RAG 技术详解与实战应用

第12讲：实践：用缓存、异步与向量引擎加速你的RAG

目录

上节回顾

提升RAG启动效率

持久化存储

- 利用向量数据库实现RAG系统的数据持久化存储
- 了解了两种主流向量数据库Chroma、Milvus的特点、适用场景

内存、硬盘与数据库

- 了解内存与硬盘的关系
- 内存缓存机制
- 标量数据库与向量数据库

构建索引加速检索

索引的定义

- “以空间换时间”，加速数据检索的特殊数据结构
- 向量索引-高效的ANN搜索

Milvus

- 内置多种先进向量索引，支持复杂查询，助力企业级应用落地

系统工程优化

- 并行与并发
- 如何使用Python实现多任务并行

模型推理性能优化

模型推理性能指标

- 首字延迟、吞吐量

性能优化策略

- 模型蒸馏
- 更好的推理框架
- 参数精度→模型量化

目录

向量数据库实现持久化存储

为了防止程序重启导致数据丢失，使用**向量数据库，**将存储在**硬盘**中，以实现数据的**持久化存储**。LazyLLM 提供了多种存储组件同时支持基于**内存**和**向量数据库**两种存储方式，以供开发者灵活选择。

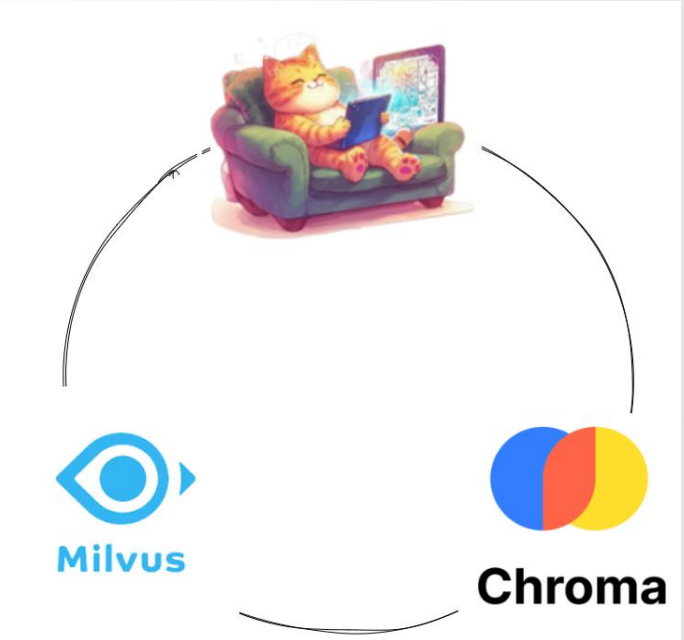
组件名	存储介质	功能描述	适用场景
MapStore	内存	基于内存的存储，提供节点存储的基本能力，能以最快的速度实现数据读写	技术探索阶段，将少量文档处理数据保存在内存中，以技术选型与算法迭代的快速验证
ChromadbStore	硬盘	使用 Chroma向量数据库，实现数据的持久化存储。	轻量级实验，快速POC，缩短从构思到可运行原型的研发周期
MilvusStore	硬盘	使用Milvus向量数据库，实现数据的本地存储/分布式远程存储。	企业级部署，适用于数据体量大，对并发、检索性能都有较高要求的场景

LazyLLM支持的向量数据库

在基于LazyLLM实现RAG系统时，只需在实例化Document时传入存储配置，即可实现不同存储类型的切换。

doc = lazyllm.Document(dataset_path=...,..., **store_conf=store_conf**)

- **type：存储类型**
  - map - 基于内存kv存储
  - chroma - 使用轻量级向量数据库ChromaDB
  - milvus - 基于高性能向量数据库Milvus
- **kwargs：不同存储所需额外配置参数（字典的形式）**
  - chroma:
    - dir（必填）：存储数据目录
  - milvus:
    - uri（必填）：存储数据地址，文件路径或远程连接的url
    - index_kwargs（可选，dict/list[dict]）：内置索引配置，主要包含索引类型index_type和度量类型metric_type,支持指定不同向量模型使用不同索引配置。目前支持 稠密和稀疏向量，感兴趣的开发者可以前往官网查看详细参数



LazyLLM支持的向量数据库

目前ChromaDB仅作为数据的持久化存储使用，而Milvus在支持数据持久化存储的同时，也支持了多样内置向量索引的选择，以便开发者轻松实现高效的向量检索。

以下是使用不同存储时的store_conf示例：

MapStore

无需传入任何配置

ChromadbStore

```
chroma_store_conf = {
    'type': 'chroma',
    'kwargs': {
        'dir': 'dbs/chroma1',
    }
}
```

MilvusStore

```
milvus_store_conf = {
    'type': 'milvus',
    'kwargs': {
        'uri': 'dbs/milvus1.db',
        'index_kwargs': {
            'index_type': 'HNSW',
            'metric_type': 'COSINE',
        }
    },
}
```

请注意，目前 LazyLLM 支持 Milvus 2.4.x，请注意您的 pymilvus >= 2.4.11 , milvus-lite == 2.4.10

实践：系统启动性能对比

分别使用内存、ChromaDB和Milvus作为数据存储，测试三者首次启动和二次启动耗时，以展示持久化存储对于系统二次启动效率的提升。

```
    embedding_model = OnlineEmbeddingModule(                                    # 线上embedding模型
        source="doubao",
        embed_model_name="doubao-embedding-large-text-240915“
    )
1.  def test_store(store_conf: dict=None):                                      # 传入存储配置
        """接收存储配置，测试不同配置下系统启动性能"""
2.      st1 = time.time()

3.      docs = lazyllm.Document(dataset_path=DOC_PATH, embed=embedding_model, store_conf=store_conf)   # 实例化Document，传入存储配置
4.      docs.create_node_group(name=‘sentence’, parent=“MediumChunk”, transform=lambda x: x.split(‘。’))  # 创建按句子划分节点组

5.      if store_conf and store_conf.get(‘type’) == “milvus”:                   # milvus配置中已指定度量类型
6.          retriever1 = lazyllm.Retriever(docs, group_name="sentence", topk=3)  # 无需similarity参数
7.      else:
8.          retriever1 = lazyllm.Retriever(docs, group_name="sentence", similarity='cosine', topk=3)   # similariy=cosine，使用向量检索

9.      retriever1.start()                                                      # 启动检索器，完成即系统启动完毕
10.     et1 = time.time()

11.     st2 = time.time()
12.     res = retriever1("牛车水")                                               # 测试单次检索耗时
13.     et2 = time.time()
14.     nodes = "\n======\n".join([node.text for node in res])
15.     msg = f“Init time: {et1 - st1}, retrieval time: {et2 - st2}s\n”          # 输出系统启动耗时、检索耗时
16.     LOG.info(msg)
17.     LOG.info(nodes)
18.     return msg
```

