint()

query_vector = embeddings.embed_query("我叫慕小课，我喜欢打篮球游泳")

print(query_vector)
print(len(query_vector))
```

相关资料信息：

1. 百度千帆预设模型列表：https://cloud.baidu.com/doc/WENXINWORKSHOP/s/alj562vvu

2. 百度千帆嵌入文档：https://python.langchain.com/v0.2/docs/integrations/text_embedding/baidu_qianfan_endpoint/
3. 百度千帆嵌入翻译文档：http://imooc-langchain.shortvar.com/docs/integrations/text_embedding/baidu_qianfan_endpoint/

Faiss 向量数据库的配置与使用

Faiss 向量数据库的配置与使用

目前关于向量数据库的技术进展非常迅速，所以不同服务商提供的向量数据库使用差异非常大，数据的存储结构、支持的相似性检索方式、集合、条件筛选等功能差异也比较大，在 LangChain 中对于向量数据库基类只做了通用性的封装，减轻了部分迁移成本，在使用向量数据库时一定要综合衡量下各个向量数据库的差异。

按照部署方式和提供的服务类型进行划分，向量数据库可以划分成几种：

1. **本地文件向量数据库**：用户将向量数据存储到本地文件系统中，通过数据库查询的接口来检索向量数据，例如：**Faiss。**
2. **本地部署 API 向量数据库**：这类数据库不仅允许本地部署，而且提供了方便的 API 接口，使用户可以通过网络请求来访问和查询向量数据，这类数据库通常提供了更复杂的功能和管理选项，例如：**Milvus、Annoy、Weaviate** 等。
3. **云端 API 向量数据库**：将向量数据存储在云端，通过 API 提供向量数据的访问和管理功能，例如：**TCVectorDB、Pinecone** 等。

要想快速上手向量数据库的使用，只需要把向量数据库看成是 `Excel电子表格` 使用即可，按照使用办公软件的流程：**安装、写入数据、查找数据、删除数据、更新数据、保存数据**等相同的流程去学习+使用向量数据库即可。

而且向量数据库没有 SQL 数据库这么多复杂的查询功能，也没有事务，学习起来其实比绝大部分传统数据库都要容易上手。

01. Faiss 向量数据库简介

Faiss 是 Facebook 团队开源的向量检索工具，针对高维空间的海量数据，提供高效可靠的相似性检索方式，被广泛用于推荐系统、图片和视频搜索等业务。Faiss 支持 Linux、macOS 和 Windows 操作系统，在百万级向量的相似性检索表现中，Faiss 能实现 < 10ms 的响应（需牺牲搜索准确度）。

Faiss 官网：https://faiss.ai/，Faiss 仓库：https://github.com/facebookresearch/faiss

Faiss 使用 C++ 开发，提供了 Python 接口，可以通过 pip 安装 Faiss 库：

```
# CPU环境下使用
pip install faiss-cpu

# GPU环境下使用并且已经安装了CUDA，则可以使用GPU版本
pip install faiss-gpu
```

目前绝大部分向量数据库都支持多种对数据的操作方法：**新增数据、检索数据、带得分的数据检索、带筛选条件的数据检索、删除数据等**，但是几乎都不支持 `修改数据`，这是因为向量数据库通常使用特定的索引结构（如向量索引树或近似最临搜索算法），这些结构结构需要再数据插入后进行构建和优化，如果允许修改数据，索引结构可能需要频繁更新，这会显著增加系统的复杂性和开销，所以一般是删除后再新增。

Faiss 也类似，不过由于 Faiss 是本地文件向量数据库，还额外支持了将**向量数据持久化到本地、从本地文件夹加载向量数据库**等操作。

02. Faiss 向量数据库使用技巧

2.1 数据的导入与相似性搜索

在 LangChain 中，提供了 `from_texts` 和 `from_documents` 两个通用方法，这两个方法可以快捷从 `文本` 和 `文档` 中导入数据到向量数据库中，由于向量数据库存储的向量，所以需要传入文本嵌入模型，让向量数据库自动将传入的文本转换成向量。

示例如下：

```
import dotenv
from langchain_community.vectorstores import FAISS
from langchain_openai import OpenAIEmbeddings

dotenv.load_dotenv()

embedding = OpenAIEmbeddings(model="text-embedding-3-small")

db = FAISS.from_texts([
    "笨笨是一只很喜欢睡觉的猫咪",
    "我喜欢在夜晚听音乐，这让我感到放松。",
    "猫咪在窗台上打盹，看起来非常可爱。",
    "学习新技能是每个人都应该追求的目标。",
    "我最喜欢的食物是意大利面，尤其是番茄酱的那种。",
    "昨晚我做了一个奇怪的梦，梦见自己在太空飞行。",
    "我的手机突然关机了，让我有些焦虑。",
    "阅读是我每天都会做的事情，我觉得很充实。",
    "他们一起计划了一次周末的野餐，希望天气能好。",
    "我的狗喜欢追逐球，看起来非常开心。",
], embedding)

print(db.index.ntotal)
```

完成数据的填充后，即可在向量数据库中进行对应的检索，LangChain 为所有的向量数据库都设计封装了一致的搜索接口，最常用的有以下 4 种：

1. `similarity_search()`：基础相似度搜索，传递 `query(搜索语句)`、`k(返回条数)`、`filter(过滤器)`、`fetch_k(富余条数)` 等。
2. `similarity_search_with_score()`：携带得分的相似性搜索，参数和 `similarity_search()` 函数保持一致，只是会返回得分，这里的得分并不是相似性得分，而是欧几里得距离。
3. `similarity_search_with_relevance_scores()`：携带相关性得分的相似性搜索，得分范围是 0-1
4. `as_retriever()`：将向量数据库转换成检索器，检索器是 Runnable 可运行组件。

在向量数据库中进行检索并携带得分（这里的得分并不是相似性得分，默认是欧几里得距离），效果如下：

```
print(db.similarity_search_with_score("我养了一只猫，叫笨笨"))
```

输出内容：

```
10
[(Document(page_content='我养了一只猫，叫大笨'), 0.11375141),
(Document(page_content='猫咪在窗台上打盹，看起来非常可爱。'), 1.0895041),
(Document(page_content='我的狗喜欢追逐球，看起来非常开心。'), 1.3836973),
(Document(page_content='我的手机突然关机了，让我有些焦虑。'), 1.5533546)]
```

由于不同文本嵌入模型生成向量的范围不一致，LangChain 封装的 Faiss 计算相关性得分的时候，可能会出现 bug（比如出现负数）。

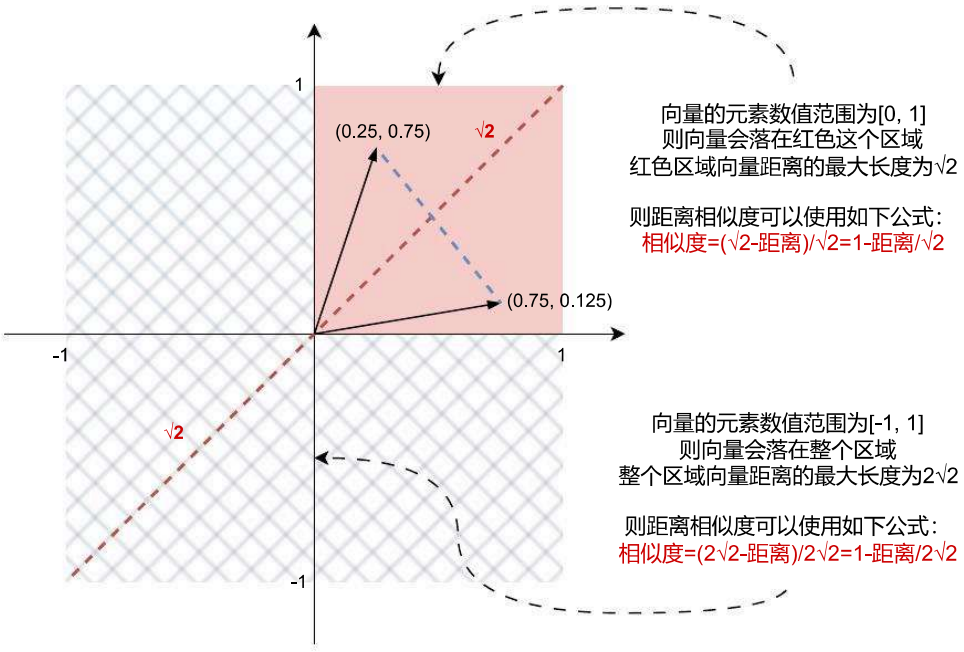
```
print(db.similarity_search_with_relevance_scores("我养了一只猫，叫笨笨"))

