```
    def split_text(self, text: str) -> List[str]:
        if self.pdf:            # 如果pdf参数为True，那么对文本进行预处理
            # 替换掉连续的3个及以上的换行符为一个换行符
            text = re.sub(r"\n{3,}", r"\n", text)
            # 将所有的空白字符（包括空格、制表符、换页符等）替换为一个空格
            text = re.sub('\s', " ", text)
            # 将连续的两个换行符替换为一个空字符
            text = re.sub("\n\n", "", text)

        # 导入pipeline模块，用于创建一个处理流程
        from modelscope.pipelines import pipeline

        # 创建一个document-segmentation任务的处理流程
        # 用的模型为damo/nlp_bert_document-segmentation_chinese-base，计算设备为cpu
        p = pipeline(
            task="document-segmentation",
            model='damo/nlp_bert_document-segmentation_chinese-base',
            device="cpu")
        result = p(documents=text)    # 对输入的文本进行处理，返回处理结果
        sent_list = [i for i in result["text"].split("\n\t") if i]  # 将处理结果按照换行符和制表符进行切分，得到句子列表
        return sent_list              # 返回句子列表
```

其中，有三点值得注意下

- 参数use_document_segmentation指定是否用语义切分文档
  此处采取的文档语义分割模型为达摩院开源的：***nlp_bert_document-segmentation_chinese-base***（这是其论文）
- 另，如果使用模型进行文档语义切分，那么需要安装：

```
modelscope[nlp]：pip install "modelscope[nlp]" -f https://modelscope.oss-cn-beijing.aliyuncs.com/releases/repo.html
```

- 且考虑到使用了三个模型，可能对于低配置gpu不太友好，因此这里将模型load进cpu计算，有需要的话可以替换device为自己的显卡id

3.6 knowledge_base：存储用户上传的文件并向量化

knowledge_bas下面有两个文件，一个content 即用户上传的原始文件，vector_store则用于存储向量库文件，即本地知识库本体，因为content因人而异 谁上传啥就是啥 所以没啥好分析，而vector_store下面则有两个文件，一个index.faiss，一个index.pkl

