

第三章：LangChain4j

聊完了大模型的基本使用，接下来我们学习LangChain4j，之前我们调用大模型都是借助于Apifox发送http请求完成的，但是在实际开发中我们肯定不能这么调用，我们需要写代码调用大模型，这样才能真正的做到讲大模型和我们传统的软件深度融合。

那究竟应该怎么写代码才能调用大模型呢？我们可以借助于一些工具库来完成。目前市面上有关java调用大模型的工具库，主流的有两种，一种是LangChain4j，一种是SpringAI。有关使用SpringAI如何调用大模型，黑马已经提供了对应的课程《SpringAI+DeepSeek》，有需要的同学可以直接上B站观看。咱们本次课程，主要给大家介绍如何使用LangChain4j调用大模型。



LangChain4j

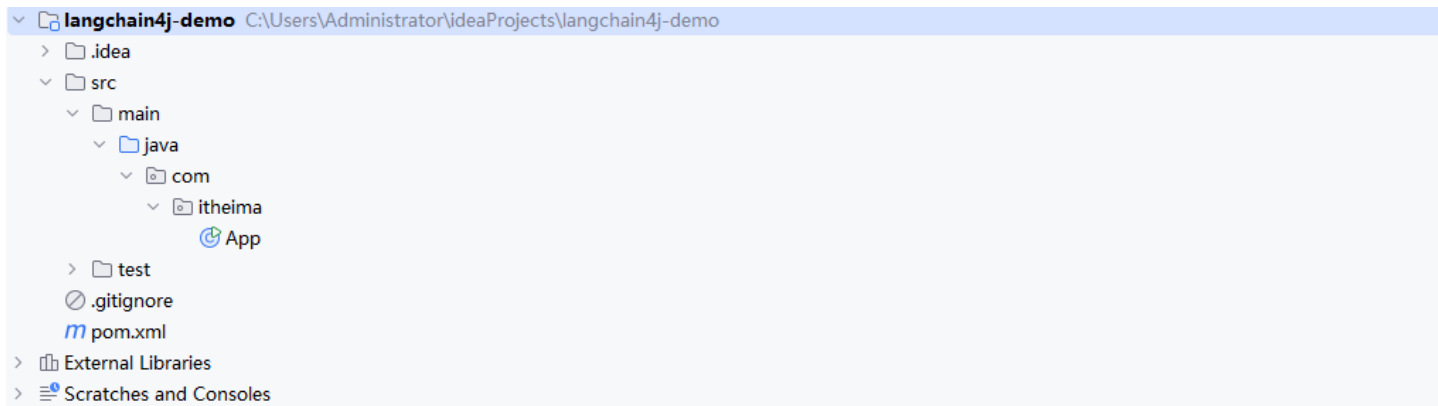


Spring AI

LangChain4j的官网是：<https://docs.langchainj.dev>。里面提供了langchain4j的详细使用教程，大家有兴趣的可以自己去看一看，咱们课程中就不带着大家看了，直接教大家怎么操作。

3.1 快速入门

3.1.1 创建一个普通的maven工程



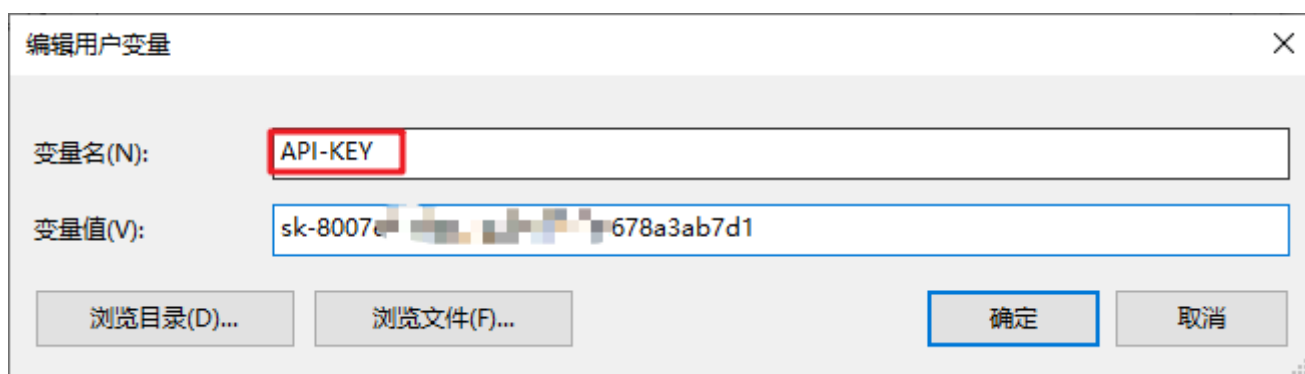
3.1.2 引入依赖

pom.xml

```
1 <dependency>
2   <groupId>dev.langchain4j</groupId>
3   <artifactId>langchain4j-open-ai</artifactId>
4   <version>1.0.1</version>
5 </dependency>
```

3.1.3 构建聊天对象OpenAiChatModel

构建OpenAiChatModel对象的时候，需要指定大模型的url地址，百炼平台的API-KEY，以及调用的模型名称。这里的API-KEY给大家说明一下，API-KEY可以直接写死到代码中，也可以配置到操作系统的环境变量中，然后通过代码获取再使用。这里推荐大家把API-KEY配置到系统的环境变量中再使用，因为如果直接写死在代码里面，会存在API-KEY泄露的风险。所以在写代码前，请先在系统的用户变量中创建一个名字叫API-KEY的环境变量，值就是你在百炼平台申请的api-key。最后一定记得重启IDEA！



下面是构建OpenAiChatModel对象的代码：

App.java

```
1 OpenAiChatModel model = OpenAiChatModel.builder()
2   .baseUrl("https://dashscope.aliyuncs.com/compatible-mode/v1")//url参考百
   炼平台API文档
3   .apiKey(System.getenv("API-KEY"))//获取环境变量API-KEY使用
4   .modelName("qwen-plus")//设置模型名称
```

```
5         .build();
```

3.1.4 调用方法与大模型交互

```
App.java
1  public class App {
2      public static void main(String[] args) {
3          //2. 构建OpenAiChatModel对象
4          OpenAiChatModel model = OpenAiChatModel.builder()
5              .baseUrl("https://dashscope.aliyuncs.com/compatible-mode/v1")
6              .apiKey(System.getenv("API-KEY"))
7              .modelName("qwen-plus")
8              .build();
9
10         //3. 调用chat方法, 交互
11         String result = model.chat("东哥帅不帅?");
12         System.out.println(result);
13     }
14 }
```