# 输出
[(Document(page_content='笨笨是一只很喜欢睡觉的猫咪', metadata={'page': 1}),
0.4592331743070337), (Document(page_content='猫咪在窗台上打盹，看起来非常可爱。',
metadata={'page': 3}), 0.22960424668403867), (Document(page_content='我的狗喜欢追逐
球，看起来非常开心。', metadata={'page': 10}), 0.02157827632118159),
(Document(page_content='我的手机突然关机了，让我有些焦虑。', metadata={'page': 7}),
-0.09838758604956)]
```

Faiss 计算相关性得分的核心代码如下（默认使用欧几里得距离计算相关性）：

```
# langchain_community/vectorstores/faiss.py -> FAISS
def _euclidean_relevance_score_fn(distance: float) -> float:
    return 1.0 - distance / math.sqrt(2)
```

这个公式计算正确的前提是向量只有 2 维，并且向量的每个值范围是 [0, 1]，这个公式的几何意义如下：

欧几里得距离相似性计算公式逻辑



通用相似度计算公式为：1/(1+距离)

扩展到三维向量空间中（向量的范围是 [0, 1]），两个向量/点之间最大距离为 `√3`，并不是 `√2`，所以直接套用公式，可能会出现负数得分，在 N 维向量空间下，两点的最大距离是 `√N`，所以出现负数的概率大大增加。

可以修改成以下方式进行修正：

```
def _euclidean_relevance_score_fn(distance: float) -> float:
    return 1.0 / (1.0 + distance)
```

> **Important**
>
> 在使用 LangChain 封装的向量数据库时，一定要注意测试和校验下文本嵌入模型生成向量的数值范围，避免出现明显的错误。由于向量数据库目前更新太快，而且 LangChain 封装了太多的第三方组件（数百个），在很多场合下，LangChain 可能没有对每一种情况进行测试，有可能会出现一些莫名其妙的计算结果。

2.2 带过滤的相似性搜索

在绝大部分向量数据库中，除了存储向量数据，还支持存储对应的元数据，这里的元数据可以是**文本原文、扩展信息、页码、归属文档id、作者、创建时间**等等任何自定义信息，一般在向量数据库中，会通过元数据来实现对数据的检索。

```
向量数据库记录 = 向量(vector)+元数据(metadata)+id
```

比较遗憾的是 Faiss 原生并不支持过滤，所以在 LangChain 封装的 FAISS 中对过滤功能进行了相应的处理。首先获取比 `k` 更多的结果 `fetch_k`（默认为 20 条），然后先进行搜索，接下来再搜索得到的 `fetch_k` 条结果上进行过滤，得到 `k` 条结果，从而实现带过滤的相似性搜索。

而且 Faiss 的搜索都是针对 `元数据` 的，在 Faiss 中执行带过滤的相似性搜索非常简单，只需要在搜索时传递 `filter` 参数即可，`filter` 可以传递一个元数据字典，也可以接收一个函数（函数的参数为元数据字典，返回值为布尔值）。

例如下方的代码只会对 `page>5` 的文档进行检索，代码如下：

```
import dotenv
from langchain_community.vectorstores import FAISS
from langchain_openai import OpenAIEmbeddings

dotenv.load_dotenv()

embedding = OpenAIEmbeddings(model="text-embedding-3-small")

texts: list = [
    "笨笨是一只很喜欢睡觉的猫咪",
    "我喜欢在夜晚听音乐，这让我感到放松。",
    "猫咪在窗台上打盹，看起来非常可爱。",
    "学习新技能是每个人都应该追求的目标。",
    "我最喜欢的食物是意大利面，尤其是番茄酱的那种。",
    "昨晚我做了一个奇怪的梦，梦见自己在太空飞行。",
    "我的手机突然关机了，让我有些焦虑。",
    "阅读是我每天都会做的事情，我觉得很充实。",
    "他们一起计划了一次周末的野餐，希望天气能好。",
    "我的狗喜欢追逐球，看起来非常开心。",
]
```

```
metadatas: list = [
    {"page": 1},
    {"page": 2},
    {"page": 3},
    {"page": 4},
    {"page": 5},
    {"page": 6},
    {"page": 7},
    {"page": 8},
    {"page": 9},
    {"page": 10},
]
db = FAISS.from_texts(texts, embedding, metadatas)

print(db.index_to_docstore_id)
print(db.similarity_search_with_score("我养了一只猫，叫笨笨", filter=lambda x:
x["page"] > 5))
```

输出结果：

```
{0: '452f290d-3afa-4989-a168-2d222a92093e', 1: '71bc9dcf-751c-4e65-9b61-
5003c43c8474', 2: '66a7bc83-df40-4036-b7c3-1b747ca4ee98', 3: '829e5148-139b-4185-
95c1-341681d6ca5a', 4: '24038a82-a083-4ec5-99cd-adaf81036f98', 5: 'b0f16e08-8cf3-
4f08-87fb-d635604dee82', 6: '668e6593-5f2c-4f86-95ff-93cf679e05a7', 7: '9c6359ae-
42c4-438e-bcf0-d35037c857e4', 8: '7f2c926e-d390-46f8-8485-6685c898bc45', 9:
'b347b82e-ec1a-4583-baa8-61d5f68e92a0'}
[(Document(page_content='我的狗喜欢追逐球，看起来非常开心。', metadata={'page': 10}),
1.3836973), (Document(page_content='我的手机突然关机了，让我有些焦虑。', metadata=
{'page': 7}), 1.5533546), (Document(page_content='阅读是我每天都会做的事情，我觉得很充
实。', metadata={'page': 8}), 1.5989475), (Document(page_content='他们一起计划了一次
周末的野餐，希望天气能好。', metadata={'page': 9}), 1.7179501)]
```

2.3 删除指定数据

在 Faiss 中，支持删除向量数据库中特定的数据，目前仅支持传入数据条目 id 进行删除，并不支持条件筛选（但是可以通过条件筛选找到符合的数据，然后提取 id 列表，然后批量删除）。

代码示例：

```
print("删除前数量:", db.index.ntotal)
# 获取向量数据库的索引id列表信息
db.delete([db.index_to_docstore_id[0]])
print("删除后数量:", db.index.ntotal)
```

输出结果：

```
删除前数量: 10
删除后数量: 9
```

2.4 保存和加载本地数据

除了从文本和文档列表中加载数据到向量数据库，Faiss 还支持将整个数据库持久化到本地文件，亦或者从本地文件一键加载数据，这样就不需要在每次使用向量数据库的时候重新创建，可以极大提升向量数据库的使用效率，两个方法如下：

1. `save_local()`：将向量数据库持久化到本地，传递 `folder_path` 和 `index` 分别代表文件夹路径与索引名字。
2. `load_local()`：将本地的数据加载到向量数据库，传递 `folder_path`、`embeddings` 和 `index` 分别代表文件夹路径、嵌入模型、索引名字。

示例：

```
db.save_local("./vector-store/")
new_db = FAISS.load_local("./vector-store/", embedding,
allow_dangerous_deserialization=True)
docs = new_db.similarity_search("我养了一只猫，叫笨笨")
```

输出结果：

