向量数据库

快速入门

腾讯云

文档目录

快速入门
购买免费测试版实例

最近更新时间：2024-05-13 16:10:21

操作场景

您可根据本文的介绍，购买和配置您的第一台免费体验版向量数据库（Tencent Cloud VectorDB）实例。

> **说明：**
> 免费测试版实例，为单可用区，1节点部署，仅用于快速体验或测试数据库的相似性检索能力。

地域

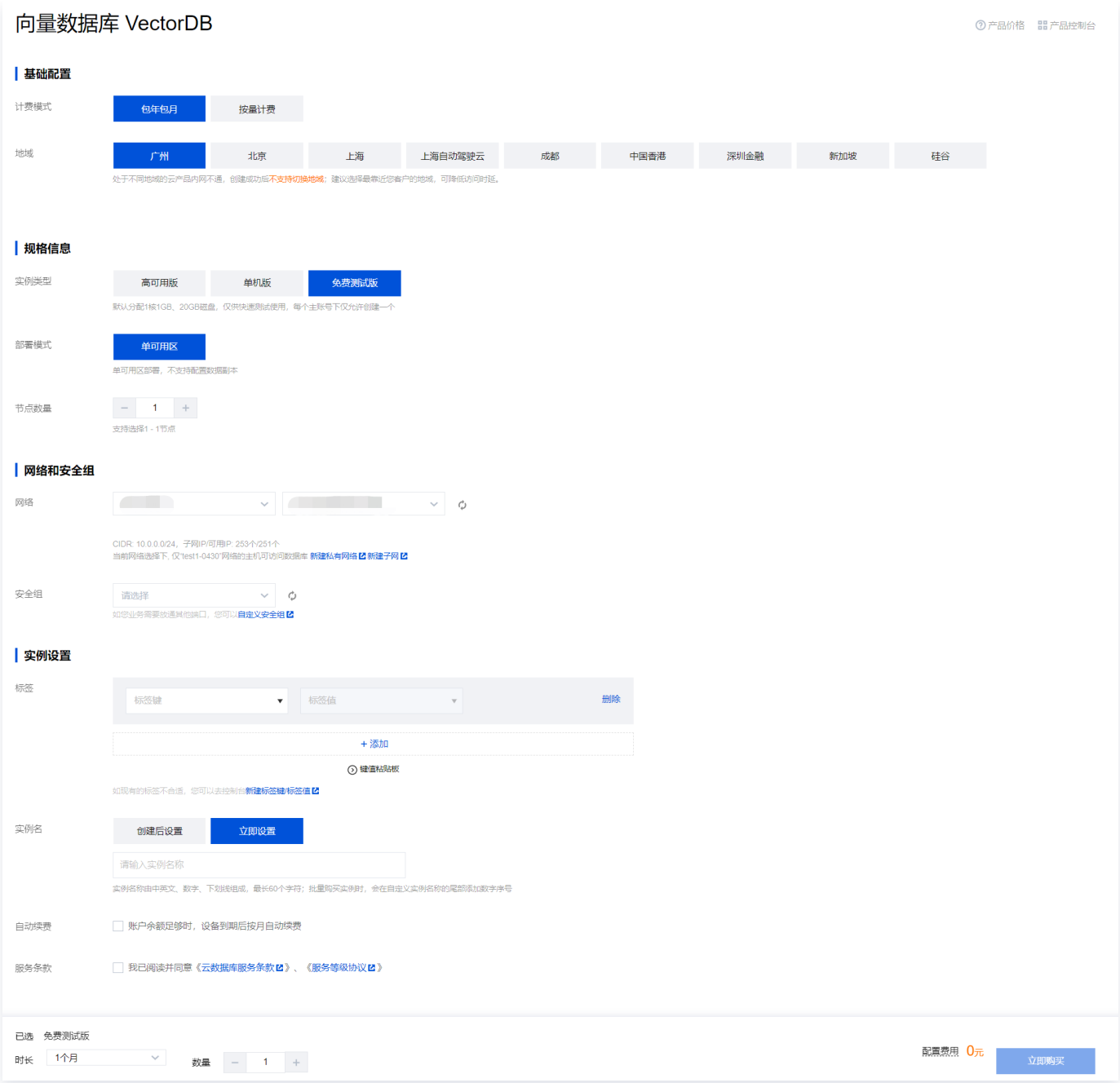
当前支持北京、上海、广州、成都、上海自动驾驶云、深圳金融、中国香港、新加坡、硅谷，其他地域在规划准备中。

前提条件

- 已注册腾讯云账号并完成实名认证。
  - 如需注册腾讯云账号：请单击 注册腾讯云账号 。
  - 如需完成实名认证：请单击 实名认证 。
- 已创建数据库实例的私有网络与安全组，请参见 私有网络 与 安全组 。

操作步骤

1. 使用腾讯云账号登录 向量数据库 VectorDB 购买页。
2. 请参见下表，配置如下相关参数，购买**免费测试版**实例，如下图所示。



分类	界面参数	配置说明
基础配置	计费模式	选择**包年包月**，免费测试版实例仅支持包年包月计费模式。
	地域	处于不同地域的云产品内网不通，新建后不能切换地域，请选择最靠近业务的地域。
规格信息	实例类型	请选择**免费测试版**。
	部署模式	**实例类型**为**免费测试版**与**单机版**，仅支持**单可用区**部署。
	节点数量	**实例类型**为**免费测试版**，取值只能为：1。
网络和	网络	• 在下拉列表分别选择已配置的私有网络及子网。

安全组		• 若现有网络不满足需求，请单击**新建私有网络**或**新建子网**，创建所需的网络环境。具体操作，请参见 创建私有网络 。
	安全组	您可以在**选择已有安全组**下拉框中选择已有的安全组，也可以单击**自定义安全组**，设置新的安全组入站规则。具体操作，请参见 安全组 。
实例设置	实例名	请在输入框输入实例名称。仅支持长度不超过 60 的中文/英文/数字/-/_。
	自动续费	体验免费测试版，可不设置自动续费。购买之后，若需要继续使用，可开通自动续费。
	服务条款	勾选**我已阅读并同意云数据库服务条款**。
	时长	购买实例使用的时长，免费测试版仅能试用1个月。
	数量	购买实例数量。一个账号仅能购买一个免费测试版的实例。

3. 单击右下角的**立即购买**，自动返回实例列表页面，当前实例**状态**为**创建中**，等待实例状态更新为**运行中**即可使用免费版实例。

下一步

待购买的免费测试版实例运行正常，便可以进行 连接前准备事项：配置网络环境、安装 SDK 等。

连接前先决条件

最近更新时间：2024-05-13 16:13:41

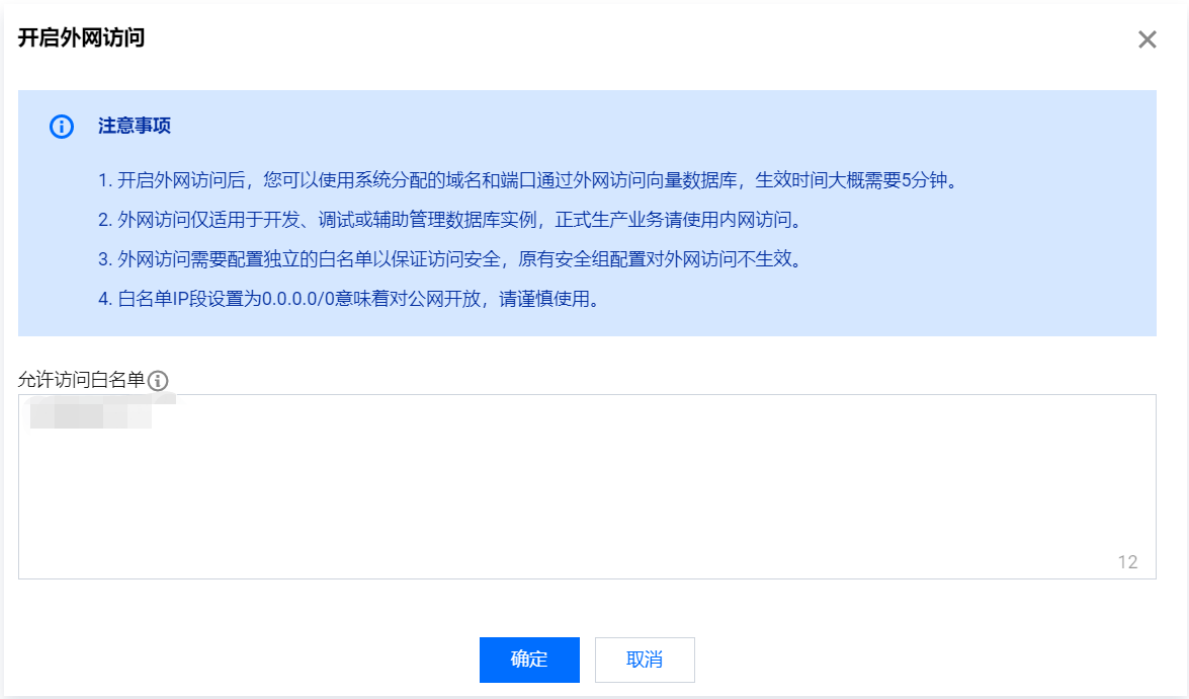
开始连接操作向量数据库之前，您需要确保已准备好必要的网络环境，并已获取连接账户与访问密钥。此外，客户端还需要安装适用的 SDK。

配置网络环境

开启外网功能，系统自动分配域名与端口，在本地 Linux 系统的客户端便可通过外网快速连接数据库。

外网方式（推荐）

1. 登录 向量数据库控制台 。
2. 在实例列表中，找到需开启外网访问的实例。
3. 单击**实例 ID** 进入**实例详情**页面，在**网络信息**区域，单击**外网地址**后面的**开启**。
4. 在**开启外网访问**的小窗口，在**允许访问白名单**的输入框，配置外网访问的白名单列表。输入格式支持如下形式：
   - IP 地址：如192.168.1.0。
   - CIDR 段：如192.168.1.0/24。
   - 多个 IP 或 CIDR 段：如：192.168.1.0,192.168.1.21。多个 IP 或 CIDR 段之间使用英文逗号分隔。最多支持填写10个 IP 或 CIDR 段。



5. 单击**确定**，等待任务执行完成，在**网络信息**区域的**外网地址**，可查看系统分配的外网地址。

内网方式

- 申请与腾讯云向量数据库在同一地域同一个 VPC 内的 Linux 云服务器 CVM，通过内网快速连接向量数据库。
- 在腾讯云 CVM 安全组中需配置**出站规则**，把腾讯云向量数据库的 IP 及端口添加到出站规则中。在腾讯云向量数据库安全组中配置**入站规则**，把 CVM 的 IP 地址及向量数据库的端口添加到入站规则中，才能连接成功。 操作详情，可参见 安全组 。

获取登录密钥

1. 登录 向量数据库控制台 。
2. 在实例列表中，找到目标实例。
3. 单击目标**实例 ID**，或在其**操作**列，单击**管理**，进入**实例详情**页面。
4. 在**实例详情**页面，选择**密钥管理**。
5. 在**密钥管理**页面，可查看到访问数据库的账户及其对应的 **API 密钥**。单击**复制**，可直接复制 **root** 账户默认的密钥。