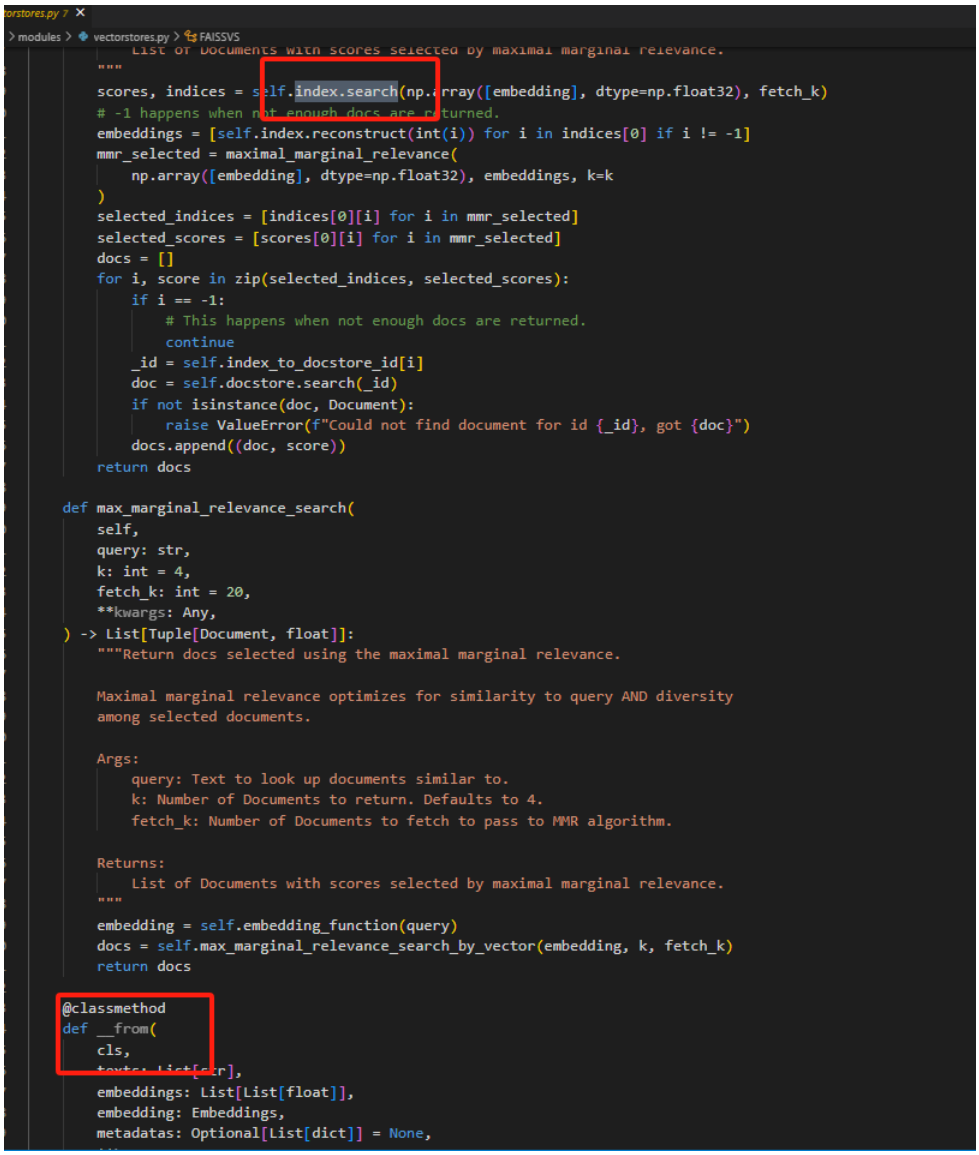
3.7 chains：向量搜索/匹配

如之前所述，本节开头图中“FAISS索引、FAISS搜索”中的“FAISS”是Facebook AI推出的一种用于有效搜索大规模高维向量空间中相似度的库，在大规模数据集中快速找到与给定向量最相似的向量是很多AI应用的重要组成部分，例如在推荐系统、自然语言处理、图像检索等领域

3.7.1 chains/modules /vectorstores.py文件：根据查询向量query在向量数据库中查找与query相似的文本向量

主要是关于

1. FAISS (Facebook AI Similarity Search)的使用，具体体现在*max_marginal_relevance_search_by_vector* 中(**如下图的最上面部分**)
2. 以及一个FAISS向量存储类(FAISSVS，FAISSVS类继承自FAISS类)的定义，包含两个方法
   一个 max_marginal_relevance_search (**如下图的中间部分**，其最后会调用上面的max_marginal_relevance_search_by_vector)
   一个 __from (**如下图的最下面部分**)



接下来，我们逐一分析下这几个函数

- **max_marginal_relevance_search**
  分两步，给定查询语句，首先将查询语句转换为嵌入向量「*embedding = self.embedding_function(query)*」，然后调用 *max_marginal_relevance_search_by_vector* 函数进行MMR搜索

```python
#  使用最大边际相关性返回被选中的文本
def max_marginal_relevance_search(
    self,
    query: str,            # 查询
    k: int = 4,            # 返回的文档数量，默认为 4
    fetch_k: int = 20,     # 用于传递给 MMR 算法的抓取文档数量
    **kwargs: Any,
) -> List[Tuple[Document, float]]:

    # 查询向量化
    embedding = self.embedding_function(query)
    # 调用：max_marginal_relevance_search_by_vector
    docs = self.max_marginal_relevance_search_by_vector(embedding, k, fetch_k)
    return docs
```

  下面看下其中 *max_marginal_relevance_search_by_vector* 的实现
  该函数通过给定的嵌入向量，使用最大边际相关性(Maximal Marginal Relevance, MMR)方法来返回相关的文本
  MMR是一种解决查询结果多样性和相关性的算法，具体来说，它不仅要求返回的文本与查询尽可能相似，而且希望返回的文本集之间尽可能多样

```python
# 使用最大边际相关性返回被选中的文档，最大边际相关性旨在优化查询的相似性和选定文本之间的多样性
def max_marginal_relevance_search_by_vector(
    self, embedding: List[float], k: int = 4, fetch_k: int = 20, **kwargs: Any
) -> List[Tuple[Document, float]]:

    # 使用索引在文本中搜索与嵌入向量相似的内容，返回最相似的fetch_k个文本的得分和索引
    scores, indices = self.index.search(np.array([embedding], dtype=np.float32), fetch_k)