实践：系统启动性能对比

```
   # chroma db存储配置
1. chroma_store_conf = {
       'type': 'chroma’,
       'kwargs': {
           'dir': 'dbs/chroma1’,
       }
   }
   # milvus存储配置
2. milvus_store_conf = {
       'type': 'milvus',
       'kwargs': {
           'uri': 'dbs/milvus1.db',
           'index_kwargs': {
               'index_type': 'HNSW',
               'metric_type': 'COSINE',
           }
       },
   }

   # 测试集，依次测试使用内存、chroma、milvus时的系统启动性能
3. test_conf = {
       "map": None,
       "chroma": chroma_store_conf,
       "milvus": milvus_store_conf
   }
4. start_times = ""
5. for store_type, store_conf in test_conf.items():
       # 调用测试函数
6.     res = test_store(store_conf=store_conf)
7.     start_times += f"Store type: {store_type}: {res}"
8. print(start_times)
```

```
Store type: map: Init time: 17.713993787765503, retrieval time: 0.3607618808746338s
Store type: chroma: Init time: 17.690227270126343, retrieval time: 0.36835598945617676s
Store type: milvus: Init time: 15.303590059280396, retrieval time: 0.022578477859497075
```

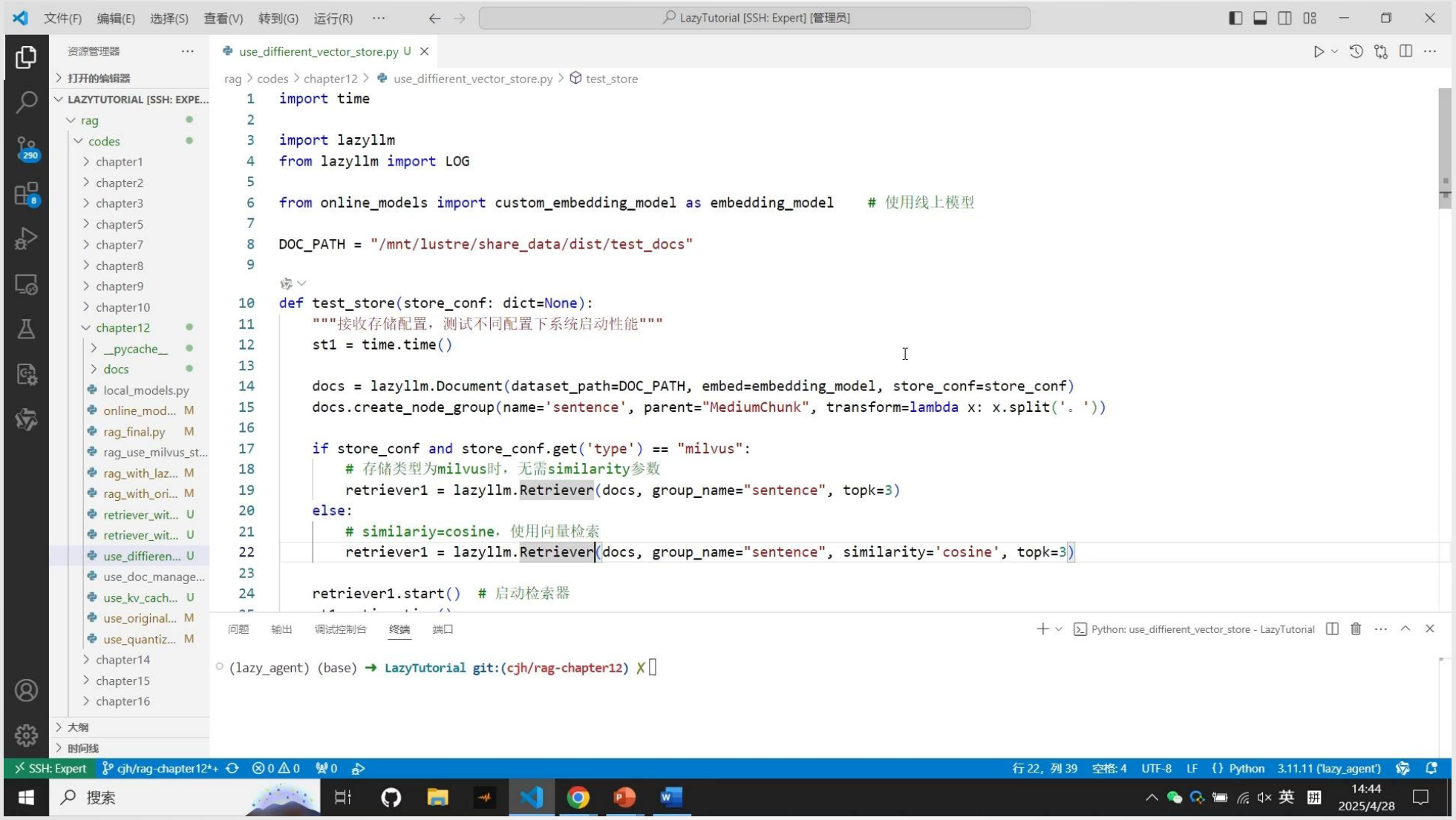
第一次启动

```
Store type: map: Init time: 17.303208112716675, retrieval time: 0.3642408847808838s
Store type: chroma: Init time: 13.890714883804321, retrieval time: 0.3575289249420166s
Store type: milvus: Init time: 1.8017945289611816, retrieval time: 0.020511150360107422s
```

第二次启动

存储类型	第一次启动/检索耗时（s）	第二次启动/检索耗时（s）
Map	17.71/0.36	17.30/0.36
ChromaDB	17.69/0.37	13.89（↓21.5%）/0.36
Milvus	15.30/0.02	1.80（↓88.2%）/0.02