3.1.5 查看日志信息

为了查看与大模型交互过程中具体发送的请求消息和大模型响应的数据，可以打开日志开关，我们只需要在构建OpenAiChatModel对象的时候调用logRequests和logResponses方法设置一下即可。

```
App.java
1  public class App {
2      public static void main(String[] args) {
3          //2. 构建OpenAiChatModel对象
4          OpenAiChatModel model = OpenAiChatModel.builder()
5              .baseUrl("https://dashscope.aliyuncs.com/compatible-mode/v1")
6              .apiKey(System.getenv("API-KEY"))
7              .modelName("qwen-plus")
8              .logRequests(true) //设置打印请求日志
9              .logResponses(true) //设置打印响应日志
10             .build();
11
12         //3. 调用chat方法, 交互
13         String result = model.chat("东哥帅不帅?");
14         System.out.println(result);
15     }
16 }
```

```
D:\soft\jdk17\bin\java.exe ...
15:38:25.356 [main] INFO dev.langchain4j.http.client.log.LoggingHttpClient -- HTTP request:
- method: POST
- url: https://dashscope.aliyuncs.com/compatible-mode/v1/chat/completions
- headers: [Authorization: Beare...d1], [User-Agent: langchain4j-openai], [Content-Type: application/json]
- body: {
  "model": "qwen-plus",
  "messages": [ {
    "role": "user",
    "content": "东哥帅不帅?"
  } ],
  "stream": false
}

15:38:29.961 [main] INFO dev.langchain4j.http.client.log.LoggingHttpClient -- HTTP response:
- status code: 200
- headers: [:status: 200], [content-length: 799], [content-type: application/json], [date: Mon, 16 Jun 2025 07:38:29 GMT], [req-arrive-time: 1750059505599], [req-co
- body: {"choices":[{"message":{"role":"assistant","content":"东哥，如果是指京东的刘强东的话，他是一个非常有影响力的企业家，关于外貌，帅不帅其实是很主观的事情，不同的人有不同的审美标准。在很多人眼中，他可能因为才华和成就而显得“帅气”。你也可以分享下你心中对“帅”的
东哥，如果是指京东的刘强东的话，他是一个非常有影响力的企业家，关于外貌，帅不帅其实是很主观的事情，不同的人有不同的审美标准。在很多人眼中，他可能因为才华和成就而显得“帅气”。你也可以分享下你心中对“帅”的
```

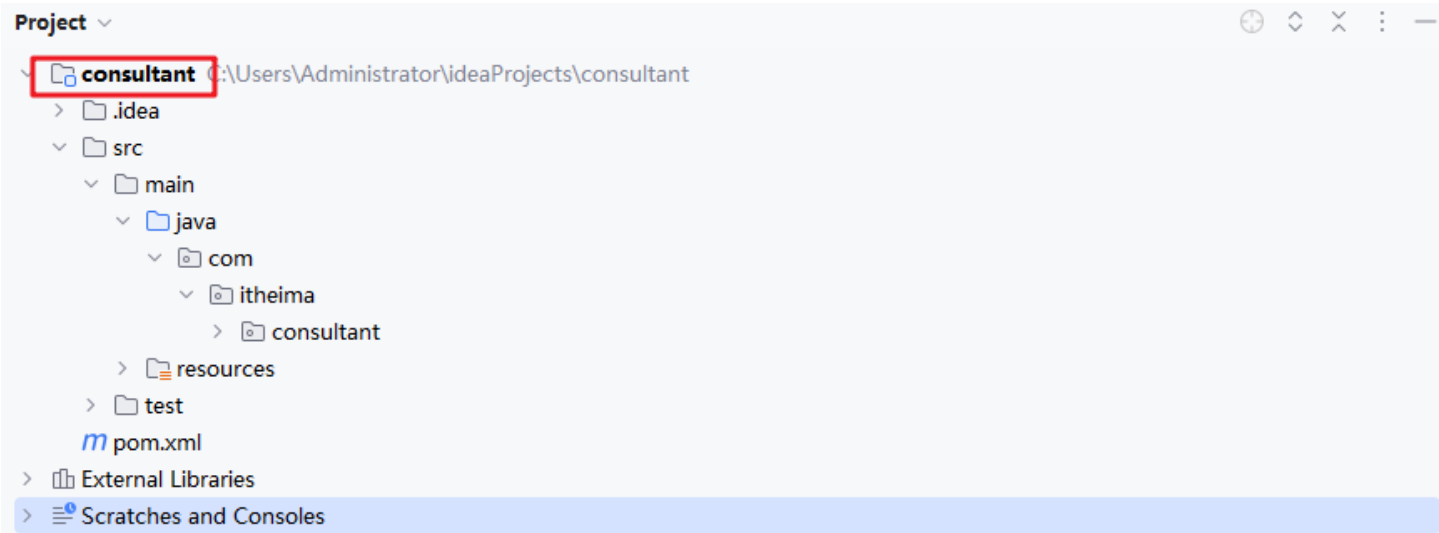
请求消息

响应消息

3.2 Spring整合LangChain4j

因为将来我们的java项目，必然是离不开Spring的，既然我们做的是大模型与传统软件的结合，那么毫无疑问，langchain4j的使用必然要和spring结合起来才可以。

3.2.1 创建SpringBoot项目



3.2.2 引入LangChain4j起步依赖

```

pom.xml
1  <dependency>
2      <groupId>dev.langchain4j</groupId>
3      <artifactId>langchain4j-open-ai-spring-boot-starter</artifactId>
4      <version>1.0.1-beta6</version>
5  </dependency>
```

3.2.3 在application.yml中配置调用大模型的信息

```
application.yml

1  langchain4j:
2    open-ai:
3      chat-model:
4        base-url: https://dashscope.aliyuncs.com/compatible-mode/v1
5        api-key: ${API-KEY}
6        model-name: qwen-plus
```

起步依赖会检测到配置信息，自动的往IOC容器中注入一个OpenAiChatModel对象。

3.2.4 开发接口，调用大模型

```
ChatController.java

1  @RestController
2  public class ChatController {
3    @Autowired
4    private OpenAiChatModel model;
5    @RequestMapping("/chat")
6    public String chat(String message){
7        String result = model.chat(message);
8        return result;
9    }
10 }
```

3.2.5 查看日志信息

为了查看与大模型交互过程中具体发送的请求消息和大模型响应的数据，我们需要在application.yml配置文件中开启配置即可。

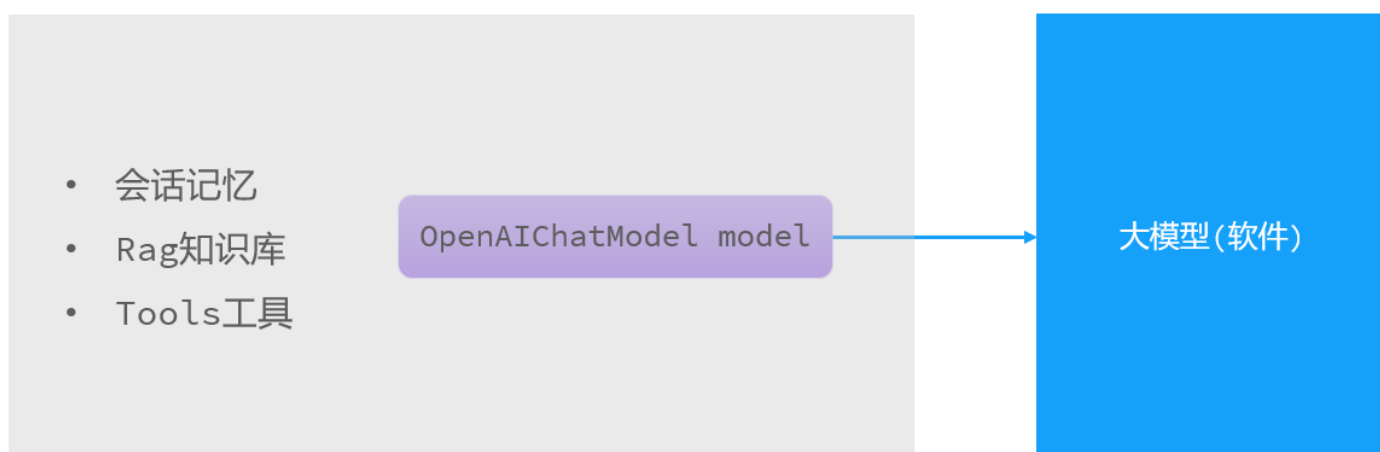
```
application.yml

1  langchain4j:
2    open-ai:
3      chat-model:
4        base-url: https://dashscope.aliyuncs.com/compatible-mode/v1
5        api-key: ${API-KEY}
6        model-name: qwen-plus
7        log-requests: true #请求消息日志
8        log-responses: true #响应消息日志
9  logging:
```

```
10     level:
11     dev.langchain4j: debug #日志级别
```

