1. 常见向量数据库的介绍

1.0 复习知识

当计算两个向量的相似度时，常见的方法包括余弦相似度、内积和L2距离（欧几里得距离）。下面是每个方法的公式和示例计算。

1. 余弦相似度

公式:

$$\text{cosine similarity}(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|}$$

其中，$\mathbf{a} \cdot \mathbf{b}$ 是向量 $\mathbf{a}$ 和 $\mathbf{b}$ 的内积，$\|\mathbf{a}\|$ 和 $\|\mathbf{b}\|$ 是向量的欧几里得范数（L2范数）。

计算示例:

假设有两个向量:

$$\mathbf{a} = [1, 2, 3], \quad \mathbf{b} = [4, 5, 6]$$

1. 计算内积: $\mathbf{a} \cdot \mathbf{b} = 1 \times 4 + 2 \times 5 + 3 \times 6 = 32$
2. 计算范数: $\|\mathbf{a}\| = \sqrt{1^2 + 2^2 + 3^2} = \sqrt{14}$, $\|\mathbf{b}\| = \sqrt{4^2 + 5^2 + 6^2} = \sqrt{77}$
3. 计算余弦相似度: $\text{cosine similarity}(\mathbf{a}, \mathbf{b}) = \frac{32}{\sqrt{14} \times \sqrt{77}} \approx 0.974$

2. 内积（Dot Product）

公式:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i$$

即向量 $\mathbf{a}$ 和 $\mathbf{b}$ 的内积。

计算示例:

假设有两个向量:

$$\mathbf{a} = [1, 2, 3], \quad \mathbf{b} = [4, 5, 6]$$

1. 计算内积：$\mathbf{a} \cdot \mathbf{b} = 1 \times 4 + 2 \times 5 + 3 \times 6 = 32$

3. L2 距离（欧几里得距离）

公式：

$$\text{L2 distance}(\mathbf{a}, \mathbf{b}) = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2}$$

即计算向量 $\mathbf{a}$ 和 $\mathbf{b}$ 每个元素差值的平方和，再取平方根。

计算示例：

假设有两个向量：

$$\mathbf{a} = [1, 2, 3], \quad \mathbf{b} = [4, 5, 6]$$

1. 计算每个分量的差值：$(1-4)^2 = 9, (2-5)^2 = 9, (3-6)^2 = 9$
2. 计算平方和：$9 + 9 + 9 = 27$
3. 计算L2距离：$\text{L2 distance}(\mathbf{a}, \mathbf{b}) = \sqrt{27} \approx 5.196$

总结：

- **余弦相似度** 衡量两个向量的夹角相似度，值范围为 [−1, 1]，值越大表示越相似。
- **内积** 衡量两个向量的相似度，值越大表示越相似，但不考虑向量的大小。
- **L2距离** 衡量两个向量之间的“直线”距离，值越小表示越相似。

1.1 什么是向量数据库

向量数据库是一种专门用于存储和查询高维向量的数据库。随着机器学习和深度学习技术的发展，向量表示（如词向量、图像特征、用户行为向量等）变得越来越重要。这些向量通常是多维的，表示数据的特征和属性。