```

```
    # 通过索引从文本中重构出嵌入向量，-1表示没有足够的文本返回
embeddings = [self.index.reconstruct(int(i)) for i in indices[0] if i != -1]
    # 使用最大边际相关性算法选择出k个最相关的文本
    mmr_selected = maximal_marginal_relevance(
        np.array([embedding], dtype=np.float32), embeddings, k=k
    )

    selected_indices = [indices[0][i] for i in mmr_selected]    # 获取被选中的文本的索引
    selected_scores = [scores[0][i] for i in mmr_selected]      # 获取被选中的文本的得分
    docs = []
    for i, score in zip(selected_indices, selected_scores):     # 对于每个被选中的文本索引和得分
        if i == -1:  # 如果索引为-1，表示没有足够的文本返回
            continue

        _id = self.index_to_docstore_id[i]       # 通过索引获取文本的id
        doc = self.docstore.search(_id)          # 通过id在文档库中搜索文本
        if not isinstance(doc, Document):        # 如果搜索到的文本不是Document类型，抛出错误
            raise ValueError(f"Could not find document for id {_id}, got {doc}")
        docs.append((doc, score))        # 将文本和得分添加到结果列表中
    return docs                          # 返回结果列表
```

注意，上面第6-7行代码中，直接调用的index的search函数

```
# 使用索引在文本中搜索与嵌入向量相似的内容，返回最相似的fetch_k个文本的得分和索引
scores, indices = self.index.search(np.array([embedding], dtype=np.float32), fetch_k)
```

这里面就有来头了，通过和我司杜老师的讨论确定，这个search是根据构建索引时所用的度量指标找到最近的k个向量的，在构建索引时
→ 如果是faiss.IndexFlatIP，就是内积(METRIC_INNER_PRODUCT)，可以认为是余弦相似度
→ 如果是faiss.IndexFlatL2，就是欧氏距离(METRIC_L2)，更多计算距离的方式详见上文的2.2.2节

其具体的代码实现如下所示(来源：faiss/IndexFlat.cpp#L27)

```
void IndexFlat::search(
    idx_t n,                          // 搜索的查询向量的数量
    const float* x,                   // 指向查询向量数据的指针
    idx_t k,                          // 每个查询向量返回的最近邻个数
    float* distances,                 // 返回的距离数组
    idx_t* labels,                    // 返回的标签数组
    const SearchParameters* params) const {      // 搜索参数

    // 如果params非空，则使用params中的选择器，否则使用nullptr
    IDSelector* sel = params ? params->sel : nullptr;

    // 检查k（最近邻的数量）必须大于0
    FAISS_THROW_IF_NOT(k > 0);

    // distances和labels被视为堆（用于存储最近邻的结果）
    if (metric_type == METRIC_INNER_PRODUCT) {        // 如果度量类型是内积
        float_minheap_array_t res = {size_t(n), size_t(k), labels, distances};
        // 使用内积计算最近邻
        knn_inner_product(x, get_xb(), d, n, ntotal, &res, sel);

    } else if (metric_type == METRIC_L2) {            // 如果度量类型是L2距离
        float_maxheap_array_t res = {size_t(n), size_t(k), labels, distances};
        // 使用L2距离计算最近邻
        knn_L2sqr(x, get_xb(), d, n, ntotal, &res, nullptr, sel);

    } else if (is_similarity_metric(metric_type)) {   // 如果度量类型是其他相似度度量
        float_minheap_array_t res = {size_t(n), size_t(k), labels, distances};
        // 使用其他相似度度量计算最近邻
        knn_extra_metrics(
            x, get_xb(), d, n, ntotal, metric_type, metric_arg, &res);

    } else { // 其他情况
        FAISS_THROW_IF_NOT(!sel);     // 确保选择器为空
        float_maxheap_array_t res = {size_t(n), size_t(k), labels, distances};
        // 使用其他相似度度量计算最近邻
        knn_extra_metrics(
            x, get_xb(), d, n, ntotal, metric_type, metric_arg, &res);
    }
}
```

