

# RAG

---

大模型的商业化落地挑战

RAG系统

RAG(Retrieval Augmented Generation)-检索增强生成LLM的局限性

为什么会用到RAG

RAG vs Fine-tuning

RAG概念

RAG系统工作流程图解

RAG系统的搭建流程

智能客服系统的几种思路

向量与Embeddings的定义

向量间的相似度计算

文档的加载与切割（基于文档的LLM回复系统搭建）

把文本切分成chunks

按照句子来切分

按照固定字符数切分

按固定字符数 结合overlapping window

递归方法 RecursiveCharacterTextSplitter

向量检索

关键字搜索

向量数据库

Pinecone

Milvus

Chroma

Faiss

如何选型向量数据库

chromadb演示

基于向量检索的RAG实现

各大平台RAG实现

阿里云-百炼RAG

智普RAG

医疗项目

混合检索



## 大模型的商业化落地挑战

- 针对B端、C端的效果
- 隐私问题
- 幻觉问题

## RAG系统

### RAG(Retrieval Augmented Generation)-检索增强生成LLM的局限性

将大模型应用于实际业务场景时会发现，通用的基础大模型基本无法满足我们的实际业务需求，主要有以下几方面原因

- LLM的知识不是实时的，不具备知识更新
- LLM可能不知道你私有的领域/业务知识
- LLM有时会在回答中生成看似合理但实际上错误的信息

## 为什么会用到RAG

- 1、提高准确性: 通过检索相关的信息, RAG可以提高生成文本的准确性。
- 2、减少训练成本: 与需要大量数据来训练的大型生成模型相比, RAG可以通过检索机制来减少所需的训练数据量, 从而降低训练成本。
- 3、适应性强: RAG模型可以适应新的或不断变化的数据。由于它们能够检索最新的信息, 因此在新数据和事件出现时, 它们能够快速适应并生成相关的文本。

## RAG vs Fine-tuning

RAG (检索增强生成) 是把内部的文档数据先进行embedding, 借助检索先获得大致的知识范围答案, 再结合prompt给到LLM, 让LLM生成最终的答案