3.3 AiServices工具类

接下来我们学习LangChain4j提供的工具类AiServices，一个非常宝藏的工具。在之前的案例中，我们访问大模型是借助于OpenAiChatModel的chat方法完成的。其实这种方式在实际开发中并不是很常用，因为如果使用这种方式调用大模型，将来我们完成一些高阶的功能，比如会话记忆/RAG知识库/Tools工具的时候，在调用chat方法访问大模型前，我们需要自己做很多很多的工作，完成起来是比较复杂的。



为了简化我们程序员的使用，LangChain4j提供了AiServices工具类，封装了有关model对象和其它一些功能的操作，用起来会非常简单。接下来我们先来聊一聊AiServices工具类的基本使用。

3.3.1 AiServices工具类基本使用

3.3.1.1 引入AiServices相关依赖

pom.xml

```
1 <dependency>
2     <groupId>dev.langchain4j</groupId>
3     <artifactId>langchain4j-open-ai-spring-boot-starter</artifactId>
4     <version>1.0.1-beta6</version>
5 </dependency>
```

3.3.1.2 声明用于封装聊天方法的接口

ConsultantService.java

```

1  public interface ConsultantService {
2      //用于聊天的方法,message为用户输入的内容
3      public String chat(String message);
4  }

```

3.3.1.3 使用AiServices工具类创建接口的动态代理对象

由于创建好的代理对象，将来在ChatController中需要使用，所以这些代码将会放到统一的配置类CommonConfig中完成。

CommonConfig.java

```

1  @Configuration
2  public class CommonConfig {
3      @Autowired
4      private OpenAiChatModel model;
5      @Bean
6      public ConsultantService consultantService() {
7          ConsultantService cs = AiServices.builder(ConsultantService.class)
8              .chatModel(model) //设置对话时使用的模型对象
9              .build();
10         return cs;
11     }
12 }

```

3.3.1.4 ChatController中注入ConsultantService并使用

ChatController.java

```

1  @RestController
2  public class ChatController {
3      @Autowired
4      private ConsultantService consultantService;
5      @RequestMapping("/chat")
6      public String chat(String message){
7          String result = consultantService.chat(message);
8          return result;
9      }
10 }

```

3.3.2 AiServices工具类声明式使用

为了简化AiServices工具类的使用，LangChain4j提供了声明式使用方法，想为哪个接口创建代理对象，只需要在该接口上添加@AiService注解并指定要使用的模型，将来LangChain4j扫描到该注解后

会自动的创建该接口的代理对象并注入到IOC容器中。接下来修改ConsultantService中的代码，并重新测试。

ConsultantService.java

```
1  @AiService(  
2      wiringMode = AiServiceWiringMode.EXPLICIT,  
3      chatModel = "openAiChatModel"  
4  )  
5  public interface ConsultantService {  
6      //用于聊天的方法,message为用户输入的内容  
7      public String chat(String message);  
8  }
```

上述代码中，AiService注解的wiringMode用于指定装配模式，默认的取值为AiServiceWiringMode.AUTOMATIC，表示自动装配的意思，这里咱们设置为手动装配：AiServiceWiringMode.EXPLICIT。chatModel注解用于指定对话时需要使用的模型对象在IOC容器中的名字，由于IOC容器中Bean对象的名字默认是类名首字母小写，所以这里的取值为openAiChatModel。

实际上，在使用AiService注解时，我们不手动的指定这两个属性的值，也就是说采用AiService的自动装配模式也是可以的。

ConsultantService.java

```
1  @AiService  
2  public interface ConsultantService {  
3      //用于聊天的方法,message为用户输入的内容  
4      public String chat(String message);  
5  }
```

只是如果我们手动设置的话，大家更容易理解这里究竟在做什么，所以将来咱们在使用AiService注解的时候，都采用手动装配的方式。

3.4 流式调用

在第二章大模型的使用中我们有讲到，调用大模型有两种方式：流式调用和阻塞式调用。在我们前面演示的过程中，其实都是用的是阻塞式调用，结果是一次性响应的，接下来我们学习如何使用LangChain4j发起流式调用。

3.4.1 流式调用步骤

3.4.1.1 引入依赖

pom.xml

```
1  <dependency>
2      <groupId>org.springframework.boot</groupId>
3      <artifactId>spring-boot-starter-webflux</artifactId>
4  </dependency>
5  <dependency>
6      <groupId>dev.langchain4j</groupId>
7      <artifactId>langchain4j-reactor</artifactId>
8      <version>1.0.1-beta6</version>
9  </dependency>
```

3.4.1.2 配置流式模型对象

之前咱们配置的是阻塞式对话模型对象，在流式调用中，我们需要使用LangChain4j的流式模型对象。和之前一样，也需要在配置文件中完成配置。

代码块

```
1  langchain4j:
2      open-ai:
3          streaming-chat-model: #流式模型配置
4              base-url: https://dashscope.aliyuncs.com/compatible-mode/v1
5              api-key: ${API-KEY}
6              model-name: qwen-plus
7              log-requests: true
8              log-responses: true
```

3.4.1.3 调整ConsultantService中的代码

ConsultantService中的chat方法的返回值类型，需要修改为支持流式处理的类型Flux，同时还需要在AiService注解中，通过streamingChatModel属性，配置一下流式调用的模型对象，值为openAistreamingChatModel

ConsultantService.java

```
1  @AiService(
2      wiringMode = AiServiceWiringMode.EXPLICIT,
3      chatModel = "openAiChatModel",
4      streamingChatModel = "openAistreamingChatModel"
5  )
6  public interface ConsultantService {
7      public Flux<String> chat(String message);
```

```
8 }
```

3.4.1.4 调整ChatController中的代码

ChatController.java

```
1  @RestController
2  public class ChatController {
3      @Autowired
4      private ConsultantService consultantService;
5
6      @RequestMapping(value = "/chat", produces = "text/html;charset=utf-8")
7      public Flux<String> chat(String memoryId, String message) {
8          Flux<String> result = consultantService.chat(memoryId, message);
9          return result;
10     }
11 }
```

其中@RequestMapping注解的produces属性，用于解决乱码问题。

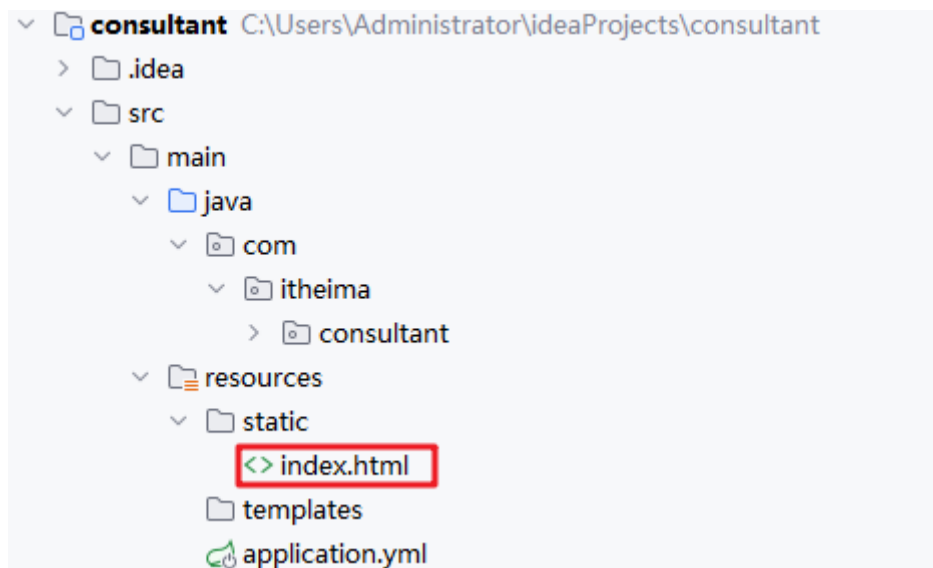
3.4.2 对接前端页面

我们将来成熟的项目肯定不能让用户通过地址栏输入，所以我们需要提供前端页面供用户更方便的使用，有关前端的知识不是本次课程的核心内容，所以这里就不带着大家一点一点儿写了，我已经提前给大家准备好了，大家只需要把资料中提供的前端页面直接拷贝到当前项目的static目录下，浏览器就能直接访问了。