实践：系统启动性能对比 – 实机演示



目录

高效索引加速数据检索

建立索引能够使检索器高效定位和检索相关信息，本部分我们将会学习：

·如何使用LazyLLM的索引组件 IndexBase 编写并应用字典树索引

·了解高性能向量数据库Milvus的原生使用方式

·学习如何在LazyLLM中使用Milvus向量数据库的向量索引

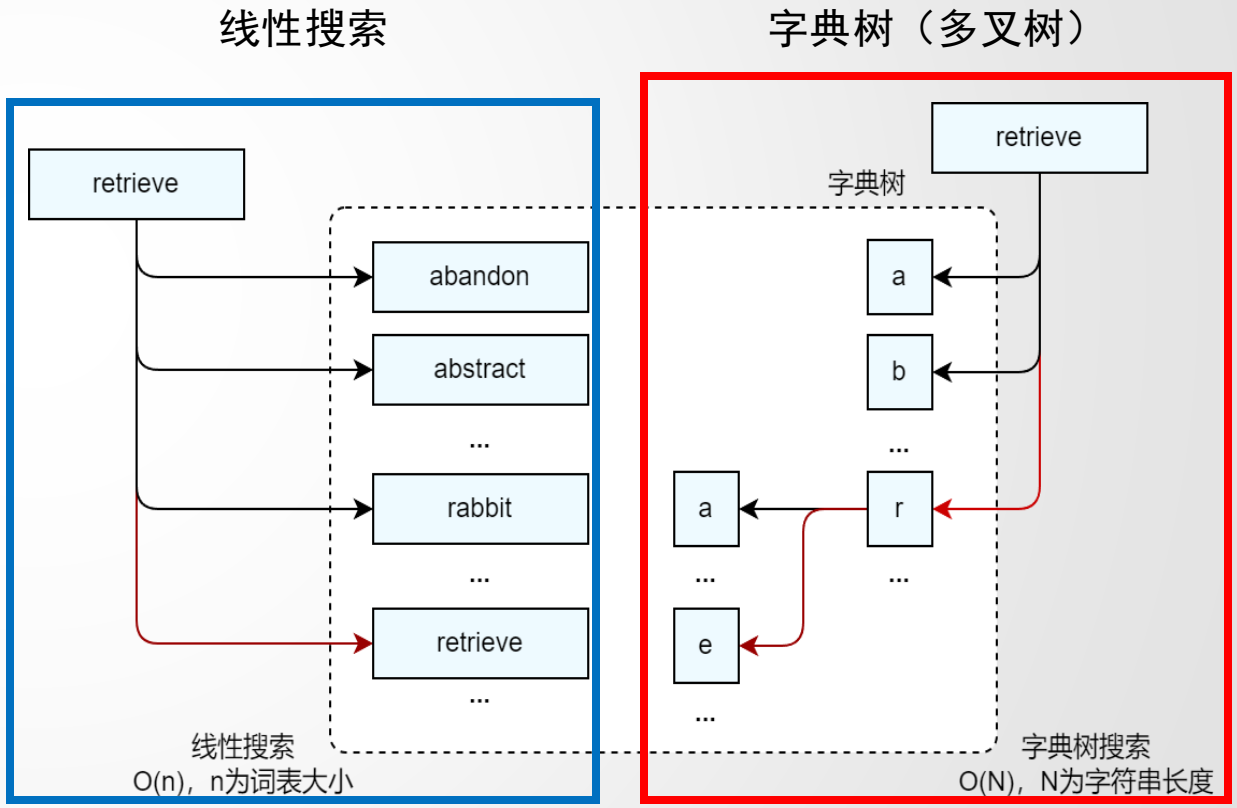
小试牛刀：用IndexBase构建字典树

LazyLLM索引组件的基类，所有索引均使用IndexBase组件创建，随后索引组件可被注册在Document当中，并在后续的检索过程中使用。

IndexBase需要实现的三个方法：

- update：接收节点列表，为节点建立/更新索引。
- remove：接收节点id和节点组名，在索引中删除对应节点。
- query：根据接收的查询以及相关参数，在索引中查询符合要求的节点。

注意：使用默认存储（MapStore）的情况下，节点的更新与删除会自动关联Index的更新与删除。



小试牛刀：用IndexBase构建字典树

```python
class TrieNode:
    def __init__(self):
        self.children: Dict[str, TrieNode] = {}
        self.is_end_of_word: bool = False
        self.uids: set[str] = set()
```

1. 定义字典树节点，包含叶节点字典集合、词尾判断、节点id集合

```python
class TrieTreeIndex(IndexBase):
    def __init__(self, store: 'StoreBase'):
        self.store = store
        self.root = TrieNode()
        self.uid_to_word: Dict[str, str] = {}
```

2. 字典树索引类：
· 继承IndexBase，接收store存储，用于对应节点查找
· 初始化方法中创建root节点作为检索入口

```python
    @override
    def update(self, nodes: List['DocNode']) -> None:
        if not nodes or nodes[0]._group != 'words':
            return
        for n in nodes:
            uid = n._uid
            word = n.text
            self.uid_to_word[uid] = word
            node = self.root
            for char in word:
                node = node.children.setdefault(char, TrieNode())
            node.is_end_of_word = True
            node.uids.add(uid)
```

3. 索引更新方法
· 索引仅用于特定节点组检索，故设定如果节点组名不是tree则不建立索引。
· 对于每个单词，从root节点进入，遍历每个字母：
如果当前字母不在当前节点叶子节点集合，则创建叶子结点，key为当前字母。
随后转移至改叶子节点，检查下一字母
直到字母遍历完毕，叶子节点位置即为词尾，标记并记录id

小试牛刀：用IndexBase构建字典树

```
@override
def remove(self, uids: List[str], group_name: Optional[str] = None) -> None:
    if group_name != 'words':
        return
    for uid in uids:
        word = self.uid_to_word.pop(uid, None)
        if not word:
            continue
        self._remove(self.root, word, 0, uid)

def _remove(self, node: TrieNode, word: str, index: int, uid: str) -> bool:
    if index == len(word):
        if uid not in node.uids:
            return False
        node.uids.remove(uid)
        node.is_end_of_word = bool(node.uids)
        return not node.children and not node.uids
    char = word[index]
    child = node.children.get(char)
    if not child:
        return False
    should_delete = self._remove(child, word, index + 1, uid)
    if should_delete:
        del node.children[char]
        return not node.children and not node.uids
    return False
```

1. 实现索引删除方法，从root入口，递归删除对应单词索引

2. 递归停止条件：到达词尾，删除索引

3.如果子节点已经可以删了，就把这个 char 对应的 child 从 children 里删掉。再判断当前节点是否也可以删（没有子节点且没有其他 uid）。

小试牛刀：用IndexBase构建字典树

```python
    @override
    def query(self, query: str, group_name: str, **kwargs) -> List[str]:
        node = self.root                                    # 查询时以根节点为入口节点
        for char in query:                                  # 顺序遍历query的每个字符
            node = node.children.get(char)                  # 若存在字符对应的叶子节点，则转移至叶子结点，直到完成遍历
            if node is None:
                return []
        return self.store.get_nodes(group_name=group_name, uids=list(node.uids)) if node.is_end_of_word else []
```

将自定义索引成功注册并使用：

```python
docs = lazyllm.Document(dataset_path=DOC_PATH, embed=embedding_model)
docs.create_node_group(name='words', transform=(lambda d: d.split('\r\n')))          # 创建节点组
docs.register_index("trie_tree", TrieTreeIndex, docs1.get_store())                   # 注册索引
retriever = lazyllm.Retriever(docs1, group_name="words", index="trie_tree", topk=1)  # 创建检索器，指定索引类型
retriever.start()                                                                    # 检索器初始化
```

注意：在注册自定义索引时，请关注使用的存储是否支持hook机制，以自动同步索引。例如 Milvus 这种专门做向量检索的数据库，本身架构不允许外部代码主动注册额外索引，故只能使用预定义索引。

小试牛刀：用IndexBase构建字典树

为了方便性能对比，同样定义线性搜索索引。

```
1. class LinearSearchIndex(IndexBase):
2.     def __init__(self):
3.         self.nodes = []
4.
5.     @override
6.     def update(self, nodes: List['DocNode']) -> None:
7.         if not nodes or nodes[0]._group != 'words’:
8.             return
9.         for n in nodes:
10.             self.nodes.append(n)                                    # 顺序加入每个单词
11.     @override
12.     def remove(self, uids: List[str], group_name: Optional[str] = None) -> None:
13.         if group_name != 'words’:
14.             return
15.         for uid in uids:
16.             for i, n in enumerate(self.nodes):
17.                 if n._uid == uid:
18.                     del self.nodes[i]
19.                     break
20.     @override
21.     def query(self, query: str, **kwargs) -> List[str]:
22.         res = []
23.         for n in self.nodes:                                        # 线性搜索：顺序遍历每个单词
24.             if n.text == query:
25.                 res.append(n)
26.                 break                                               # 假设每个单词出现一次，精准匹配
27.         return res
```