Fine-tuning (微调) 是用一定量的数据集对LLM进行局部参数的调整, 以期望LLM更加理解我们的业务逻辑, 有更好的zero-shot能力。

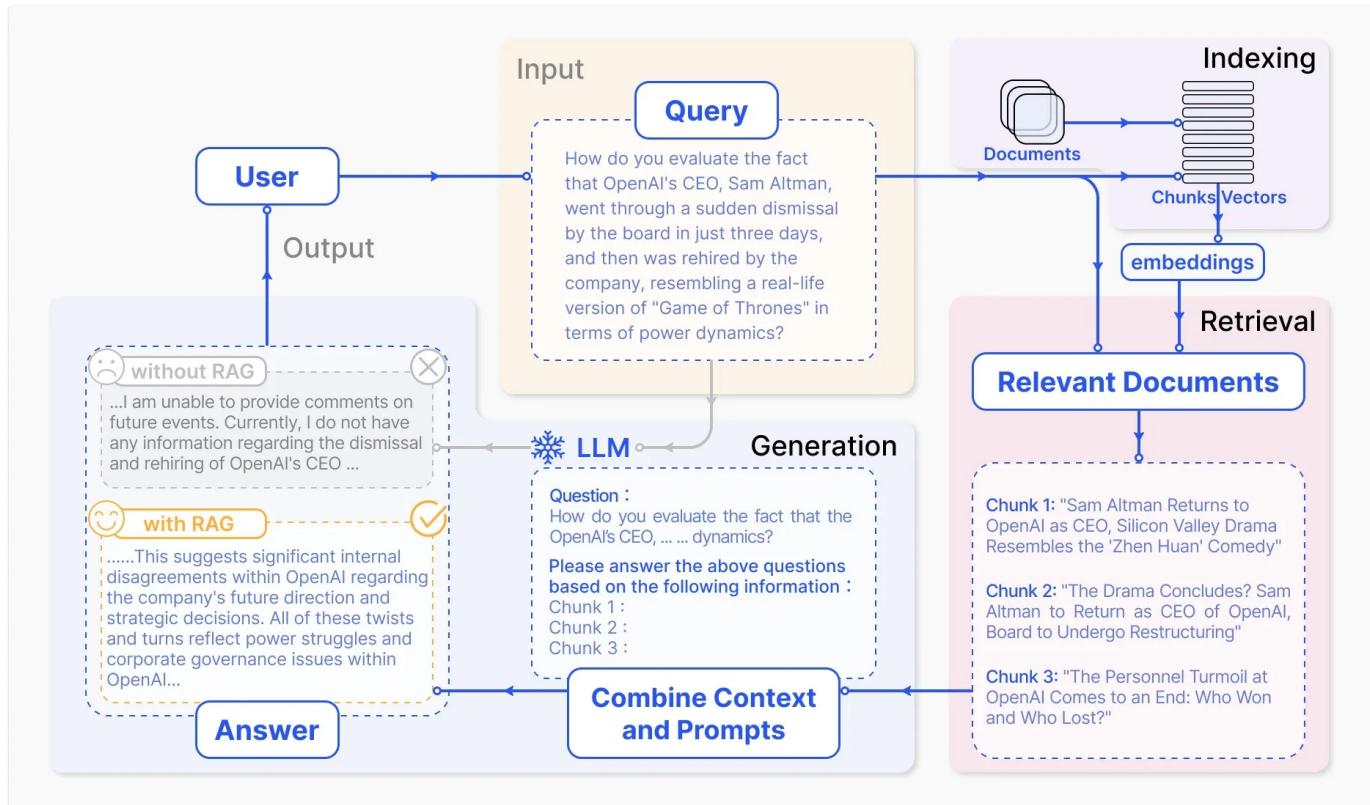
## RAG概念

RAG (Retrieval Augmented Generation) 顾名思义, 通过检索外部数据, 增强大模型的生成效果。

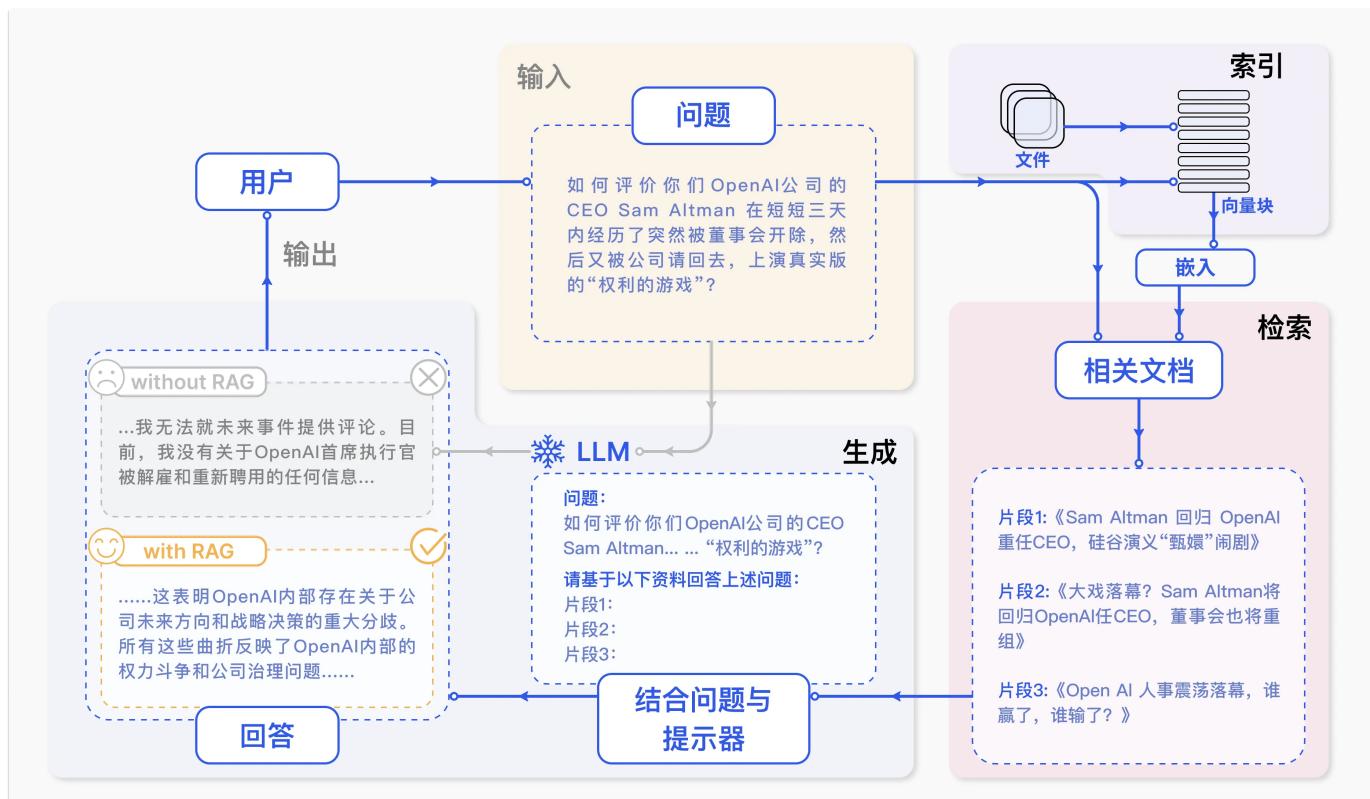
RAG即检索增强生成, 为LLM提供了从某些数据源检索到的信息, 并基于此修正生成的答案。RAG 基本上是Search + LLM 提示, 可以通过大模型回答查询, 并将搜索算法所找到的信息作为大模型的上下文。查询和检索到的上下文都会被注入到发送到 LLM 的提示语中。

# RAG系统工作流程图解

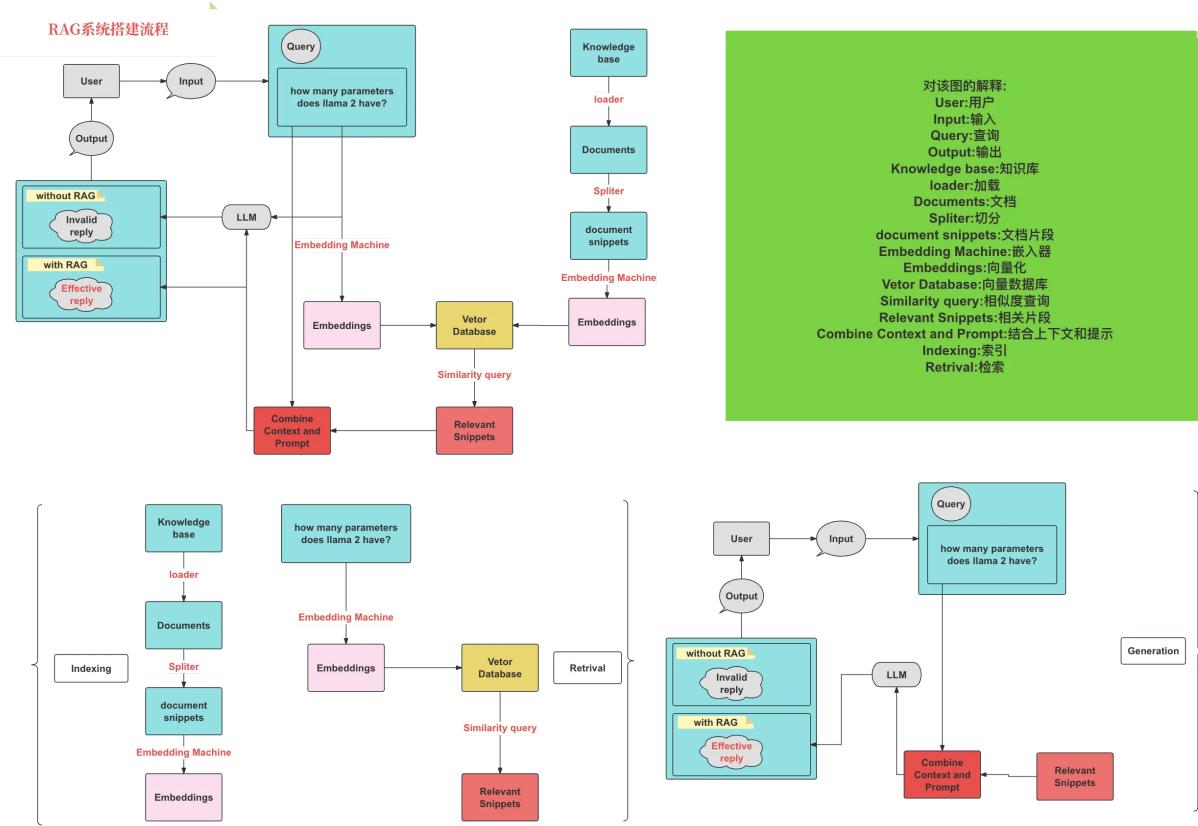
RAG论文：<https://arxiv.org/pdf/2312.10997>



中文版



# RAG系统的搭建流程



**索引 (Indexing) :** 索引首先清理和提取各种格式的原始数据，如 PDF、HTML、Word 和 Markdown，然后将其转换为统一的纯文本格式。为了适应语言模型的上下文限制，文本被分割成更小的、可消化的块（chunk）。然后使用嵌入模型将块编码成向量表示，并存储在向量数据库中。这一步对于在随后的检索阶段实现高效的相似性搜索至关重要。知识库分割成 chunks，并将 chunks 向量化至向量库中