在这样的背景下，向量数据库提供了高效的存储、检索和相似度搜索功能。

向量数据库其实和其他数据库差不多，只不过他存的是向量。

一般使用向量数据库的流程是：创建数据库->创建集合->创建索引->插入数据->搜索。

1.2 常见的向量数据库

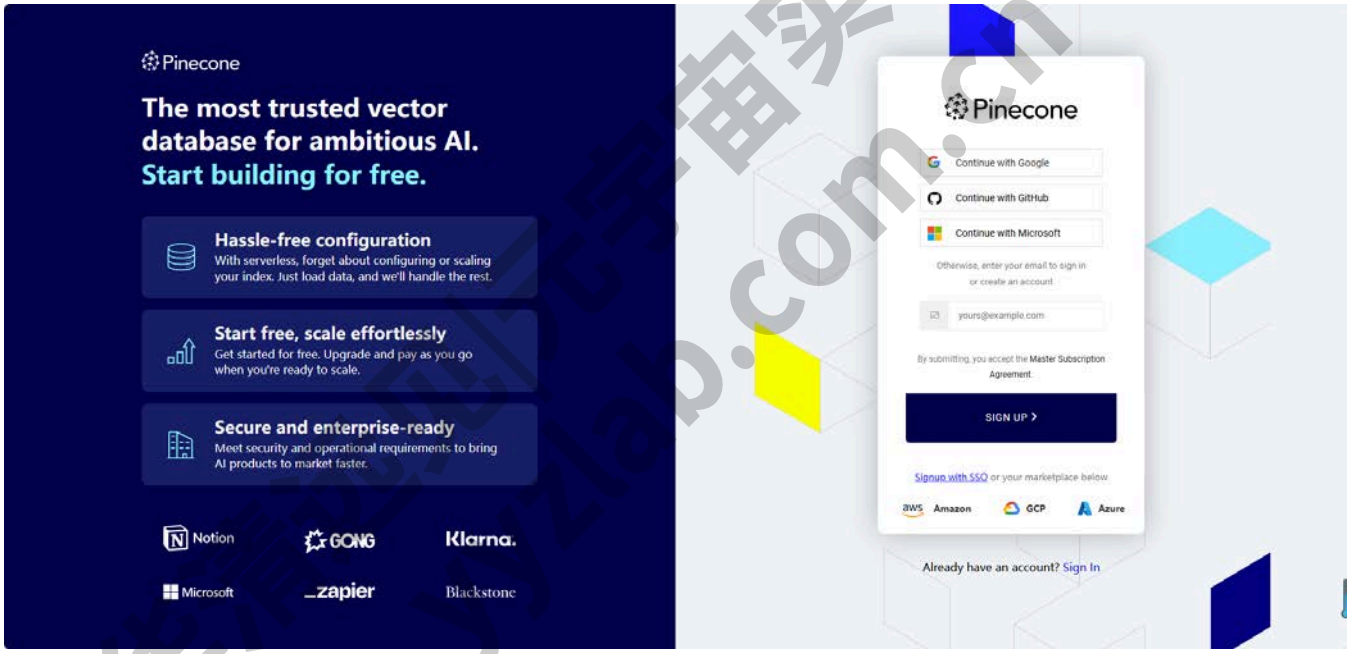
1.2.1 Pinecone

专为构建和运行向量数据库而设计，提供简单易用的API。支持高效的相似性搜索和实时数据更新。提供管理和监控功能，易于扩展。

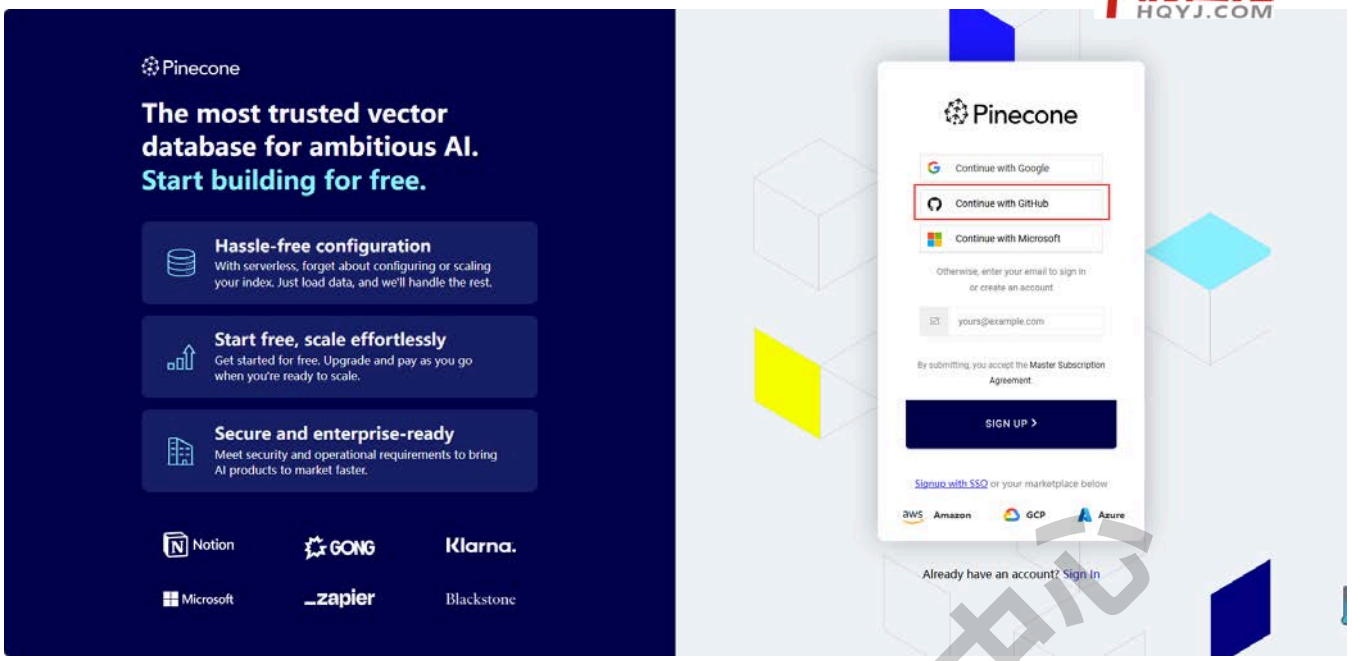
注册Pinecone获取api key

安装 Pinecone

打开 https://app.pinecone.io

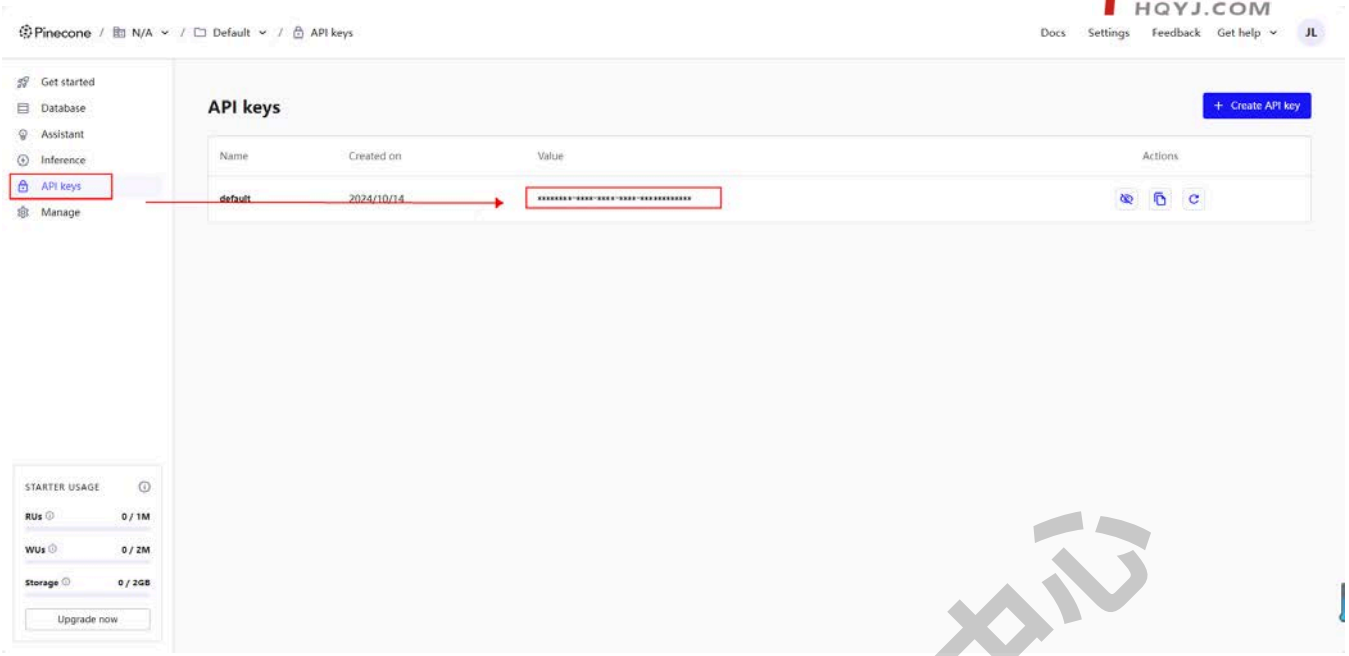


注册登录，这里直接使用github账号登录



随便填一填信息，后面一样随便填写，然后一直continue

获取api key




```
pip install pinecone==5.4.1
```

初始化客户端

```
# 初始化客户端
pc = Pinecone(api_key="4a646aa7-de8d-461e-ac3f-367a37a95bfa")
```

创建 Pinecone 服务端无服务器索引，查询时不注释会报错显示已存在。name为索引库的名字，dimension为加入索引库的向量维度，metric为相似度计算方法这里设置为余弦相似度，spec默认就行。

```
# 创建 Pinecone 服务端无服务器索引（创建时打开注释，查询时注释上。）
index_name = "example"
pc.create_index(
    name=index_name,
    dimension=1024, # Replace with your model dimensions
    metric="cosine", # Replace with your model metric
    spec=ServerlessSpec(
        cloud="aws",
        region="us-east-1"
    )
)
```

向example中加入向量。

```
# 向索引中插入向量（创建时打开注释，查询时注释上。）
index = pc.Index('example')

index.upsert(
    vectors=[
        {"id": "vec1", "values": embeddings[0]},
        {"id": "vec2", "values": embeddings[1]},
        {"id": "vec3", "values": embeddings[2]},
    ],
)
```

查询相似度，vector是要查询的向量，top_k是查询出最接近的3个，include_values为是否显示向量的值。

```
# 查询相似向量
query_results = index.query(
    vector=query_embedding[0].tolist(),
    top_k=3,
    # include_values=True
)
print(query_results)

# {'matches': [{'id': 'vec2', 'score': 0.670338273, 'values': []},
#              {'id': 'vec1', 'score': 0.451030284, 'values': []},
#              {'id': 'vec3', 'score': 0.0896421522, 'values': []}],
#  'namespace': '',
#  'usage': {'read_units': 5}}
```