```
[Document(page_content='笨笨是一只很喜欢睡觉的猫咪', metadata={'page': 1}),
Document(page_content='猫咪在窗台上打盹，看起来非常可爱。', metadata={'page': 3}),
Document(page_content='我的狗喜欢追逐球，看起来非常开心。', metadata={'page': 10}),
Document(page_content='我的手机突然关机了，让我有些焦虑。', metadata={'page': 7})]
```

Pinecone 向量数据库的配置与使用

Pinecone 向量数据库的配置与使用

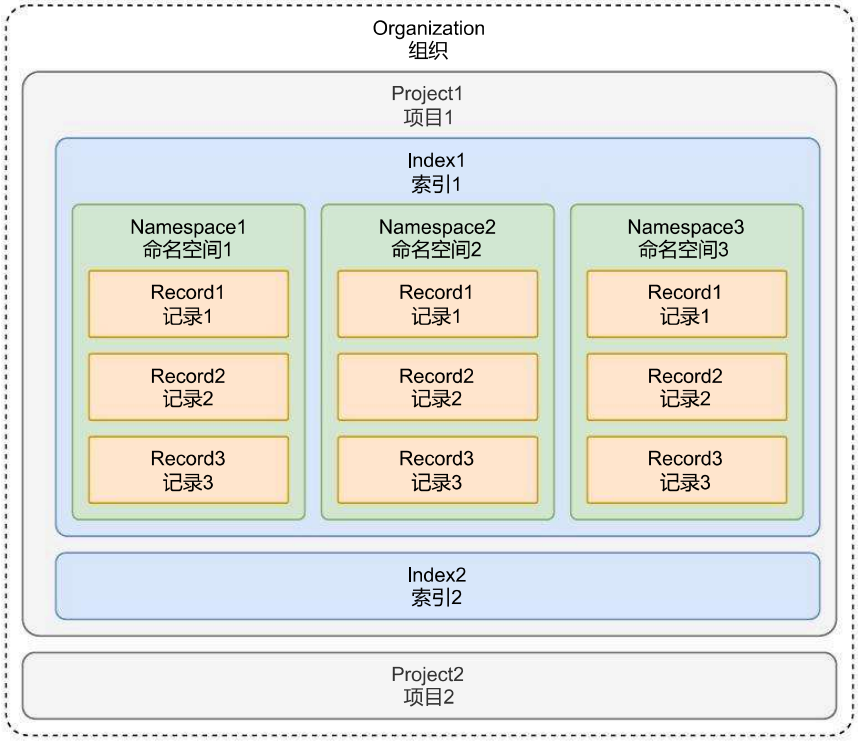
01. Pinecone 向量数据库简介

Pinecone 是一个托管的、云原生的向量数据库，具有极简的 API，并且无需在本地部署即可快速使用，Pinecone 服务提供商还为每个账户设置了足够的免费空间，**在开发阶段，可以快速基于 Pinecone 快速开发 AI 应用。**

相关资料：

1. Pinecone 官网：https://www.pinecone.io/
2. Pinecone 翻译文档：https://www.pinecone-io.com/
3. langchain-pinecone 翻译文档：http://imooc-langchain.shortvar.com/docs/integrations/vectorstores/pinecone/

Pinecone 向量数据库的设计架构与 Faiss 差异较大，Pinecone 由于是一个面向商业端的向量数据库，在功能和概念上会更加丰富，有几个核心概念+架构图如下：



概念的解释如下：

1. **组织**：组织是使用相同结算方式的一个或者多个项目的集合，例如个人账号、公司账号等都算是一个组织。
2. **项目**：项目是用来管理向量数据库、索引、硬件资源等内容的整合，可以将不同的项目数据进行区分。
3. **索引**：索引是 Pinecone 中数据的最高组织单位，在索引中需要定义向量的存储维度、查询时使用的相似性指标，并且在 Pinecone 中支持两种类型的索引：无服务器索引（根据数据大小自动扩容）和 Pod 索引（预设空间/硬件）。

4. **命名空间**：命名空间是索引内的分区，用于将索引中的数据区分成不同的组，以便于在不同的组内存储不同的数据，例如知识库、记忆的数据可以存储到不同的组中，类似 Excel 中的 `Sheet表`。
5. **记录**：记录是数据的基本单位，一条记录涵盖了 ID、向量(values)、元数据(metadata) 等。

所以在 Pinecone 中使用向量数据库，要确保 `组织`、`项目`、`索引`、`命名空间`、`记录` 等内容均配置好才可以使用，并且由于 Pinecone 是云端向量数据库，使用时还需配置对应的 API 秘钥（可在注册好 Pinecone 后管理页面的 API Key 中设置）。

对于 Pinecone，LangChain 团队也封装了响应的包，安装命令：

```
pip install -U langchain-pinecone
```

然后在 `.env` 文件中配置对应的 API 秘钥，如下：

```
# Pinecone向量数据库
PINECONE_API_KEY=xxx
```

接下来就可以像 Faiss 一样去使用 LangChain 封装好的向量数据库了（会有稍许差异，这是不同向量数据库之间的一些差异）。

02. Pinecone 向量数据库使用技巧

2.1 数据的导入

Pinecone 和 Faiss 一样拥有 `from_texts` 和 `from_documents` 方法，支持快捷从 `文本列表` 和 `文档列表` 中构建向量数据库，也支持通过 `构造函数` 实例化 Pinecone 向量数据库后，通过 `add_texts` 的方式添加数据。

在配置好 `Index(llmops)`、`namespace(dataset)`、`api_key` 后，可执行以下代码完成向量数据库数据的添加：

```
import dotenv
from langchain_openai import OpenAIEmbeddings
from langchain_pinecone import PineconeVectorStore

dotenv.load_dotenv()

embedding = OpenAIEmbeddings(model="text-embedding-3-small")
db = PineconeVectorStore(index_name="llmops", embedding=embedding,
namespace="dataset")
db.add_texts([
    "笨笨是一只很喜欢睡觉的猫咪",
    "我喜欢在夜晚听音乐，这让我感到放松。",
    "猫咪在窗台上打盹，看起来非常可爱。",
    "学习新技能是每个人都应该追求的目标。",
    "我最喜欢的食物是意大利面，尤其是番茄酱的那种。",
    "昨晚我做了一个奇怪的梦，梦见自己在太空飞行。",
    "我的手机突然关机了，让我有些焦虑。",
    "阅读是我每天都会做的事情，我觉得很充实。",
    "他们一起计划了一次周末的野餐，希望天气能好。",
    "我的狗喜欢追逐球，看起来非常开心。",
], [
    {"page": 1},
```

```
    {"page": 2},
    {"page": 3},
    {"page": 4},
    {"page": 5},
    {"page": 6, "account_id": 1},
    {"page": 7},
    {"page": 8},
    {"page": 9},
    {"page": 10},
], namespace="dataset")

print(db.similarity_search_with_relevance_scores("我养了一只猫，叫笨笨"))
```

输出内容：

```
[(Document(page_content='笨笨是一只很喜欢睡觉的猫咪', metadata={'page': 1.0}), 0.8095565), (Document(page_content='猫咪在窗台上打盹，看起来非常可爱。', metadata={'page': 3.0}), 0.7276270835), (Document(page_content='我的狗喜欢追逐球，看起来非常开心。', metadata={'page': 10.0}), 0.6540447475), (Document(page_content='我的手机突然关机了，让我有些焦虑。', metadata={'page': 7.0}), 0.611671284)]
```

2.2 带过滤的相似性搜索

和 Faiss 不同，Pinecone 支持原生的带过滤的相似性检索功能（元数据筛选），使用元数据过滤器会精确检索与过滤器匹配的最临近结果数，在 Pinecone 中，过滤器格式为 json，其中 `键` 是元数据字段对应字符串，`值` 是 字符串、数字、布尔值、列表、json 中的一种。

例如下方是精确筛选的过滤器：

```
{
    "genre": "action",
    "year": 2020,
    "length_hrs": 1.5
}

{
    "color": "blue",
    "fit": "straight",
    "price": 29.99,
    "is_jeans": true
}
```

除此之外，筛选器的值还可以传递 json 数据，用于支持更加复杂的条件搜索，而且还可以和 AND/OR 配合，

筛选条件	描述	支持的类型
`$eq`	将元数据值等于指定值的向量进行匹配	数字、字符串、布尔值
`$ne`	匹配元数据值不等于指定值的向量	数字、字符串、布尔值
`$gt`	匹配元数据值大于指定值的向量	数字
`$gte`	匹配元数据值大于或等于指定值的向量	数字

筛选条件	描述	支持的类型
`$lt`	匹配元数据值小于指定值的向量	数字
`$lte`	匹配元数据值小于或等于指定值的向量	数字
`$in`	将向量与指定数组中的元数据值进行匹配	字符串，数字
`$nin`	与不在指定数组中的元数据值的向量匹配	字符串，数字
`$exists`	与存在/不存在特定元数据值的向量匹配	布尔值
`$and`	多个条件同时符合	-
`$or`	多个条件只需要满足一个	-

例如检索页数小于等于 5 页的数据，代码示例：

```
import dotenv
from langchain_openai import OpenAIEmbeddings
from langchain_pinecone import PineconeVectorStore

dotenv.load_dotenv()

embedding = OpenAIEmbeddings(model="text-embedding-3-small")
db = PineconeVectorStore(index_name="llmops", embedding=embedding,
namespace="dataset")

resp = db.similarity_search_with_relevance_scores(
    "我养了一只猫叫笨笨",
    filter={"page": {"$lte": 5}}
)
```

输出内容：

```
[(Document(page_content='笨笨是一只很喜欢睡觉的猫咪', metadata={'page': 1.0}),
0.8121996819999999), (Document(page_content='猫咪在窗台上打盹，看起来非常可爱。',
metadata={'page': 3.0}), 0.7371905), (Document(page_content='我喜欢在夜晚听音乐，这让
我感到放松。', metadata={'page': 2.0}), 0.5996375455), (Document(page_content='我最
喜欢的食物是意大利面，尤其是番茄酱的那种。', metadata={'page': 5.0}),
0.5745788215000001)]
```

例如下方，在 `page=5` 并且 `account_id=1` 的向量数据里进行相似性检索：

```
resp = db.similarity_search_with_relevance_scores(
    "我养了一只猫叫笨笨",
    filter={"$and": [{"page": 5}, {"account_id": 1}]}
)