小试牛刀：用IndexBase构建字典树

我们准备一个庞大的词表，其中包含37w个英文字符串（大多数为英文单词）。

对比使用线性搜索与字典树索引的检索耗时：

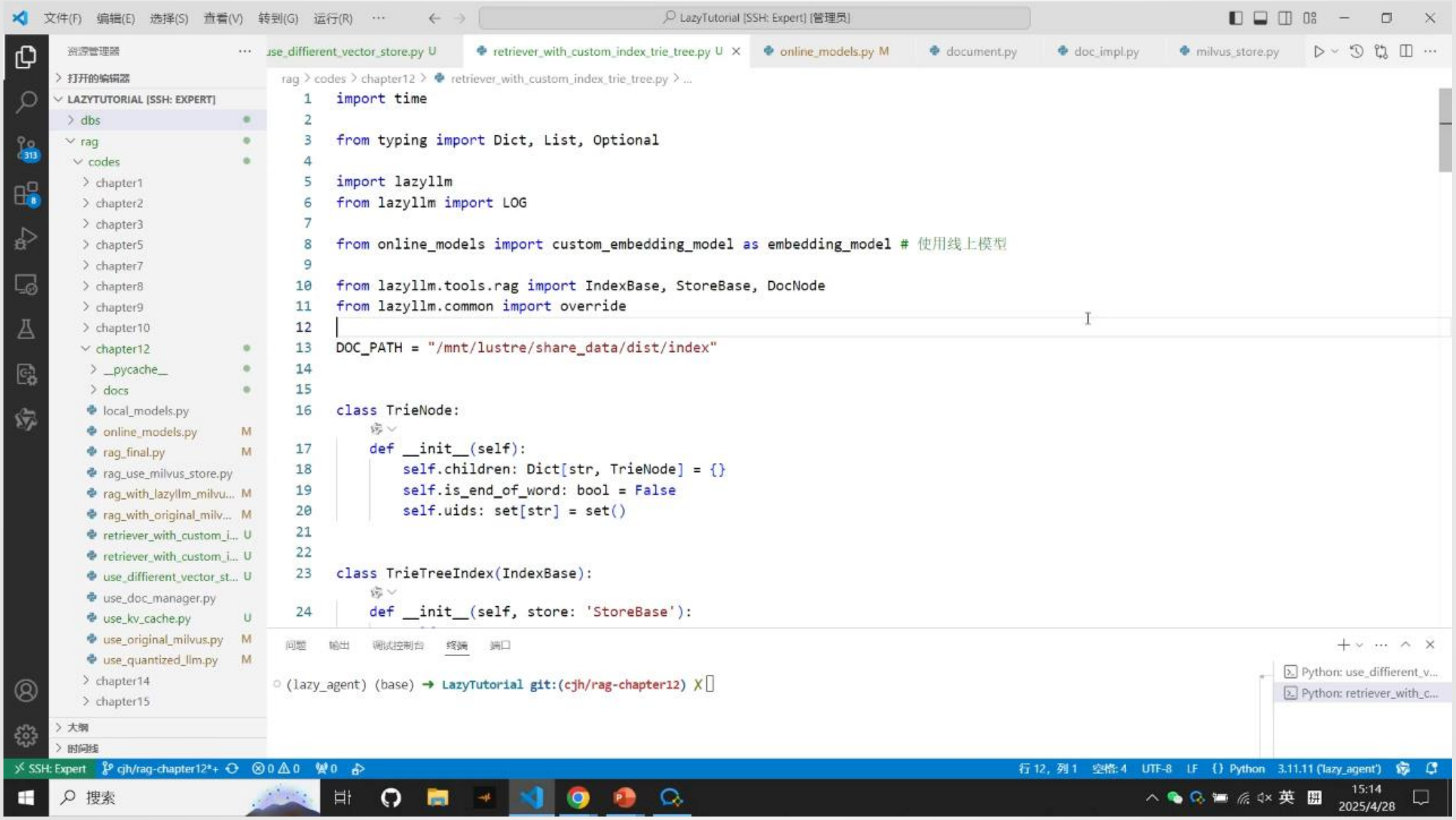
```
1. docs = lazyllm.Document(dataset_path=DOC_PATH, embed=embedding_model)
2. docs.create_node_group(name='words', transform=(lambda d: d.split('\r\n')))          # 创建节点组
3. docs.register_index("trie_tree", TrieTreeIndex, docs.get_store())                    # 注册索引
4. docs.register_index("linear_search", LinearSearchIndex)
5. retriever1 = lazyllm.Retriever(docs, group_name="words", index="linear_search", topk=1) # 创建检索器，指定索引类型
6. retriever2 = lazyllm.Retriever(docs, group_name="words", index="trie_tree", topk=1)
7. retriever1.start()                                                                   # 检索器初始化
8. retriever2.start()

9. for query in ["a", "lazyllm", "zwitterionic"]:                                       # 分别检索词表中靠前、中间、靠后单词
10.     st = time.time()
11.     res = retriever1(query)
12.     et = time.time()
13.     LOG.info(f"query: {query}, linear time: {et - st}, linear res: {res[0].text}")

14.     st = time.time()
15.     res = retriever2(query)
16.     et = time.time()
17.     LOG.info(f"query: {query}, trie time: {et - st}, trie res: {res[0].text}")
```

```
query: a, linear time: 6.175041198730469e-05, linear res: a
query: a, trie time: 7.152557373046875e-05, trie res: a
query: lazyllm, linear time: 0.0384678840637207, linear res: lazyllm
query: lazyllm, trie time: 0.00010895729064941406, trie res: lazyllm
query: zwitterionic, linear time: 0.08076953887939453, linear res: zwitterionic
query: zwitterionic, trie time: 0.00010061264038085938, trie res: zwitterionic
```

线性搜索：搜索时间随单词位置增加

字典树：搜索效率高且稳定

小试牛刀：用IndexBase构建字典树



使用Milvus内置向量索引

向量索引是向量数据库的核心能力之一，实际生产中往往使用向量数据库（如Milvus）实现高效向量检索。

我们首先了解一下本次的主角——Milvus的原生基础使用方法，包含Milvus的安装以及基本功能的使用（初始化、数据注入、数据检索）。

在本次实践中使用 Milvus Lite，它是pymilvus中包含的一个 python 库，可以嵌入到应用程序中。Milvus 还支持在Docker和Kubernetes上部署，适用于生产用例。

安装命令：pip install -U pymilvus

	索引方式	度量类型
浮点索引	FLAT IVF_FLAT,IVF_PQ,IVF_SQ8 HNSW, HNSW_PQ,SCANN ...	欧氏距离 L2 内积 IP 余弦相似度 COSINE
二进制索引	BIN_FLAT、BIN_IVF_FLAT	Jaccard (JACCARD) 哈明 (HAMMING)
稀疏索引	SPARSE_INVERTED_INDEX、 SPARSE_WAND	内积 IP

原生Milvus基本操作一览

设置向量数据库（实例化客户端以创建本地数据库）：

```
1. from pymilvus import MilvusClient

# 创建milvus客户端，传入本地数据库的存储路径，若路径不存在则创建
2. client = MilvusClient("dbs/origin_milvus.db")
```