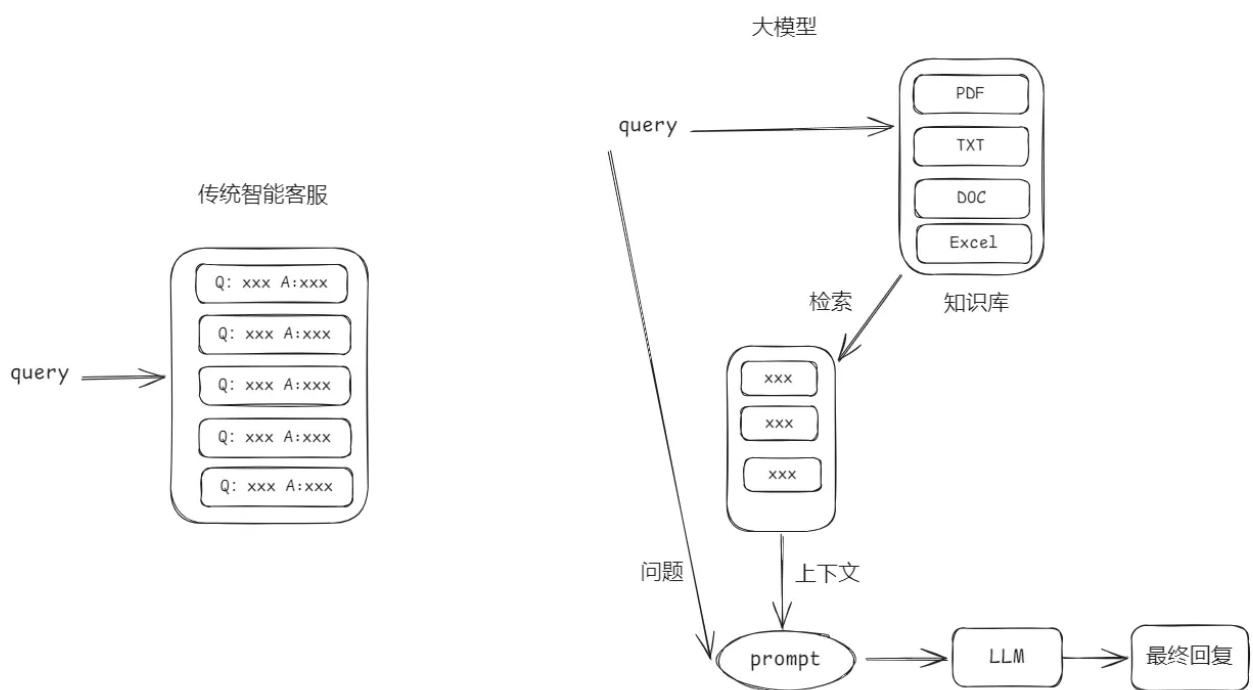
**检索 (Retrieval) :** 在收到用户查询 (Query) 后，RAG 系统采用与索引阶段相同的编码模型将查询转换为向量表示，然后计算索引语料库中查询向量与块向量的相似性得分。该系统优先级和检索最高 k (Top-K) 块，显示最大的相似性查询。

这些块随后被用作 prompt 中的扩展上下文。Query 向量化，匹配向量空间中相近的 chunks

RAG具体实现流程：加载文件 => 读取文本 => 文本分割 =>文本向量化 =>输入问题向量化 =>在文本向量中匹配出与问题向量最相似的 top k 个 =>匹配出的文本作为上下文和问题一起添加到 prompt 中 =>提交给 LLM 生成回答

## 智能客服系统的几种思路

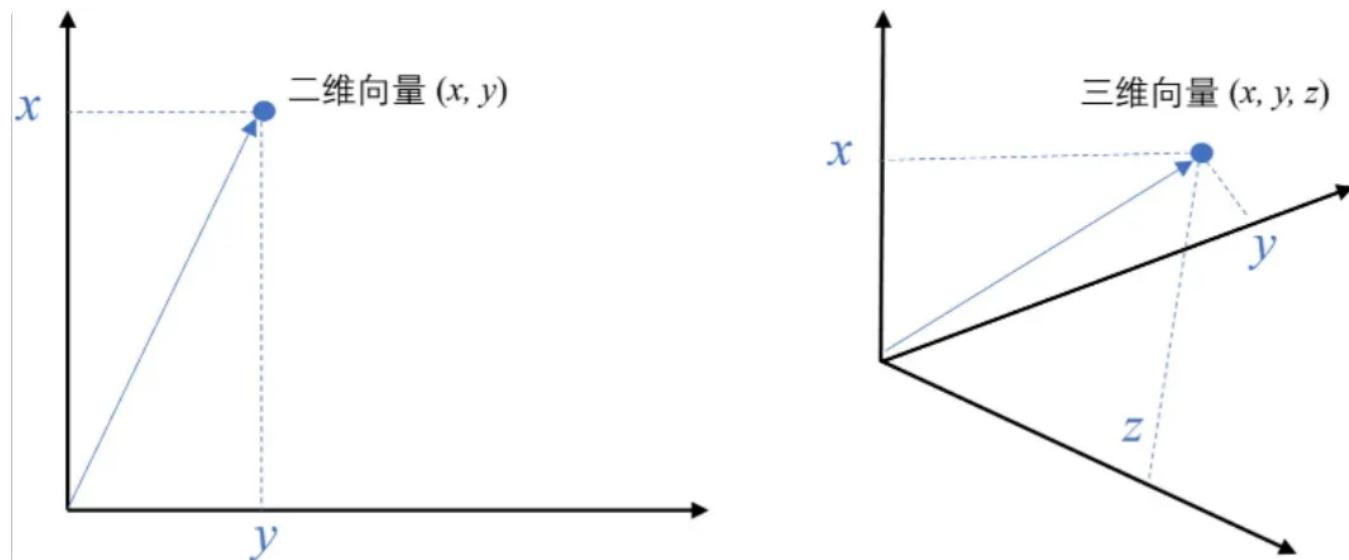
### 传统VS大模型



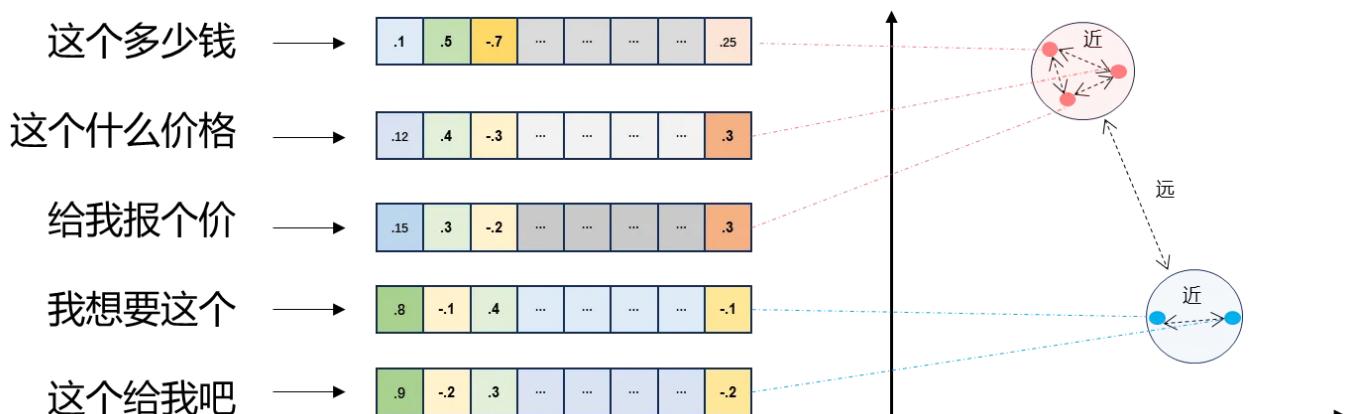
## 向量与Embeddings的定义

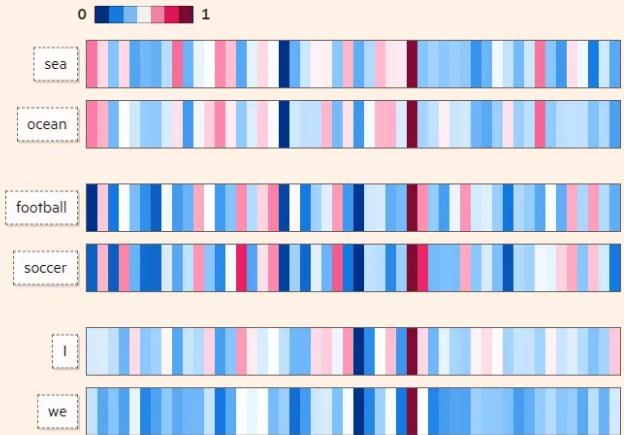
在数学中，向量（也称为欧几里得向量、几何向量），指具有大小（magnitude）和方向的量。它可以形象化地表示为带箭头的线段。箭头所指：代表向量的方向；线段长度：代表向量的大小。