如果不需要example索引则删除

```
pc.delete_index(index_name)
```

结果解释：

当然，以下是详细的解释以及带有注释的代码：

查询结果解释

查询结果是一个字典，包含以下键：

- `matches`：匹配的结果列表。
  - 每个匹配项是一个字典，包含以下键：
    - `id`：向量的唯一标识符。
    - `score`：相似度分数（余弦相似度）。
    - `values`：向量的实际值（在本例中为空列表，因为设置了 `include_values=False`）。
- `namespace`：命名空间（默认为空字符串）。
- `usage`：使用情况统计信息。
  - `read_units`：读取单元的数量。

具体到你的查询结果：

```
{
    'matches': [
        {'id': 'vec2', 'score': 0.670338333, 'values': []},
        {'id': 'vec1', 'score': 0.451030374, 'values': []},
        {'id': 'vec3', 'score': 0.0896422863, 'values': []}
    ],
    'namespace': '',
    'usage': {'read_units': 5}
}
```

- **matches**：
  - 第一个匹配项：`{'id': 'vec2', 'score': 0.670338333, 'values': []}`
    - `id`：`'vec2'` 对应的是“厨房里有一只黑色的猫”。
    - `score`：`0.670338333` 表示与查询“黑色的猫在哪？”的相似度最高。
  - 第二个匹配项：`{'id': 'vec1', 'score': 0.451030374, 'values': []}`
    - `id`：`'vec1'` 对应的是“院子里有一只可爱的小猫”。

    - `score`: `0.451030374` 表示与查询的相似度次之。
  - 第三个匹配项：`{'id': 'vec3', 'score': 0.0896422863, 'values': []}`
    - `id`: `'vec3'` 对应的是“大模型真简单”。
    - `score`: `0.0896422863` 表示与查询的相似度最低。
- **namespace**: 空字符串，表示没有使用命名空间。
- **usage**:
  - `read_units`: `5` 表示本次查询消耗了5个读取单元。

1.2.2 FAISS (Facebook AI Similarity Search)

一个开源库，专门用于高效的相似性搜索。

支持多种索引结构，适合处理非常大的数据集。具有强大的性能优化，能够在CPU和GPU上运行。

安装FAISS

```
pip install faiss-cpu==1.9.0
```

创建索引

创建一个 Faiss 索引，并将句子的嵌入向量添加进去，通过faiss.write_index可以将向量持久化到本地上。

持久化是指将数据保存到非易失性存储介质上的过程，以便在程序关闭后仍然能够保留这些数据。

这样在运行代码后，在你指定的位置会新建一个faiss_index.index文件。

```
# 向量数据库
import faiss
import numpy as np

# 假设我们已经有句子的向量
# embeddings 是一个 2D 数组，形如 (3, 1024) 表示 3 个句子的 1024 维向量
embeddings = np.array(embeddings)