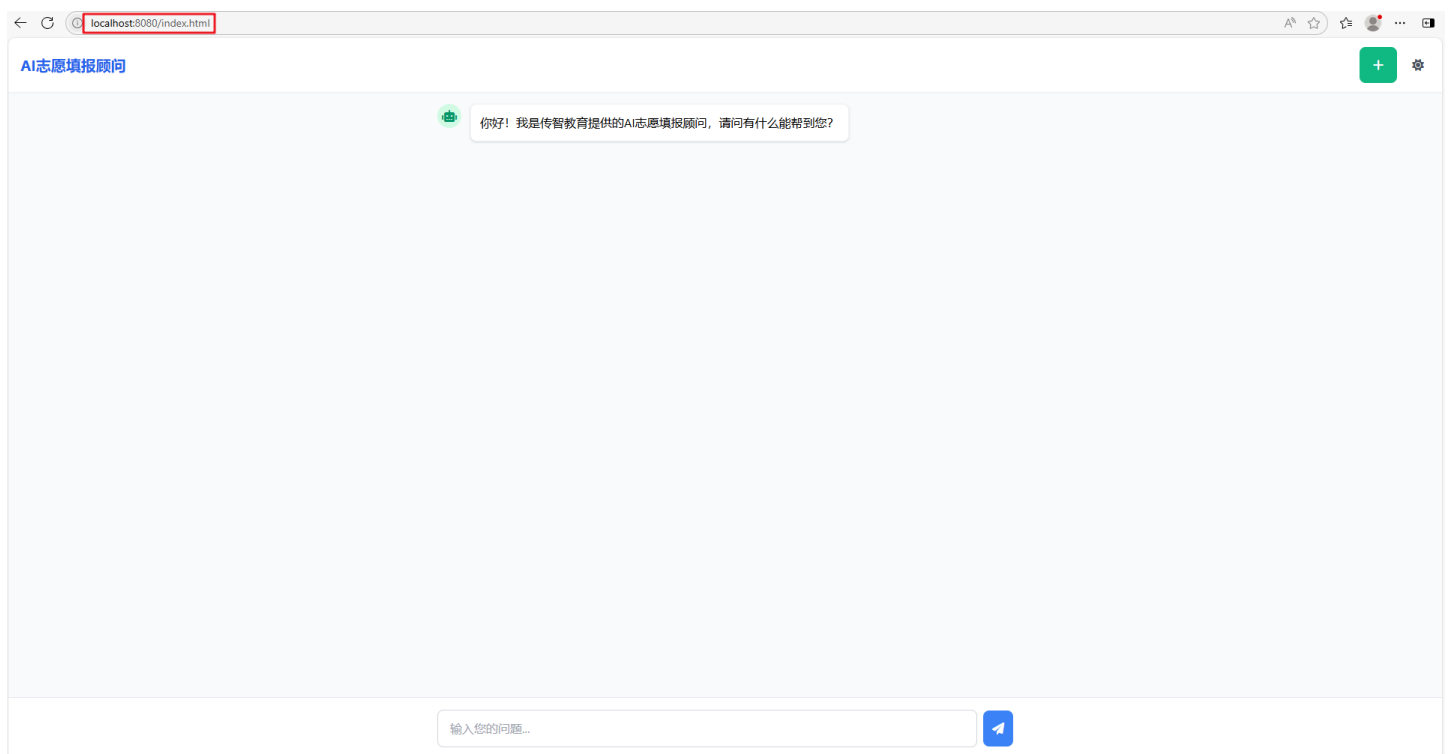
资料位置：

LangChain4J > 03_资料 > 02_素材 > 02_前端页面			
	名称	修改日期	类型
	 index.html	2025/5/21 15:30	Chrome HTML D...

项目结构：



浏览器访问：



3.5 消息注解

接下来我们学习langchain4j中提供的消息注解，将来我们开发的项目叫做Ai志愿填报顾问，它只能回答志愿填报相关的问题，如果用户问其他的问题，则不予回答。比如你问它特朗普靠谱吗？它是不能回答你的。如果来实现这样的效果，我们就需要通过设定系统消息的方式来完成了。



在LangChain4j中,提供了两个有关设置消息的注解,一个是SystemMessage, 另外一个UserMessage。

3.5.1 SystemMessage

先看SystemMessage，顾名思义，它是用于设置系统消息的，你可以直接在接口的方法上添加这个注解，在注解中书写系统消息即可。当然了, 如果我们的系统消息很长, 直接在代码中写不方便，它还提供了另外一种使用方式，通过fromResource属性，指定一个外部的文件。这样我们就可以把系统消息一次性的写入到外部文件中，管理起来也比较方便。

ConsultantService.java

```
1  @AiService(  
2      wiringMode = AiServiceWiringMode.EXPLICIT, //手动装配
```

```

3         chatModel = "openAiChatModel", //指定模型
4         streamingChatModel = "openAiStreamingChatModel"
5     )
6     //@AiService
7     public interface ConsultantService {
8         //@SystemMessage("你是东哥的助手小月月,人美心善又多金!")
9         @SystemMessage(fromResource = "system.txt")
10        public Flux<String> chat(String message);
11    }

```


咱们**Ai志愿填报顾问**项目中使用到的系统消息在资料中有提供，大家可以直接把资料中提供的system.txt复制到当前项目的resources目录下进行测试。

资料位置：

📁 > LangChain4J > 03_资料 > 02_素材 > 02_系统提示词

名称	修改日期	类型	大小
 system.txt	2025/5/27 10:46	文本文档	2 KB

项目结构：

📁 consultant	C:\Users\Administrator\ideaProjects\consultant
📁 .idea	
📁 src	
📁 main	
📁 java	
📁 com	
📁 itheima	
📁 consultant	
📁 resources	
📁 static	
📁 templates	
📄 application.yml	
 system.txt	
📁 test	
📄 pom.xml	

3.5.2 UserMessage

假设现在没有SystemMessage，那么我们可以借助于UserMessage注解完成同样的效果，我们可以在用户消息前后，拼接提前预设的内容下面给出一个使用示例：

```
1 @UserMessage("你是东哥的助手小月月，温柔貌美又多金。{{it}}")
2 public Flux<String> chat(String message);
```

上面示例中的参数message是用户传递的消息，我们在使用UserMessage注解的时候，可以通过{{it}}的方式，动态的获取到用户传递的消息，然后再往它的前后拼接上预设的内容即可，想拼什么拼什么。这里有一点需要说明，这个花括号内的it是固定的，不能随便写。假设你不想使用it这个名字，langchain4j提供了一个V注解，用于解决这个问题。我们在参数前面通过V注解给这个参数起一个名字，然后在花括号内写上同样的名字就能获取到了，下面是一个使用示例：

代码块

```
1 @UserMessage("你是东哥的助手小月月，温柔貌美又多金。{{msg}}")
2 public Flux<String> chat(@V("msg") String message);
```