例如，二维空间中的向量可以表示为  $(x, y)$ ，表示从原点  $(0,0)$  到点  $(x,y)$  的有向线段



1. 将文本转成一组浮点数：每个下标  $i$ ，对应一个维度
2. 整个数组对应一个  $n$  维空间的一个点，即文本向量又叫 Embeddings
3. 向量之间可以计算距离，距离远近对应语义相似度大小





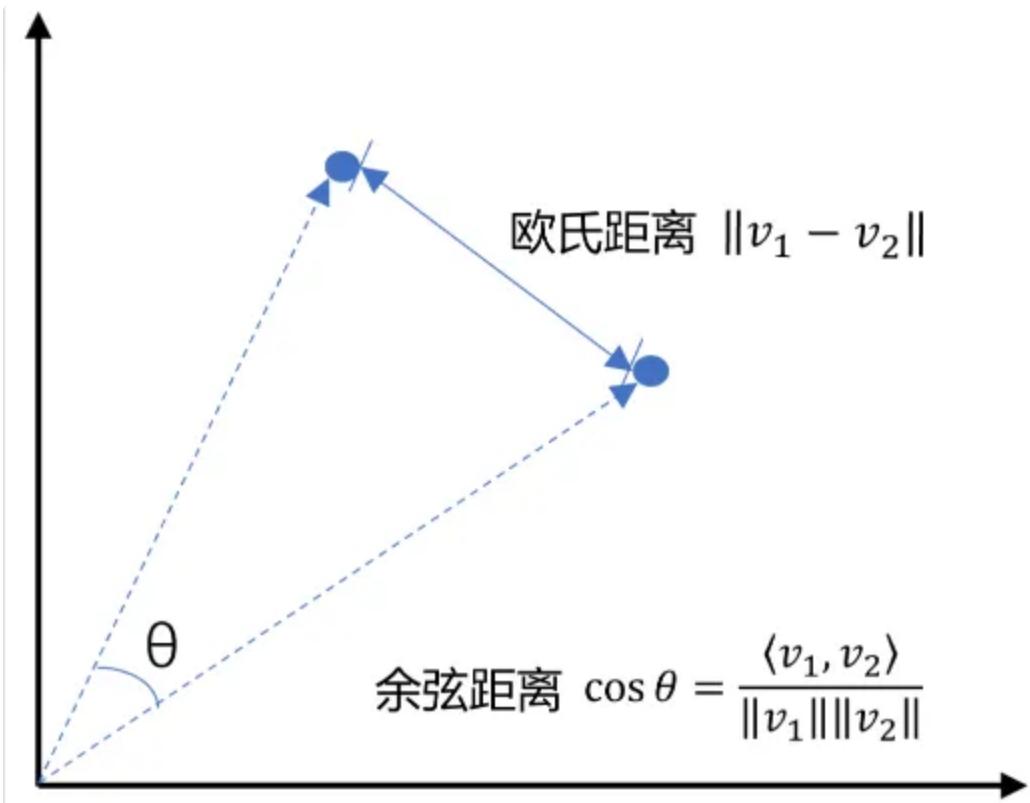
由于我们无法准确知道每个值代表什么，有意思的是，我们发现那些意思相近的单词，它们的词嵌入往往很像。

```

1  from openai import OpenAI
2  from dotenv import load_dotenv
3  load_dotenv()
4  import os
5
6  os.environ["http_proxy"] = "http://127.0.0.1:7897"
7  os.environ["https_proxy"] = "http://127.0.0.1:7897"
8
9  client = OpenAI()
10
11
12  def get_embeddings(texts, model="text-embedding-3-large"):
13      # texts 是一个包含要获取嵌入表示的文本的列表,
14      # model 则是用来指定要使用的模型的名称
15      # 生成文本的嵌入表示。结果存储在data中。
16      data = client.embeddings.create(input=texts, model=model).data
17      # print(data)
18      # 返回了一个包含所有嵌入表示的列表
19      return [x.embedding for x in data]
20
21
22  test_query = ["大模型"]
23
24  vec = get_embeddings(test_query)
25  # "大模型" 文本嵌入表示的列表。
26  # print(vec)
27  # "大模型" 文本的嵌入表示。
28  # print(vec[0])
29  # "大模型" 文本的嵌入表示的维度。3072
30  # print(len(vec[0]))

```

## 向量间的相似度计算

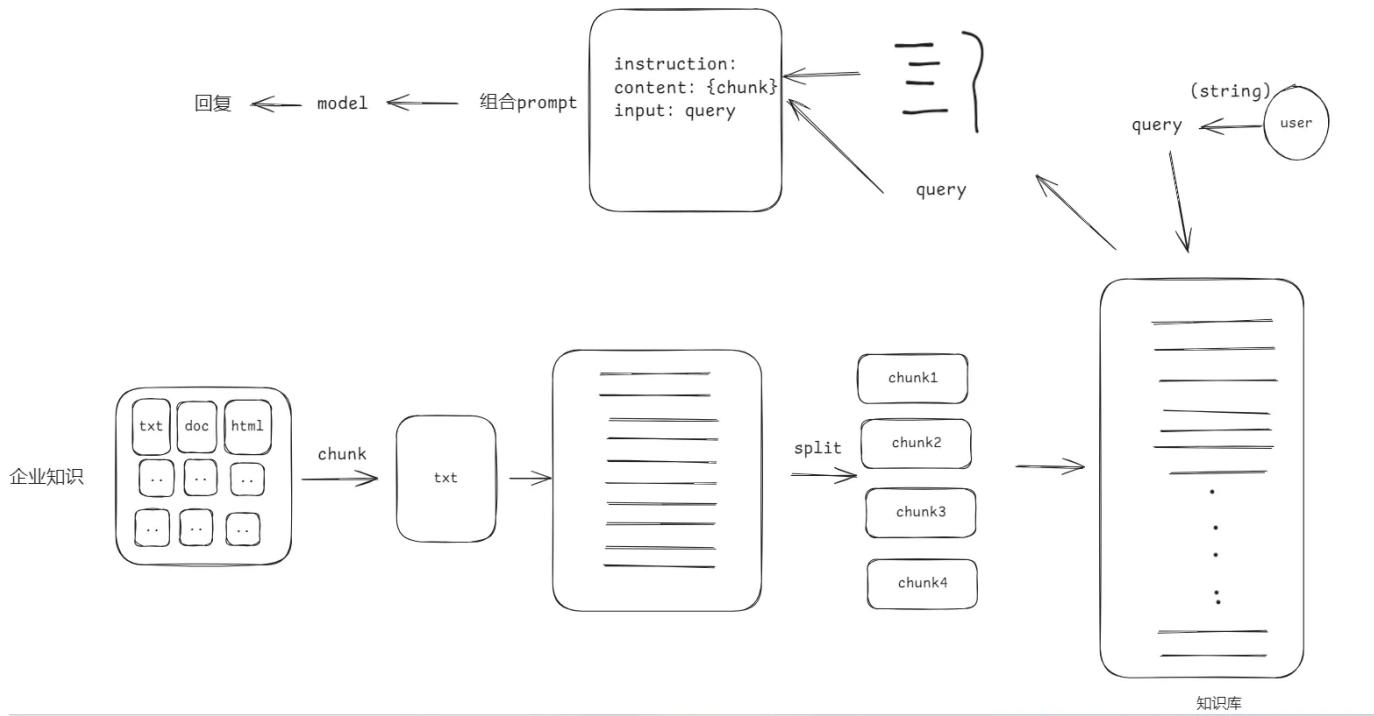


```
1  from openai import OpenAI
2  from dotenv import load_dotenv
3  import numpy as np
4  from numpy import dot
5  from numpy.linalg import norm
6  load_dotenv()
7  import os
8
9  os.environ["http_proxy"] = "http://127.0.0.1:7897"
10 os.environ["https_proxy"] = "http://127.0.0.1:7897"
11
12 client = OpenAI()
13
14 def cos_sim(a, b):
15     '''余弦距离 -- 越大越相似'''
16     return dot(a, b)/(norm(a)*norm(b))
17
18 def l2(a, b):
19     '''欧式距离 -- 越小越相似'''
20     x = np.asarray(a)-np.asarray(b)
21     return norm(x)
22
23 def get_embeddings(texts, model="text-embedding-3-large"):
24     data = client.embeddings.create(input=texts, model=model).data
25     # print(data)
26     # 返回了一个包含所有嵌入表示的列表
27     return [x.embedding for x in data]
```