安装 SDK

腾讯云向量数据库（Tencent Cloud VectorDB）提供了 Python、Java、Go 语言的 SDK， 辅助您快速访问数据库。

- Python 3.6 及以上版本。
- Java 8 或更高版本。
- Go 1.15 或更高版本。

Python

```
pip install tcvectordb
```

说明：
使用 pip 安装 Python SDK 之前，请确保 pip 指向3.6及以上的 Python 版本，否则选用相应的 pip3 安装。

Java

Gradle 项目，请在 build.gradle 文件中添加如下依赖。

```
com.tencent.tcvectordb:vectordatabase-sdk-java:1.1.2
```

Maven 项目，请在 pom.xml 文件中添加如下依赖。

```
<dependency>
    <groupId>com.tencent.tcvectordb</groupId>
    <artifactId>vectordatabase-sdk-java</artifactId>
    <version>1.1.2</version>
</dependency>
```

Go

```
go get -u github.com/tencent/vectordatabase-sdk-go/tcvectordb
```

下一步

准备好以上事项，便可以开始您的 第一次相似性检索 体验。

第一次相似性检索
选择相似性检索场景

最近更新时间：2024-01-10 16:20:52

腾讯云向量数据库（Tencent Cloud VectorDB）提供了三种常见场景管理数据库进行相似性检索的教程，辅助您快速体验产品能力。

相似性检索	场景描述
直接写入向量数据并检索	适用于已有自身的向量数据，无需使用腾讯云向量数据库进行数据向量化的场景。
基于文本信息写入向量数据	适用于无向量数据，需要应用腾讯云向量数据库的 Embedding 功能输入原始文本进行数据向量化的场景。
基于文件写入向量数据	适用于无向量数据，需要应用 AI 套件上传 md 文件进行数据向量化的场景。

写入向量数据并检索

最近更新时间：2024-01-25 14:18:01

本章节介绍如何连接向量数据库写入数据，并进行第一次相似性检索。以 Linux 操作系统为例，使用 Python、Java、Go SDK 与 HTTP API 示例代码分别进行演示。运行本章节所提供的示例代码，您将初步了解向量数据库的检索能力。

导入 SDK 依赖模块

Python

```
import tcvectordb
from tcvectordb.model.enum import FieldType, IndexType, MetricType,
ReadConsistency
from tcvectordb.model.index import Index, VectorIndex, FilterIndex,
HNSWParams
from tcvectordb.model.collection import UpdateQuery
from tcvectordb.model.document import Document, SearchParams, Filter
```

Java

```
import com.tencentcloudapi.client.VectorDBClient;
import com.tencentcloudapi.model.*;
```

Go

```
package main

import (
    "context"
    "log"
    "time"
```

```
    "github.com/tencent/vectordatabase-sdk-go/tcvectordb"
)
```

创建 Client

导入 SDK 所需的模块之后，需先创建一个向量数据库的客户端对象，与向量数据库服务器连接才能进行交互。

> **说明：**
> 如下示例 **url** 与 **key** 需要分别替换为已购买的免费版实例的**外网访问地址**与 API Key。请登录 **向量数据库控制台**，在**实例详情**页面**网络信息**区域直接复制**外网地址**，在**密钥管理**页面直接复制密钥。

Python

```
client = tcvectordb.VectorDBClient(url='http://10.0.X.X', username='root',
key='eC4bLRy2va******************************',
read_consistency=ReadConsistency.EVENTUAL_CONSISTENCY, timeout=30)
```

Java

```
public class VectorDBExample {
    public static void main(String[] args) {
        // 创建VectorDB Client
        ConnectParam connectParam = ConnectParam.newBuilder()
                .withUrl("http://10.0.X.X:80")
                .withUsername("root")
                .withKey("eC4bLRy2va********************************")
                .withTimeout(30)
                .build();
        VectorDBClient client = new VectorDBClient(connectParam,
ReadConsistencyEnum.EVENTUAL_CONSISTENCY);
    }
}
```

Go

```
func main() {
    var defaultOption = &tcvectordb.ClientOption{
        Timeout:           time.Second * 5,
        MaxIdldConnPerHost: 2,
        IdleConnTimeout:    time.Minute,
        ReadConsistency:    tcvectordb.EventualConsistency,
    }
    client, err := tcvectordb.NewClient("http://10.0.X.X:80", "root",
"eC4bLRy2va**********************************", defaultOption)
    if err != nil {
        panic(err)
    }
}
```

创建数据库

基于创建的客户端对象，创建数据库。

Python

Python SDK 通过 create_database() 接口创建数据库 **db-test**。

```
db = client.create_database(database_name='db-test')
```

Java

Java SDK 通过 createDatabase() 创建数据库 **db-test**。

```
Database db = client.createDatabase("db-test");
```

Go

Go SDK 通过 CreateDatabase() 创建数据库 db-test。

```
var (
    ctx             = context.Background()
    database          = "db-test"
)
db, _ := client.CreateDatabase(ctx, database)
```

Curl

HTTP 使用 /database/create 接口创建数据库 db_test。如下示例，url 地址与 api_key 需要分别替换为已购买的免费版实例的外网地址与API Key。

```
curl -i -X POST \
 -H 'Content-Type: application/json' \
 -H 'Authorization: Bearer
account=root&api_key=A5VOgsMpGWJhUI0WmUbY********************' \
http://10.0.X.X:80/database/create \
 -d '{
 "database": "db-test"
 }'
```

创建集合

1. 创建集合之前，需先设计索引结构，指定索引字段。如下示例，创建一个可写入 3 维向量数据存储书籍的集合 book-vector，其书籍信息字段包括： id、vector、bookName，分别对各字段构建索引。有关索引的具体信息，请参见 Index 。
   - 主键索引（Primary Key Index）：固定且必须，对应字段 id，每条数据的唯一标识。并对主键 id 构建 Filter 索引，以便可通过 id 的条件表达式进行特定行的检索。
   - 向量索引（Vector Index）：固定且必须，对应字段 vector，对向量数据构建索引，指定向量数据的维度、数据存储的索引类型、相似性计算方法及相关索引参数。
   - Filter 索引（Filter Index）：需根据检索需求选取可作为条件查询过滤数据的字段。通常，向量数据对应的文本字段，不具有过滤属性，则无需对该字段建立索引，否则，将浪费较大的内存也无实际意义。如下示例，预按书籍的名称过滤数据，对 bookName 字段建立 Filter 索引。
2. 创建集合，免费测试版实例，其分片 shard 只能为 1，副本 replicas 仅能为 0。

Python

Python SDK 通过接口 create_collection() 创建集合 **book-vector**。

```
index = Index(
        FilterIndex(name='id', field_type=FieldType.String,
index_type=IndexType.PRIMARY_KEY),
        VectorIndex(name='vector', dimension=3, index_type=IndexType.HNSW,
                metric_type=MetricType.COSINE, params=HNSWParams(m=16,
efconstruction=200)),
        FilterIndex(name='bookName', field_type=FieldType.String,
index_type=IndexType.FILTER)
    )
# create a collection
# 免费测试版实例，其分片 shard 只能为 1，副本 replicas 仅能为 0。
coll = db.create_collection(
        name='book-vector',
        shard=1,
        replicas=0,
        description='this is a collection book vector',
        index=index
    )
print(vars(coll))
```

Java

Java SDK 通过接口 createCollection 创建集合 **book-vector**。

```
// 免费测试版实例，其分片 shard 只能为 1，副本 replicas 仅能为 0。
CreateCollectionParam collectionParam = CreateCollectionParam.newBuilder()
        .withName("book-vector")
        .withShardNum(1)
        .withReplicaNum(0)
        .withDescription("this is a collection book vector")
        .addField(new FilterIndex("id", FieldType.String, IndexType.PRIMARY_KEY))
        .addField(new VectorIndex("vector", 3, IndexType.HNSW,
                MetricType.COSINE, new HNSWParams(16, 200)))
        .addField(new FilterIndex("bookName", FieldType.String, IndexType.FILTER))
        .build();
Collection collection = db.createCollection(collectionParam);
```

Go

Go SDK 通过接口 CreateCollection() 创建集合 **book-vector**。

```
var (
    ctx            = context.Background()
    database       = "db-test"
    collectionName = "book-vector"
)
index := tcvectordb.Indexes{
    VectorIndex: []tcvectordb.VectorIndex{
        {
            Dimension: 3,
            FilterIndex: tcvectordb.FilterIndex{
                FieldName: "vector",
                FieldType: tcvectordb.Vector,
                IndexType: tcvectordb.HNSW,
            },
            MetricType: tcvectordb.COSINE,
            Params: &tcvectordb.HNSWParam{
                M:              16,
                EfConstruction: 200,
            },
        },
    },
    FilterIndex: []tcvectordb.FilterIndex{
        {
            FieldName: "id",
            FieldType: tcvectordb.String,
            IndexType: tcvectordb.PRIMARY,
        },
        {
            FieldName: "bookName",
            FieldType: tcvectordb.String,
            IndexType: tcvectordb.FILTER,
        },
    },
}

// 免费测试版实例，其分片 shard 只能为 1，副本 replicas 仅能为 0。
coll, _ := db.CreateCollection(ctx, collectionName, 1, 0, "this is a collection book
vector", index)
```

```
log.Printf("CreateCollection success: %v: %v", coll.DatabaseName,
coll.CollectionName)
```

Curl

HTTP 通过 /collection/create 创建集合 **book-vector** 。免费测试版实例，其分片 shard 只能为 1，副本 replicas 仅能为 0。

```
curl -i -X POST \
  -H 'Content-Type: application/json' \
  -H 'Authorization: Bearer
account=root&api_key=A5VOgsMpGWJhUI0WmUbY********************' \
  http://10.0.X.X:80/collection/create \
  -d '{
    "database": "db-test",
    "collection": "book-vector",
    "replicaNum": 0,
    "shardNum": 1,
    "description": "this is a collection book vector",
    "indexes": [
        {
            "fieldName": "id",
            "fieldType": "string",
            "indexType": "primaryKey"
        },
        {
            "fieldName": "vector",
            "fieldType": "vector",
            "indexType": "HNSW",
            "dimension": 3,
            "metricType": "COSINE",
            "params": {
                "M": 16,
                "efConstruction": 200
            }
        },
        {
            "fieldName": "bookName",
            "fieldType": "string",
            "indexType": "filter"
        }
    ]
```

```
}'
```

插入数据

如下示例，为书籍集合 **book-vector** 写入3条数据。

> **说明：**
> - 向量数据库支持动态 Schema，写入数据时可以写入任何字段，无需提前定义，类似 MongoDB。如下示例，page 与 author 为新定义的书籍信息字段。
> - 创建集合时，并未对 page 与 author 构建 Filter 索引，因此，二者不具有过滤属性，仅 bookName 具有过滤属性。

Python

Python SDK 通过接口 upsert() 为集合 **book-vector** 写入向量数据。

```
res = coll.upsert(
        documents=[
            Document(id='0001', vector=[
                0.2123, 0.23, 0.213], author='罗贯中', bookName='三国演义',
page=21),
            Document(id='0002', vector=[
                0.2123, 0.22, 0.213], author='吴承恩', bookName='西游记',
page=22),
            Document(id='0003', vector=[
                0.2123, 0.21, 0.213], author='曹雪芹', bookName='红楼梦',
page=23)
        ]
    )
```