- **__from**
  用于从一组文本和对应的嵌入向量创建一个FAISSVS实例。该方法首先创建一个FAISS索引并添加嵌入向量，然后创建一个文本存储以存储与每个嵌入向量关联的文本

```python
# 从给定的文本、嵌入向量、元数据等信息构建一个FAISS索引对象
def __from(
    cls,
    texts: List[str],                 # 文本列表，每个文本将被转化为一个文本对象
    embeddings: List[List[float]],    # 对应文本的嵌入向量列表
    embedding: Embeddings,            # 嵌入向量生成器，用于将查询语句转化为嵌入向量
    metadatas: Optional[List[dict]] = None,
    **kwargs: Any,
) -> FAISS:

    faiss = dependable_faiss_import()      # 导入FAISS库
    index = faiss.IndexFlatIP(len(embeddings[0]))      # 使用FAISS库创建一个新的索引，索引的维度等于嵌入文本向量的长度
    index.add(np.array(embeddings, dtype=np.float32))  # 将嵌入向量添加到FAISS索引中

    # quantizer = faiss.IndexFlatL2(len(embeddings[0]))
    # index = faiss.IndexIVFFlat(quantizer, len(embeddings[0]), 100)
    # index.train(np.array(embeddings, dtype=np.float32))
    # index.add(np.array(embeddings, dtype=np.float32))

    documents = []
    for i, text in enumerate(texts):      # 对于每一段文本
        # 获取对应的元数据，如果没有提供元数据则使用空字典
        metadata = metadatas[i] if metadatas else {}

        # 创建一个文本对象并添加到文本列表中
        documents.append(Document(page_content=text, metadata=metadata))

    # 为每个文本生成一个唯一的ID
    index_to_id = {i: str(uuid.uuid4()) for i in range(len(documents))}

    # 创建一个文本库，用于存储文本对象和对应的ID
    docstore = InMemoryDocstore(
        {index_to_id[i]: doc for i, doc in enumerate(documents)}
    )

    # 返回FAISS对象
    return cls(embedding.embed_query, index, docstore, index_to_id)
```

从上面代码的第11-13行中

```
faiss = dependable_faiss_import()     # 导入FAISS库
  index = faiss.IndexFlatIP(len(embeddings[0]))     # 使用FAISS库创建一个新的索引，索引的维度等于嵌入文本向量的长度
  index.add(np.array(embeddings, dtype=np.float32)) # 将嵌入向量添加到FAISS索引中
```

可知，构建index的时候就已经指定了用IndexFlatIP(余弦相似度)的方式计算距离(你可以通过这个代码链接验证下：chains/modules/vectorstores.py#L103)

以上就是这段代码的主要内容，通过使用FAISS和MMR，它可以帮助我们在大量文本中找到与给定查询最相关的文本

3.7.2 chains /local_doc_qa.py代码文件：向量搜索

1. 导入包和模块
   代码开始的部分是一系列的导入语句，导入了必要的 Python 包和模块，包括文件加载器，文本分割器，模型配置，以及一些 Python 内建模块和其他第三方库
2. 改写 HuggingFaceEmbeddings 类的哈希方法
   代码定义了一个名为 ***_embeddings_hash*** 的函数，并将其赋值给 HuggingFaceEmbeddings 类的 __hash__ 方法。这样做的目的是使 HuggingFaceEmbeddings 对象可以被哈希，即可以作为字典的键或者被加入到集合中
3. 载入向量存储器
   定义了一个名为 ***load_vector_store*** 的函数，这个函数用于从本地加载一个向量存储器，返回 FAISS 类的对象。其中使用了 lru_cache 装饰器，可以缓存最近使用的 CACHED_VS_NUM 个结果，提高代码效率
4. 文件树遍历
   tree 函数是一个递归函数，用于遍历指定目录下的所有文件，返回一个包含所有文件的完整路径和文件名的列表。它可以忽略指定的文件或目录
5. 加载文件：
   load_file 函数根据文件后缀名选择合适的加载器和文本分割器，加载并分割文件
6. 生成提醒:
   generate_prompt 函数用于根据相关文档和查询生成一个提醒。提醒的模板由 prompt_template 参数提供
7. 创建文档列表
   search_result2docs