⚠ 为了后续Ai志愿填报顾问这个项目的效果，测试完毕后请记得注释掉UserMessage相关的代码，保留 @SystemMessage(fromResource = "system.txt")设置。

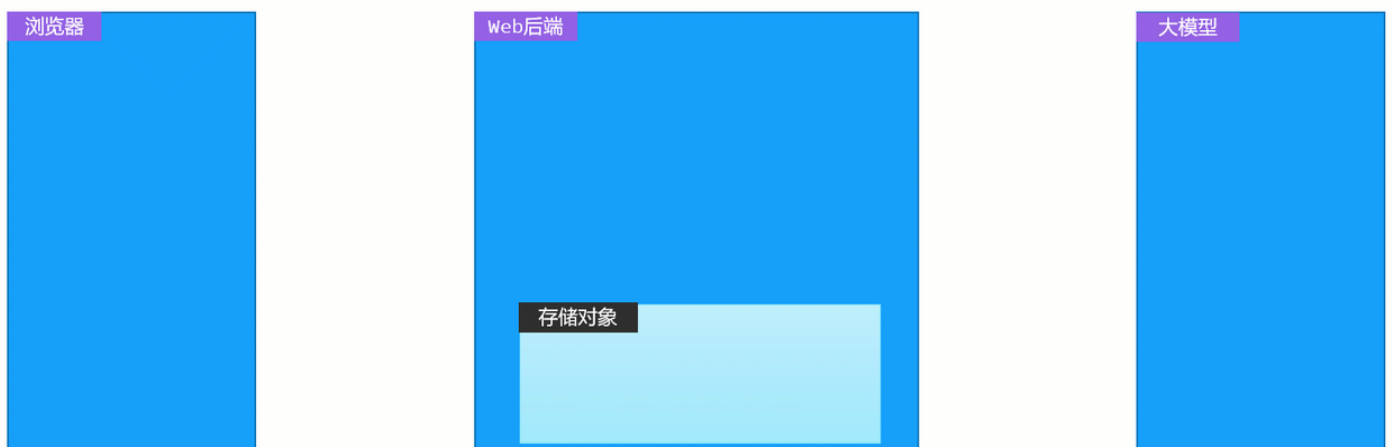
3.6 会话记忆

之前我们学习大模型使用的时候有讲过，大模型不具备记忆能力，每次会话都是独立的。要想让大模型产生记忆的效果，唯一的方法就是把之前聊天的内容和新的内容一起发送给大模型。

之前我们在apifox中演示会话记忆效果的时候，是手动拼接多条消息的，有了langchain4j了之后就不需要这么麻烦了，它能够帮我们记录聊天消息并自动发送！

3.5.1 会话记忆原理

我们通过下面这幅图给大家解释一下LangChain4j是如何实现会话记忆效果的。图中三个框，分别代表浏览器、web后端和大模型。将来我们借助于langchain4j可以准备一个专门用于存储会话记录的存储对象。



当用户问西北大学是211吗？它会把消息传递给后端，后端接收到消息后，会自动把消息存放到存储对象中，然后再获取存储对象中记录的所有会话消息，一块发送给大模型，当然现在存储对象中只记录了一条消息，所以只把一条消息发送给大模型。大模型根据接收到的消息，生成答案，比如说是的，再把答案响应给web后端，此时web后端会把得到的响应消息往存储对象中拷贝一份，然后再把响应消息发送给用户。

用户接收到答案后，接着问,是985吗？这条消息发送给web后端后，web后端依然会自动的把消息存放到存储对象中，此时存储对象中就存放了三条消息了，紧接着获取到存储对象中所有的会话消息，一并发送给大模型，这一次大模型就能够根据用户发送的所有会话记录进行推断回答了，这就是会话记忆的原理！

3.5.2 会话记忆基本实现

langchain4j提供了一个接口叫做ChatMemory，该接口中提供了add方法用于添加一条记录，messages方法用于获取所有的会话记录，clear方法用于清除所有的会话记录，这里还有一个id方法，它是用于唯一的标识一个存储对象，当然这个id暂时我们用不着，等会儿我们讲解会话记忆隔离的时候再给大家详细的讲解。同时LangChain4j还提供了该接口的两个实现类，一个是TokenWindowChatMemory，另外一个MessageWindowChatMemory, 咱们暂时先使用MessageWindowChatMemory来存储会话记录。

ChatMemory.java

```
1  public interface ChatMemory {
2      Object id(); //记忆存储对象的唯一标识
3
4      void add(ChatMessage var1); //添加一条会话记忆
5
6      List<ChatMessage> messages(); //获取所有会话记忆
7
8      void clear(); //清除所有会话记忆
9  }
```

3.5.2.1 定义会话记忆对象

我们需要在CommonConfig类中，构建MessageWindowChatMemory对象，并注入到IOC容器中。构建的时候我们可以指定该对象中最大的会话存储数量。给大家解释一下这里为什么要有一个最大的会话存储数量。首先是因为咱们大模型的上下文不是无限的，一般目前大模型支持的上下文最大在10w个token左右，也就是说你发送给大模型的消息不是无限制的，你发的太多了大模型也吃不消。这是第一个原因，另外一个原因是如果会话记录存储的太多，费用就会越贵。前面我们讲过，所有发送给大模型的消息都会转换成token，而平台就是按照token数量收费的，你发送的越多收费越高，所以这里我们需要设置一个合适的数量，一般设置20就够用了。如果要存储的消息超过了20条，那么最早存储的消息就会被淘汰，在存储对象中最多保留最新的20条消息。

CommonConfig.java

```
1  @Bean
2  public ChatMemory chatMemory() {
3      return MessageWindowChatMemory.builder()
4          .maxMessages(20) //最大保存的会话记录数量
5          .build();
6  }
```

3.5.2.2 配置会话记忆对象

我们需要在ConsultantService接口上的AiService注解中借助于chatMemory属性完成配置，值就是IOC容器中ChatMemory对象的名字，也就是我们构建该对象时使用的方法名。


ConsultantService.java

```
1  @AiService(
2      wiringMode = AiServiceWiringMode.EXPLICIT,
3      chatModel = "openAiChatModel",
4      streamingChatModel = "openAiStreamingChatModel",
5      chatMemory = "chatMemory" //配置会话记忆对象
6  )
7  public interface ConsultantService {
8      @SystemMessage(fromResource = "system.txt")
9      public Flux<String> chat(String message);
10 }
```

3.5.3 会话记忆隔离

刚才我们借助于MessageWindowChatMemory实现了会话记忆的效果，看起来还不错，但是还是有一些小问题的。当不同的用户访问我们的程序时，无法区分不同用户的会话记录，因为刚才实现的会话记忆，所有用户存储会话记录都是用的是同一个会话记忆对象，所以会话记忆并没有做到隔离，那应该怎么办呢？

还能不能记得当时我们介绍ChatMemory接口的时候，它提供了一个id方法，我们当时说它是用来唯一的标识某一个会话记忆对象的，在这里，我们需要借助它来完成会话记忆隔离。同样的，我们先通过一副动图来讲解一下LangChain4j是如何做到会话记忆隔离的。



在LangChain4j中可以准备一个容器，专门用于存储当前程序中所有的会话记忆对象。假设有一个用户访问我们的程序，此时它除了要把用户问题message携带给后端，还需要携带一个memoryId，假设它携带的memoryId为1，此时LangChain4j会先从容器中找有没有一个ChatMemory对象的id为1，如果有就使用，但是很明显现在没有。所以它会新建一个ChatMemory对象，并把当前的memoryId 1 设置给这个ChatMemory对象，并把会话记录存储到该对象中使用。