Java

Java SDK 通过 upsert() 接口为集合 **book-vector** 批量插入数据。

```
Document doc1 = Document.newBuilder()
        .withId("0001")
        .withVector(Arrays.asList(0.2123, 0.23, 0.213))
```

```
        .addDocField(new DocField("bookName", "三国演义"))
        .addDocField(new DocField("author", "吴承恩"))
        .addDocField(new DocField("page", 21))
        .build();
Document doc2 = Document.newBuilder()
        .withId("0002")
        .withVector(Arrays.asList(0.2123, 0.22, 0.213))
        .addDocField(new DocField("bookName", "西游记"))
        .addDocField(new DocField("author", "吴承恩"))
        .addDocField(new DocField("page", 22))
        .build();
Document doc3 = Document.newBuilder()
        .withId("0003")
        .withVector(Arrays.asList(0.2123, 0.21, 0.213))
        .addDocField(new DocField("bookName", "红楼梦"))
        .addDocField(new DocField("author", "曹雪芹"))
        .addDocField(new DocField("page", 23))
        .build();
InsertParam insertParam = InsertParam.newBuilder()
        .addDocument(doc1)
        .addDocument(doc2)
        .addDocument(doc3)
        .build();
collection.upsert(insertParam);
```

Go

Go SDK 通过 Upsert() 接口为集合 book-vector 批量插入数据。

```
result, err := coll.Upsert(ctx, []tcvectordb.Document{
    {
        Id:     "0001",
        Vector: []float32{0.2123, 0.23, 0.213},
        Fields: map[string]tcvectordb.Field{
            "bookName": {Val: "三国演义"},
            "author":   {Val: "罗贯中"},
            "page":     {Val: 21},
        },
    },
    {
        Id:     "0002",
        Vector: []float32{0.2123, 0.22, 0.213},
```

```
        Fields: map[string]tcvectordb.Field{
            "bookName": {Val: "西游记"},
            "author":   {Val: "吴承恩"},
            "page":     {Val: 22},
        },
    },
    {
        Id:     "0003",
        Vector: []float32{0.2123, 0.21, 0.213},
        Fields: map[string]tcvectordb.Field{
            "bookName": {Val: "红楼梦"},
            "author":   {Val: "曹雪芹"},
            "page":     {Val: 23},
        },
    },
}, &tcvectordb.UpsertDocumentParams{})

log.Printf("upsert result: %+v", result)
```

Curl

如下示例，通过 /document/upsert 给集合 book-vector 批量插入数据。插入数据以 Document 为最小单元，插入3 个 Document。

```
curl -i -X POST \
  -H 'Content-Type: application/json' \
  -H 'Authorization: Bearer
account=root&api_key=A5VOgsMpGWJhUI0WmUbY********************' \
  http://10.0.X.X:80/document/upsert \
  -d '{
    "database": "db-test",
    "collection": "book-vector",
    "documents": [
        {
            "id": "0001",
            "vector": [
                0.2123,
                0.23,
                0.213
            ],
            "author": "罗贯中",
            "bookName": "三国演义",
```

```
        "page": 21
      },
      {
        "id": "0002",
        "vector": [
          0.2123,
          0.22,
          0.213
        ],
        "author": "吴承恩",
        "bookName": "西游记",
        "page": 22
      },
      {
        "id": "0003",
        "vector": [
          0.2123,
          0.21,
          0.213
        ],
        "author": "曹雪芹",
        "bookName": "红楼梦",
        "page": 23
      }
    ]
}'
```

执行成功，返回如下信息：

```
{
  "code": 0,
  "msg": "operation success",
  "affectedCount": 3
}
```

相似性检索

相似度检索 是基于向量数据之间的相似度计算方法来检索与查询向量最相似的数据。

Python

检索与输入向量数据相似的数据

Python SDK 提供了 search() 按照 Vector 搜索的能力，可根据指定的多个向量查找 TopK 个相似性结果。如下示例，检索与 **vectors** 字段指定的三组向量数据分别相似，且满足 **bookName** 条件表达式的 Top3 数据。

```
# 1. vectors 指定了需检索的向量数据。
# 2. filter 指定了 bookName 字段的条件表达式，过滤数据。
# 3. limit 限制每个单元返回的相似性数据的条数，如 vector 写入三组向量数据，limit 为 3，则每组向量返回 top3 的相似数据。
# 4. params 指定索引类型对应的查询参数，HNSW 类型需要设置 ef，指定查询的遍历范围。
doc_lists = coll.search(
        vectors=[[0.3123, 0.43, 0.213],[0.315, 0.4, 0.216],[0.40, 0.38, 0.26]],
        filter=Filter(Filter.In("bookName",["三国演义", "西游记"])),
        params=SearchParams(ef=200),
        limit=3
    )
for i, docs in enumerate(doc_lists):
            print(i)
            for doc in docs:
                print(doc)
```

检索结果，如下所示。

> **说明：**
>
> - 输出结果的顺序，与搜索时设置的 vectors 配置的向量值的顺序一致。如下示例，0下面的三行结果对应 `[0.3123, 0.43, 0.213]` 向量的相似度查询结果。1下面的三行结果对应 `[0.315, 0.4, 0.216]` 的查询结果。
> - 每一个查询结果都返回 TopK 条相似度计算的结果。其中，K为 **limit** 设置的数值，如果插入的数据不足 K 条，则返回实际检索到的 Document 数量。
> - 检索结果会按照与查询向量的相似程度进行排列，相似度最高的结果会排在最前面，相似度最低的结果则排在最后面。相似程度则通过 L2（欧几里得距离）、IP（内积）或 COSINE（余弦相似度）计算得出的分数来衡量，输出参数 score 表示相似性计算分数。其中，欧式距离（L2）计算所得的分数越小与搜索值越相似；而余弦相似度（COSINE）与 内积（IP）计算所得的分数越大与搜索值越相似。

```
0
{'id': '0001', 'score': 0.971423, 'bookName': '三国演义', 'page': 21, 'author': '罗贯中'}
```

```
{'id': '0002', 'score': 0.966884, 'author': '吴承恩', 'page': 22, 'bookName': '西游记'}
1
{'id': '0001', 'score': 0.978463, 'page': 21, 'bookName': '三国演义', 'author': '罗贯
中'}
{'id': '0002', 'score': 0.974783, 'author': '吴承恩', 'bookName': '西游记', 'page': 22}
2
{'id': '0001', 'score': 0.986069, 'bookName': '三国演义', 'page': 21, 'author': '罗贯
中'}
{'id': '0002', 'score': 0.985201, 'author': '吴承恩', 'bookName': '西游记', 'page': 22}
```

检索与指定 Document ID 相似的数据

Python SDK 还提供了 searchById() 按 id 检索的能力。如下示例，检索与 **id**（Document ID）为 0001、0002 分别相似，且满足 **bookName** 条件表达式相似度最高的 Top3 数据。

```
# 1. document_ids 指定了需检索文档的 id。
# 2. filter 指定了 bookName 字段的条件表达式，过滤数据。
# 3. limit 限制每个单元返回的相似性数据的条数，如 document_ids 传入2个数据 id，
limit 为 3，则返回每个 id 相似的 top3 向量。
# 4. params 指定索引类型对应的查询参数。HNSW 类型需要设置 ef，指定查询的遍历范
围。
doc_lists = coll.searchById(
            document_ids=['0001','0002'],
            filter=Filter(Filter.In("bookName",["三国演义", "西游记"])),
            params=SearchParams(ef=200),
            limit=3
        )
for i, docs in enumerate(doc_lists):
            print(i)
            for doc in docs:
                print(doc)
```

查看输出，如下所示。

> **说明：**
> 输出的 Document ID 顺序与查询时配置的参数 document_ids 输入的顺序一致。查询结果中0下面的三行为 id 为0001进行相似度查询的结果，1下面的三行为 id 为0002进行相似度查询的结果。

```
0
{'id': '0001', 'score': 1.0, 'author': '罗贯中', 'bookName': '三国演义', 'page': 21}
{'id': '0002', 'score': 0.999773, 'page': 22, 'author': '吴承恩', 'bookName': '西游记'}
1
{'id': '0002', 'score': 1.0, 'author': '吴承恩', 'bookName': '西游记', 'page': 22}
```

```
{'id': '0001', 'score': 0.999773, 'bookName': '三国演义', 'page': 21, 'author': '罗贯中'}
```

Java

检索与输入向量相似的数据

Java SDK 提供了 search() 接口按照 Vector 搜索的能力。如下示例，检索与 Vectors 字段指定的三组向量数据分别相似，且满足 bookName 条件表达式的 Top3 条数据。

```
SearchByVectorParam searchByVectorParam =
SearchByVectorParam.newBuilder()
        // Vectors 指定了需检索的向量数据
        .withVectors(Arrays.asList(Arrays.asList(0.3123, 0.43, 0.213),
Arrays.asList(0.315, 0.4, 0.216), Arrays.asList(0.40, 0.38, 0.26)))
        // 若使用 HNSW 索引，则需要指定参数 ef，ef 越大，召回率越高，但也会影响检索速度
        .withParams(new HNSWSearchParams(200))
        // 指定返回的最相似的 Top K 的 K 值
        .withLimit(3)
        // 设置标量字段的 Filter 表达式，过滤所需查询的文档
        .withFilter(new Filter(Filter.in("bookName", Arrays.asList("三国演义","西游记"))))
        .build();
// 输出相似度检索结果
List<List<Document>> svDocs = collection.search(searchByVectorParam);
int i = 0;
for (List<Document> docs : svDocs) {
    System.out.println("\tres: " + i++);
    for (Document doc : docs) {
        System.out.println("\tres: " + doc.toString());
    }
}
```

检索结果，如下所示。

> **说明：**
> - 输出数组的 Document 顺序与搜索时设置的 Vectors 配置的向量值的顺序一致。

- 每一个查询结果都返回 TopK 条相似度计算的结果。其中，K 为 **limit** 设置的数值，如果检索的数据不足 K 条，则返回实际检索到的 Document 数量。
- 检索结果会按照与查询向量的相似程度进行排列，相似度最高的结果会排在最前面，相似度最低的结果则排在最后面。相似程度则通过 L2（欧几里得距离）、IP（内积）或 COSINE（余弦相似度）计算得出的分数来衡量，输出参数 score 表示相似性计算分数。其中，欧式距离（L2）计算所得的分数越小与搜索值越相似；而余弦相似度（COSINE）与 内积（IP） 计算所得的分数越大与搜索值越相似。

```
res: 0
res: {"id":"0001","score":0.971423,"page":21,"bookName":"三国演义","author":"吴承恩"}
res: {"id":"0002","score":0.966884,"bookName":"西游记","page":22,"author":"吴承恩"}
res: 1
res: {"id":"0001","score":0.978463,"page":21,"author":"吴承恩","bookName":"三国演义"}
res: {"id":"0002","score":0.974783,"bookName":"西游记","page":22,"author":"吴承恩"}
res: 2
res: {"id":"0001","score":0.986069,"bookName":"三国演义","author":"吴承恩","page":21}
res: {"id":"0002","score":0.985201,"author":"吴承恩","bookName":"西游记","page":22}
```

根据 ID 进行相似度检索

Java SDK 还提供了 searchById() 接口按 id 检索的能力。如下示例，检索与 id（Document ID）为 0001、0002分别相似，且满足 **bookName** 条件表达式的 Top3 条数据。

```
SearchByIdParam searchByIdParam = SearchByIdParam.newBuilder()
        .withDocumentIds(Arrays.asList("0001", "0002"))
        // 若使用 HNSW 索引，则需要指定参数 ef，ef 越大，召回率越高，但也会影响检索速度
        .withParams(new HNSWSearchParams(200))
         // 指定 Top K 的 K 值
        .withLimit(3)
        // 使用 filter 过滤数据
        .withFilter(new Filter(Filter.in("bookName", Arrays.asList("三国演义","西游记"))))
        .build();
List<List<Document>> siDocs = collection.searchById(searchByIdParam);
int i = 0;
```

```
for (List<Document> docs : siDocs) {
    System.out.println("\tres: " + i++);
    for (Document doc : docs) {
        System.out.println("\tres: " + doc.toString());
    }
}
```