```

```python
# 保存索引到磁盘文件
faiss.write_index(index, "db/faiss_index.index")

# 初始化 Faiss 索引
index = faiss.IndexFlatIP(1024)  # 使用内积进行检索（标准化后的内积就是余弦相似度）
# index = faiss.IndexFlatL2(1024)  # 使用L2距离进行检索

index.add(embeddings)  # 添加向量到索引
```

`标准化后的内积就是余弦相似度` 如何理解：

`faiss.IndexFlatIP` 使用内积进行相似度计算，其背后的原因是，内积在某些情况下可以与余弦相似度等价。如果你将向量进行标准化，即每个向量除以其欧几里得范数，那么内积结果就相当于计算了它们之间的余弦相似度。也就是说，标准化后的内积和余弦相似度是相等的。

标准化后的内积等于余弦相似度

假设我们有两个向量 $\mathbf{a}$ 和 $\mathbf{b}$，并且对它们进行了标准化：

$$\hat{\mathbf{a}} = \frac{\mathbf{a}}{\|\mathbf{a}\|}, \quad \hat{\mathbf{b}} = \frac{\mathbf{b}}{\|\mathbf{b}\|}$$

那么它们之间的内积就是：

$$\hat{\mathbf{a}} \cdot \hat{\mathbf{b}} = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|}$$

这个结果正是余弦相似度的公式。因此，如果在实际应用中，向量已经被标准化，使用内积进行检索实际上就是在计算余弦相似度。

查询并检索

```
# 假设我们想查询 "黑色的猫在哪？"
query_embedding = model.encode(["黑色的猫在哪？"])

# 进行检索，返回最相似的 2 个结果
D, I = index.search(query_embedding, k=3)

# 输出结果
print(f"最相似的句子索引: {I}")
print(f"对应的距离: {D}")

# 最相似的句子索引: [[1 0 2]]
# 对应的距离: [[0.6703382  0.45103022 0.08964214]]
```

1.2.3 Chroma

提供易于使用的API，特别适合构建和使用向量数据库。支持多种向量存储后端，具有良好的扩展性。适用于文本、图像等多种类型的数据。

安装Chroma

```
pip install chromadb==0.5.20
```

创建 Chroma 客户端

创建一个 Chroma 客户端实例，并选择一个持久化的存储路径（可选）。如果没有指定路径，默认情况下 Chroma 将在内存中运行。
这样在运行代码后，在你指定的位置会新建一个chroma.sqlite3文件。

```
from chromadb import Client, PersistentClient
from chromadb.config import Settings

# 初始化 Chroma 客户端
# client = Client()
# 持久化
client = PersistentClient(path="db")
```

创建或连接到一个集合

metadata={"hnsw:space": "cosine"}表示使用余弦相似度，还可以使用"L2"距离、"ip"内积，注意在Chroma中他们的公式如下：

Distance	parameter	Equation
Squared L2	`l2`	$d = \sum (A_i - B_i)^2$
Inner product	`ip`	$d = 1.0 - \sum (A_i \times B_i)$
Cosine similarity	`cosine`	$d = 1.0 - \frac{\sum (A_i \times B_i)}{\sqrt{\sum (A_i^2)} \cdot \sqrt{\sum (B_i^2)}}$

因此默认返回的是一个cosine，返回值越大两个向量越接近，对于余弦相似度计算，使用1-d可以还原我们更直观的相似度。

```
# 创建集合
collection = client.create_collection("example_collection", metadata={"hnsw:space": "cosine"},)
```

插入数据

```
# 插入向量
collection.add(
    embeddings=embeddings,
    ids=["vec0", "vec1", "vec2"],
)
```

查询并检索

```
# 查询
results = collection.query(
    query_embeddings=query_embedding,
    n_results=3
)

# 输出结果
print(f"最相似的句子索引: {results['ids']}")
print(f"对应的距离: {[1-d for d in results['distances'][0]]}")

# 最相似的句子索引: [['vec1', 'vec0', 'vec2']]
# 对应的相似度: [0.6703380942344666, 0.45103007555007935,
0.08964204788208008]
```