创建Collections，存储向量及相关元数据：

相当于SQL数据库中建表

```
1. if client.has_collection(collection_name="demo_collection"): # 如果已存在同名collection，则先删除
2.     client.drop_collection(collection_name="demo_collection")

3. client.create_collection(
4.     collection_name="demo_collection",
5.     dimension=1024,                                    # 设置存储向量维度
6. )
```

注：create_collection支持更多的入参，如主键列名primary_field_name、向量列名vector_field_name、向量索引参数index_params、相似度参数metric_type等，此处仅介绍基本用法，其余均使用默认参数。

原生Milvus基本操作一览

数据入库：

```
docs = [                                                    # 测试文段列表
    "Artificial intelligence was founded as an academic discipline in 1956.",
    "Alan Turing was the first person to conduct substantial research in AI.",
    "Born in Maida Vale, London, Turing was raised in southern England.",
]
vecs =[embedding_model(doc) for doc in docs]                # 文段向量化
data = [                                                    # 将数据转换为入库所需格式
    { "id" : i,  "vector" : vecs[i],  "text" : docs[i],  "subject" :  "history" }   # subject为自定义label，
    for i in range(len(vecs))
]
res = client.insert(collection_name= "demo_collection" , data=data)   # insert方法数据入库
```

向量检索：

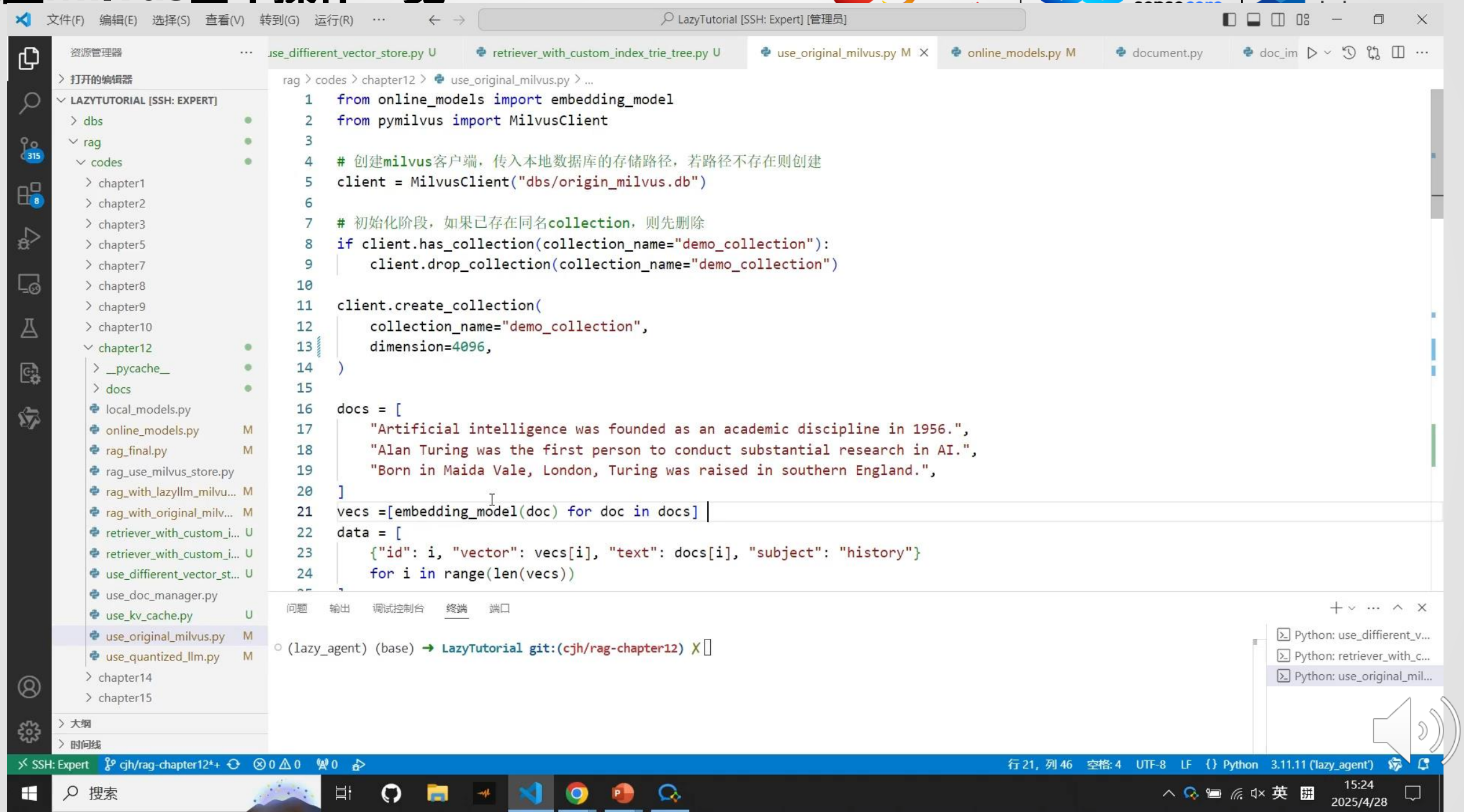
```
query = "Who is Alan Turing?"
q_vec = embedding_model(query)                  # query向量化
res = client.search(
    collection_name="demo_collection",          # 指定collection
    data=[q_vec],
    limit=2,                                    # 指定检索数量（top_k）
    output_fields=["text", "subject"],          # 指定检索结果中包含的字段
)
```

原生Milvus基本操作一览

标量过滤检索：

```
1. docs2 = [                                                    # 另一组文段
      "Machine learning has been used for drug design.",
      "Computational synthesis with AI algorithms predicts molecular properties.",
      "DDR1 is involved in cancers and fibrosis.",
   ]
2. vecs2 =[embedding_model(doc) for doc in docs2]
3. data2 = [
      {"id": 3 + i, "vector": vecs2[i], "text": docs2[i], "subject": "biology"}
      for i in range(len(vecs2))
   ]
4. client.insert(collection_name= “demo_collection” , data=data2)      # 数据插入同一个collection中
5. res = client.search(
      collection_name="demo_collection",
      data=[embedding_model("tell me AI related information")],
      filter= “subject ==  ‘biology’ ” ,                            # 检索时使用filter实现标量过滤
      limit=2,
      output_fields=["text", "subject"],
   )
```

原生Milvus基本操作一览



使用原生Milvus搭建RAG

```
1. import lazyllm
2. from lazyllm.tools.rag import SimpleDirectoryReader, SentenceSplitter
3. from lazyllm.tools.rag.doc_node import MetadataMode

4. from pymilvus import MilvusClient

5. from online_models import embedding_model, llm, rerank_model                    # llm、embedding、reranker均使用线上模型

6. DOC_PATH = "/xxx/xxx"

7. client = MilvusClient("dbs/rag_milvus.db")                                      # milvus client初始化

8. if client.has_collection(collection_name="demo_collection"):
9.     client.drop_collection(collection_name="demo_collection")
10. client.create_collection(                                                      # 创建collection
        collection_name="demo_collection",
        dimension=1024,
    )
11. docs = SimpleDirectoryReader(input_dir=DOC_PATH)()                             # 使用reader加载文件目录
12. block_transform = SentenceSplitter(chunk_size=256, chunk_overlap=25)           # 定义分块策略
13. nodes = []
14. for doc in docs:
15.     nodes.extend(block_transform(doc))

16. vecs = [embedding_model(node.get_text(MetadataMode.EMBED)) for node in nodes]  # 切片向量化
17. data = [
        {"id": i, "vector": vecs[i], "text": nodes[i].text}
        for i in range(len(vecs))
    ]
18. res = client.insert(collection_name= "demo_collection" , data=data)            # 数据入库
```