检索结果，如下所示。

> **说明：**
> 输出数组的 Document 顺序与查询时配置的参数 **DocumentIds** 输入的顺序一致。如下 documents 中第一个中括号为 id 为 0001的相似数据，第二个中括号为 id 为 0002 的相似数据。

```
res: 0
res: {"id":"0001","score":1.0,"bookName":"三国演义","page":21,"author":"吴承恩"}
res: {"id":"0002","score":0.999773,"author":"吴承恩","bookName":"西游
记","page":22}
res: 1
res: {"id":"0002","score":1.0,"author":"吴承恩","bookName":"西游记","page":22}
res: {"id":"0001","score":0.999773,"bookName":"三国演义","author":"吴承
恩","page":21}
```

Go

检索与输入向量相似的数据

Go SDK 提供了 Search() 接口按照 Vector 搜索的能力。如下示例，检索与指定的三组向量数据分别相似，且满足 **bookName** 条件表达式的 Top3 条数据。

```
filter := tcvectordb.NewFilter(`bookName in ("三国演义","西游记")`)
searchRes, _ := coll.Search(ctx, [][]float32{
    {0.3123, 0.43, 0.213},
    {0.315, 0.4, 0.216},
    {0.40, 0.38, 0.26},
}, &tcvectordb.SearchDocumentParams{
    Filter:         filter,
    Params:         &tcvectordb.SearchDocParams{Ef: 200},
    Limit:          3,
})
```

```
for i, docs := range searchRes.Documents {
    log.Printf("doc %d result: ", i)
    for _, doc := range docs {
        log.Printf("document: %+v", doc)
    }
}
```

检索结果，如下所示。

> **说明：**
> - 输出数组的 Document 顺序与查询时配置的向量数据的顺序一致。
> - 每一个查询结果都返回 TopK 条相似度计算的结果。其中，K 为 **limit** 设置的数值，如果检索的数据不足 K 条，则返回实际检索到的 Document 数量。
> - 检索结果会按照与查询向量的相似程度进行排列，相似度最高的结果会排在最前面，相似度最低的结果则排在最后面。相似程度则通过 L2（欧几里得距离）、IP（内积）或 COSINE（余弦相似度）计算得出的分数来衡量，输出参数 **Score** 表示相似性计算分数。其中，欧式距离（L2）计算所得的分数越小与搜索值越相似；而余弦相似度（COSINE）与 内积（IP）计算所得的分数越大与搜索值越相似。

```
2024/01/03 17:19:11 doc 0 result:
2024/01/03 17:21:29 document: {Id:0001 Vector:[] Score:0.971423
Fields:map[author:罗贯中 bookName:三国演义 page:21]}
2024/01/03 17:21:29 document: {Id:0002 Vector:[] Score:0.966884
Fields:map[author:吴承恩 bookName:西游记 page:22]}
2024/01/03 17:21:29 doc 1 result:
2024/01/03 17:21:29 document: {Id:0001 Vector:[] Score:0.978463
Fields:map[author:罗贯中 bookName:三国演义 page:21]}
2024/01/03 17:21:29 document: {Id:0002 Vector:[] Score:0.974783
Fields:map[author:吴承恩 bookName:西游记 page:22]}
2024/01/03 17:21:29 doc 2 result:
2024/01/03 17:21:29 document: {Id:0001 Vector:[] Score:0.986069
Fields:map[author:罗贯中 bookName:三国演义 page:21]}
2024/01/03 17:21:29 document: {Id:0002 Vector:[] Score:0.985201
Fields:map[author:吴承恩 bookName:西游记 page:22]}
```

检索与指定 id 相似的数据

Go SDK 还提供了 SearchById() 接口按 id 检索的能力。如下示例，检索与 id（Document ID）为 0001、0002分别相似，且满足 **bookName** 条件表达式的 Top3 条数据。

```
documentId := []string{"0001", "0002"}
```

```
filter := tcvectordb.NewFilter(`bookName in ("三国演义","西游记")`)
searchRes, err := coll.SearchById(ctx, documentId,
&tcvectordb.SearchDocumentParams{
    Filter:         filter,
    Params:         &tcvectordb.SearchDocParams{Ef: 200},
    Limit:          3,
})
for i, docs := range searchRes.Documents {
    log.Printf("doc %d result: ", i)
    for _, doc := range docs {
        log.Printf("document: %+v", doc)
    }
}
```

检索结果，如下所示。

```
2024/01/03 17:24:27 doc 0 result:
2024/01/03 17:24:27 document: {Id:0001 Vector:[] Score:1 Fields:map[author:罗
贯中 bookName:三国演义 page:21]}
2024/01/03 17:24:27 document: {Id:0002 Vector:[] Score:0.999773
Fields:map[author:吴承恩 bookName:西游记 page:22]}
2024/01/03 17:24:27 doc 1 result:
2024/01/03 17:24:27 document: {Id:0002 Vector:[] Score:1 Fields:map[author:吴
承恩 bookName:西游记 page:22]}
2024/01/03 17:24:27 document: {Id:0001 Vector:[] Score:0.999773
Fields:map[author:罗贯中 bookName:三国演义 page:21]}
```

Curl

检索与输入向量相似的数据

HTTP 支持 /document/search 接口按照 Vector 检索的能力，查询与指定向量 vectors 字段相似的数据。如下示例，检索与指定的三组向量数据分别相似，且满足 **bookName** 条件表达式的 Top3 条数据。

```
curl -i -X POST \
  -H 'Content-Type : application/json' \
  -H 'Authorization : Bearer
account=root&api_key=A5VOgsMpGWJhUI0WmUbY********************' \
  http : //10.0.X.X:80/document/search \
  -d ' {
```

```
    "database": "db-test",
    "collection": "book-vector",
    "search": {
      "vectors": [
        [
          0.3123,
          0.43,
          0.213
        ],
        [
          0.315,
          0.4,
          0.216
        ],
        [
          0.40,
          0.38,
          0.26
        ]
      ],
      "params": {
        "ef": 200
      },
      "filter": "bookName in (\"三国演义\",\"西游记\")",
      "limit": 3
    }
}'
```

执行成功，返回如下信息：

说明：

- 输出数组的 Document 顺序与查询时配置的参数 `vectors` 输入的顺序一致。
- 每一个查询结果都返回 TopK 条相似度计算的结果。其中，K 为 **limit** 设置的数值，如果检索的数据不足 K 条，则返回实际的 Document 数量。
- 检索结果会按照与查询向量的相似程度进行排列，相似度最高的结果会排在最前面，相似度最低的结果则排在最后面。相似程度则通过 L2（欧几里得距离）、IP（内积）或 COSINE（余弦相似度）计算得出的分数来衡量，输出参数 score 表示相似性计算分数。其中，欧式距离（L2）计算

> 所得的分数越小与搜索值越相似；而余弦相似度（COSINE）与 内积（IP）计算所得的分数越大与搜索值越相似。


```
{
 "code": 0,
 "msg": "operation success",
 "documents": [
  [
   {
    "id": "0001",
    "score": 0.971423,
    "page": 21,
    "author": "罗贯中",
    "bookName": "三国演义"
   },
   {
    "id": "0002",
    "score": 0.966884,
    "author": "吴承恩",
    "page": 22,
    "bookName": "西游记"
   }
  ],
  [
   {
    "id": "0001",
    "score": 0.978463,
    "author": "罗贯中",
    "bookName": "三国演义",
    "page": 21
   },
   {
    "id": "0002",
    "score": 0.974783,
    "page": 22,
    "bookName": "西游记",
    "author": "吴承恩"
   }
  ],
  [
   {
    "id": "0001",
    "score": 0.986069,
    "bookName": "三国演义",
```

```
      "author": "罗贯中",
      "page": 21
    },
    {
      "id": "0002",
      "score": 0.985201,
      "author": "吴承恩",
      "page": 22,
      "bookName": "西游记"
    }
  ]
 ]
}
```

检索与指定 ID 相似的数据

HTTP 还提供了 /document/search 按 id 检索的能力。如下示例，检索与 id（Document ID）为 0001、0002分别相似，且满足 **bookName** 条件表达式的 Top3 条数据。

```
curl -i -X POST \
  -H 'Content-Type: application/json' \
  -H 'Authorization: Bearer
account=root&api_key=A5VOgsMpGWJhUI0WmUbY********************' \
  http://10.0.X.X:80/document/search \
  -d '{
    "database": "db-test",
    "collection": "book-vector",
    "search": {
      "documentIds": [
        "0001",
        "0002"
      ],
      "params": {
        "ef": 200
      },
      "retrieveVector": true,
      "filter": "bookName in (\"三国演义\",\"西游记\")",
      "limit": 3
    }
}'
```

执行成功，返回如下信息：

> **说明：**

> 输出数组的 Document 顺序与查询时配置的参数 documentIds 输入 id 的顺序一致。如下 documents 中第一个中括号为 id 为 0001的相似数据，第二个中括号为 id 为 0002 的相似数据。


```
{
 "code": 0,
 "msg": "operation success",
 "documents": [
  [
   {
    "id": "0001",
    "score": 1.0,
    "bookName": "三国演义",
    "author": "罗贯中",
    "page": 21
   },
   {
    "id": "0002",
    "score": 0.999773,
    "page": 22,
    "bookName": "西游记",
    "author": "吴承恩"
   }
  ],
  [
   {
    "id": "0002",
    "score": 1.0,
    "page": 22,
    "author": "吴承恩",
    "bookName": "西游记"
   },
   {
    "id": "0001",
    "score": 0.999773,
    "page": 21,
    "author": "罗贯中",
    "bookName": "三国演义"
   }
  ]
 ]
}
```

删除数据库

Python

Python SDK 通过 drop_database() 接口删除数据库 db-test。

```
client.drop_database(database_name='db-test')
```

Java

Java SDK 通过 dropDatabase() 删除数据库 db-test。

```
client.dropDatabase("db-test");
```

Go

Go SDK 通过 DropDatabase() 删除数据库 db-test。

```
result, _ := client.DropDatabase(context.Background(), database)
```

Curl

HTTP 使用 /database/drop 接口删除数据库 db_test。如下示例，url 地址与 api_key 需要分别替换为已购买的免费版实例的外网地址与API Key。

```
curl -i -X POST \
  -H 'Content-Type: application/json' \
  -H 'Authorization: Bearer
account=root&api_key=A5VOgsMpGWJhUI0WmUbY********************' \
  http://10.0.X.X:80/database/drop \
  -d '{
    "database": "db-test"
}'
```

应用 Embedding 相似性检索

最近更新时间：2024-01-10 16:20:52

腾讯云向量数据库（Tencent Cloud VectorDB）默认开通 Embedding 功能。本章节介绍如何应用 Embedding 功能写入原始文本，并基于输入的文本信息进行相似性检索。以 Linux 操作系统为例，使用 Python、Java、Go SDK 与 HTTP API 示例代码分别演示。运行本章节所提供的示例代码，您将初步了解基于文本信息进行数据写入检索的能力。