1.2.4 Milvus

Milvus 是一款云原生向量数据库，它具备高可用、高性能、易拓展的特点，用于海量向量数据的实时召回。

安装 Milvus

设置工作目录和配置文件，在创建启动容器之前，我们要先设置好工作目录和配置文件。

新建 milvus 后在milvus文件夹下新建以下文件夹。

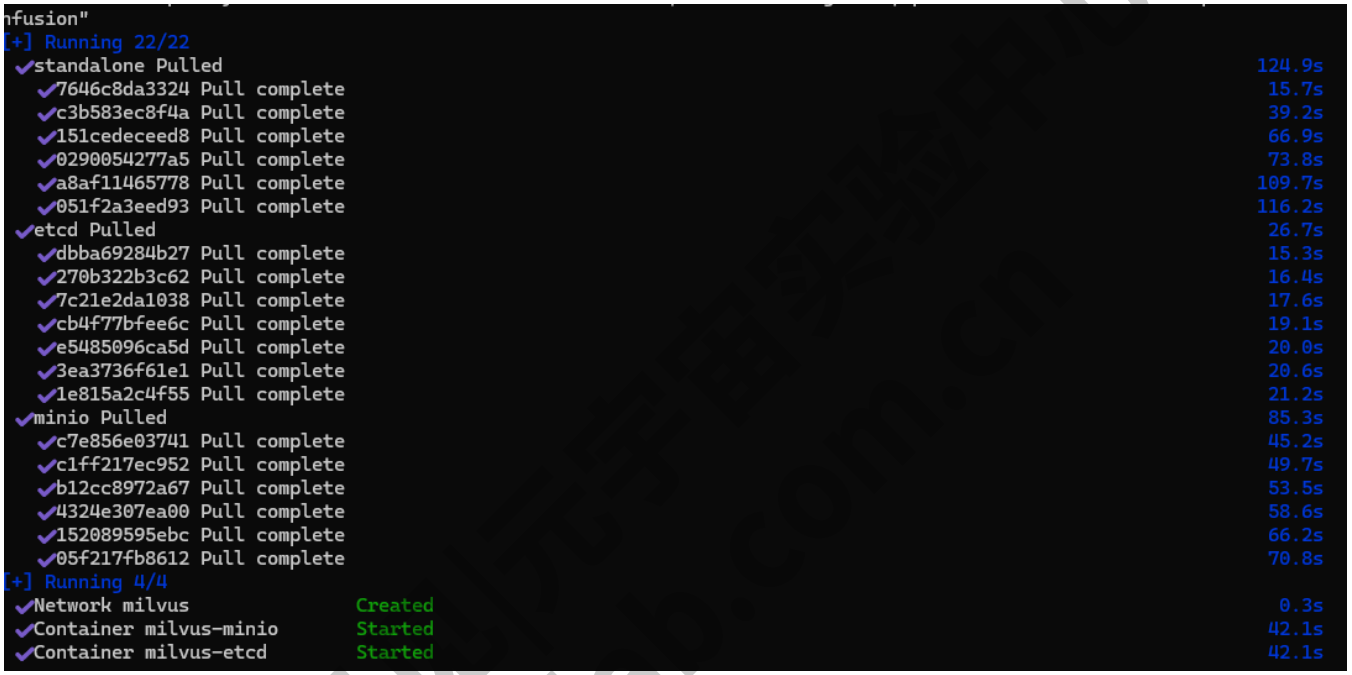
名称	修改日期	类型	大小
conf	2024/10/15 11:58	文件夹	
db	2024/10/15 11:58	文件夹	
logs	2024/10/15 11:58	文件夹	
pic	2024/10/15 11:59	文件夹	
volumes	2024/10/15 13:14	文件夹	
wal	2024/10/15 11:59	文件夹	
docker-compose.yml	2024/10/15 11:46	YML 文件	2 KB

下载 docker-compose.yml ， v2.4.13是版本。将 docker-compose.yml 复制到刚刚新建的 milvus 文件夹中。

```
wget https://github.com/milvus-io/milvus/releases/download/v2.4.13/milvus-standalone-docker-compose.yml -O docker-compose.yml
```

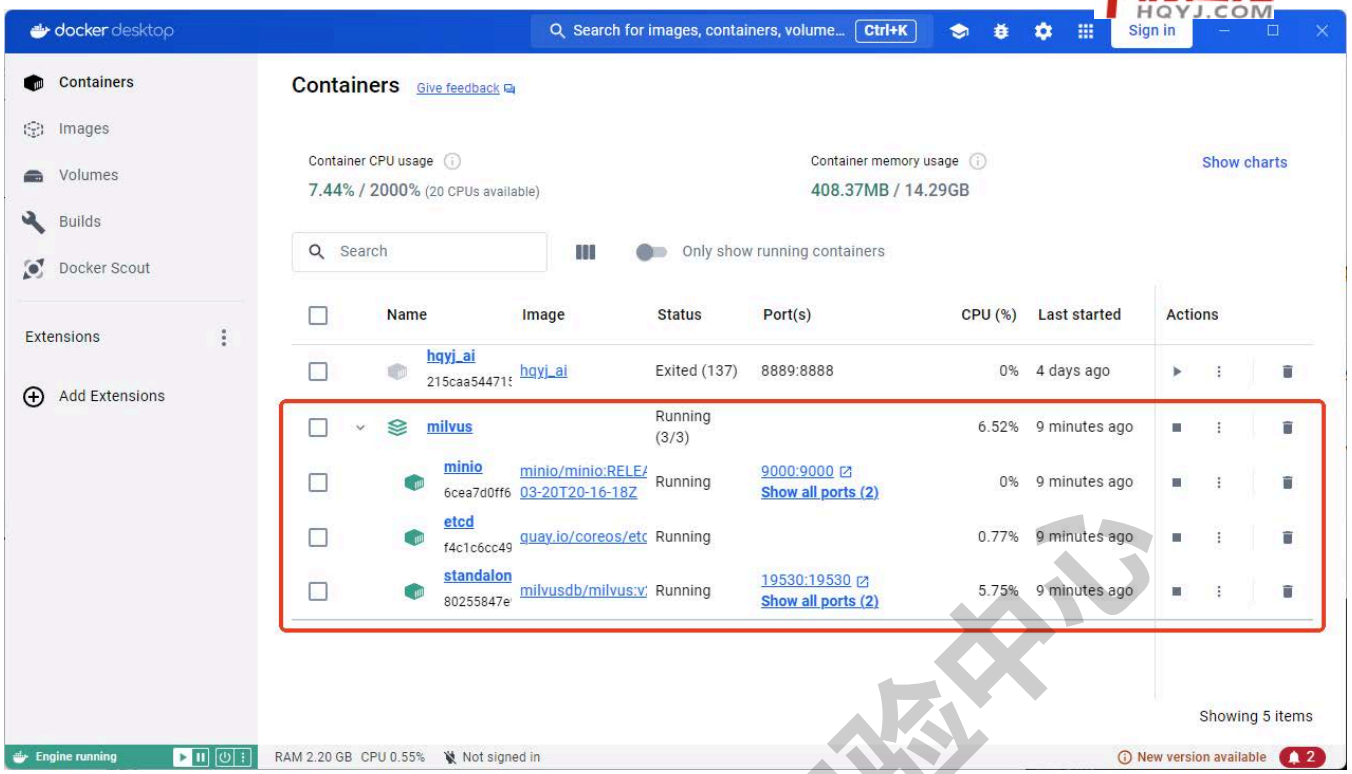
使用 Docker 部署 Milvus，在 milvus 文件夹中打开终端，运行以下命令

```
docker-compose up -d
```