且能支持跨语言

```
1 # 且能支持跨语言
2 query = "global conflicts"
3 # query = "国际争端"
4 documents = [
5     "联合国安理会上，俄罗斯与美国，伊朗与以色列“吵”起来了",
6     "土耳其、芬兰、瑞典与北约代表将继续就瑞典“入约”问题进行谈判",
7     "日本岐阜市陆上自卫队射击场内发生枪击事件 3人受伤",
8     "孙志刚被判死缓 减为无期徒刑后终身监禁 不得减刑、假释",
9     "以色列立法禁联合国机构，美表态担忧，中东局势再生波澜",
10 ]
11
12 query_vec = get_embeddings([query])[0]
13
14 doc_vecs = get_embeddings(documents)
15
16
17 print("Cosine distance:")
18 print(cos_sim(query_vec, query_vec))
19 for vec in doc_vecs:
20     print(cos_sim(query_vec, vec))
21
22
23 print("\nEuclidean distance:")
24 print(l2(query_vec, query_vec))
25 for vec in doc_vecs:
26     print(l2(query_vec, vec))
```

## 文档的加载与切割（基于文档的LLM回复系统搭建）



## 把文本切分成chunks

- 按照句子来切分
- 按照字符数来切分
- 按固定字符数 结合overlapping window
- 递归方法 RecursiveCharacterTextSplitter

### 按照句子来切分

```

1 import re
2
3 text = "自然语言处理（NLP），作为计算机科学、人工智能与语言学的交融之地，致力于赋予计算机解析和处理人类语言的能力。在这个领域，机器学习发挥着至关重要的作用。利用多样的算法，机器得以分析、领会乃至创造我们所理解的语言。从机器翻译到情感分析，从自动摘要到实体识别，NLP的应用已遍布各个领域。随着深度学习技术的飞速进步，NLP的精确度与效能均实现了巨大飞跃。如今，部分尖端的NLP系统甚至能够处理复杂的语言理解任务，如问答系统、语音识别和对话系统等。NLP的研究推进不仅优化了人机交流，也对提升机器的自主性和智能水平起到了关键作用。"
4
5 # 正则表达式匹配中文句子结束的标点符号
6 sentences = re.split(r'（。|？|！|……）', text)
7
8 # 重新组合句子和结尾的标点符号
9 chunks = [sentence + (punctuation if punctuation else '') for sentence, punctuation in zip(sentences[::2], sentences[1::2])]
10
11 for i, chunk in enumerate(chunks):
12     print(f"块 {i+1}: {len(chunk)}: {chunk}")

```

## 按照固定字符数切分

```

1 def split_by_fixed_char_count(text, count):
2     return [text[i:i+count] for i in range(0, len(text), count)]
3
4 # 假设我们按照每100个字符来切分文本
5 chunks = split_by_fixed_char_count(text, 100)
6
7 for i, chunk in enumerate(chunks):
8     print(f"块 {i+1}: {len(chunk)}: {chunk}")

```

## 按固定字符数 结合overlapping window

```

1 def sliding_window_chunks(text, chunk_size, stride):
2     return [text[i:i+chunk_size] for i in range(0, len(text), stride)]
3
4 chunks = sliding_window_chunks(text, 100, 50) # 100个字符的块, 步长为50
5
6 for i, chunk in enumerate(chunks):
7     print(f"块 {i+1}: {len(chunk)}: {chunk}")

```

## 递归方法 RecursiveCharacterTextSplitter

```

1 from langchain.text_splitter import RecursiveCharacterTextSplitter
2
3 text = """
4 自然语言处理（NLP），作为计算机科学、人工智能与语言学的交融之地，致力于赋予计算机解析和
5 处理人类语言的能力。在这个领域，机器学习发挥着至关重要的作用。利用多样的算法，机器得以分
6 析、领会乃至创造我们所理解的语言。从机器翻译到情感分析，从自动摘要到实体识别，NLP的应用
7 已遍布各个领域。随着深度学习技术的飞速进步，NLP的精确度与效能均实现了巨大飞跃。如今，部
8 分尖端的NLP系统甚至能够处理复杂的语言理解任务，如问答系统、语音识别和对话系统等。NLP的研
9 究推进不仅优化了人机交流，也对提升机器的自主性和智能水平起到了关键作用。
10 """
11
12 splitter = RecursiveCharacterTextSplitter(
13     chunk_size=50,
14     chunk_overlap=10,
15     length_function=len,
16 )
17
18 chunks = splitter.split_text(text)
19
20 for i, chunk in enumerate(chunks):
21     print(f"块 {i + 1}: {len(chunk)}: {chunk}")

```

## 向量检索

检索的方式有那些

列举两种：

1、关键字搜索：通过用户输入的关键字来查找文本数据。

2、语义搜索：不仅考虑关键词的匹配，还考虑词汇之间的语义关系，以提供更准确的搜索结果。

## 关键字搜索

先看一个最基础的实现

安装模块

```
▼ Python |  
1 pip install redis
```

Redis介绍：<https://www.cnblogs.com/almira998/p/17189227.html>

导入模块

```
▼ Python |  
1 import redis  
2 import json  
3 import os  
4 from dotenv import load_dotenv  
5 load_dotenv()  
6  
7 os.environ["http_proxy"] = "http://127.0.0.1:7897"  
8 os.environ["https_proxy"] = "http://127.0.0.1:7897"
```