导入 SDK 依赖模块

Python

```
import tcvectordb
from tcvectordb.model.enum import FieldType, IndexType, MetricType,
ReadConsistency, EmbeddingModel
from tcvectordb.model.index import Index, VectorIndex, FilterIndex,
HNSWParams, IVFFLATParams
from tcvectordb.model.collection import UpdateQuery
from tcvectordb.model.document import Document, SearchParams, Filter
from tcvectordb.model.collection import Embedding, UpdateQuery
```

Java

```
import com.tencentcloudapi.client.VectorDBClient;
import com.tencentcloudapi.model.*;
```

Go

```
package main

import (
    "context"
    "log"
```

```
    "time"

    "github.com/tencent/vectordatabase-sdk-go/tcvectordb"
)
```

创建 Client

导入 SDK 所需的模块之后，需先创建一个向量数据库的客户端对象，与向量数据库服务器连接才能进行交互。

> **说明：**
> 如下示例 **url** 与 **key** 需要分别替换为已购买的免费版实例的**外网访问地址**与 API Key。请登录 **向量数据库控制台**，在**实例详情**页面**网络信息**区域直接复制**外网地址**，在**密钥管理**页面直接复制密钥。

Python

```
client = tcvectordb.VectorDBClient(url='http://10.0.X.X', username='root',
key='eC4bLRy2va*****************************',
read_consistency=ReadConsistency.EVENTUAL_CONSISTENCY, timeout=30)
```

Java

```
public class VectorDBExample {
    public static void main(String[] args) {
        // 创建VectorDB Client
        ConnectParam connectParam = ConnectParam.newBuilder()
                .withUrl("http://10.0.X.X:80")
                .withUsername("root")
                .withKey("eC4bLRy2va*******************************")
                .withTimeout(30)
                .build();
        VectorDBClient client = new VectorDBClient(connectParam,
ReadConsistencyEnum.EVENTUAL_CONSISTENCY);
    }
}
```

Go

```
func main() {
    var defaultOption = &tcvectordb.ClientOption{
        Timeout:           time.Second * 5,
        MaxIdldConnPerHost: 2,
        IdleConnTimeout:    time.Minute,
        ReadConsistency:    tcvectordb.EventualConsistency,
    }
    client, err := tcvectordb.NewClient("http://10.0.X.X:80", "root",
"eC4bLRy2va**********************************", defaultOption)
    if err != nil {
        panic(err)
    }
}
```

创建数据库

Python

Python SDK 通过 create_database() 接口创建数据库 **db-test**。

```
db = client.create_database(database_name='db-test')
```

Java

Java SDK 通过 createDatabase() 创建数据库 **db-test**。

```
Database db = client.createDatabase("db-test");
```

Go

Go SDK 通过 CreateDatabase() 创建数据库 **db-test**。

```
var (
    ctx             = context.Background()
    database          = "db-test"
)
db, _ := client.CreateDatabase(ctx, database)
```

Curl

HTTP 使用 /database/create 接口创建名为 **db_test** 的数据库。其中， **url** 地址与 **api_key** 需要分别替换为已购买的免费版实例的外网地址与 API Key。

```
curl -i -X POST \
 -H 'Content-Type: application/json' \
 -H 'Authorization: Bearer
account=root&api_key=A5VOgsMpGWJhUI0WmUbY********************' \
http://10.0.X.X:80/database/create \
 -d '{
 "database": "db-test"
 }'
```

创建集合

1. 创建集合之前，需先设计索引结构，指定索引字段。如下示例，创建一个可写入3维向量数据存储书籍的集合 book-emb，其书籍信息字段包括： id、vector、bookName，分别对各字段构建索引。有关索引的具体信息，请参见 Index 。
   - 主键索引（Primary Key Index）：固定且必须，对应字段 **id**，每条数据的唯一标识。并对主键 **id** 构建 Filter 索引，以便可通过 id 的条件表达式进行特定行的检索。
   - 向量索引（Vector Index）：固定且必须，对应字段 **vector**，对向量数据构建索引，指定向量数据存储的索引类型、相似性计算方法及相关索引参数。

     > **说明：**
     > 应用 Embedding 功能，向量数据的维度 Dimension 可以不配置，默认会使用 Embedding 模型支持的数据维度。如下示例，**model** 指定的模型为 **BGE_BASE_ZH**，默认数据维度为 768 维。若配置 Dimension，则务必与 Embedding 的模型维数一致。

   - Filter 索引（Filter Index）：需根据检索需求选取可作为条件查询过滤数据的字段。通常，向量数据对应的文本字段，不具有过滤属性，则无需对该字段建立索引，否则，将浪费较大的内存也无实际意义。如下示

例，预按书籍的名称过滤数据，对 **bookName** 字段建立 Filter 索引。

2. 应用 Embedding 功能，则需配置 Embedding 参数。如下示例，自定义文本信息的字段名为 **text**、选择 Embedding 模型 **bge-base-zh**，向量数据字段固定为 **vector**。
3. 创建集合，免费测试版实例，其分片 shard 只能为 1，副本 replicas 仅能为 0。

Python

Python SDK 通过接口 create_collection() 创建集合 **book-vector**。

```
# 第一步：设计索引字段
index = Index(
        FilterIndex(name='id', field_type=FieldType.String,
index_type=IndexType.PRIMARY_KEY),
        VectorIndex(name='vector', dimension=768,
index_type=IndexType.HNSW,
                    metric_type=MetricType.COSINE, params=HNSWParams(m=16,
efconstruction=200)),
        FilterIndex(name='bookName', field_type=FieldType.String,
index_type=IndexType.FILTER)
    )

# 第二步：配置 Embedding 参数
# 1. 指定文本字段与向量字段，向量字段固定为vector
# 2. 指定 Embedding 模型，推荐使用 BGE_BASE_ZH。
ebd = Embedding(vector_field='vector', field='text',
model=EmbeddingModel.BGE_BASE_ZH)

# 第三步，创建 Collection
coll = db.create_collection(
        name='book-emb',
        shard=1,
        replicas=0,
        description='this is an embedding collection',
        embedding=ebd,
        index=index
    )
print(vars(coll))
```

Java

Java SDK 通过接口 createCollection 创建集合 book-emb。

```
CreateCollectionParam collectionParam = CreateCollectionParam.newBuilder()
        .withName("book-emb")
        .withShardNum(1)
        .withReplicaNum(0)
        .withDescription("this is an embedding collection")
        .addField(new FilterIndex("id", FieldType.String, IndexType.PRIMARY_KEY))
        .addField(new VectorIndex("vector", BGE_BASE_ZH.getDimension(),
IndexType.HNSW,
                MetricType.COSINE, new HNSWParams(16, 200)))
        .addField(new FilterIndex("bookName", FieldType.String, IndexType.FILTER))
        .withEmbedding(
                Embedding
                        .newBuilder()
                        .withModel(BGE_BASE_ZH)
                        .withField("text")
                        .withVectorField("vector")
                        .build())
        .build();
Collection collection = db.createCollection(collectionParam);
```

Go

Go SDK 通过接口 CreateCollection() 创建集合 book-emb。

```
var (
    embeddingCollection = "book-emb"
)
// 第一步：设计索引字段
index := tcvectordb.Indexes{
    VectorIndex: []tcvectordb.VectorIndex{
        {
            FilterIndex: tcvectordb.FilterIndex{
                FieldName: "vector",
                FieldType: tcvectordb.Vector,
                IndexType: tcvectordb.HNSW,
            },
            MetricType: tcvectordb.COSINE,
            Params: &tcvectordb.HNSWParam{
                M:              16,
```

```
            EfConstruction: 200,
        },
    },
},
FilterIndex: []tcvectordb.FilterIndex{
    {
        FieldName: "id",
        FieldType: tcvectordb.String,
        IndexType: tcvectordb.PRIMARY,
    },
    {
        FieldName: "bookName",
        FieldType: tcvectordb.String,
        IndexType: tcvectordb.FILTER,
    },
},
}
// 第二步：配置 Embedding 参数
param := &tcvectordb.CreateCollectionParams{
    Embedding: &tcvectordb.Embedding{
        Field:      "text",
        VectorField: "vector",
        Model:      tcvectordb.BGE_BASE_ZH,
    },
}
// 第三步：创建集合
coll, _ := db.CreateCollection(ctx, embeddingCollection, 1, 0, "this is an
embedding collection", index, param)
log.Printf("CreateCollection success: %v: %v", coll.DatabaseName,
coll.CollectionName)
```

Curl

HTTP 通过 /collection/create 创建集合 **book-emb**。

```
curl -i -X POST \
  -H 'Content-Type: application/json' \
  -H 'Authorization: Bearer
account=root&api_key=A5VOgsMpGWJhUI0WmUbY********************' \
  http://10.0.X.X:80/collection/create \
  -d '{
    "database": "db-test",
```

```
    "collection": "book-emb",
    "replicaNum": 0,
    "shardNum": 1,
    "description": "this is an embedding collection",
    "embedding": {
        "field": "text",
        "vectorField": "vector",
        "model": "bge-base-zh"
    },
    "indexes": [
        {
            "fieldName": "id",
            "fieldType": "string",
            "indexType": "primaryKey"
        },
        {
            "fieldName": "vector",
            "fieldType": "vector",
            "indexType": "HNSW",
            "metricType": "COSINE",
            "params": {
                "M": 16,
                "efConstruction": 200
            }
        },
        {
            "fieldName": "bookName",
            "fieldType": "string",
            "indexType": "filter"
        }
    ]
}'
```

基于文本信息写入数据

如下示例，为书籍集合 **book-emb** 写入3条数据，其中 **text** 字段为写入的文本信息。Embedding 将文本信息自动向量化后写入数据库，并保存原始文本。

> **说明：**
> - 向量数据库支持动态 Schema，写入数据时可以写入任何字段，无需提前定义，类似 MongoDB。如下示例，page 与 author 为新定义的书籍信息字段。

- 创建集合时，并未对 page 与 author 构建 Filter 索引，因此，二者不具有过滤属性，仅 bookName 具有过滤属性。
- 写入数据，可能存在一定延迟。

Python

Pyhton SDK 通过 upsert() 接口为集合 **book-emb**，写入原始文本。

```
res = coll.upsert(
        documents=[
            Document(id='0001', text="话说天下大势，分久必合，合久必分。",
author='罗贯中', bookName='三国演义', page=21),
            Document(id='0002', text="混沌未分天地乱，茫茫渺渺无人间。",
author='吴承恩', bookName='西游记', page=22),
            Document(id='0003', text="甄士隐梦幻识通灵，贾雨村风尘怀闺秀。",
author='曹雪芹', bookName='红楼梦', page=23)
        ],
    )
```

Java

在 createCollection() 建表时，配置 Embedding 模型相关参数之后，便可以通过 **upsert()** 接口可直接传入原始文本。如下示例，基于 createCollection() 创建的集合 **book-emb**，写入原始文本。

```
Document doc1 = Document.newBuilder()
        .withId("0001")
        .addDocField(new DocField("text", "话说天下大势，分久必合，合久必分。"))
        .addDocField(new DocField("bookName", "三国演义"))
        .addDocField(new DocField("author", "罗贯中"))
        .addDocField(new DocField("page", 21))
        .build();
Document doc2 = Document.newBuilder()
        .withId("0002")
        .addDocField(new DocField("text", "混沌未分天地乱，茫茫渺渺无人间。"))
        .addDocField(new DocField("bookName", "西游记"))
        .addDocField(new DocField("author", "吴承恩"))
        .addDocField(new DocField("page", 22))
        .build();
Document doc3 = Document.newBuilder()
```

```
        .withId("0003")
        .addDocField(new DocField("text", "甄士隐梦幻识通灵，贾雨村风尘怀闺秀。"))
        .addDocField(new DocField("bookName", "红楼梦"))
        .addDocField(new DocField("author", "曹雪芹"))
        .addDocField(new DocField("page", 23))
        .build();

InsertParam insertParam = InsertParam.newBuilder()
        .addDocument(doc1)
        .addDocument(doc2)
        .addDocument(doc3)
        .build();
collection.upsert(insertParam);
```