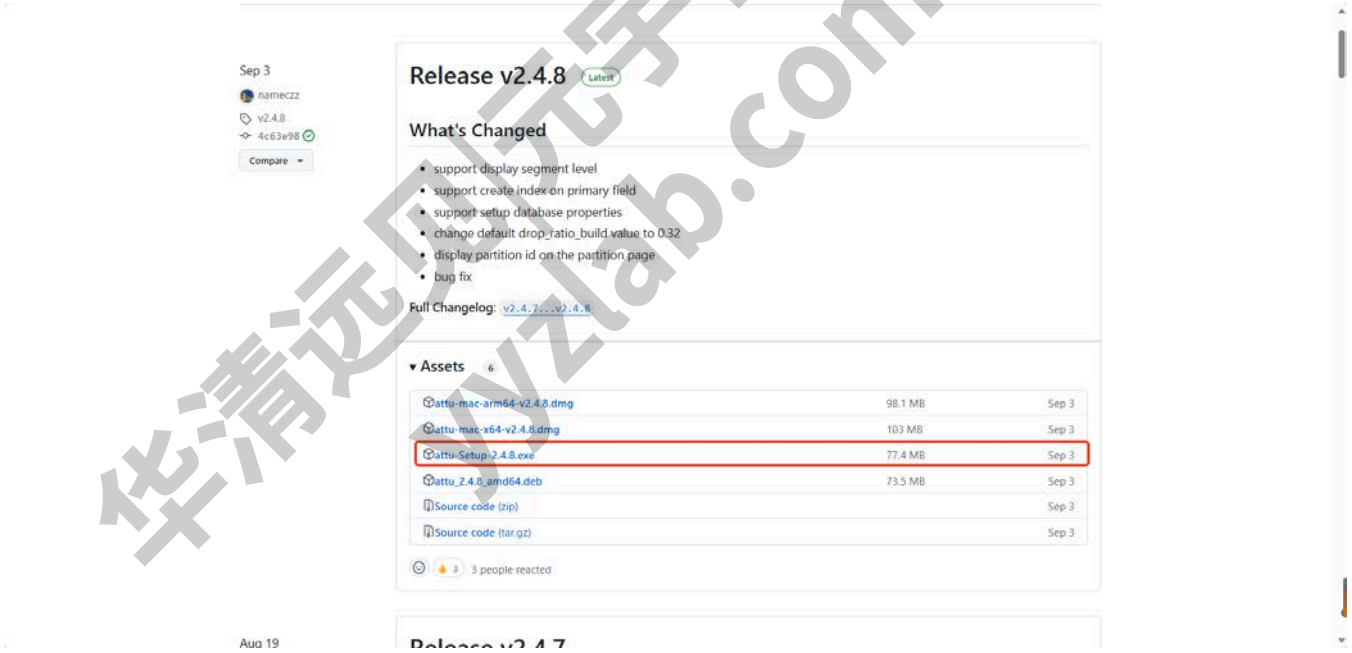
设置端口 `docker port milvus-standalone 19530/tcp`。

从下面信息可以看出，这个Docker环境中有四个正在运行的容器，分别与Milvus数据库服务、Etcd键值存储服务、MinIO对象存储服务相关联。“standalone”指的是一个独立运行的Milvus实例。