使用原生Milvus搭建RAG

```
# 检索与生成
prompt = '你是一个友好的 AI 问答助手，你需要根据给定的上下文和问题提供答案。\
        根据以下资料回答问题：\
        {context_str} \n    '
llm.prompt(lazyllm.ChatPrompter(instruction=prompt, extro_keys=[ 'context_str' ])) # 初始化prompt

query = "证券管理的基本规范？"
q_vec = embedding_model(query)                                          # query向量化
res = client.search(                                                    # 向量检索
    collection_name="demo_collection",
    data=[q_vec],
    limit=12,
    output_fields=["text"],
)

contexts = [res[0][i].get( 'entity' ).get( "text" , "" ) for i in range(len(res[0]))]       # 提取检索结果

rerank_res = rerank_model(text=query, documents=contexts, top_n=3)      # 重排序
rerank_contexts = [contexts[res[0]] for res in rerank_res]

context_str = "\n------------------\n".join(rerank_contexts)
res = llm({ "query" : query, "context_str" : context_str})              # 召回信息拼接，大模型生成
print(res)
```

使用原生Milvus搭建RAG

文件(F) 编辑(E) 选择(S) 查看(V) 转到(G) 运行(R) ··· LazyTutorial [SSH: Expert] [管理员]

资源管理器

打开的编辑器

LAZYTUTORIAL [SSH: EXPERT]

- dbs
- rag
  - codes
    - chapter1
    - chapter2
    - chapter3
    - chapter5
    - chapter7
    - chapter8
    - chapter9
    - chapter10
    - chapter12
      - __pycache__
      - docs
      - local_models.py
      - online_models.py M
      - rag_final.py M
      - rag_use_milvus_store.py
      - rag_with_lazyllm_milvu... M
      - rag_with_original_milv... M
      - retriever_with_custom_i... U
      - retriever_with_custom_i... U
      - use_diffierent_vector_st... U
      - use_doc_manager.py
      - use_kv_cache.py U
      - use_original_milvus.py M
      - use_quantized_llm.py M
    - chapter14
    - chapter15

大纲

时间线

use_diffierent_vector_store.py U | retriever_with_custom_index_trie_tree.py U | rag_with_original_milvus.py M | document.py | doc_impl.py | milv

rag > codes > chapter12 > rag_with_original_milvus.py > ...

```python
import lazyllm
from lazyllm.tools.rag import SimpleDirectoryReader, SentenceSplitter
from lazyllm.tools.rag.doc_node import MetadataMode

from pymilvus import MilvusClient

from online_models import custom_embedding_model as embedding_model, llm, rerank_model

DOC_PATH = "/mnt/lustre/share_data/dist/test_docs"

# milvus client初始化
client = MilvusClient("dbs/rag_milvus.db")
if client.has_collection(collection_name="demo_collection"):
    client.drop_collection(collection_name="demo_collection")
client.create_collection(
    collection_name="demo_collection",
    dimension=1024,
)

# 加载本地文件，并解析 -- block切片 -- 向量化 -- 入库
docs = SimpleDirectoryReader(input_dir=DOC_PATH)()
block_transform = SentenceSplitter(chunk_size=256, chunk_overlap=25)

nodes = []
for doc in docs:
```

问题 输出 调试控制台 终端 端口

(lazy_agent) (base) ➜ LazyTutorial git:(cjh/rag-chapter12) X

Python: use_diffierent_v...
Python: retriever_with_c...
Python: use_original_mil...

SSH: Expert | cjh/rag-chapter12*+ | ⊗0 ⚠0 | 0 | 行 6，列 1 | 空格: 4 | UTF-8 | LF | {} Python | 3.11.11 ('lazy_agent')

搜索 | 英 | 15:30 2025/4/28

Milvus in LazyLLM

从头学习如何使用Milvus向量数据库成本略大，使用LazyLLM一键接入Milvus。

配置方式：store_conf中额外添加字段 indices ， SmartEmbeddingIndex快速实现Milvus向量索引。

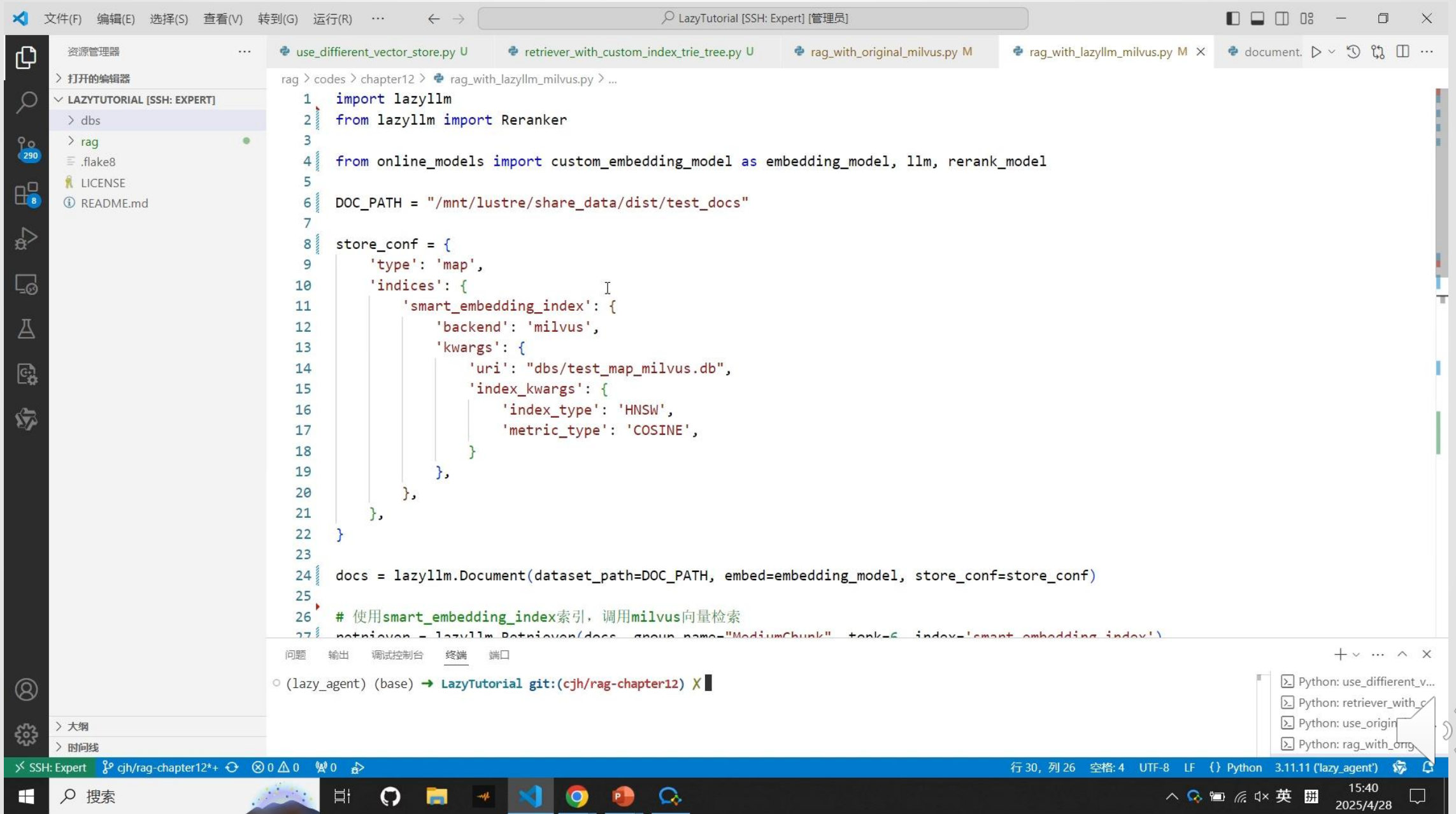
```
store_conf = {
    'type': 'map',
    'indices': {
        'smart_embedding_index': {
            'backend' : 'milvus' ,                          # 设定索引使用Milvus后端
            'kwargs': {
                'uri' : "dbs/test.db" ,                     # Milvus数据存储地址
                'index_kwargs' : {
                    'index_type' : 'HNSW' ,                 # 设置索引类型
                    'metric_type' : 'COSINE' , # 设置度量类型
                }
            },
        },
    },
}
```