连接Redis，存储数据，搜索数据

```
1 # 连接 Redis
2 r = redis.Redis(host='localhost', port=6379, decode_responses=True)
3
4 # 读取数据
5 with open('train_zh.json', 'r', encoding='utf-8') as f:
6     data = [json.loads(line) for line in f]
7
8 # 取出问题和输出数据
9 instructions = [entry['instruction'] for entry in data[0:1000]]
10 outputs = [entry['output'] for entry in data[0:1000]]
11 # print("instructions", instructions)
12 # print("outputs", outputs)
13
14 # 将数据存储到 Redis
15 for instruction, output in zip(instructions, outputs):
16     r.set(instruction, output) # 存入 Redis, 值序列化为 JSON
17
18 # 搜索函数: 根据关键字搜索 instruction 中包含该关键字的条目
19 def search_instructions(keyword, top_n=3):
20     # 通过模糊匹配
21     keys = r.keys(pattern="*" + keyword + "*")
22     data = []
23     for key in keys:
24         data.append(r.get(key))
25     return data[:top_n]
```

## 查看结果

```
1 result = search_instructions("白癜风")
```

## LLM 接口封装

```
1 from openai import OpenAI
2
3 client = OpenAI()
4
5 def get_completion(prompt, model="gpt-3.5-turbo"):
6     '''封装 openai 接口'''
7     messages = [{"role": "user", "content": prompt}]
8     response = client.chat.completions.create(
9         model=model,
10        messages=messages,
11        temperature=0, # 模型输出的随机性, 0 表示随机性最小
12    )
13    return response.choices[0].message.content
```

## Prompt模板

```
1 def build_prompt(prompt_template, **kwargs):
2     '''将 Prompt 模板赋值'''
3     prompt = prompt_template
4     for k, v in kwargs.items():
5         if isinstance(v, str):
6             val = v
7         elif isinstance(v, list) and all(isinstance(elem, str) for elem in
8             v):
9             val = '\n'.join(v)
10        else:
11            val = str(v)
12        prompt = prompt.replace(f"__{k.upper()}__", val)
13
14
15 prompt_template = """
16 你是一个问答机器人。
17 你的任务是根据下述给定的已知信息回答用户问题。
18 确保你的回复完全依据下述已知信息。不要编造答案。
19 如果下述已知信息不足以回答用户的问题，请直接回复"我无法回答您的问题"。
20
21 已知信息：
22 __INFO__
23
24 用户问：
25 __QUERY__
26
27 请用中文回答用户问题。
28 """
```

## RAG Pipeline初探

```
1 # user_query = "白癜风"
2 user_query = "得了白癜风，怎么办？"
3
4 # 1. 检索
5 search_results = search_instructions(user_query, 3)
6
7 # 2. 构建 Prompt
8 prompt = build_prompt(prompt_template, info=search_results, query=user_query)
9 print("==Prompt==")
10 print(prompt)
11
12 # 3. 调用 LLM
13 response = get_completion(prompt)
14
15 print("==回答==")
16 print(response)
```

## 向量数据库

在人工智能时代，向量数据库已成为数据管理和AI模型不可或缺的一部分。向量数据库是一种专门设计用来存储和查询向量嵌入数据的数据库。这些向量嵌入是AI模型用于识别模式、关联和潜在结构的关键数据表示。

随着AI和机器学习应用的普及，这些模型生成的嵌入包含大量属性或特征，使得它们的表示难以管理。这就是为什么数据从业者需要一种专门为处理这种数据而开发的数据库，这就是向量数据库的用武之地

### Pinecone

Pinecone: [www.pinecone.io/](http://www.pinecone.io/)

Pinecone的关键特性包括：

- 重复检测：帮助用户识别和删除重复的数据

- 排名跟踪：跟踪数据在搜索结果中的排名，有助于优化和调整搜索策略
- 数据搜索：快速搜索数据库中的数据，支持复杂的搜索条件
- 分类：对数据进行分类，便于管理和检索
- 去重：自动识别和删除重复数据，保持数据集的纯净和一致性

## Milvus

Milvus: [milvus.io/](https://milvus.io/)

Milvus的关键特性包括：

- 毫秒级搜索万亿级向量数据集
- 简单管理非结构化数据
- 可靠的向量数据库，始终可用
- 高度可扩展和适应性强
- 混合搜索
- 统一的Lambda结构
- 受到社区支持，得到行业认可

## Chroma

Chroma: [www.trychroma.com/](https://www.trychroma.com/)

Chroma的关键特性包括：

- 功能丰富：支持查询、过滤、密度估计等多种功能
- 即将添加的语言链（LangChain）、LlamaIndex等更多功能
- 相同的API可以在Python笔记本中运行，也可以扩展到集群，用于开发、测试和生产

## Faiss

Faiss:<https://github.com/facebookresearch/faiss>

Faiss的关键特性包括：

- 不仅返回最近的邻居，还返回第二近、第三近和第k近的邻居
- 可以同时搜索多个向量，而不仅仅是单个向量（批量处理）
- 使用最大内积搜索而不是最小欧几里得搜索
- 也支持其他距离度量，但程度较低。
- 返回查询位置附近指定半径内的所有元素（范围搜索）
- 可以将索引存储在磁盘上，而不仅仅是RAM中

## 如何选型向量数据库

在选择适合项目的向量数据库时，需要根据项目的具体需求、团队的技术背景和资源情况来综合评估。以下是一些建议和注意事项：

### 向量嵌入的生成

- 如果已经有了自己的向量嵌入生成模型，那么需要的是一个能够高效存储和查询这些向量的数据库
- 如果需要数据库服务来生成向量嵌入，那么应该选择提供这类功能的产品

### 延迟要求

- 对于需要实时响应的应用程序，低延迟是关键。需要选择能够提供快速查询响应的数据库
- 如果应用程序允许批量处理，那么可以选择那些优化了大批量数据处理的数据库

### 开发人员的经验

- 根据团队的技术栈和经验，选择一个易于集成和使用的数据库
- 如果团队成员对某些技术或框架更熟悉，那么选择一个能够与之无缝集成的数

据库会更有利

## chromadb演示

### 安装模块

```
▼ Python |  
1 pip install chromadb
```

```
1  from openai import OpenAI
2  import chromadb
3  from chromadb.config import Settings
4  from dotenv import load_dotenv
5  import json
6  load_dotenv()
7  import os
8
9  os.environ["http_proxy"] = "http://127.0.0.1:7897"
10 os.environ["https_proxy"] = "http://127.0.0.1:7897"
11
12 client = OpenAI()
13
14
15 def get_embeddings(texts, model="text-embedding-ada-002"):
16     '''封装 OpenAI 的 Embedding 模型接口'''
17     data = client.embeddings.create(input=texts, model=model).data
18     return [x.embedding for x in data]
19
20
21 with open('train_zh.json', 'r', encoding='utf-8') as f:
22     data = [json.loads(line) for line in f]
23
24
25 # print(data[0:100])
26 instructions = [entry['instruction'] for entry in data[0:1000]]
27 outputs = [entry['output'] for entry in data[0:1000]]
28
29
30 class MyVectorDBConnector:
31     def __init__(self, collection_name, embedding_fn):
32         chroma_client = chromadb.Client(Settings(allow_reset=True))
33
34         # 为了演示, 实际不需要每次 reset()
35         chroma_client.reset()
36
37         # 创建一个 collection
38         self.collection = chroma_client.get_or_create_collection(name=collection_name)
39         self.embedding_fn = embedding_fn
40
41     def add_documents(self, instructions, outputs):
42         '''向 collection 中添加文档与向量'''
43         # get_embeddings(instructions)
44         embeddings = self.embedding_fn(instructions)
```

```
45
46     self.collection.add(
47         embeddings=embeddings, # 每个文档的向量
48         documents=outputs, # 文档的原文
49         ids=[f"id{i}" for i in range(len(outputs))] # 每个文档的 id
50     )
51
52     # print(self.collection.count())
53
54
55     def search(self, query, top_n):
56         '''检索向量数据库'''
57         results = self.collection.query(
58             query_embeddings=self.embedding_fn([query]),
59             n_results=top_n
60         )
61         return results
62
63
64     # 创建一个向量数据库对象
65     vector_db = MyVectorDBConnector("demo", get_embeddings)
66
67     # 向向量数据库中添加文档
68     vector_db.add_documents(instructions, outputs)
69
70     # user_query = "白癜风"
71     user_query = "得了白癜风怎么办? "
72     results = vector_db.search(user_query, 2)
73     # print(results)
74
75     for para in results['documents'][0]:
76         print(para + "\n")
```