# 输出，没有符合条件的数据
[]
```

改成 `$or` 则为符合其中一个条件即可：

```
resp = db.similarity_search_with_relevance_scores(
    "我养了一只猫叫笨笨",
    filter={"$or": [{"page": 5}, {"account_id": 1}]}
)

# 输出
[(Document(page_content='我最喜欢的食物是意大利面，尤其是番茄酱的那种。', metadata=
{'page': 5.0}), 0.574801199), (Document(page_content='昨晚我做了一个奇怪的梦，梦见自己
在太空飞行。', metadata={'account_id': 1.0, 'page': 6.0}), 0.5700240435)]
```

关于 Pinecone 更详细的筛选用法文档：https://docs.pinecone.io/guides/data/filter-with-metadata

2.3 删除指定数据

Pinecone 向量数据库中同样支持删除数据，并且支持按照 `id列表`、`按照命名空间删除全部数据`、`条件删除` 等多种策略，`delete()` 函数参数如下：

1. ids：需要删除的 id 列表，非必填参数。
2. delete_all：是否删除全部数据，一般结合 namespace 一起使用，代表删除命名空间内的所有数据，非必填参数。
3. namespace：需要删除的命名空间数据，非必填参数。
4. filter：过滤器，格式为 json/dict，删除符合条件的所有数据，在无服务器索引的情况下，不支持通过元数据筛选器删除对应的数据，只在 Pod 索引下支持，检索器的用法和相似性搜索一模一样。

例如，删除 `id=21c694f3-1e12-47b5-af9d-c93de29ad093` 对应的向量数据，如下：

```
db.delete(["21c694f3-1e12-47b5-af9d-c93de29ad093"])
```

关于 Pineone 删除数据的用法文档：https://docs.pinecone.io/guides/data/delete-data

2.4 获取原始实例

如果 LangChain 封装的 VectorStore 对应的方法并不能满足相应的需求，一般情况下，还可以获取到向量数据库的原始实例，例如 Pinecone 提供了 `update` 方法，但是在 LangChain 封装的 PineconeVectorStore 并没有提供这个方法，所以可以考虑从原始实例上调用该方法。

可以通过 `.get_pinecone_index()` 函数来获取对应的 `Index`，从而进行相应的操作：

```
llmops_index = db.get_pinecone_index("llmops")

llmops_index.update(id="xxx", values=[xxx], namespace="dataset")
```

关于原始实例操作数据的相关文档：https://docs.pinecone.io/guides/data/upsert-data

TCVectorDB 向量数据库的配置与使用

TCVectorDB 向量数据库的配置与使用

01. TCVectorDB 向量数据库简介

虽然目前绝大部分开源向量数据库都是海外的，配置起来也非常简单，性能也很高，但是因为网络的原因，如果将向量数据库部署到海外，而产品面向国内，网络延迟是一个必要考虑的问题，所以考虑国内服务提供商的云向量数据库往往是最佳的选择。

腾讯云向量数据库（TCVectorDB）是一款全托管的自研企业级分布式数据库服务， 专用于存储、检索、分析多维向量数据，该数据库支持多种索引类型和相似度计算方法，索引支持千亿级向量规模，可支持百万级 QPS 及毫秒级查询延迟。而且目前认证过的腾讯云账号可以免费使用 TCVectorDB 一个月时间，相关文档：

1. TCVectorDB 产品链接：https://cloud.tencent.com/product/vdb
2. TCVectorDB 产品文档：https://cloud.tencent.com/document/product/1709
3. LangChain TCVectorDB 翻译文档：http://imooc-langchain.shortvar.com/docs/integrations/vectorstores/tencentvectordb/

TCVectorDB 和 Pinecone 的设计理念非常接近，在腾讯云向量数据库中也有对应的 `数据库`、`集合`、`记录` 等概念：

1. **数据库**：分为普通向量数据库和 AI 数据库，其中 AI 数据库无需外部配置文本分割、Embedding、文档解析等功能，底层全部有腾讯云实现，而普通向量数据库则需要外部程序处理，它只能接收传递的数据，并没有处理功能，但是功能更加可定制化。
2. **集合**：集合是数据库的下一个单位，类似与传统数据库中的 `表`，在集合中，需要设置集合名称、分片数、索引等信息。
3. **记录**：集合的每一条数据就是记录。

TCVectorDB 默认只能在内网中链接使用，在生产环境中，也尽可能不将数据库暴露到外网中，但是在开发中，则需要配置并开启外网访问功能，配置好外网访问功能、API 秘钥后，需要在项目中导入对应的环境变量。

```
TC_VECTOR_DB_URL=xxx
TC_VECTOR_DB_USERNAME=root
TC_VECTOR_DB_KEY=xxx
TC_VECTOR_DB_DATABASE=llmops
TC_VECTOR_DB_TIMEOUT=30
```

接下来安装对应的 Python 包，命令如下：

```
pip install tcvectordb
```

接下来就可以像 Faiss、Pinecone 一样正常使用即可，整体功能和 Pinecone 几乎一模一样，只是 `filter`、`namespace` 等概念的操作有些许差异。

02. TCVectorDB 向量数据库使用技巧

2.1 内置 Embedding 导入数据与查询

TCVectorDB 向量数据库内置了 Embedding 模型，并支持 5 种嵌入方式，如下：

模型名	适用语言类型	维度	最大 Token 数量	分类得分	聚类得分	检索得分
bge-base-zh（推荐）	中文	768	512	67.06	47.64	69.53
m3e-base	中文	768	512	67.52	47.68	56.91
text2vec-large-chinese	中文	1024	512	60.66	30.02	41.94
e5-large-v2	英文	1024	512	75.24	44.49	50.56
multilingual-e5-base	多语言	768	514	63.35	40.68	40.68

使用内置的 Embedding 模型速度会更快（而且目前 LangChain 封装的 TCVectorDB 使用内置 Embedding 没有 bug），不过限制也比较多，迁移数据库时，所有的文本全部需要重新生成。

内置 Embedding 模型文档说明&计费：https://cloud.tencent.com/document/product/1709/98014

示例：

```
import os

import dotenv
from langchain_community.vectorstores import TencentVectorDB
from langchain_community.vectorstores.tencentvectordb import (
    ConnectionParams,
)
from langchain_openai import OpenAIEmbeddings

dotenv.load_dotenv()

embedding = OpenAIEmbeddings(model="text-embedding-3-small")
db = TencentVectorDB(
    embedding=None,
    connection_params=ConnectionParams(
        url=os.environ.get("TC_VECTOR_DB_URL"),
        username=os.environ.get("TC_VECTOR_DB_USERNAME"),
        key=os.environ.get("TC_VECTOR_DB_KEY"),
        timeout=int(os.environ.get("TC_VECTOR_DB_TIMEOUT")),
    ),
    database_name=os.environ.get("TC_VECTOR_DB_DATABASE"),
    collection_name="dataset-builtin",
)

texts = [
    "笨笨是一只很喜欢睡觉的猫咪",
```

```
    "我喜欢在夜晚听音乐，这让我感到放松。",
    "猫咪在窗台上打盹，看起来非常可爱。",
    "学习新技能是每个人都应该追求的目标。",
    "我最喜欢的食物是意大利面，尤其是番茄酱的那种。",
    "昨晚我做了一个奇怪的梦，梦见自己在太空飞行。",
    "我的手机突然关机了，让我有些焦虑。",
    "阅读是我每天都会做的事情，我觉得很充实。",
    "他们一起计划了一次周末的野餐，希望天气能好。",
    "我的狗喜欢追逐球，看起来非常开心。",
]

ids = db.add_texts(texts)
print("添加文档id列表:", ids)

print(db.similarity_search_with_score("我养了一只猫，叫笨笨"))
```

输出示例：

```
添加文档id列表: ['1721204162283324200-8452797147690134545-0', '1721204162283324200-
4748609318240602995-1', '1721204162283324200-5687737768520280782-2',
'1721204162283324200--327905253905429573-3', '1721204162283324200-
-4217514370195306218-4', '1721204162283324200-5510932568003309936-5',
'1721204162283324200-7945499047074485108-6', '1721204162283324200-
8077162730065074156-7', '1721204162283324200-36196153077512649-8',
'1721204162283324200-3472252428185052217-9']