注：如果存储类型type=milvus的情况下，由于kwargs中已经包含了索引配置信息，因此无需额外配置indices，即可实现内置的向量检索。

Milvus in LazyLLM

```
1. docs = lazyllm.Document(dataset_path=DOC_PATH, embed=embedding_model, store_conf=store_conf)

2. retriever = lazyllm.Retriever(
       docs,
       group_name="MediumChunk",
       topk=6,
       index='smart_embedding_index'                                  # 使用smart_embedding_index索引，调用milvus向量检索
   )
3. retriever.start()

4. reranker = Reranker('ModuleReranker', model=rerank_model, topk=3)   # 定义reranker

5. prompt = '你是一个友好的 AI 问答助手，你需要根据给定的上下文和问题提供答案。\
       根据以下资料回答问题：\
       {context_str} \n    '
6. llm.prompt(lazyllm.ChatPrompter(instruction=prompt, extro_keys=['context_str']))

7. query = "证券管理有哪些准则？"

8. nodes = retriever(query=query)                                     # query检索
9. rerank_nodes = reranker(nodes, query)                              # 节点重排
10. context_str = "\n======\n".join([node.get_content() for node in rerank_nodes]) # 召回节点拼接
11. res = llm({"query": query, "context_str": context_str})           # 大模型生成
12. print(res)
```

Milvus in LazyLLM



Milvus的分布式使用方式

Milvus不仅支持本地db文件链接，同时支持远程服务器端点的接入。

使用 Docker 完成服务部署(Milvus Standalone / Distributed)
或
直接使用云服务厂商（如Zilliz Cloud）提供的Milvus云服务

```
client = MilvusClient("dbs/origin_milvus.db")
```

获取远端地址与身份验证

```
client = MilvusClient(
    uri="http://localhost:19530",
    token="username:password  “
)
```

Milvus的分布式使用方式

例如，我们期望在Linux系统上部署Milvus Standalone服务。

- 安装Docker，并根据 Milvus 官方文档检查硬件与软件要求。
- 在Docker中安装Milvus，安装脚本：

```
# 下载安装脚本
curl -sfL https://raw.githubusercontent.com/milvus-io/milvus/master/scripts/standalone_embed.sh -o standalone_embed.sh

# 启动Docker容器
bash standalone_embed.sh start
```

安装结果：

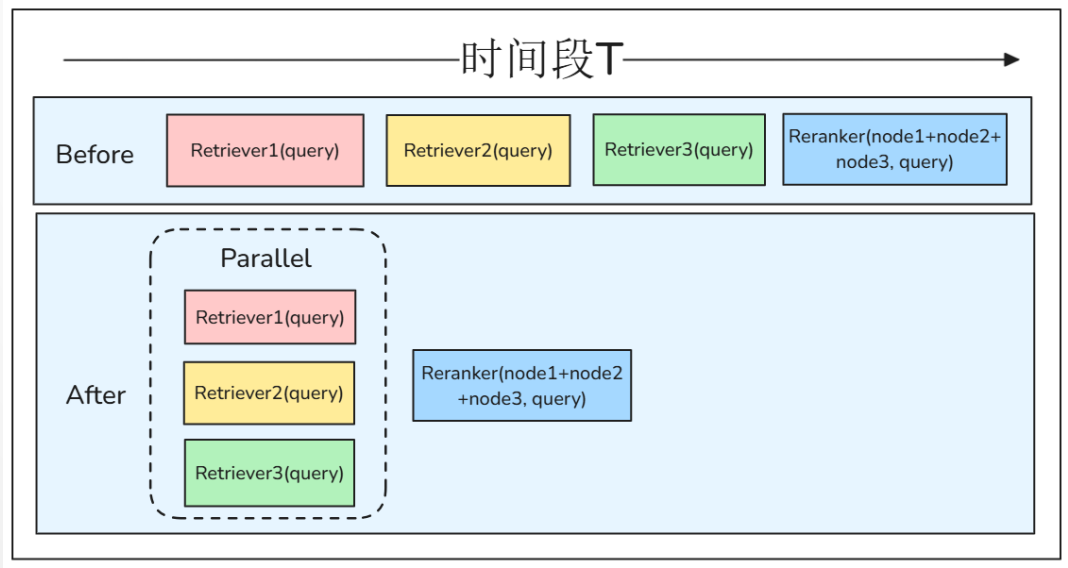
- 名为 Milvus 的 docker 容器在19530端口启动。要更改 Milvus 的默认配置，需将设置添加到当前文件夹中的user.yaml文件，然后重新启动服务。
- Milvus 数据卷被映射到当前文件夹中的volumes/milvus。

工程优化：使用Flow实现高效RAG管道

定义document
定义retriever1
定义retriever2
定义reranker
...
nodes1 = retriever1(query=query)
nodes2 = retriever2(query=query)
rerank_nodes = reranker(nodes1 + nodes2, query)