假设又有一个用户访问我们的程序，它携带的memoryId为2，同样的，LangChain4j也会从容器中找有没有一个ChatMemory对象的id为2，很显然还是没有，所以会创建一个新的ChatMemory对象，并把memoryId 2设置给这个ChatMemory对象，并把会话记录存储到该对象中使用。

注意，假设第二个用户继续访问我们的程序，它携带了同样的memoryId 2给后端，此时LangChain4j从容器中查找的时候发现已经存在一个ChatMemory对象的id为2，所以直接复用这个已经存在的ChatMemory对象，这样我们就可以借助于ChatMemory的id值实现不同会话之间的记忆隔离效果。

了解完了原理，接下来我们学习如何写代码才能事项会话记忆隔离。

3.5.3.1 定义会话记忆对象提供者

LangChain4j中提供了一个类ChatMemoryProvider，将来LangChain4j如果从容器中没有找到指定id的ChatMemory对象，就会调用ChatMemoryProvider对象的get方法获取一个新的ChatMemory对象使用，因此我们需要提供这个ChatMemoryProvider对象，实现get方法。这里的get方法，会接收一个参数，这个参数就是memoryId，返回一个结果就是ChatMemory对象。我们只需要在get方法中写清楚根据memoryId如何构建ChatMemory对象并返回的逻辑即可。当然这里我们依然构建的是MessageWindowChatMemory对象，只不过我们在构建的时候，除了要指定最大的会话记录数量外，还需要把memoryId设置给当前的ChatMemory对象。

CommonConfig.java

```
1  @Bean
2  public ChatMemoryProvider chatMemoryProvider() {
3      ChatMemoryProvider chatMemoryProvider = new ChatMemoryProvider() {
4          @Override
```

```

5         public ChatMemory get(Object memoryId) {
6             return MessageWindowChatMemory.builder()
7                 .id(memoryId)//id值
8                 .maxMessages(20)//最大会话记录数量
9                 .build();
10        }
11    };
12    return chatMemoryProvider;
13 }

```

3.5.3.2 配置会话记忆对象提供者

我们需要在AiService注解中，借助于chatMemoryProvider这个属性指定一下会话记忆对象提供者，跟之前的套路都是一样的，只不过我们既然提供了ChatMemoryProvider，之前提供的这个公有的ChatMemory就没有必要了，可以把它注释掉。

ConsultantService.java

```

1  @AiService(
2      wiringMode = AiServiceWiringMode.EXPLICIT,
3      chatModel = "openAiChatModel",
4      streamingChatModel = "openAiStreamingChatModel",
5      //chatMemory = "chatMemory",
6      chatMemoryProvider = "chatMemoryProvider"//配置会话记忆对象提供者
7  )

```

3.5.3.3 ConsultantService接口的方法中添加参数memoryId

我们在ConsultantService接口中，给chat方法添加一个参数memoryId，并且需要添加注解@MemoryId明确的告诉LangChain4j，将来我的第一个参数就是用于标识ChatMemory对象的id值，将来你就拿这个参数的值去容器中帮我匹配对象的ChatMemory对象，如果匹配到就复用，如果没有匹配到就调用ChatMemoryProvider对象的get方法获取一个新的使用。

这里有个小细节要注意，如果chat方法只有一个参数，那langchain4j会默认把这个参数当做用户消息来处理，如果chat方法有多个参数，我们就必须手动的指定哪个参数对应的是用户消息，所以我们需要在message参数前面手动的添加UserMessage注解，用于标识message对应的就是用户消息。

ConsultantService.java

```

1  @AiService(
2      wiringMode = AiServiceWiringMode.EXPLICIT,
3      chatModel = "openAiChatModel",
4      streamingChatModel = "openAiStreamingChatModel",
5      //chatMemory = "chatMemory",
6      chatMemoryProvider = "chatMemoryProvider"//配置会话记忆对象提供者

```

```

7   )
8   public interface ConsultantService {
9       @SystemMessage(fromResource = "system.txt")
10      public Flux<String> chat(@MemoryId String memoryId, @UserMessage String
11                               message);
12  }

```

3.5.3.4 ChatController中chat接口接收前端传递的memoryId

ChatController.java

```

1   @RequestMapping(value = "/chat", produces = "text/html; charset=utf-8")
2   public Flux<String> chat(String memoryId, String message) {
3       Flux<String> result = consultantService.chat(memoryId, message);
4       return result;
5   }

```

3.5.3.5 前端访问/chat接口是提交memoryId参数

在咱们之前提供的index.html前端页面中，已经提交了这个参数，只是之前的代码我们一直没有接收这个参数。所以这一块我们无需写任何代码，直接启动测试即可。

3.5.4 会话记忆持久化

刚才我们完成了会话记忆隔离，其实我们的会话记忆还是有一些瑕疵的，只要后端重启，会话记忆就没有了，丢失了。先分析一下为什么会存在这种问题，之前我们一直构建的用于存储会话记录的对象是MessageWindowChatMemory，而这个对象内部维护了一个成员变量ChatMemoryStore，其实我们使用MessageWindowChatMemory对象的add方法添加会话记录的时候，真正用于存储的对象是这个ChatMemoryStore，所以要分析为什么重启之后会话记录会丢失，我们得分析ChatMemoryStore是如何存储会话记录的。

MessageWindowChatMemory.java

```

1   public class MessageWindowChatMemory implements ChatMemory {
2       private final String id;
3       private final ChatMemoryStore store; // 这个对象用于存储会话记录
4
5       public void add(ChatMessage message) {
6           this.store.updateMessages(this.id, messages);
7       }
8       public List<ChatMessage> messages() {
9           return this.store.getMessages(this.id);
10      }

```

```

11     public void clear() {
12         this.store.deleteMessages(this.id);
13     }
14 }

```

ChatMemoryStore是一个接口，它里面提供了getMessages、updateMessages、deleteMessages方法分别用于根据memoryId获取会话记录，根据memoryId更新会话记录以及根据memoryId删除会话记录。

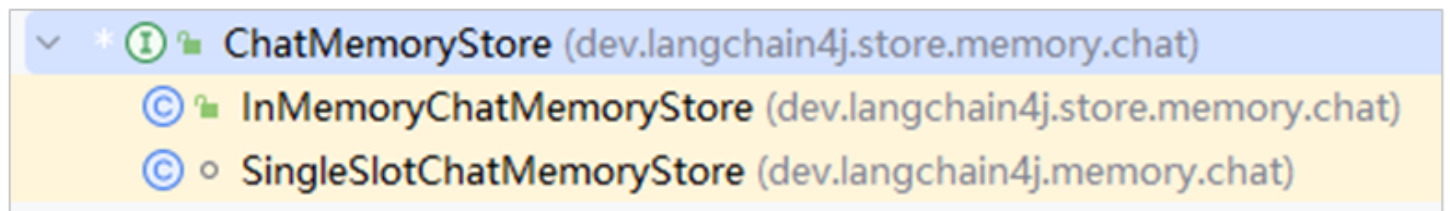
ChatMemoryStore.java

```

1  public interface ChatMemoryStore {
2      List<ChatMessage> getMessages(Object memoryId);
3      void updateMessages(Object memoryId, List<ChatMessage> messages);
4      void deleteMessages(Object memoryId);
5  }

```

LangChain4j为该接口提供了两个实现类，分别是InMemoryChatMemoryStore和SingleSlotMemoryStore。而我们MessageWindowChatMemory中默认使用的Store对象就是这个SingleChatMemoryStore。