[(Document(page_content='笨笨是一只很喜欢睡觉的猫咪'), 0.874507),
(Document(page_content='我的狗喜欢追逐球，看起来非常开心。'), 0.757709),
(Document(page_content='猫咪在窗台上打盹，看起来非常可爱。'), 0.748665),
(Document(page_content='昨晚我做了一个奇怪的梦，梦见自己在太空飞行。'), 0.735823)]
```

2.2 外部 Embedding 导入数据与查询

除了使用内置的 Embedding 模型，TCVectorDB 也支持传递自定义的文本嵌入模型，但是在 **LangChain 中封装的 TencentVectorDB 存在一个 bug**，当传递外部 Embedding 模型时，必须配置 meta_fields 属性，并且指定字段名称为 `text`，然后将 metadatas 中的数据添加上 `text` 这个字段，向量数据库才会记录原文信息。

`meta_fields` 的字段是告知 TCVectorDB 需要存储 metadata 中的哪些字段，如果没有设置，metadata 中的所有字段均不会被保存。

如果没有记录原文信息，在进行检索操作时，会因为找不到原文，从而没法创建 Document 文档，引发报错。

出现 bug 的源码：

```
# langchain_community/vectorstores/tencentvectordb.py->TencentVectorDB::add_texts
def add_texts(
    self,
    texts: Iterable[str],
    metadatas: Optional[List[dict]] = None,
    timeout: Optional[int] = None,
    batch_size: int = 1000,
    ids: Optional[List[str]] = None,
    **kwargs: Any,
```

```
) -> List[str]:
    ...
    if embeddings:
        doc_attrs["vector"] = embeddings[id]
    else:
        doc_attrs["text"] = texts[id]
    ...
```

修复示例:

```
import os

import dotenv
from langchain_community.vectorstores import TencentVectorDB
from langchain_community.vectorstores.tencentvectordb import (
    ConnectionParams,
    MetaField,
    META_FIELD_TYPE_STRING,
)
from langchain_openai import OpenAIEmbeddings

dotenv.load_dotenv()

embedding = OpenAIEmbeddings(model="text-embedding-3-small")
db = TencentVectorDB(
    embedding=embedding,
    connection_params=ConnectionParams(
        url=os.environ.get("TC_VECTOR_DB_URL"),
        username=os.environ.get("TC_VECTOR_DB_USERNAME"),
        key=os.environ.get("TC_VECTOR_DB_KEY"),
        timeout=int(os.environ.get("TC_VECTOR_DB_TIMEOUT")),
    ),
    database_name=os.environ.get("TC_VECTOR_DB_DATABASE"),
    collection_name="dataset-external",
    meta_fields=[
        # 配置text字段，防止LangChain在检索时，找不到字段构建Document文档，从而触发错误
        MetaField(name="text", data_type=META_FIELD_TYPE_STRING),
    ]
)

texts = [
    "笨笨是一只很喜欢睡觉的猫咪",
    "我喜欢在夜晚听音乐，这让我感到放松。",
    "猫咪在窗台上打盹，看起来非常可爱。",
    "学习新技能是每个人都应该追求的目标。",
    "我最喜欢的食物是意大利面，尤其是番茄酱的那种。",
    "昨晚我做了一个奇怪的梦，梦见自己在太空飞行。",
    "我的手机突然关机了，让我有些焦虑。",
    "阅读是我每天都会做的事情，我觉得很充实。",
    "他们一起计划了一次周末的野餐，希望天气能好。",
    "我的狗喜欢追逐球，看起来非常开心。",
]
# 将原始文本数据合并到metadata中，
metadata = [{"text": text, "page": index} for index, text in enumerate(texts)]
```

```
ids = db.add_texts(texts, metadata)
print("添加文档id列表:", ids)

print(db.similarity_search("我养了一只猫，叫笨笨"))
```

输出内容：

```
添加文档id列表: ['1721204290046443100--7979142186803955240-0',
'1721204290046443100--2991801275249710134-1', '1721204290046443100-
7229055024046586038-2', '1721204290046443100-68200102425904647 95-3',
'1721204290046443100-1761951776563722534-4', '1721204290046443100-
7105227574091896456-5', '1721204290046443100-385681627517092469-6',
'1721204290046443100-2611196434988287836-7', '1721204290046443100-
5925412479147078471-8', '1721204290046443100--5880598133876375674-9']

[Document(page_content='笨笨是一只很喜欢睡觉的猫咪', metadata={'text': '笨笨是一只很喜欢
睡觉的猫咪'}), Document(page_content='猫咪在窗台上打盹，看起来非常可爱。', metadata=
{'text': '猫咪在窗台上打盹，看起来非常可爱。'}), Document(page_content='我的狗喜欢追逐球，
看起来非常开心。', metadata={'text': '我的狗喜欢追逐球，看起来非常开心。'}),
Document(page_content='我的手机突然关机了，让我有些焦虑。', metadata={'text': '我的手机突
然关机了，让我有些焦虑。'})]
```

2.3 TCVectorDB 带过滤的相似性搜索

TCVectorDB 也支持为相似性搜索添加过滤器，表达式的格式为 <field_name><operator><value>，多个表达式之间支持 and（与）、or（或）、not（非）关系。其中：

1. **<field_name>**：表示要过滤的字段名。
2. **<operator>**：要使用的运算符。
    - string：匹配单个字符串值（=）、排除单个字符串值（!=）、匹配任意一个字符串值（in）、排除所有字符串值（not in）。其对应的 Value 必须使用英文双引号括起来。
    - uint64：大于（>）、大于等于（>=）、等于（=）、小于（<）、小于等于（<=）、不等于（!=）。
3. **<value>**：表示要匹配的值。

例如：`uint64_val > 30`、`game_tag="Detective"` 等。

并且在 TCVectorDB 使用非向量字段进行检索时，必须在构建 `Collection` 时添加上对应的索引，否则无法检索会报错。

详细文档：：https://cloud.tencent.com/document/product/1222/98734#2e2e982a-a487-4a05-8410-58c6bda12180

代码示例：

```
import os

import dotenv
from langchain_community.vectorstores import TencentVectorDB
from langchain_community.vectorstores.tencentvectordb import (
    ConnectionParams,
    MetaField,
    META_FIELD_TYPE_UINT64,
```

```
)
from langchain_openai import OpenAIEmbeddings

dotenv.load_dotenv()

db = TencentVectorDB(
    embedding=None,
    connection_params=ConnectionParams(
        url=os.environ.get("TC_VECTOR_DB_URL"),
        username=os.environ.get("TC_VECTOR_DB_USERNAME"),
        key=os.environ.get("TC_VECTOR_DB_KEY"),
        timeout=int(os.environ.get("TC_VECTOR_DB_TIMEOUT")),
    ),
    database_name=os.environ.get("TC_VECTOR_DB_DATABASE"),
    collection_name="dataset-filter",
    meta_fields=[
        MetaField(name="page", data_type=META_FIELD_TYPE_UINT64),
    ]
)

texts = [
    "笨笨是一只很喜欢睡觉的猫咪",
    "我喜欢在夜晚听音乐，这让我感到放松。",
    "猫咪在窗台上打盹，看起来非常可爱。",
    "学习新技能是每个人都应该追求的目标。",
    "我最喜欢的食物是意大利面，尤其是番茄酱的那种。",
    "昨晚我做了一个奇怪的梦，梦见自己在太空飞行。",
    "我的手机突然关机了，让我有些焦虑。",
    "阅读是我每天都会做的事情，我觉得很充实。",
    "他们一起计划了一次周末的野餐，希望天气能好。",
    "我的狗喜欢追逐球，看起来非常开心。",
]
metadatas = [
    {"page": 1},
    {"page": 2},
    {"page": 3},
    {"page": 4},
    {"page": 5},
    {"page": 6, "account_id": 1},
    {"page": 7},
    {"page": 8},
    {"page": 9},
    {"page": 10},
]

ids = db.add_texts(texts, metadatas)
print("添加文档id列表:", ids)

print(db.similarity_search_with_score("我养了一只猫，叫笨笨", expr="page>=9"))
```

输出内容:

```
添加文档id列表: ['1721205246238323600--5707869341580869917-0',
'1721205246238323600-9777153695249071 79-1', '1721205246238323600-
6780760367391118320-2', '1721205246238323600-9214408441339088766-3',
'1721205246238323600--5359114623033048472-4', '1721205246238323600-
5350375289397780096-5', '1721205246238323600-1143848279187318822-6',
'1721205246238323600-1555278682065468720-7', '1721205246238323600-
-8454962785458887831-8', '1721205246238323600--7242128883936657791-9']
[(Document(page_content='我的狗喜欢追逐球，看起来非常开心。', metadata={'page': 10}),
0.757709), (Document(page_content='他们一起计划了一次周末的野餐，希望天气能好。',
metadata={'page': 9}), 0.675508)]
```

Weaviate 向量数据库的配置与使用

Weaviate 向量数据库的配置与使用

01. Weaviate 介绍

Weaviate 是完全使用 Go 语言构建的开源向量数据库，提供了强大的数据存储和检索功能。并且 Weaviate 提供了多种部署方式，以满足不同用户和用例的需求，部署方式如下：

1. **Weaviate 云**：使用 Weaviate 官方提供的云服务，支持数据复制、零停机更新、无缝扩容等功能，适用于评估、开发和生产场景。
2. **Docker 部署**：使用 Docker 容器部署 Weaviate 向量数据库，适用于评估和开发等场景。
3. **K8s 部署**：在 K8s 上部署 Weaviate 向量数据库，适用于开发和生产场景。
4. 嵌入式 Weaviate：基于本地文件的方式构建 Weaviate 向量数据库，适用于评估场景，不过嵌入式 Weaviate 只适用于 Linux、macOS 系统，在 Windows 下不支持。

Weaviate 和 Pinecone/TCVectorDB 一样，也存在着集合的概念，在 Weaviate 中集合类似传统关系型数据库中的表，负责管理一类数据/数据对象，要使用 Weaviate 的流程其实也非常简单：

1. **创建部署 Weaviate 数据库（使用 Weaviate 云、Docker 部署）。**
2. **安装 Python 客户端/LangChain 集成包。**
3. **连接 Weaviate（本地连接、云端连接）。**
4. **创建数据集/集合（代码创建、可视化管理界面创建），在 Weaviate 中，集合的名字必须以大写字母开头，并且只能包含字母、数字和下划线，否则创建的时候会出错，和 Python 的类名规范几乎一致。**
5. **添加数据/向量。**
6. **相似性搜索/带过滤器的相似性搜索。**

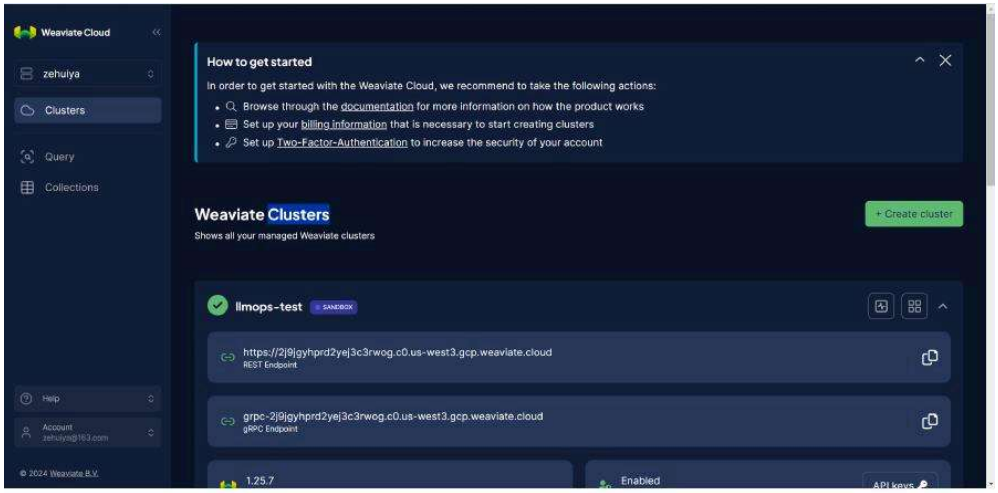
参考资料：

1. Weaviate 官网：https://weaviate.io/
2. Weaviate 快速上手指南：https://weaviate.io/developers/weaviate/quickstart
3. LangChain Weaviate 集成包翻译文档：https://imooc-langchain.shortvar.com/docs/integrations/vectorstores/weaviate

02. Weaviate 云服务

Weaviate 官方为所有注册登录的账号提供了无限量的 Weaviate 云服务（免费账号每个实例使用时间最大为 14 天，付费账户不限），通过邮箱注册登录 Weaviate 后，找到后台管理系统的 `Clusters(集群)` 即可快速创建 Weaviate 向量数据库实例。

Weaviate 后台管理面板：https://console.weaviate.cloud/dashboard



创建好 Weaviate 云服务器集群后，平台提供了 REST 和 gRPC 两种链接方式的地址与 API 秘钥，在客户端中即可快速连接到云服务。

03. Docker 部署 Weaviate 向量数据库

在 Docker 上部署 Weaviate 向量数据库非常简单，如果使用默认值，则不需要 `docker-compose.yml` 文件来运行镜像（适用于开发场景），安装好 Docker 之后，执行如下命令：：

```
docker run -d --name weaviate-dev -p 8080:8080 -p 50051:50051
cr.weaviate.io/semitechnologies/weaviate:1.24.20
```

上述的命令就会快速创建一个叫 `weaviate-dev` 的容器并在后台运行，该容器暴露了两个端口，8080 和 50051，其中 8080 端口为 REST 接口连接端口、50051 为 gRPC 服务连接端口。

除此之外，使用 Docker 部署的 Weaviate 向量数据库服务，还有以下几个常见命令：

```
# 启动 weaviate 服务
docker start weaviate

# 关闭 weaviate 服务
docker stop weaviate

# 移除 weaviate 容器
docker rm weaviate-dev

# 查看当前 docker 容器所有镜像
docker images

# 移除 weaviate 镜像
docker rmi cr.weaviate.io/semitechnologies/weaviate

# 查看当前运行的 docker 服务
docker ps

# 查看所有 docker 容器（涵盖启动和未启动）
docker ps -a
```

04. Weaviate 向量数据库使用技巧

创建好 Weaviate 数据库服务后，接下来就可以安装 Python 客户端/LangChain 集成包，命令如下：

```
pip install -Uqq langchain-weaviate
```

下一步如果使用的是 Weaviate 云服务，可以直接从可视化界面创建 `Collection`，亦或者在使用时 LangChain 自动检测对应的数据集是否存在，如果不存在则直接创建。

然后就可以考虑连接 Weaviate 服务了，Weaviate 框架针对不同的部署方式提供的不同的连接方法：

1. `weaviate.connect_to_local()`：连接到本地的部署服务，需配置连接 URL、端口号。
2. `weaviate.connect_to_wcs()`：连接到远程的 Weaviate 服务，需配置连接 URL、连接秘钥。

示例如下：

```
import weaviate

# 连接192.168.2.120:8080并创建weaviate客户端
client = weaviate.connect_to_local("192.168.2.120", "8080")
```

连接到远程的 Weaviate 服务代码如下：

```
import weaviate
from weaviate.auth import AuthApiKey

client = weaviate.connect_to_wcs(
    cluster_url="https://2j9jgyhprd2yej3c3rwog.c0.us-west3.gcp.weaviate.cloud",
    auth_credentials=AuthApiKey("BAn9bGZdZbdGCmUyfdegQoKFctyMmxaQdDFb")
)
```

创建好客户端后，接下来可以基于客户端创建 LangChain 向量数据库实例，在实例化 LangChain VectorDB 时，需要传递 client（客户端）、 index_name（集合名字）、text（原始文本的存储键）、embedding（文本嵌入模型），如下：

```
import dotenv
import weaviate
from langchain_openai import OpenAIEmbeddings
from langchain_weaviate import WeaviateVectorStore

dotenv.load_dotenv()

# 1.连接weaviate向量数据库
client = weaviate.connect_to_local("192.168.2.120", "8080")

# 2.实例化WeaviateVectorStore
embedding = OpenAIEmbeddings(model="text-embedding-3-small")
db = WeaviateVectorStore(client=client, index_name="DatasetTest",
text_key="text", embedding=embedding)
```

实例化 LangChain VectorDB 后，就可以像 Faiss、Pinecone、TCVectorDB 一样去使用了，例如执行新增数据后完成检索示例如下：

```python
import dotenv
import weaviate
from langchain_openai import OpenAIEmbeddings
from langchain_weaviate import WeaviateVectorStore

dotenv.load_dotenv()

# 1.连接weaviate向量数据库
client = weaviate.connect_to_local("192.168.2.120", "8080")

# 2.实例化WeaviateVectorStore
embedding = OpenAIEmbeddings(model="text-embedding-3-small")
db = WeaviateVectorStore(client=client, index_name="dataset-test",
text_key="text", embedding=embedding)

# 3.新增数据
ids = db.add_texts([
    "笨笨是一只很喜欢睡觉的猫咪",
    "我喜欢在夜晚听音乐，这让我感到放松。",
    "猫咪在窗台上打盹，看起来非常可爱。",
    "学习新技能是每个人都应该追求的目标。",
    "我最喜欢的食物是意大利面，尤其是番茄酱的那种。",
    "昨晚我做了一个奇怪的梦，梦见自己在太空飞行。",
    "我的手机突然关机了，让我有些焦虑。",
    "阅读是我每天都会做的事情，我觉得很充实。",
    "他们一起计划了一次周末的野餐，希望天气能好。",
    "我的狗喜欢追逐球，看起来非常开心。",
])