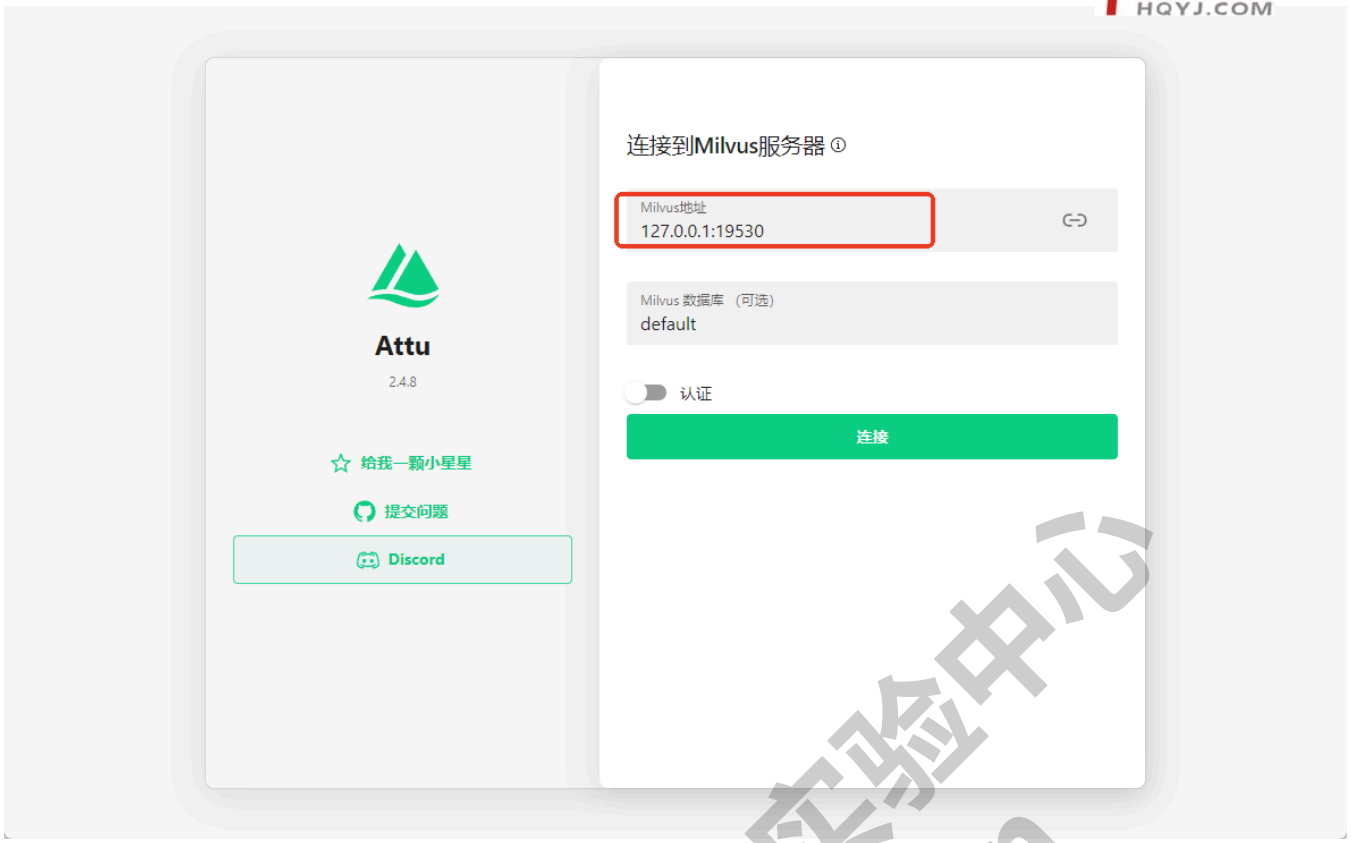


安装可视化界面（可选）

进入官网 https://github.com/zilliztech/attu/releases 下载安装。



安装docker的端口填写



连接进入

通过python使用Milvus

安装

```
pip install pymilvus==2.5.0
```

创建数据库

通过指定的主机地址 (host) 和端口号 (port) 建立与 Milvus 服务的连接。sample_db 为自定义的数据库名字。

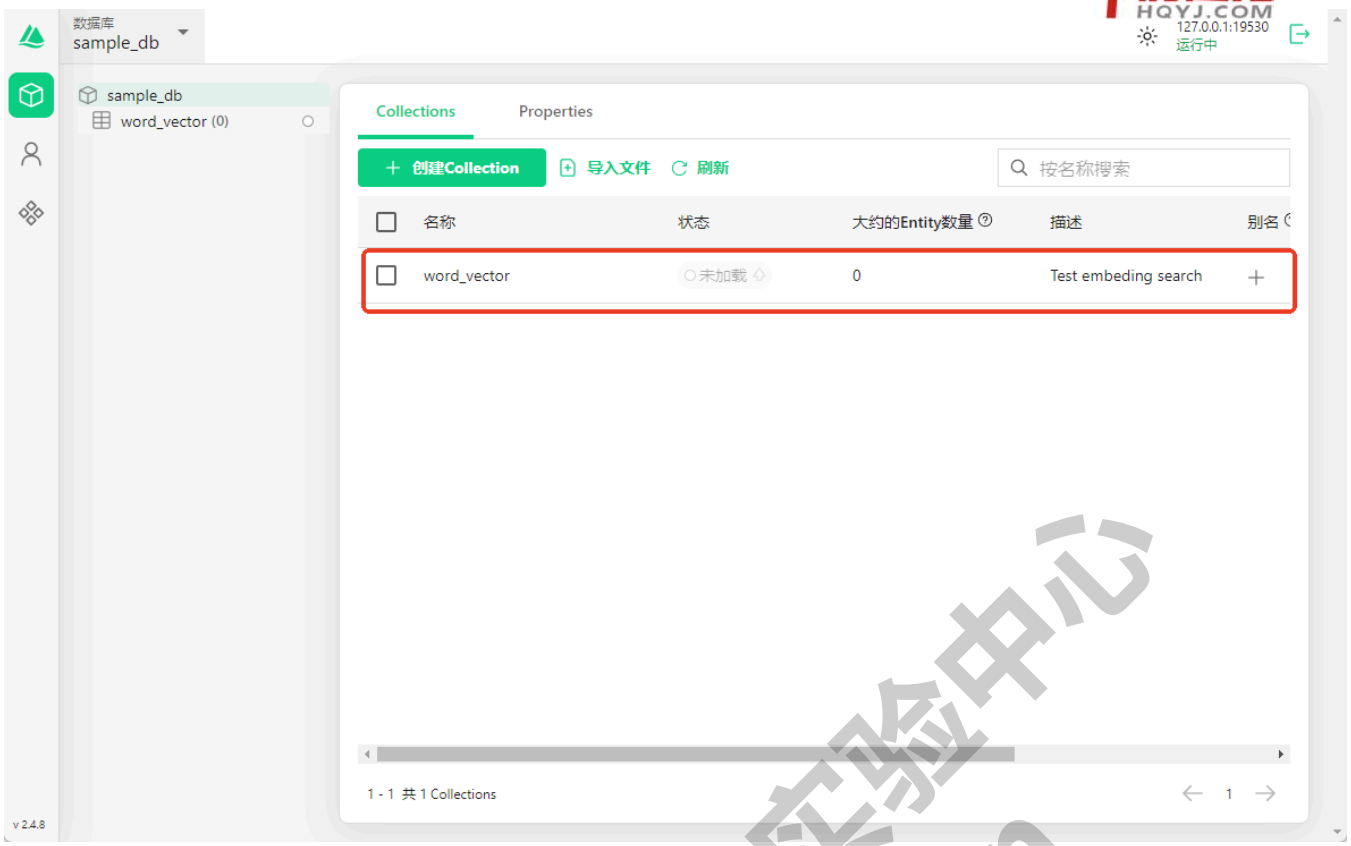
```
from pymilvus import connections, db

conn = connections.connect(host="127.0.0.1", port=19530)
database = db.create_database("sample_db")
```