Go

Go SDK 通过 Upsert() 接口为集合 book-vector 批量插入数据。

```
result, err := coll.Upsert(ctx, []tcvectordb.Document{
    {
        Id:     "0001",
        Fields: map[string]tcvectordb.Field{
            "text":  {Val: "话说天下大势，分久必合，合久必分。"},
            "bookName": {Val: "三国演义"},
            "author":   {Val: "罗贯中"},
            "page":     {Val: 21},
        },
    },
    {
        Id:     "0002",
        Fields: map[string]tcvectordb.Field{
            "text":  {Val: "混沌未分天地乱，茫茫渺渺无人间。"},
            "bookName": {Val: "西游记"},
            "author":   {Val: "吴承恩"},
            "page":     {Val: 22},
        },
    },
    {
        Id:     "0003",
        Fields: map[string]tcvectordb.Field{
            "text":  {Val: "甄士隐梦幻识通灵，贾雨村风尘怀闺秀。"},
            "bookName": {Val: "红楼梦"},
```

```
            "author":   {Val: "曹雪芹"},
            "page":     {Val: 23},
        },
    },
}, &tcvectordb.UpsertDocumentParams{})
log.Printf("upsert result: %+v", result)
```

Curl

如下示例，通过 /document/upsert 为集合 book-emb，写入原始文本。

```
curl -i -X POST \
  -H 'Content-Type: application/json' \
  -H 'Authorization: Bearer
account=root&api_key=A5VOgsMpGWJhUI0WmUbY**********' \
  http://10.0.X.X:80/document/upsert \
  -d '{
    "database": "db-test",
    "collection": "book-emb",

    "documents": [
        {
            "id": "0001",
            "text": "话说天下大势，分久必合，合久必分。",
            "author": "罗贯中",
            "bookName": "三国演义",
            "page": 21
        },
        {
            "id": "0002",
            "text": "混沌未分天地乱，茫茫渺渺无人间。",
            "author": "吴承恩",
            "bookName": "西游记",
            "page": 22
        },
        {
            "id": "0003",
            "text": "甄士隐梦幻识通灵，贾雨村风尘怀闺秀。",
            "author": "曹雪芹",
            "bookName": "红楼梦",
            "page": 23
        }
```

```
    ]
}'
```

相似性检索

基于 Embedding 功能的相似性检索，支持检索与输入的文本信息相似的文本。如下示例，检索与 **embeddingItems** 参数输入的文本信息相似度最高，且满足 **bookName** 条件表达式的文本。

> **说明：**
> - **params**：指定索引类型对应的查询参数。其中，**ef** 为 HWSN 索引类型对应的检索参数，指定寻找节点邻居遍历的范围，默认为200。**ef** 越大，召回率越高。
> - **filter**：指定了 bookName 字段的条件表达式，过滤数据。
> - **limit**： 限制每个单元返回的相似性数据的条数，如 limit 为3，则返回 top3 的相似数据。
> - **retrieve_vector** 指定是否输出向量字段。示例中，文本信息被 Embedding 向量化为768 维数据，数据量大，不变展示，设置为 False。
> - **output_fields**：可自定义需要输出的字段。若不自定，则返回所有字段。

Python

Python SDK 提供了 searchByText() 接口按照输入的文本批量进行相似性查询的能力。如下示例，在集合 **book-emb** 中，检索与 **embeddingItems** 参数配置的文本信息相似，且满足 **bookName** 条件表达式的文本。

```
doc_lists = coll.searchByText(
            embeddingItems=['天下大势，分久必合，合久必分'],
            filter=Filter(Filter.In("bookName",["三国演义", "西游记"])),
            params=SearchParams(ef=200),
            limit=3,
            retrieve_vector=False,
            output_fields=['bookName','author','text']
        )

for i, docs in enumerate(doc_lists.get("documents")):
            print(i)
            for doc in docs:
                print(doc)
```

检索结果，如下所示。

说明：
检索结果将按照相似程度的高低排列。相似度最高的结果会排在最前面，最低的结果则排在后面。相似程度则通过 L2（欧几里得距离）、IP（内积）或 COSINE（余弦相似度）计算得出的分数来衡量。输出参数 **score** 表示相似性计算分数。其中，欧式距离（L2）计算所得的分数越小与搜索值越相似；而余弦相似度（COSINE）与 内积（IP）计算所得的分数越大与搜索值越相似。

```
{'id': '0001', 'score': 0.979274, 'text': '话说天下大势，分久必合，合久必分。',
'bookName': '三国演义', 'author': '罗贯中'}
{'id': '0002', 'score': 0.790986, 'bookName': '西游记', 'text': '混沌未分天地乱，茫茫渺
渺无人间。', 'author': '吴承恩'}
```

Java

Java SDK 通过 searchByEmbeddingItems() 接口，在集合 **book-emb** 中，检索与 **EmbeddingItems** 参数的文本信息最相似，且满足 **bookName** 条件表达式的文本。

```
 SearchByEmbeddingItemsParam searchByEmbeddingItemsParam =
SearchByEmbeddingItemsParam.newBuilder()
        .withEmbeddingItems(Arrays.asList("天下大势，分久必合，合久必分"))
        // 若使用 HNSW 索引，则需要指定参数 ef，ef 越大，召回率越高，但也会影响检索速
度
        .withParams(new HNSWSearchParams(200))
        // 设置标量字段的 Filter 表达式，过滤所需查询的文档
        .withRetrieveVector(false)
         // 指定 Top K 的 K 值
        .withLimit(5)
        // 使用 filter 过滤数据
        .withFilter(new Filter(Filter.in("bookName", Arrays.asList("三国演义","西游
记"))))
        // 指定返回的 fields
        .withOutputFields(Arrays.asList("author", "bookName", "text"))
        .build();
List<List<Document>> siDocs =
collection.searchByEmbeddingItems(searchByEmbeddingItemsParam);
int i = 0;
for (List<Document> docs : siDocs) {
    System.out.println("\tres: " + i++);
    for (Document doc : docs) {
        System.out.println(doc.toString());
    }
```

```
}
```

检索结果，如下所示。

> **说明：**
> 检索结果将按照相似程度的高低排列。相似度最高的结果会排在最前面，最低的结果则排在后面。相似程度则通过 L2（欧几里得距离）、IP（内积）或 COSINE（余弦相似度）计算得出的分数来衡量。输出参数 **score** 表示相似性计算分数。其中，欧式距离（L2）计算所得的分数越小与搜索值越相似；而余弦相似度（COSINE）与 内积（IP） 计算所得的分数越大与搜索值越相似。

```
res: 0
res: {"id":"0001","score":0.979274,"bookName":"三国演义","author":"罗贯
中","text":"话说天下大势，分久必合，合久必分。"}
res: {"id":"0002","score":0.790986,"bookName":"西游记","author":"吴承恩",
"text":"沌未分天地乱，茫茫渺渺无人间。"}
```

Go

Go SDK 通过 SearchByText() 接口，在集合 **book-emb** 中，检索与 text 字段的文本信息最相似，且满足 **bookName** 条件表达式的文本。

```
filter := tcvectordb.NewFilter(`bookName in ("三国演义","西游记")`)
searchRes, _ := coll.SearchByText(ctx, map[string][]string{"text": {"天下大势，分
久必合，合久必分"}}, &tcvectordb.SearchDocumentParams{
    Params:         &tcvectordb.SearchDocParams{Ef: 200},
    RetrieveVector: false,
    Limit:          3,
    Filter:         filter,
    OutputFields:   []string{"author", "bookName", "text"},
})
for i, docs := range searchRes.Documents {
    log.Printf("doc %d result: ", i)
    for _, doc := range docs {
        log.Printf("document: %+v", doc)
    }
}
```

检索结果，如下所示。

说明：
检索结果将按照相似程度的高低排列。相似度最高的结果会排在最前面，最低的结果则排在后面。相似程度则通过 L2（欧几里得距离）、IP（内积）或 COSINE（余弦相似度）计算得出的分数来衡量。输出参数 **score** 表示相似性计算分数。其中，欧式距离（L2）计算所得的分数越小与搜索值越相似；而余弦相似度（COSINE）与 内积（IP） 计算所得的分数越大与搜索值越相似。

```
2024/01/04 17:42:05 doc 0 result:
2024/01/04 17:42:05 document: {Id:0001 Vector:[] Score:0.979274
Fields:map[author:罗贯中 bookName:三国演义 page:21 text:话说天下大势，分久必合，合久必分。]}
2024/01/04 17:42:05 document: {Id:0002 Vector:[] Score:0.790986
Fields:map[author:吴承恩 bookName:西游记 page:22 text:混沌未分天地乱，茫茫渺渺无人间。]}
```

Curl

HTTP 使用 /document/search 接口，在集合 **book-emb** 中，检索与 **embeddingItems** 参数的文本信息相似最高，且满足 **bookName** 条件表达式的文本。

```
curl -i -X POST \
  -H 'Content-Type: application/json' \
  -H 'Authorization: Bearer
account=root&api_key=A5VOgsMpGWJhUI0WmUbY********************' \
  http://10.0.X.X:80/document/search \
  -d '{
    "database": "db-test",
    "collection": "book-emb",
    "search": {
        "embeddingItems": [
            "天下大势，分久必合，合久必分"
        ],
        "limit": 3,
        "params": {
            "ef": 200
        },
        "retrieveVector": false,
        "filter": "bookName in (\"三国演义\",\"西游记\")",
        "outputFields": [
            "id",
            "author",
```

```
        "text",
        "bookName"
      ]
    }
}'
```

检索结果，如下所示。

> **说明：**
> 检索结果将按照相似程度的高低排列。相似度最高的结果会排在最前面，最低的结果则排在后面。相似程度则通过 L2（欧几里得距离）、IP（内积）或 COSINE（余弦相似度）计算得出的分数来衡量。输出参数 **score** 表示相似性计算分数。其中，欧式距离（L2）计算所得的分数越小与搜索值越相似；而余弦相似度（COSINE）与 内积（IP） 计算所得的分数越大与搜索值越相似。


```
{
 "code": 0,
 "msg": "operation success",
 "documents": [
  [
   {
    "id": "0001",
    "score": 0.979274,
    "author": "罗贯中",
    "bookName": "三国演义",
    "text": "话说天下大势，分久必合，合久必分。"
   },
   {
    "id": "0002",
    "score": 0.790986,
    "text": "混沌未分天地乱，茫茫渺渺无人间。",
    "bookName": "西游记",
    "author": "吴承恩"
   }
  ]
 ]
}
```


删除数据库

Python

Python SDK 通过 drop_database() 接口删除数据库 db-test。

```
client.drop_database(database_name='db-test')
```

Java

Java SDK 通过 dropDatabase() 删除数据库 db-test。

```
client.dropDatabase("db-test");
```