Pipeline实现RAG系统整体数据管道

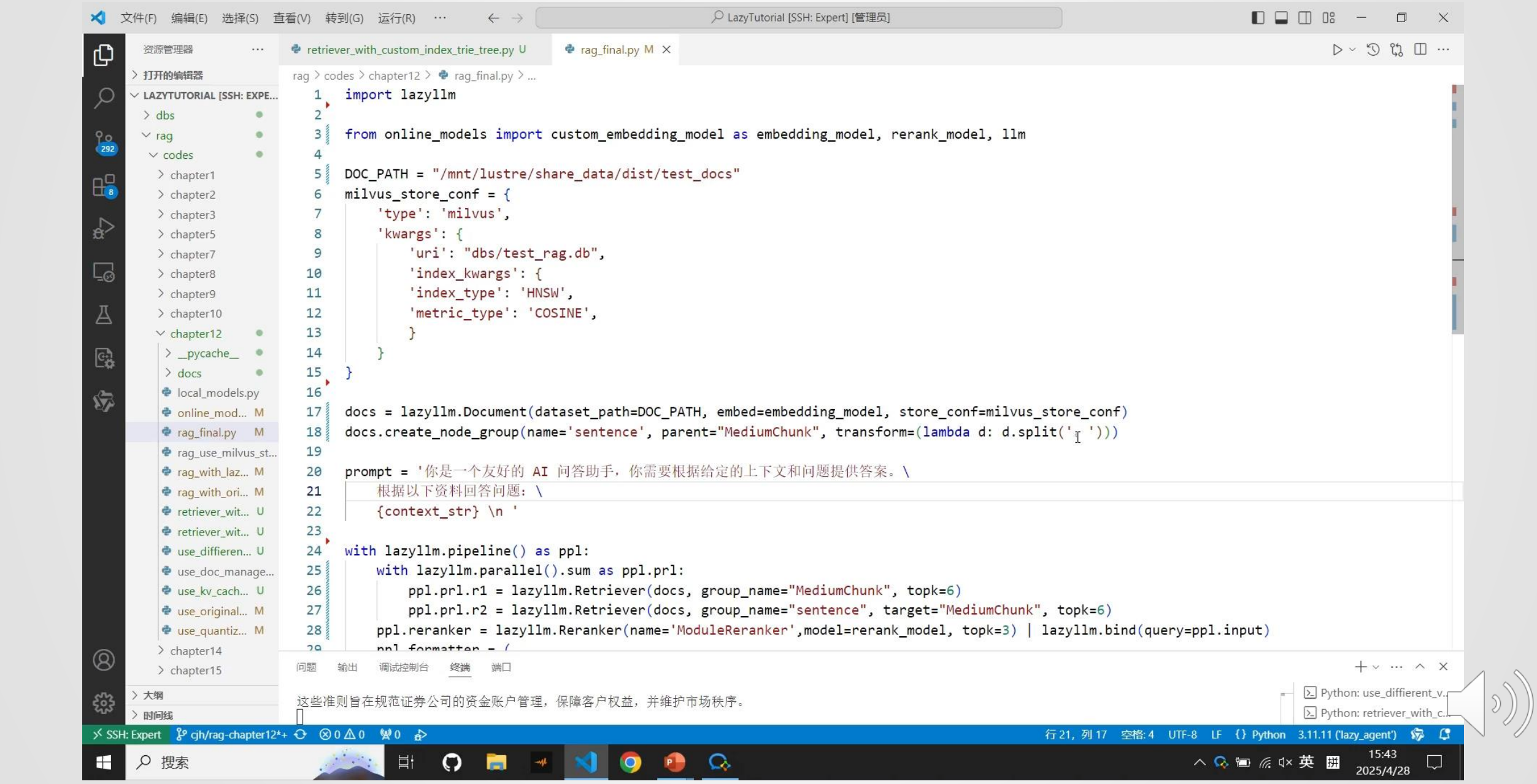
Parallel实现多路召回并行执行



使用Flow实现高效RAG管道

```
1. docs = lazyllm.Document(dataset_path=DOC_PATH, embed=embedding_model, store_conf=milvus_store_conf)
2. docs.create_node_group(name='sentence', parent="MediumChunk", transform=(lambda d: d.split('。')))

3. prompt = '你是一个友好的 AI 问答助手，你需要根据给定的上下文和问题提供答案。\
      根据以下资料回答问题：\
      {context_str} \n    '

5. with lazyllm.pipeline() as ppl:
6.     with lazyllm.parallel().sum as ppl.prl:
7.         ppl.prl.r1 = lazyllm.Retriever(docs, group_name="MediumChunk", topk=6)
8.         ppl.prl.r2 = lazyllm.Retriever(docs, group_name="sentence", target="MediumChunk", topk=6)
9.     ppl.reranker = lazyllm.Reranker(name='ModuleReranker',model=rerank_model, topk=3) | lazyllm.bind(query=ppl.input)
10.    ppl.formatter = (
          lambda nodes, query: dict(
              context_str="\n".join(node.get_content() for node in nodes),
              query=query)
      ) | lazyllm.bind(query=ppl.input)
11.    ppl.llm = llm.prompt(lazyllm.ChatPrompter(instruction=prompt, extro_keys=['context_str']))

12. w = lazyllm.WebModule(ppl, port=23492, stream=True).start().wait()
```

使用Flow实现高效RAG管道



目录

使用不同推理框架部署大模型

LazyLLM支持使用LightLLM、vLLM、LMDeploy三种推理框架实现模型部署，开发者可以根据自己的实际需求，灵活选择推理框架，以实现最适合自己的模型推理体验。

```
llm = lazyllm.TrainableModule('internlm2-chat-20b')
```

```
llm = lazyllm.TrainableModule( 'internlm2-chat-20b' ).deploy_method(lazyllm.deploy.Lightllm)

llm = lazyllm.TrainableModule( 'internlm2-chat-20b' ).deploy_method(lazyllm.deploy.Vllm)

llm = lazyllm.TrainableModule( 'internlm2-chat-20b' ).deploy_method(lazyllm.deploy.LMDeploy)
```

```
llm = TrainableModule('DeepSeek-R1').deploy_method(
    (deploy.Vllm, {
        'tensor-parallel-size': 8,
        'pipeline-parallel-size': 2,
        'max-num-batched-tokens': 131072,
        'launcher': launchers.sco(nnode=2, ngpus=8, sync=False)
    })
)
```

传入推理框架所需入参

使用量化模型

从开源社区寻找大模型的量化版本

运行BF16或FP16的 Qwen2-72B-Instruct 模型运行需要至少144GB显存（例如2xA100-80G或5xV100-32G）
而运行 Int4 模型至少需要 48GB 显存（例如1xA100-80G或2xV100-32G），是原来的33%

使用量化模型

```
import time
from lazyllm import TrainableModule, deploy

start_time = time.time()
llm = TrainableModule('Qwen2-72B-Instruct').deploy_method(                    # 启动原始模型
        (deploy.Vllm, {
            'tensor-parallel-size': 2,
        })).start()
end_time = time.time()
print("原始模型加载耗时：", end_time-start_time)

start_time = time.time()
llm_awq = TrainableModule('Qwen2-72B-Instruct-AWQ').deploy_method(            # 启动量化模型
        (deploy.Vllm, {
            'tensor-parallel-size': 2,
        })).start()
end_time = time.time()
print("AWQ量化模型加载耗时：", end_time-start_time)

query = "生成一份1000字的人工智能发展相关报告"                                  # 测试query

start_time = time.time()
llm(query)                                                                    # 原始模型生成
end_time = time.time()
print("原始模型耗时：", end_time-start_time)

start_time = time.time()
llm_awq(query)                                                                # 量化模型生成
end_time = time.time()
print("AWQ量化模型耗时：", end_time-start_time)
```

原始模型加载耗时：
129.6051540374756
AWQ量化模型加载耗时：
86.4980857372284

原始模型耗时：
13.104065895080566
AWQ量化模型耗时：
8.81701111793518

Q&A

1. 面对性能测试中的不稳定性，有什么处理方法么

商汤 sensetime | 大装置 sensecore | 深蓝学院 shenlanxueyuan.com

感谢聆听

Thanks for Listening