创建collection（集合）

建立连接，创建两个字段id和embedding，name设置名字，dtype设置类型，is_primary表示设置为主键（和SQL类似），auto_id表示自动分配id。dim表示存入的向量维度。

```
# 创建 Collection
# 导入必要的类和模块
```

```python
from pymilvus import CollectionSchema, FieldSchema, DataType
from pymilvus import Collection, db, connections

# 连接到 Milvus 服务
conn = connections.connect(host="127.0.0.1", port=19530)
# 使用名为 sample_db 的数据库
db.using_database("sample_db")

# 定义主键字段 id，类型为 INT64，并设置为主键（is_primary=True），同时开启自动增长（auto_id=True）
id = FieldSchema(name="id", dtype=DataType.INT64, is_primary=True, auto_id=True)

# 定义向量字段 embedding，类型为 FLOAT_VECTOR，维度为 1024
embedding = FieldSchema(name="embedding", dtype=DataType.FLOAT_VECTOR, dim=1024)

# 创建集合模式（CollectionSchema），包含 id 和 embedding 字段，并提供描述信息
schema = CollectionSchema(
    fields=[id, embedding],
    description="Test embedding search",
)

# 定义集合名称为 word_vector，并基于之前定义的模式创建集合
collection_name = "word_vector"
collection = Collection(name=collection_name, schema=schema)
```



创建索引

创建索引时还可以设定以下参数用于设定索引方式：

- FLAT：准确率高， 适合数据量小，暴力求解相似。
- IVF-FLAT：量化操作， 准确率和速度的平衡
- IVF: inverted file 先对空间的点进行聚类，查询时先比较聚类中心距离，再找到最近的N个点。
- IVF-SQ8：量化操作，disk cpu GPU 友好
- SQ8：对向量做标量量化，浮点数表示转为int型表示，4字节->1字节。
- IVF-PQ：快速，但是准确率降低，把向量切分成m段，对每段进行聚类；查询时，查询向量分端后与聚类中心计算距离，各段相加后即为最终距离。使用对称距离(聚类中心之前的距离)不需要计算直接查表，但是误差回更大一些。
- HNSW：基于图的索引，高效搜索场景，构建多层的NSW。
- ANNOY：基于树的索引，高召回率

metric_type 包括 "IP" 内积、"COSINE" 余弦相似度、"L2" L2距离

```
# 创建索引
# 导入必要的类和模块
from pymilvus import Collection, utility, connections, db
```

```python

# 连接到 Milvus 服务
conn = connections.connect(host="127.0.0.1", port=19530)

# 使用名为 sample_db 的数据库
db.using_database("sample_db")

# 定义索引参数，这里指定度量类型为余弦相似度（COSINE）
index_params = {
    "metric_type": "COSINE",
}

# 获取名为 "word_vector" 的集合
collection = Collection("word_vector")

# 在名为 "embedding" 的字段上创建索引，索引参数为之前定义的 index_params
collection.create_index(
    field_name="embedding",
    index_params=index_params
)

# 获取 "word_vector" 集合索引构建的进度
utility.index_building_progress("word_vector")
```

插入数据

将embedding后的数据存入向量数据库。

```
# 插入数据
# 导入必要的类和模块
from pymilvus import Collection, db, connections

# 连接到 Milvus 服务
conn = connections.connect(host="127.0.0.1", port=19530)

# 使用名为 sample_db 的数据库
db.using_database("sample_db")

# 指定要操作的集合名称
coll_name = 'word_vector'

# 获取名为 coll_name 的集合
collection = Collection(coll_name)

# 插入数据，假设 embeddings 是一个已经准备好的向量列表或数组
mr = collection.insert([embeddings])

# 打印插入操作的结果
print(mr)
```

查询并检索

连接Milvus 服务将之前创建的集合加载到内存中，设定搜索参数执行搜索。

```
# 向量搜索
# 导入必要的类和模块
from pymilvus import Collection, db, connections
from pymilvus.orm import utility

# 连接到 Milvus 服务
conn = connections.connect(host="127.0.0.1", port=19530)

# 使用名为 sample_db 的数据库
db.using_database("sample_db")

```

```
# 指定要操作的集合名称
coll_name = 'word_vector'

# 获取名为 coll_name 的集合
collection = Collection(coll_name)

# 将集合加载到内存中，以便进行搜索操作
collection.load()

# 定义搜索参数，这里指定度量类型为余弦相似度（COSINE）
search_params = {"metric_type": "COSINE"}

# 假设 query_embedding 已经是一个包含查询向量的列表或数组
# 执行搜索操作
results = collection.search(
    data=query_embedding.tolist(),  # 将查询向量转换为列表形式
    anns_field="embedding",         # 指定要搜索的向量字段名
    param=search_params,            # 搜索参数
    limit=5,                        # 返回最多的结果数量
    expr=None                       # 可选的过滤表达式，默认为 None 表示不过滤
)

# 输出搜索结果
for result in results:
    print(result)

# ['id: 453238947178891095, distance: 0.6703383326530457, entity: {}',
# 'id: 453238947178891094, distance: 0.45103031396865845, entity: {}',
# 'id: 453238947178891096, distance: 0.08964216709136963, entity: {}']

```