Go

Go SDK 通过 DropDatabase() 删除数据库 db-test。

```
result, _ := client.DropDatabase(context.Background(), database)
```

Curl

HTTP 使用 /database/drop 接口删除数据库 db_test。如下示例，url 地址与 api_key 需要分别替换为已购买的免费版实例的外网地址与 API Key。

```
curl -i -X POST \
  -H 'Content-Type: application/json' \
  -H 'Authorization: Bearer
account=root&api_key=A5VOgsMpGWJhUI0WmUbY********************' \
  http://10.0.X.X:80/database/drop \
  -d '{
    "database": "db-test"
}'
```

基于 AI 套件快速导入文件并检索

最近更新时间：2024-05-11 16:59:51

腾讯云向量数据库（Tencent Cloud VectorDB）AI 套件 功能支持直接写入 Markdown 文件。本章节介绍如何应用 AI 套件上传文件写入数据，并基于输入的文本信息对文件内容进行相似性检索的方法。以 Linux 操作系统为例，使用 Python、Java、Go SDK 示例代码分别演示。运行本章节所提供的示例代码，您将初步了解 AI 套件一站式文档检索的解决方案。

导入 SDK 依赖模块

Python

```
import tcvectordb
from tcvectordb.model.ai_database import AIDatabase
from tcvectordb.model.collection_view import Embedding, SplitterProcess,
Language, CollectionView
from tcvectordb.model.document import Filter, Document
from tcvectordb.model.document_set import DocumentSet
from tcvectordb.model.enum import FieldType, IndexType, ReadConsistency
from tcvectordb.model.index import Index, FilterIndex
```

Java

```
import com.tencentcloudapi.client.VectorDBClient;
import com.tencentcloudapi.model.*
```

Go

```
package main

import (
    "context"
    "time"
```

```
    "log"
    "github.com/tencent/vectordatabase-sdk-go/tcvectordb"
)
```

创建 Client

导入 SDK 所需的模块之后，需先创建一个向量数据库的客户端对象，与向量数据库服务器连接才能进行交互。

> **说明：**
> 如下示例 **url** 与 **key** 需要分别替换为已购买的免费版实例的 外网访问地址 与 API Key。请登录 向量数据库控制台，在**实例详情**页面**网络信息**区域直接复制**外网地址**，在**密钥管理**页面直接复制密钥。

Python

```
#create a database client object
client = tcvectordb.VectorDBClient(url='http://10.0.X.X', username='root',
key='eC4bLRy2va*****************************',
read_consistency=ReadConsistency.EVENTUAL_CONSISTENCY, timeout=30)
```

Java

```
public class VectorDBExample {
    public static void main(String[] args) {
        // 创建VectorDB Client
        ConnectParam connectParam = ConnectParam.newBuilder()
                .withUrl("http://10.0.X.X:80")
                .withUsername("root")
                .withKey("eC4bLRy2va*****************************")
                .withTimeout(30)
                .build();
        VectorDBClient client = new
VectorDBClient(connectParam,ReadConsistencyEnum.EVENTUAL_CONSISTENCY);
    }
}
```

Go

```
func main() {
    // 初始化客户端
    var defaultOption = &tcvectordb.ClientOption{
        Timeout:           time.Second * 5,
        MaxIdldConnPerHost: 2,
        IdleConnTimeout:    time.Minute,
        ReadConsistency:    tcvectordb.EventualConsistency,
    }
    client, err := tcvectordb.NewClient("http://10.0.X.X:80", "root",
"eC4bLRy2va**********************************", defaultOption)
    if err != nil {
        panic(err)
    }
}
```

创建数据库

基于已创建的客户端对象，创建数据库 **db-test-ai**。

Python

Python SDK 支持通过 create_ai_database() 接口创建数据库。

```
db = client.create_ai_database(database_name='db-test-ai')
```

Java

Java SDK 支持通过 createAIDatabase() 接口创建数据库。

```
AIDatabase db = client.createAIDatabase("db-test-ai");
```

Go

Go SDK 支持通过 CreateAIDatabase() 接口创建数据库。

```
var (
    ctx             = context.Background()
    aiDatabase          = "db-test-ai"
)
db, _ := client.CreateAIDatabase(context.Background(), aiDatabase)
log.Printf("Create AI database success, %s", db.DatabaseName)
```

创建集合视图

创建 CollectionView 之前，需要针对预上传的文件数据选取可作为 Filter 索引的标量字段，以便使用该字段的 Filter 条件表达式过滤查找文件。通常选取文件的 Metadata 信息字段。如下示例，预以文件的作者字段为 Filter 索引，将字段名 **author** 设置 Filter 索引。

Python

Python SDK 通过 create_collection_view() 创建一个名为 **coll-ai-files** 集合视图。

```
# 第一步：设计索引，为文件 meta 信息标量字段 author 配置 Filter 索引
index = Index()
index.add(FilterIndex('author', FieldType.String, IndexType.FILTER))
# 第二步：创建集合视图
coll_view = db.create_collection_view(name='coll-ai-files',
                      description='This is a collectionView',
                      index=index)
print(vars(coll_view))
```

Java

Java SDK 通过 createCollectionView() 创建一个名为 **coll-ai-files** 集合视图。

```
// link database， client 为 VectorDBClient() 创建的客户端对象
AIDatabase db = client.aiDatabase("db-test-ai");
// 初始化 ColletionView 参数
CreateCollectionViewParam collectionParam =
CreateCollectionViewParam.newBuilder()
```

```
            .withName("coll-ai-files")
            .withDescription("This is a collectionView")
            .addField(new FilterIndex("author", FieldType.String, IndexType.FILTER))
            .build();
// Create CollectionView
CollectionView db.createCollectionView(collectionParam);
```

Go

Go SDK 通过 CreateCollectionView() 创建一个名为 **coll-ai-files** 集合视图。

```
var (
    ctx            = context.Background()
    aiDatabase     = "db-test-ai"
    collectionViewName = "coll-ai-files"
)
// 第一步：设计索引，为文件 meta 信息标量字段 author 配置 Filter 索引
index := tcvectordb.Indexes{
    FilterIndex: []tcvectordb.FilterIndex{
        {
            FieldName: "author",
            FieldType: tcvectordb.String,
            IndexType: tcvectordb.FILTER,
        },
    },
}
// 第二步：创建集合视图
coll, _ := db.CreateCollectionView(ctx, collectionViewName,
tcvectordb.CreateCollectionViewParams{
    Description: "This is a collectionView",
    Indexes:     index,
})
log.Printf("CreateCollectionView success: %v: %v", coll.DatabaseName,
coll.CollectionViewName)
```

上传文件

集合视图创建完成之后，便可指定上传文件所在路径，上传文件于数据库中。如下示例，文件路径为 `/tmp/腾讯云向量数据库.md` ，将该文件内容及其向量数据存储于集合视图 **coll-ai-files** 中。

Python

Python SDK 通过 load_and_split_text() 接口上传文件 腾讯云向量数据库.md 于集合 coll-ai-files。

```
# 上传文件
# 1. 指定文件在客户端的存放路径
# 2. 自定义文件meta数据
res = coll_view.load_and_split_text(local_file_path="/tmp/腾讯云向量数据库.md",
          metadata={
          'author': 'Tencent'
      }
  )
res = coll_view.load_and_split_text(local_file_path="/tmp/腾讯云向量数据库.md")
```

Java

Java SDK 通过 loadAndSplitText() 接口上传文件 腾讯云向量数据库.md 于集合 coll-ai-files。

```
// link database， client 为 VectorDBClient() 创建的客户端对象
AIDatabase db = client.aiDatabase("db-test-ai");
// link collectionView
CollectionView collection = db.describeCollectionView("coll-ai-files");
// LocalFilePath 配置上传文件的本地路径
// DocumentSetName 指定文件存储于数据库的名称
LoadAndSplitTextParam param = LoadAndSplitTextParam.newBuilder()
            .withLocalFilePath("/tmp/腾讯云向量数据库.md")
            .Build();
// 配置文件 Meatdata 标量字段的值
Map<String, Object> metaDataMap = new HashMap<>();
metaDataMap.put("author", "Tencent");
// 调用 loadAndSplitText() 上传文件
collection.loadAndSplitText(param, metaDataMap);
```

Go

Go SDK 通过 LoadAndSplitText() 接口上传文件 腾讯云向量数据库.md 于集合 coll-ai-files。

```
metaData := map[string]interface{}{
        "author": "Tencent",
    }
UpsertResult, _ := coll.LoadAndSplitText(ctx,
tcvectordb.LoadAndSplitTextParams{
    LocalFilePath: "/tmp/腾讯云向量数据库.md",
    MetaData:      metaData,
})
log.Printf("LoadAndSplitText success: %+v", UpsertResult)
```

查询文件

上传文件可能存在延迟，查询文件，以便确认文件已在后台解析完成。如下示例，指定文件名，查询存储于数据库中的文件状态。返回参数 **indexedStatus** 将显示文件预处理、Embedding 向量化的状态。

- **New**：等待解析。
- **Loading**：文件解析中。
- **Failure**：文件解析、写入出错。
- **Ready**：文件解析、写入完成。

Python

Python SDK 通过 get_document_set() 获取文件当前状态。

```
res = coll_view.get_document_set(document_set_name="腾讯云向量数据库.md")
print(vars(res))
```

Java

Java SDK 通过 getFile() 获取文件当前状态。

```
String fileId="";
String fileName = "腾讯云向量数据库.md";
System.out.println(collection.getFile(fileName,fileId).toString());
```

Go

Go SDK 通过 GetDocumentSetByName() 获取文件当前状态。

```
result, _ := coll.GetDocumentSetByName(ctx, "腾讯云向量数据库.md")
log.Printf("GetDocumentSetByName success: %+v", result)
```

相似性内容检索

确认文件已解析完成之后，便可开始进行相似性内容检索。如下示例，检索信息 `什么是向量数据库` ，默认返回相似度最高的信息。

Python

Python SDK 通过 search() 接口进行相似性检索。

```
# content 指定需检索的文本内容
# document_set_name 指定检索的文件名
doc_list = coll_view.search(
        content='向量是指什么',
        document_set_name = ['腾讯云向量数据库.md']
    )
for doc in doc_list:
    print(vars(doc))
```