接下来我们重点分析它里面又是如何存储会话记录的。

SingleSlotChatMemoryStore.java

```

1  class SingleSlotChatMemoryStore implements ChatMemoryStore {
2      private List<ChatMessage> messages = new ArrayList(); //用于存储会话记录
3      public List<ChatMessage> getMessages(Object memoryId) {
4          return this.messages;
5      }
6      public void updateMessages(Object memoryId,
7                                 List<ChatMessage> messages) {
8          this.messages = messages;
9      }
10     public void deleteMessages(Object memoryId) {
11         this.messages = new ArrayList();
12     }
13 }

```

在SingleSlotChatMemoryStore中维护了一个集合对象messages，它就是使用这个集合存储会话消息的，所以很明显这是内存存储，一旦当服务器重启后这些消息必然会丢失！

接下来我们要做的事情就是将会话记录持久化存储到外部的存储器中，比如mysql、redis、mongo等等都可以。最直观的解决思路就是我不让MessageWindowChatMemory使用SingleSlotChatMemoryStore去维护会话记录，咱们自己提供一个ChatMemoryStore的实现类，在实现类中把消息存储到其它地方，然后再把咱们自己提供的ChatMemoryStore交给MessageWindowChatMemory即可。在咱们本次课程中, 我们把会话记录存储在redis中。

3.5.4.1 准备redis环境

首先参考资料中提供的docker desktop安装文档，在windows上搭建docker环境。

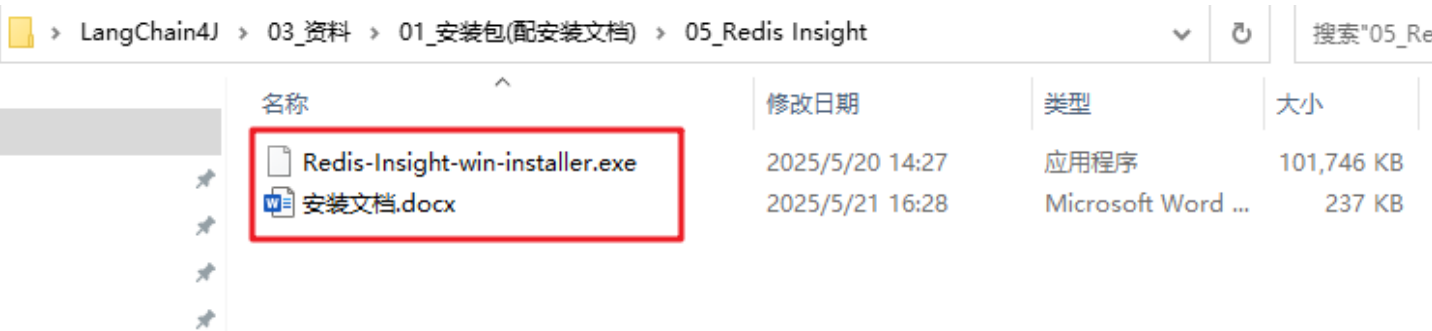


其次在命令符提示窗口中执行命令安装redis，在windows上映射的端口为6379。

代码块

```
1 docker run --name redis -d -p 6379:6379 redis
```

最后参考资料中提供的 redis insight安装文档，安装redis图形化界面客户端。



3.5.4.2 引入redis起步依赖

```

pom.xml
1  <dependency>
2      <groupId>org.springframework.boot</groupId>
3      <artifactId>spring-boot-starter-data-redis</artifactId>
4  </dependency>
```

3.5.4.3 配置redis连接信息

代码块

```
1  spring:
2    data:
3      redis:
4        host: localhost
5        port: 6379
```

3.5.4.4 提供ChatMemory实现类操作redis

定义实现类实现ChatMemory接口，重写getMessages、updateMessages、deleteMessages方法，用于操作redis，并且把实现类的对象注入到IOC容器中。

RedisChatMemoryStore.java

```
1  import dev.langchain4j.data.message.ChatMessage;
2  import dev.langchain4j.data.message.ChatMessageDeserializer;
3  import dev.langchain4j.data.message.ChatMessageSerializer;
4  import dev.langchain4j.store.memory.chat.ChatMemoryStore;
5  import org.springframework.beans.factory.annotation.Autowired;
6  import org.springframework.data.redis.core.StringRedisTemplate;
7  import org.springframework.stereotype.Repository;
8
9  import java.time.Duration;
10 import java.util.List;
11
12 @Repository
13 public class RedisChatMemoryStore implements ChatMemoryStore {
14     //注入RedisTemplate
15     @Autowired
16     private StringRedisTemplate redisTemplate;
17     @Override
18     public List<ChatMessage> getMessages(Object memoryId) {
19         //获取会话消息
20         String json = redisTemplate.opsForValue().get(memoryId);
21         //把json字符串转化成List<ChatMessage>
22         List<ChatMessage> list =
23             ChatMessageDeserializer.messagesFromJson(json);
24         return list;
25     }
26     @Override
27     public void updateMessages(Object memoryId, List<ChatMessage> list) {
28         //更新会话消息
29         //1.把list转换成json数据
```

```

30         String json = ChatMessageSerializer.messagesToJson(list);
31         //2.把json数据存储到redis中
32         redisTemplate.opsForValue().set(memoryId.toString(), json,
Duration.ofDays(1));
33     }
34
35     @Override
36     public void deleteMessages(Object memoryId) {
37         //删除会话消息
38         redisTemplate.delete(memoryId.toString());
39     }
40 }

```

3.5.4.5 配置ChatMemoryStore

将我们提供的ChatMemoryStore配置给MessageWindowChatMemory对象使用。

CommonConfig.java

```

1  @Autowired
2  private ChatMemoryStore redisChatMemoryStore;
3
4  @Bean
5  public ChatMemoryProvider chatMemoryProvider(){
6      ChatMemoryProvider chatMemoryProvider = new ChatMemoryProvider() {
7          @Override
8          public ChatMemory get(Object memoryId) {
9              return MessageWindowChatMemory.builder()
10                 .id(memoryId)
11                 .maxMessages(20)
12                 .chatMemoryStore(redisChatMemoryStore) //配置ChatMemoryStore
13                 .build();
14            }
15        };
16        return chatMemoryProvider;
17    }

```

3.7 RAG知识库

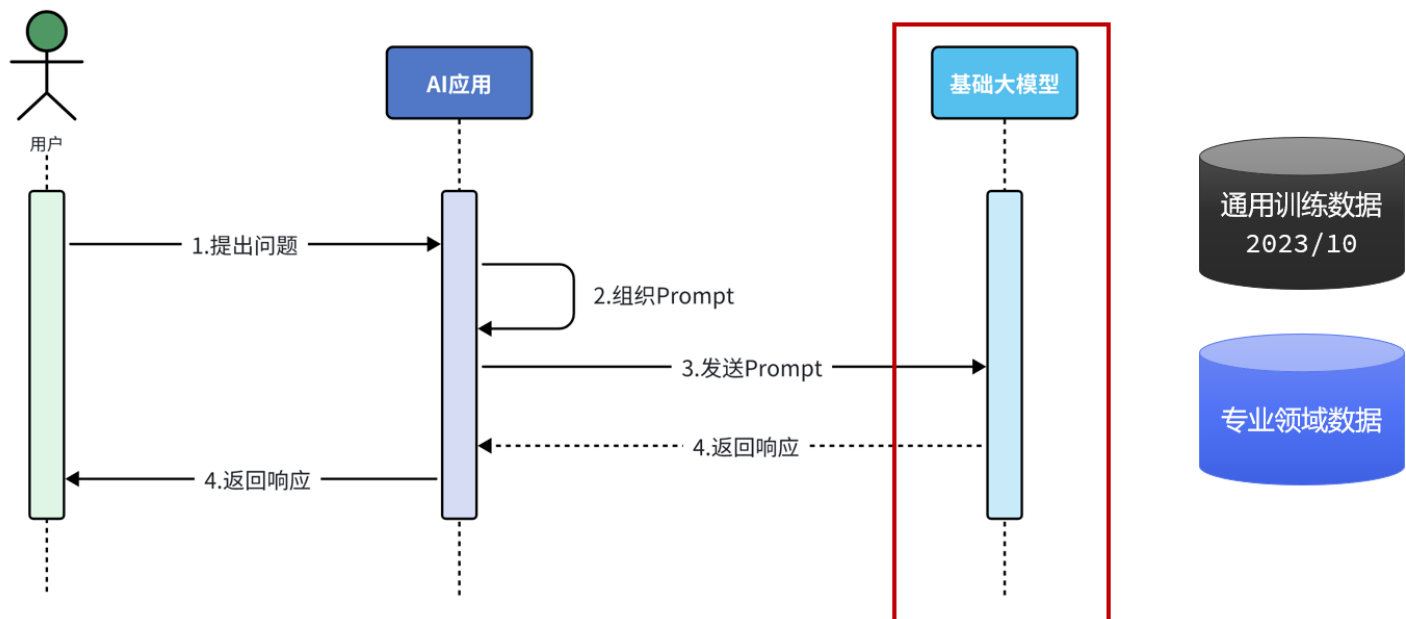
咱们目前的AI志愿填报顾问还存在一个问题，无法查询各个高校2024年最新的录取分数，其原因在于咱们使用的qwen-plus大模型最后一次训练是在2023年10月，在这个时间之后产生的新的数据，大模型是无法感知的。



