Embedding packaging explanation

The source code for this article is **here** . If you need to reproduce, you can download and run the source code.
LangChain provides an efficient development framework for developing custom applications based on LLM, which allows developers to quickly stimulate the powerful capabilities of LLM and build LLM applications. LangChain also supports Embeddings of multiple large models, and has built-in calling interfaces for Embeddings of large models such as OpenAI and LLAMA. However, LangChain does not have all large models built in. It provides strong scalability by allowing users to customize Embeddings types.

In this section, we take Zhipu AI as an example to explain how to customize Embeddings based on LangChain.

This section involves relatively more technical details of LangChain and large model calls. Students who have the energy can learn to deploy them. If you don't have the energy, you can directly use the subsequent code to support the calls.

To implement custom Embeddings, you need to define a custom class that inherits from the LangChain Embeddings base class, and then define two functions: ① embed_query method, which is used to embed a single string (query); ② embed_documents method, which is used to embed a list of strings (documents).

First we import the required third-party libraries:

python

```python
from __future__ import annotations

import logging
from typing import Dict, List, Any

from langchain.embeddings.base import Embeddings
from langchain.pydantic_v1 import BaseModel, root_validator

logger = logging.getLogger(__name__)
```

Here we define a custom Embeddings class that inherits from the Embeddings class:

```python
    class ZhipuAIEmbeddings(BaseModel, Embeddings):
        """`Zhipuai Embeddings` embedding models."""

        client: Any
        """`zhipuai.ZhipuAI"""
```

In Python, root_validator is a decorator function in the Pydantic module for custom data validation. root_validator is used to perform custom validation on the entire data model before validating the entire data model to ensure that all data conforms to the expected data structure.

root_validator receives a function as a parameter, which contains the logic to be validated. The function should return a dictionary containing the validated data. If the validation fails, a ValueError exception is raised.

Here we just need to configure it `.env` in the file and it will be automatically obtained . `ZHIPUAI_API_KEY` `zhipuai.ZhipuAI` `ZHIPUAI_API_KEY`

```python
    @root_validator()
    def validate_environment(cls, values: Dict) -> Dict:
        """
        实例化ZhipuAI为values["client"]

        Args:

            values (Dict): 包含配置信息的字典，必须包含 client 的字段.
        Returns:

            values (Dict): 包含配置信息的字典。如果环境中有zhipuai库，则将返回实例化的
        """
        from zhipuai import ZhipuAI
        values["client"] = ZhipuAI()
        return values
```

`embed_query` It is a method for calculating embedding for a single text (str). Here we override this method, call the remote API instantiated during the verification environment, `ZhipuAI` and return the embedding result.

python

```
    def embed_query(self, text: str) -> List[float]:
        """
        生成输入文本的 embedding.

        Args:
            texts (str): 要生成 embedding 的文本.

        Return:
            embeddings (List[float]): 输入文本的 embedding，一个浮点数值列表.
        """
        embeddings = self.client.embeddings.create(
            model="embedding-2",
            input=text
        )
        return embeddings.data[0].embedding
```

embed_documents is a method that calculates embeddings for a list of strings (List[str]). For this type of input, we use a loop to calculate the embeddings of the substrings in the list one by one and return them.

python

```
    def embed_documents(self, texts: List[str]) -> List[List[float]]:
        """
        生成输入文本列表的 embedding.
        Args:
            texts (List[str]): 要生成 embedding 的文本列表.

        Returns:
            List[List[float]]: 输入列表中每个文档的 embedding 列表。每个 embedding
        """
        return [self.embed_query(text) for text in texts]
```

You can add some content processing before requesting embedding. For `embed_query` example, if the text is particularly long, we can consider segmenting the text to prevent it from exceeding the maximum token limit. These are all possible and depend on everyone's subjective initiative to improve it. Here is just a simple demo.

Through the above steps, we can define the embedding calling method based on LangChain and Zhipu AI. We encapsulate this code in the zhipuai_embedding.py file.

The source code for this article is **here** . If you need to reproduce, you can download and run the source code.

Previous chapter
4. Build and use vector database