检索结果，如下所示。

关键信息	子参数	参数含义
score	-	• 表示查询向量与检索结果向量之间的相似性计算分数。 • 输出结果按照相似程度得分由高到低逐一排列。
data	text	检索到的结果。

```
{
 'score': 0.8473929762840271,
 'data': {
  'text': '### 什么是向量？\n向量是指在数学和物理中用来表示大小和方向的量。它由一组有序的数值组成，这些数值代表了向量在每个坐标轴上的分量。\n',
```

```
    'startPos': 441,
    'endPos': 508,
    'pre': [
      '## 关键概念\n如果您不熟悉向量数据库和相似性搜索领域，请优先阅读以下基本概念，便于您对向量数据库有一个初步的了解。\n'
    ],
    'next': [
      '### 什么是非结构化数据？\n非结构化数据，是指图像、文本、音频等数据。与结构化数据相比，非结构化数据不遵循预定义模型或组织方式，通常更难以处理和分析。\n'
    ]
  },
  'documentSet': {
    'documentSetId': '1192727327592550400',
    'documentSetName': '腾讯云向量数据库.md',
    'author': 'Tencent'
  }
}{
  'score': 0.7868989706039429,
  'data': {
    'text': '### 什么是 AI 中的向量表示？\n当我们处理非结构化数据时，需要将其转换为计算机可以理解和处理的形式。向量表示是一种将非结构化数据转换为嵌入向量的技术，通过多维度向量数值表述某个对象或事物的属性或者特征。腾讯云向量数据库提供的模型能力，目前在开发调试中。\n',
    'startPos': 585,
    'endPos': 784,
    'pre': [
      '### 什么是非结构化数据？\n非结构化数据，是指图像、文本、音频等数据。与结构化数据相比，非结构化数据不遵循预定义模型或组织方式，通常更难以处理和分析。\n'
    ],
    'next': [
      '### 什么是向量检索？\n向量检索是一种基于向量空间模型的信息检索方法。将非结构化的数据表示为向量存入向量数据库，向量检索通过计算查询向量与数据库中存储的向量的相似度来找到目标向量。\n'
    ]
  },
  'documentSet': {
    'documentSetId': '1192727327592550400',
    'documentSetName': '腾讯云向量数据库.md',
    'author': 'Tencent'
  }
}{
  'score': 0.7792050242424011,
  'data': {
    'text': '### 什么是向量检索？\n向量检索是一种基于向量空间模型的信息检索方法。将非结构化的数据表示为向量存入向量数据库，向量检索通过计算查询向量与数据库中存储的向
```

```
量的相似度来找到目标向量。\n',
    'startPos': 784,
    'endPos': 876,
    'pre': [
      '### 什么是 AI 中的向量表示？\n当我们处理非结构化数据时，需要将其转换为计算机可以理解和处理的形式。向量表示是一种将非结构化数据转换为嵌入向量的技术，通过多维度向量数值表述某个对象或事物的属性或者特征。腾讯云向量数据库提供的模型能力，目前在开发调试中。\n'
    ],
    'next': [
      '## 为什么是腾讯云向量数据库？\n腾讯云向量数据库作为一种专门存储和检索向量数据的服务提供给用户， 在高性能、高可用、大规模、低成本、简单易用、稳定可靠、智能运维等方面体现出显著优势。 \n'
    ]
  },
  'documentSet': {
    'documentSetId': '1192727327592550400',
    'documentSetName': '腾讯云向量数据库.md',
    'author': 'Tencent'
  }
}
```

Java

Java SDK 通过 search() 接口进行相似性检索。

```
// link database， client 为 VectorDBClient() 创建的客户端对象
AIDatabase db = client.aiDatabase("db-test-ai");
// link collectionView
CollectionView collection = db.describeCollectionView("coll-ai-files");
// 设置查询参数
// Content 配置需检索的文本信息
// DocumentSetNames 指定需要查找的文件名，可批量设置
SearchByContentsParam searchByContentsParam =
SearchByContentsParam.newBuilder()
            .withContent("向量是指什么")
            .withDocumentSetNames(Arrays.asList("腾讯云向量数据库.md"))
            .build();
// 根据配置的查询条件进行内容检索
List<Document> searchRes = collection.search(searchByContentsParam);
// 输出检索结果
int i = 0;
```

```
for (Document doc : searchRes) {
    System.out.println("\tres" +(i++)+": "+ doc.toString());
}
```

检索结果，如下所示。

关键信息	子参数	参数含义
score	-	• 表示查询向量与检索结果向量之间的相似性计算分数。 • 输出结果按照相似程度得分由高到低逐一排列。
data	text	检索到的结果。


```
res0: {
  "score": 0.8938464522361755,
  "data": {
    "text": "### 什么是向量？\n向量是指在数学和物理中用来表示大小和方向的量。它由一组有序的数值组成，这些数值代表了向量在每个坐标轴上的分量。\n",
    "endPos": 508,
    "startPos": 441,
    "next": [
      "### 什么是非结构化数据？\n非结构化数据，是指图像、文本、音频等数据。与结构化数据相比，非结构化数据不遵循预定义模型或组织方式，通常更难以处理和分析。\n"
    ],
    "pre": [
      "## 关键概念\n如果您不熟悉向量数据库和相似性搜索领域，请优先阅读以下基本概念，便于您对向量数据库有一个初步的了解。\n"
    ]
  },
  "documentSet": {
    "documentSetName": "\"腾讯云向量数据库.md\"",
    "documentSetId": "\"11825250341588882816\"",
    "docFields": [
      {
        "name": "author",
        "value": "Tencent"
      },
      {
        "name": "tags",
        "value": [
          "Embedding",
          "向量",
          "AI"
        ]
```

```
    }
   ]
  }
}res1: {
  "score": 0.8378515839576721,
  "data": {
   "text": "### 什么是 AI 中的向量表示？\n当我们处理非结构化数据时，需要将其转换为计算机可以理解和处理的形式。向量表示是一种将非结构化数据转换为嵌入向量的技术，通过多维度向量数值表述某个对象或事物的属性或者特征。腾讯云向量数据库提供的模型能力，目前在开发调试中。\n",
   "endPos": 784,
   "startPos": 585,
   "next": [
    "### 什么是向量检索？\n向量检索是一种基于向量空间模型的信息检索方法。将非结构化的数据表示为向量存入向量数据库，向量检索通过计算查询向量与数据库中存储的向量的相似度来找到目标向量。\n"
   ],
   "pre": [
    "### 什么是非结构化数据？\n非结构化数据，是指图像、文本、音频等数据。与结构化数据相比，非结构化数据不遵循预定义模型或组织方式，通常更难以处理和分析。\n"
   ]
  },
  "documentSet": {
   "documentSetName": "\"腾讯云向量数据库.md\"",
   "documentSetId": "\"11825250341588882816\"",
   "docFields": [
    {
     "name": "author",
     "value": "Tencent"
    },
    {
     "name": "tags",
     "value": [
      "Embedding",
      "向量",
      "AI"
     ]
    }
   ]
  }
}res2: {
  "score": 0.8152828216552734,
  "data": {
   "text": "### 什么是向量检索？\n向量检索是一种基于向量空间模型的信息检索方法。将非结构化的数据表示为向量存入向量数据库，向量检索通过计算查询向量与数据库中存储的
```

```
向量的相似度来找到目标向量。\n",
    "endPos": 876,
    "startPos": 784,
    "next": [
     "## 为什么是腾讯云向量数据库？\n腾讯云向量数据库作为一种专门存储和检索向量数据的服务提供给用户， 在高性能、高可用、大规模、低成本、简单易用、稳定可靠、智能运维等方面体现出显著优势。 \n"
    ],
    "pre": [
     "### 什么是 AI 中的向量表示？\n当我们处理非结构化数据时，需要将其转换为计算机可以理解和处理的形式。向量表示是一种将非结构化数据转换为嵌入向量的技术，通过多维度向量数值表述某个对象或事物的属性或者特征。腾讯云向量数据库提供的模型能力，目前在开发调试中。\n"
    ]
   },
   "documentSet": {
    "documentSetName": "\"腾讯云向量数据库.md\"",
    "documentSetId": "\"11825250341588882816\"",
    "docFields": [
     {
      "name": "author",
      "value": "Tencent"
     },
     {
      "name": "tags",
      "value": [
       "Embedding",
       "向量",
       "AI"
      ]
     }
    ]
   }
}
```


Go

Go SDK 通过 Search() 接口进行相似性检索。

```
// 等待文件解析完成
time.Sleep(time.Second * 30)
searchRes, _ := coll.Search(ctx, tcvectordb.SearchAIDocumentSetsParams{
```

```
    Content: "向量是指什么",
    DocumentSetName: []string{"腾讯云向量数据库.md"},
})
for _, doc := range searchRes.Documents {
    log.Printf("document: %+v", doc)
}
```

检索结果，如下所示。

关键信息	子参数	参数含义
score	-	• 表示查询向量与检索结果向量之间的相似性计算分数。 • 输出结果按照相似程度得分由高到低逐一排列。
data	text	检索到的结果。

```
2024/01/05 16:18:22 document: {DatabaseName:db-test-ai
CollectionViewName:coll-ai-files DocumentSetId:1192743758593921024
DocumentSetName:腾讯云向量数据库.md Score:0.8473929762840271
SearchData:{Text:### 什么是向量?
向量是指在数学和物理中用来表示大小和方向的量。它由一组有序的数值组成，这些数值代表
了向量在每个坐标轴上的分量。
 StartPos:441 EndPos:508 Pre:[## 关键概念
如果您不熟悉向量数据库和相似性搜索领域，请优先阅读以下基本概念，便于您对向量数据库
有一个初步的了解。
] Next:[### 什么是非结构化数据?
非结构化数据，是指图像、文本、音频等数据。与结构化数据相比，非结构化数据不遵循预定
义模型或组织方式，通常更难以处理和分析。
]} ScalarFields:map[author:Tencent]}
2024/01/05 16:18:22 document: {DatabaseName:db-test-ai
CollectionViewName:coll-ai-files DocumentSetId:1192743758593921024
DocumentSetName:腾讯云向量数据库.md Score:0.7868989706039429
SearchData:{Text:### 什么是 AI 中的向量表示?
当我们处理非结构化数据时，需要将其转换为计算机可以理解和处理的形式。向量表示是一种
将非结构化数据转换为嵌入向量的技术，通过多维度向量数值表述某个对象或事物的属性或者
特征。腾讯云向量数据库提供的模型能力，目前在开发调试中。
 StartPos:585 EndPos:784 Pre:[### 什么是非结构化数据?
非结构化数据，是指图像、文本、音频等数据。与结构化数据相比，非结构化数据不遵循预定
义模型或组织方式，通常更难以处理和分析。
] Next:[### 什么是向量检索?
向量检索是一种基于向量空间模型的信息检索方法。将非结构化的数据表示为向量存入向量数
据库，向量检索通过计算查询向量与数据库中存储的向量的相似度来找到目标向量。
]} ScalarFields:map[author:Tencent]}
```

2024/01/05 16:18:22 document: {DatabaseName:db-test-ai
CollectionViewName:coll-ai-files DocumentSetId:1192743758593921024
DocumentSetName:腾讯云向量数据库.md Score:0.7792050242424011
SearchData:{Text:### 什么是向量检索?
向量检索是一种基于向量空间模型的信息检索方法。将非结构化的数据表示为向量存入向量数据库，向量检索通过计算查询向量与数据库中存储的向量的相似度来找到目标向量。
StartPos:784 EndPos:876 Pre:[### 什么是 AI 中的向量表示?
当我们处理非结构化数据时，需要将其转换为计算机可以理解和处理的形式。向量表示是一种将非结构化数据转换为嵌入向量的技术，通过多维度向量数值表述某个对象或事物的属性或者特征。腾讯云向量数据库提供的模型能力，目前在开发调试中。
] Next:[## 为什么是腾讯云向量数据库?
腾讯云向量数据库作为一种专门存储和检索向量数据的服务提供给用户， 在高性能、高可用、大规模、低成本、简单易用、稳定可靠、智能运维等方面体现出显著优势。

删除数据库

Python

Python SDK 通过 drop_ai_database() 接口删除数据库 **db-test-ai**。

```
client.drop_ai_database(database_name='db-test-ai')
```

Java

Java SDK 通过 dropAIDatabase() 删除数据库 **db-test-ai**。

```
client.dropAIDatabase("db-test-ai");
```

Go

Go SDK 通过 DropAIDatabase() 删除数据库 **db-test-ai**。

```
result, _ := client.DropAIDatabase(context.Background(), aiDatabase)
```