## 基于向量检索的RAG实现

```
1  from dotenv import load_dotenv
2  load_dotenv()
3  from openai import OpenAI
4  import chromadb
5  from chromadb.config import Settings
6  import os
7  import json
8
9  os.environ["http_proxy"] = "http://127.0.0.1:7897"
10 os.environ["https_proxy"] = "http://127.0.0.1:7897"
11
12 client = OpenAI()
13
14 prompt_template = """
15 你是一个问答机器人。
16 你的任务是根据下述给定的已知信息回答用户问题。
17 确保你的回复完全依据下述已知信息。不要编造答案。
18 如果下述已知信息不足以回答用户的问题，请直接回复"我无法回答您的问题"。
19
20 已知信息：
21 __INFO__
22
23 用户问：
24 __QUERY__
25
26 请用中文回答用户问题。
27 """
28
29
30 with open('train_zh.json', 'r', encoding='utf-8') as f:
31     data = [json.loads(line) for line in f]
32
33
34 # print(data[0:100])
35 instructions = [entry['instruction'] for entry in data[0:1000]]
36 outputs = [entry['output'] for entry in data[0:1000]]
37
38 def get_completion(prompt, model="gpt-3.5-turbo"):
39     '''封装 openai 接口'''
40     messages = [{"role": "user", "content": prompt}]
41     response = client.chat.completions.create(
42         model=model,
43         messages=messages,
44         temperature=0, # 模型输出的随机性，0 表示随机性最小
45     )
```

```

46         return response.choices[0].message.content
47
48     def build_prompt(prompt_template, **kwargs):
49         '''将 Prompt 模板赋值'''
50         prompt = prompt_template
51         for k, v in kwargs.items():
52             if isinstance(v, str):
53                 val = v
54             elif isinstance(v, list) and all(isinstance(elem, str) for elem in v):
55                 val = '\n'.join(v)
56             else:
57                 val = str(v)
58         prompt = prompt.replace(f"__{k.upper()}__", val)
59     return prompt
60
61     class MyVectorDBConnector:
62         def __init__(self, collection_name, embedding_fn):
63             chroma_client = chromadb.Client(Settings(allow_reset=True))
64
65             # 为了演示, 实际不需要每次 reset()
66             chroma_client.reset()
67
68             # 创建一个 collection
69             self.collection = chroma_client.get_or_create_collection(name=collection_name)
70             self.embedding_fn = embedding_fn
71
72         def add_documents(self, instructions, outputs):
73             '''向 collection 中添加文档与向量'''
74             embeddings = self.embedding_fn(instructions)
75
76             if len(embeddings) != len(instructions) or len(instructions) != len(outputs):
77                 raise ValueError("嵌入向量、instructions 和 outputs 数量不一致")
78
79             self.collection.add(
80                 embeddings=embeddings, # 每个文档的向量
81                 documents=outputs, # 文档的原文
82                 ids=[f"id{i}" for i in range(len(outputs))] # 每个文档的 id
83             )
84
85         def search(self, query, top_n):
86             '''检索向量数据库'''
87             results = self.collection.query(
88                 query_embeddings=self.embedding_fn([query]),
89                 n_results=top_n
90             )

```

```
91         return results
92
93     def get_embeddings(texts, model="text-embedding-3-large"):
94         '''封装 OpenAI 的 Embedding 模型接口'''
95         data = client.embeddings.create(input=texts, model=model).data
96         return [x.embedding for x in data]
97
98     # 创建一个向量数据库对象
99     vector_db = MyVectorDBConnector("demo", get_embeddings)
100
101    # 向向量数据库中添加文档
102    vector_db.add_documents(instructions, outputs)
103
104    class RAG_Bot:
105        def __init__(self, vector_db, llm_api, n_results=2):
106            self.vector_db = vector_db
107            self.llm_api = llm_api
108            self.n_results = n_results
109
110        def chat(self, user_query):
111            # 1. 检索
112            search_results = self.vector_db.search(user_query, self.n_results
113            )
114
115            # 2. 构建 Prompt
116            prompt = build_prompt(
117                prompt_template, info=search_results['documents'][0], query=u
118                ser_query)
119                # print("*50)
120                # print(prompt)
121                # print("*50)
122            # 3. 调用 LLM
123            response = self.llm_api(prompt)
124            return response
125
126    # 创建一个RAG机器人
127    bot = RAG_Bot(
128        vector_db,
129        llm_api=get_completion
130    )
131
132    user_query = "拉肚子怎么办? "
133
134    response = bot.chat(user_query)
135
136    print(response)
```

# 各大平台RAG实现

## 阿里云-百炼RAG

创建应用->上传数据->知识索引

The screenshot shows the Alibaba Cloud BaiLian application management interface. On the left, there is a sidebar with various navigation options:

- 默认业务空间 (Default Business Space)
- 首页 (Home)
- 模型中心-灵积 (Model Center - Lingji)
- 模型广场 (Model Square)
- 模型体验 (Model Experience) - expanded, showing: 文本模型 (Text Model), 语音模型 (Voice Model), 视觉模型 (Visual Model)
- 模型工具 (Model Tools) - expanded, showing: 应用中心 (Application Center), 应用广场 (Application Square), 我的应用 (My Applications) - highlighted with a red box, 应用组件 (Application Components), 数据中心 (Data Center), 数据管理 (Data Management) - highlighted with a red box, 数据处理 (Data Processing), 数据应用 (Data Application) - highlighted with a red box, 知识索引 (Knowledge Index), 模型数据 (Model Data), and 应用观测 (Application Observation).

The main area is titled "我的应用" (My Applications) and describes it as a centralized application management center. It features tabs for "应用列表" (Application List) and "应用模板" (Application Templates), with "应用列表" currently selected. Below are filters for "全部" (All), "智能体应用" (Agent Application), "工作流应用" (Workflow Application), and "智能体编排应用" (Agent Orchestration Application). A specific application entry for "百炼RAG" is shown, labeled as a "RAG智能体应用" (RAG Agent Application) that has been "已发布" (Published). The application details include:

- 应用ID: 04c8bff4d4084e509c75d89b81bc5b1c
- 选用模型: qwen2.5-72b-instruct

Below the details are buttons for "管理" (Manage), "调用" (Invoke), and "更多" (More).