# 4.检索数据
print(db.similarity_search_with_score("笨笨"))
```

输出内容：

```
[(Document(page_content='笨笨是一只很喜欢睡觉的猫咪'), 0.699999988079071),
(Document(page_content='猫咪在窗台上打盹，看起来非常可爱。'), 0.2090398222208023),
(Document(page_content='我的狗喜欢追逐球，看起来非常开心。'), 0.19787956774234772),
(Document(page_content='我的手机突然关机了，让我有些焦虑。'), 0.11435992270708084)]
```

在 Weaviate 中，也支持带过滤器的相似性筛选，并且 LangChain Weaviate 社区包并没有对筛选过滤器进行二次封装，所以直接传递原生的 weaviate 过滤器即可，参考文档：https://weaviate.io/developers/weaviate/search/filters

例如需要检索 `page` 属性大于等于 `5` 的所有数据，可以构建一个 filters 后传递给检索方法，如下：

```python
from weaviate.classes.query import Filter

filters = Filter.by_property("page").greater_or_equal(5)
print(db.similarity_search_with_score("笨笨", filters=filters))
```

输出结果如下：

```
[(Document(page_content='我的狗喜欢追逐球，看起来非常开心。', metadata={'page': 10.0,
'account_id': None}), 0.699999988079071), (Document(page_content='我的手机突然关机
了，让我有些焦虑。', metadata={'page': 7.0, 'account_id': None}),
0.4045487940311432), (Document(page_content='昨晚我做了一个奇怪的梦，梦见自己在太空飞
行。', metadata={'page': 6.0, 'account_id': 1.0}), 0.318904846906662),
(Document(page_content='我最喜欢的食物是意大利面，尤其是番茄酱的那种。', metadata=
{'page': 5.0, 'account_id': None}), 0.2671944797039032)]
```

如果想获取 Weaviate 原始集合的实例，可以通过 `db._collection` 快速获得，从而去执行一些原始操作，例如：

```python
from weaviate.classes.query import MetadataQuery

collection = db._collection
response = reviews.query.near_text(
    query="a sweet German white wine",
    limit=2,
    target_vector="title_country",  # Specify the target vector for named vector
collections
    return_metadata=MetadataQuery(distance=True)
)

for o in response.objects:
    print(o.properties)
    print(o.metadata.distance)
```

对接自定义向量数据库的配置与使用

对接自定义向量数据库的配置与使用

01. 对接自定义向量数据库

向量数据库的发展非常迅猛，几乎间隔几天就有新的向量数据库发布，LangChain 不可能将所有向量数据库都进行集成，亦或者封装的包存在这一些 bug 或错误，这个时候就需要考虑创建自定义向量数据库，去实现特定的方法。

在 LangChain 实现自定义向量数据库的类有两种模式，一种是继承封装好的数据库类，一种是继承基类 VectorStore。前一种一般继承后重写部分方法进行扩展或者修复 bug，后面一种是对接新的向量数据库。

在 LangChain 中，继承 VectorStore 只需实现最基础的 3 个方法即可正常使用：

1. **add_texts**：将对应的数据添加到向量数据库中。
2. **similarity_search**：最基础的相似性搜索。
3. **from_texts**：从特定的文本列表、元数据列表中构建向量数据库。

其他方法因为使用频率并不高，VectorStore 并没有设置成虚拟方法，但是再没有实现的情况下，直接调用会报错，涵盖：

1. delete()：删除向量数据库中的数据。
2. _select_relevance_score_fn()：根据距离计算相似性得分函数。
3. similarity_search_with_score()：携带得分的相似性搜索函数。
4. similarity_search_by_vector()：传递向量进行相似性搜索。
5. max_marginal_relevance_search()：最大边界相似性搜索。
6. max_marginal_relevance_search_by_vector()：传递向量进行最大边界相关性搜索。

02. 自定义 VectorStore 示例

要在 LangChain 中对接自定义向量数据，本质上就是将向量数据库提供的方法集成到 `add_texts`、`similarity_search`、`from_texts` 方法下，例如自建一个基于内存+欧几里得距离的“向量数据库”，示例如下：

```
import uuid
from typing import List, Optional, Any, Iterable, Type

import dotenv
import numpy as np
from langchain_core.documents import Document
from langchain_core.embeddings import Embeddings
from langchain_core.vectorstores import VectorStore
from langchain_openai import OpenAIEmbeddings


class MemoryVectorStore(VectorStore):
    """自定义向量数据库"""
    store: dict = {}  # 在内存中开辟位置存储向量

    def __init__(self, embedding: Embeddings, **kwargs):
        self._embedding = embedding
```

```
    def add_texts(self, texts: Iterable[str], metadatas: Optional[List[dict]] =
None, **kwargs: Any) -> List[str]:
        """将数据添加到内存向量数据库中"""
        # 1.判断metadatas和texts的长度是否保持一致
        if metadatas is not None and len(metadatas) != len(texts):
            raise ValueError("元数据格式必须和文本数据保持一致")

        # 2.将文本转换为向量
        embeddings = self._embedding.embed_documents(texts)

        # 3.生成uuid
        ids = [str(uuid.uuid4()) for text in texts]

        # 4.将原始文本、向量、元数据、id构建字典并存储
        for idx, text in enumerate(texts):
            self.store[ids[idx]] = {
                "id": ids[idx],
                "vector": embeddings[idx],
                "text": text,
                "metadata": metadatas[idx] if metadatas is not None else {}
            }

        return ids

    def similarity_search(self, query: str, k: int = 4, **kwargs: Any) ->
List[Document]:
        """执行相似性搜索"""
        # 1.将query转换成向量
        embedding = self._embedding.embed_query(query)

        # 2.循环遍历记忆存储，计算欧几里得距离
        result: list = []
        for key, record in self.store.items():
            distance = self._euclidean_distance(embedding, record["vector"])
            result.append({
                "distance": distance,
                **record,
            })

        # 3.找到欧几里得距离最小的k条记录
        sorted_result = sorted(result, key=lambda x: x["distance"])
        result_k = sorted_result[:k]

        # 4.循环构建文档列表并返回
        documents = [
            Document(page_content=item["text"], metadata={**item["metadata"],
"score": item["distance"]})
            for item in result_k
        ]

        return documents

    @classmethod
    def from_texts(cls: Type["MemoryVectorStore"], texts: List[str], embedding:
Embeddings,
```

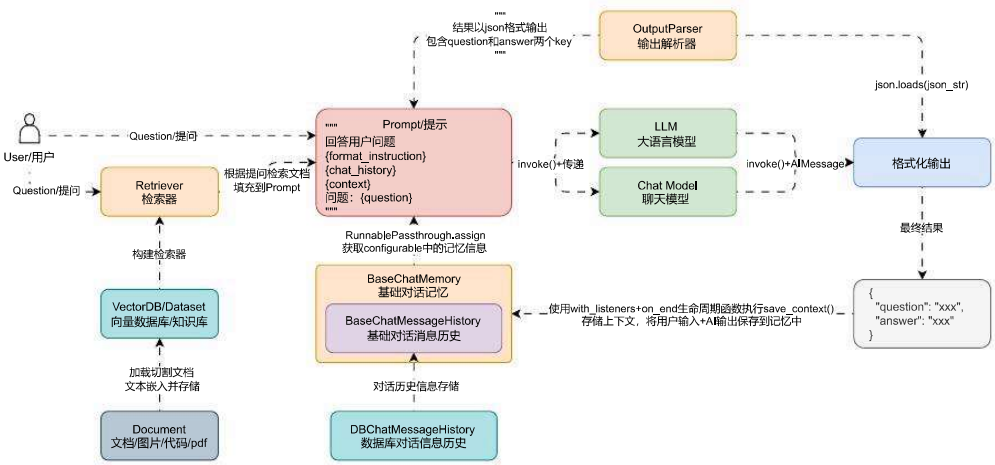
```
                    metadatas: Optional[List[dict]] = None,
                    **kwargs: Any) -> "MemoryVectorStore":
        """通过文本、嵌入模型、元数据构建向量数据库"""
        memory_vector_store = cls(embedding=embedding, **kwargs)
        memory_vector_store.add_texts(texts, metadatas)
        return memory_vector_store

    @classmethod
    def _euclidean_distance(cls, vec1, vec2) -> float:
        """计算两个向量的欧几里得距离"""
        return np.linalg.norm(np.array(vec1) - np.array(vec2))


dotenv.load_dotenv()