## Python调用

Python |

```
1  from http import HTTPStatus
2  from dashscope import Application
3  from dotenv import load_dotenv
4  load_dotenv()
5  import os
6
7  def call_agent_app():
8      response = Application.call(app_id='04c8bff4d4084e509c75d89b81bc5b1c',
9                                   prompt='llama2有多少参数',
10                                  api_key=os.getenv("ALIYUN_API_KEY"),)
11
12     if response.status_code != HTTPStatus.OK:
13         print('request_id=%s, code=%s, message=%s\n' % (response.request_id,
14                                                       response.status_code, response.message))
15     else:
16         print('request_id=%s\n output=%s\n usage=%s\n' % (response.request_id,
17                                                       response.output, response.usage))
18
19 if __name__ == '__main__':
20     call_agent_app()
```

## 智普RAG

创建应用->上传数据

The screenshot shows the BigModel control console interface. At the top, there is a navigation bar with the logo 'BigModel', '控制台' (Control Console), and '文档' (Documentation). On the left, a sidebar lists various sections: '概览' (Overview), '体验中心' (Experience Center), '模型中心' (Model Center), '模型广场' (Model Square), '私有实例' (Private Instance), '模型微调' (Model Fine-tuning), '批量处理' (Batch Processing), '应用中心' (Application Center), '基础配置' (Basic Configuration) which is highlighted with a grey background, and '知识库' (Knowledge Base). The main area is titled '我的应用' (My Applications) and displays a card for a 'RAG' application. The card features a blue icon of a robot head, the text 'RAG', and a button labeled '问答机器人' (Question Answering Robot). Below the card are four buttons: '体验' (Experience), '分享' (Share), '编辑' (Edit), and '删除' (Delete).

## 医疗项目

- 将向量化的结果保存
  - huggingface: <https://huggingface.co/models>
  - 国内的镜像: <https://hf-mirror.com/>
- 读取向量化的结果

## 混合检索

混合搜索结合了两种检索信息的方法

- 词法搜索 (BM25) : 这种传统方法根据精确的关键字匹配来检索文档。例如，如果您搜索“cat on the mat”，它将找到包含这些确切单词的文档。

- 基于嵌入的搜索（密集检索）：这种较新的方法通过比较文档的语义来检索文档。查询和文档都被转换为高维向量（嵌入），系统检索其含义（向量表示）最接近查询的文档。

通过结合这两种方法，混合搜索可以提供更好的结果。它利用基于关键字的 BM25 的精度和密集检索的语义理解，确保系统根据所使用的单词及其含义找到最相关的文档。

将 BM25 与上下文嵌入相结合的关键优势在于，它们各自的强项能够互补：

- BM25：擅长精确匹配关键词，适合特定术语至关重要的场景。
- 基于嵌入的检索：即使查询中没有确切关键词，也能够理解深层语义，捕捉意图。

这种组合让 RAG 系统能够检索到既包含正确关键词、又符合查询意图的文档，从而显著提升生成内容的质量。

实现混合搜索，在此示例中，我们将使用rank\_bm25库来实现词法搜索：

```
1  from rank_bm25 import BM25Okapi
2  from nltk.tokenize import word_tokenize
3  import jieba
4  import json
5
6  # Sample documents
7  # documents = ["The cat sat on the mat.", "The dog barked at the moon.",
8  #               "The sun is shining bright."]
9  with open('train_zh.json', 'r', encoding='utf-8') as f:
10     data = [json.loads(line) for line in f]
11
12    # print(data[0:100])
13    # Extract instructions and outputs
14    instructions = [entry['instruction'] for entry in data[0:1000]]
15    outputs = [entry['output'] for entry in data[0:1000]]
16
17    # English
18    # tokenized_corpus = [word_tokenize(doc.lower()) for doc in documents]
19    # Chinese
20    tokenized_corpus = [jieba.lcut(doc) for doc in instructions]
21    # print(tokenized_corpus)
22
23    bm25 = BM25Okapi(tokenized_corpus)
24    # print(bm25)
25
26    query = "牙齿黄，怎么办"
27    # English
28    # tokenized_query = word_tokenize(query.lower())
29    tokenized_query = jieba.lcut(query)
30
31    bm25_scores = bm25.get_scores(tokenized_query)
32    bm25_results = bm25.get_top_n(tokenized_query, outputs, n=3)
33
34    print("BM25 Results: ", bm25_results)
```

## 向量检索

```
1  from openai import OpenAI
2  import chromadb
3  from chromadb.config import Settings
4  from dotenv import load_dotenv
5  import json
6
7  load_dotenv()
8  import os
9
10 os.environ["http_proxy"] = "http://127.0.0.1:7897"
11 os.environ["https_proxy"] = "http://127.0.0.1:7897"
12
13 client = OpenAI()
14
15
16 def get_embeddings(texts, model="text-embedding-ada-002"):
17     '''封装 OpenAI 的 Embedding 模型接口'''
18     data = client.embeddings.create(input=texts, model=model).data
19     return [x.embedding for x in data]
20
21
22 class MyVectorDBConnector:
23     def __init__(self, collection_name, embedding_fn):
24         chroma_client = chromadb.Client(Settings(allow_reset=True))
25
26         # 为了演示, 实际不需要每次 reset()
27         chroma_client.reset()
28
29         # 创建一个 collection
30         self.collection = chroma_client.get_or_create_collection(name=collection_name)
31         self.embedding_fn = embedding_fn
32
33     def add_documents(self, instructions, outputs):
34         '''向 collection 中添加文档与向量'''
35         # get_embeddings(instructions)
36         embeddings = self.embedding_fn(instructions)
37
38         self.collection.add(
39             embeddings=embeddings, # 每个文档的向量
40             documents=outputs, # 文档的原文
41             ids=[f"id{i}" for i in range(len(instructions))] # 每个文档的
42             id
43         )
```

```

44         # print(self.collection.count())
45
46     def search(self, query, top_n):
47         '''检索向量数据库'''
48         results = self.collection.query(
49             query_embeddings=self.embedding_fn([query]),
50             n_results=top_n
51         )
52         return results
53
54
55 # 创建一个向量数据库对象
56 vector_db = MyVectorDBConnector("demo", get_embeddings)
57
58 # 向向量数据库中添加文档
59 # vector_db.add_documents(documents)
60 vector_db.add_documents(instructions, outputs)
61
62 results = vector_db.search(query, 3)
63 print("vector_db:", results['documents'][0])

```

为了执行混合搜索，我们结合了 BM25 和密集检索的结果。每种方法的分数均经过标准化和加权以获得最佳总体结果

```

1 import numpy as np
2
3 query_embedding = np.array(get_embeddings(query))
4 doc_embeddings = np.array(get_embeddings(instructions))
5
6 # Normalize BM25 and Dense retrieval scores
7 bm25_scores = np.array(bm25_scores)
8 bm25_scores_normalized = bm25_scores / np.max(bm25_scores)
9 dense_scores = np.linalg.norm(query_embedding - doc_embeddings, axis=1)
10 dense_scores_normalized = 1 - (dense_scores / np.max(dense_scores)) # Convert distances to similarity
11 # Combine the normalized scores (you can adjust the weights as needed)
12 combined_scores = 0.5 * bm25_scores_normalized + 0.5 * dense_scores_normalized
13 # Get the top documents based on combined scores
14 top_idx = combined_scores.argsort()[:-1]
15 # hybrid_results = [documents[i] for i in top_idx[:3]]
16 hybrid_results = [outputs[i] for i in top_idx[:3]]
17 print("Hybrid Search Results: ", hybrid_results)

```