# 1.创建初始数据与嵌入模型
texts = [
    "笨笨是一只很喜欢睡觉的猫咪",
    "我喜欢在夜晚听音乐，这让我感到放松。",
    "猫咪在窗台上打盹，看起来非常可爱。",
    "学习新技能是每个人都应该追求的目标。",
    "我最喜欢的食物是意大利面，尤其是番茄酱的那种。",
    "昨晚我做了一个奇怪的梦，梦见自己在太空飞行。",
    "我的手机突然关机了，让我有些焦虑。",
    "阅读是我每天都会做的事情，我觉得很充实。",
    "他们一起计划了一次周末的野餐，希望天气能好。",
    "我的狗喜欢追逐球，看起来非常开心。",
]
metadatas = [
    {"page": 1},
    {"page": 2},
    {"page": 3},
    {"page": 4},
    {"page": 5},
    {"page": 6, "account_id": 1},
    {"page": 7},
    {"page": 8},
    {"page": 9},
    {"page": 10},
]
embedding = OpenAIEmbeddings(model="text-embedding-3-small")

# 2.构建自定义向量数据库
db = MemoryVectorStore.from_texts(texts, embedding, metadatas)

# 3.执行检索
print(db.similarity_search("我养了一只猫，叫笨笨"))
```

构建第一个 LangChain RAG 应用

构建第一个LangChain RAG应用

01. 外挂知识库的聊天机器人架构

在 RAG 应用中，会通过外部的检索器/知识库检索人类的提问，然后将检索到的信息填充到提示模板中，一起传递给大语言模型，让其生成特定的内容，无论 RAG 应用有多么复杂，底层一定少不了这个步骤，这也是 RAG 的基础架构。



所以在 LangChain 中，也可以按照上述的流程图，将聊天机器人添加上知识库问答功能，思路其实非常简单：

1. 和 Postgres 一样实例化一个全局的 Weaviate 向量数据库，避免每次调用时才进行连接，提升效率。
2. 在聊天应用中，将 Weaviate 转换成检索器，并将生成的 Document 列表转换成字符串。
3. 将处理好的检索器拼接到 LCEL 链输入字典中，用户提问时，检索对应内容并填充到 Prompt 模板中，从而实现知识外挂。

02. 外挂知识库的聊天机器人示例

在 LLMOps 项目中，我们对接的是 Weaviate 向量数据库，可以使用云端的向量数据库，也可以使用 Docker 搭建的向量数据库，两者并没有使用差异，修改后的代码如下。

集成的向量数据库扩展：

```
# internal/extension/vector_store_extension.py

import weaviate
from flask import Flask
from langchain_openai import OpenAIEmbeddings
from langchain_weaviate import WeaviateVectorStore

vector_store: WeaviateVectorStore = None


def init_app(app: Flask):
    """初始化向量数据库存储"""
    # 1.创建向量数据库连接客户端
    client = weaviate.connect_to_local(
```

```
        host=app.config.get("WEAVIATE_HOST"),
        port=app.config.get("WEAVIATE_PORT"),
    )

    # 2.全局复制vector_store
    global vector_store
    vector_store = WeaviateVectorStore(
        client=client,
        index_name="Dataset",
        text_key="text",
        embedding=OpenAIEmbeddings(model="text-embedding-3-small"),
    )
```

配置信息：

```
# Weaviate向量数据库配置
WEAVIATE_HOST=192.168.2.120
WEAVIATE_PORT=8080
```

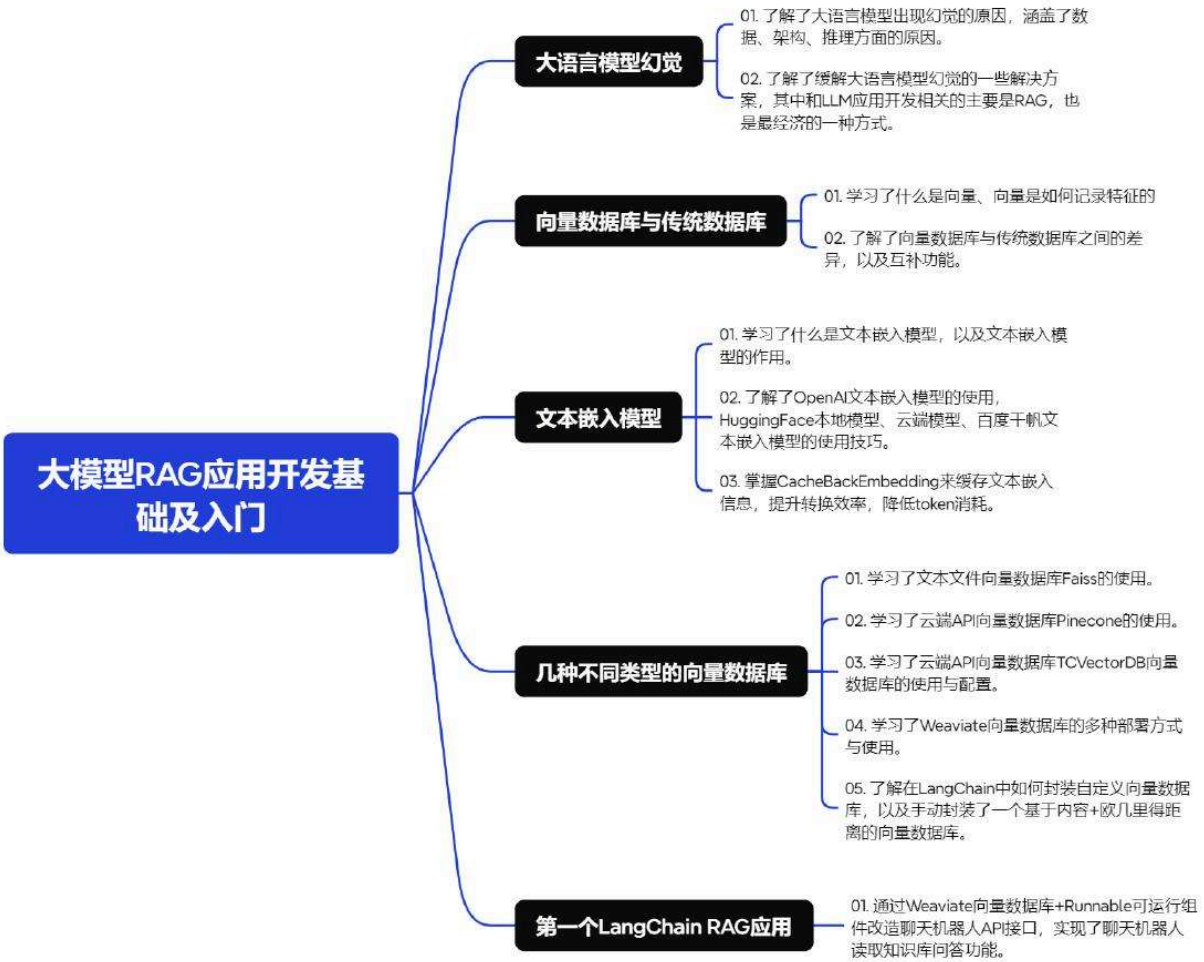
配置文件：

```
# config/config.py

class Config:
    ...
    # 向量数据库配置
    self.WEAVIATE_HOST = _get_env("WEAVIATE_HOST")
    self.WEAVIATE_PORT = int(_get_env("WEAVIATE_PORT"))
```

```
# config/default_config.py

DEFAULT_CONFIG = {
    ...
    # Weaviate向量数据库默认配置
    "WEAVIATE_HOST": "localhost",
    "WEAVIATE_PORT": 8080,
    ...
}
```

聊天机器人处理器：

```
```


大模型RAG应用开发基础及入门
大语言模型幻觉
01. 了解了大语言模型出现幻觉的原因，涵盖了数据、架构、推理方面的原因。
02. 了解了缓解大语言模型幻觉的一些解决方案，其中和LLM应用开发相关的主要是RAG，也是最经济的一种方式。
向量数据库与传统数据库
01. 学习了什么是向量、向量是如何记录特征的
02. 了解了向量数据库与传统数据库之间的差异，以及互补功能。
文本嵌入模型
01. 学习了什么是文本嵌入模型，以及文本嵌入模型的作用。
02. 了解了OpenAI文本嵌入模型的使用，HuggingFace本地模型、云端模型、百度千帆文本嵌入模型的使用技巧。
03. 掌握CacheBackEmbedding来缓存文本嵌入信息，提升转换效率，降低token消耗。
几种不同类型的向量数据库
01. 学习了文本文件向量数据库Faiss的使用。
02. 学习了云端API向量数据库Pinecone的使用。
03. 学习了云端API向量数据库TCVectorDB向量数据库的使用与配置。
04. 学习了Weaviate向量数据库的多种部署方式与使用。
05. 了解在LangChain中如何封装自定义向量数据库，以及手动封装了一个基于内容+欧几里得距离的向量数据库。
第一个LangChain RAG应用
01. 通过Weaviate向量数据库+Runnable可运行组件改造聊天机器人API接口，实现了聊天机器人读取知识库问答功能。