如果要解决这个问题，就需要使用RAG相关的知识了。

3.7.1 RAG原理

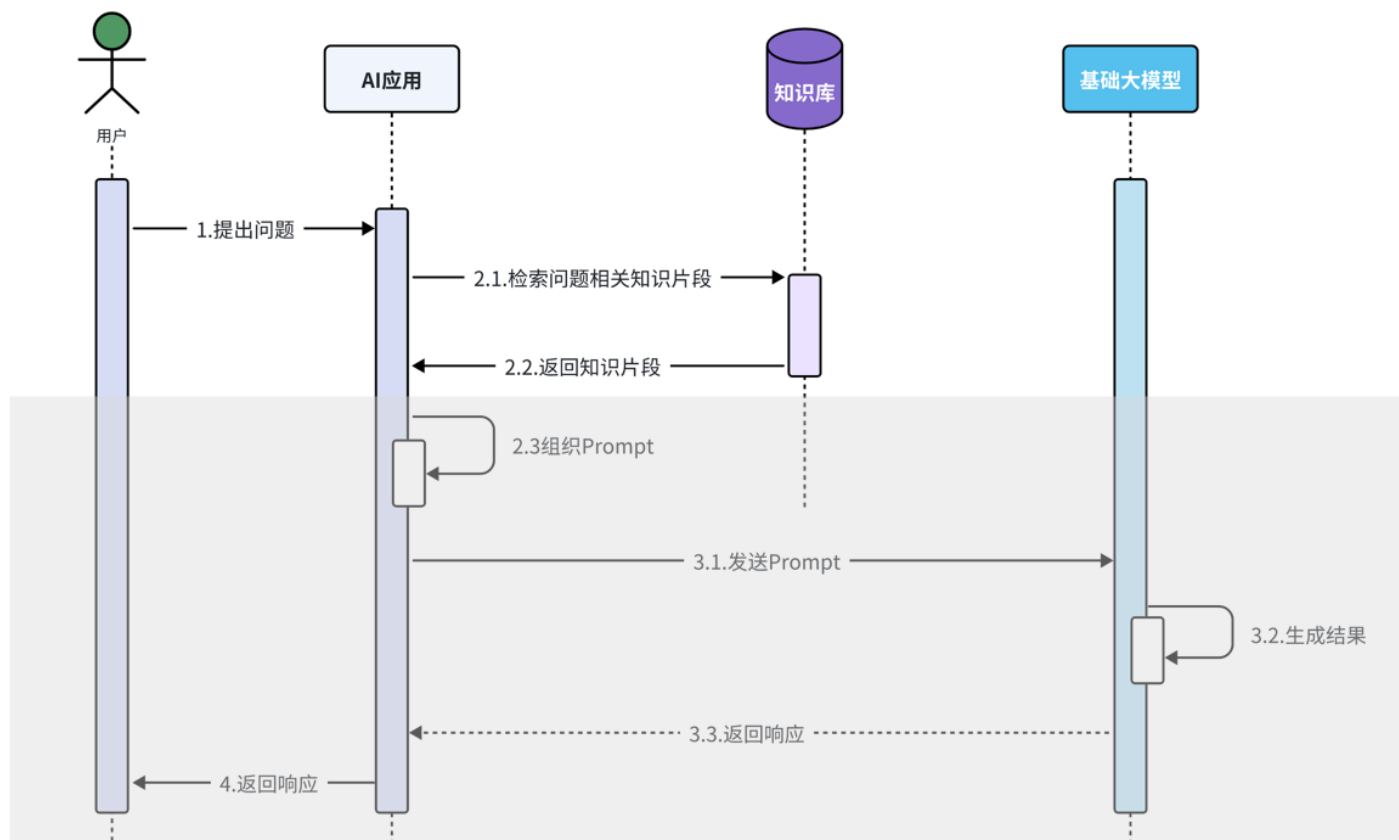
RAG全称为 Retrieval Augmented Generation，翻译过来是检索增强生成，简单理解就是通过检索外部知识库的方式增强大模型的生成能力。



正常情况下当用户把问题发送给AI应用，AI应用内部组织调用大模型的数据并发送给大模型，接下来大模型会根据接收到的数据生成结果并响应给AI应用，然后AI应用再把接收到的消息响应给用户。这是咱们目前程序的一个基本工作流程。

由于咱们一旦把大模型训练完毕后，随着时间的推移产生的新数据大模型是无法感知的，而且训练大模型的时候一般使用的都是通用的训练数据，有关专业领域的知识，大模型也是不知道的。所以，如果要想让大模型能根据最新的数据或者专业领域的知识回答问题，我们就需要给它外挂一个知识库，这就是rag要做的事情。

一旦当我们外挂了知识库后，整个工作流程会发生一些变化。



当用户把问题发送给AI应用，AI应用会先根据用户的问题从知识库中检索对应的知识片段，得到知识片段后AI应用需要结合用户的问题以及知识库中检索到的知识片段组织要发送给大模型的消息，大模型接收到消息后会同时根据用户的问题、知识库检索到的知识片段以及自身的知识储备，生成对应的结果响应给AI应用，最终再返回给用户。这是我们外挂知识库后，AI应用的工作流程。看起来比之前要复杂一些，但好消息是，下面的这一部分工作LangChain4j都能帮我们自动完成，我们需要关注的核心有两个：一个是知识库应该怎么搭建，另外一个是如何从知识库中检索出用户问题相关的知识片段。

这个知识库一般采取的是一种特殊的数据库，叫向量数据库。目前市面上常见的向量数据库有很多，比如Milvus、Chroma、Pinecone这些专用的向量数据库，还有一些传统的数据库做了向量化扩展，比如redis提供了RedisSearch用于完成向量存储，PostgreSQL提供了pgvector用于完成向量存储，不管是哪一种向量数据库原理都是一样的，使用也都大差不差。



向量数据库：

Milvus、Chroma、Pinecone

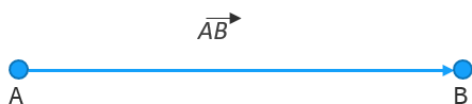
RedisSearch(Reids)、pgvector(PostgreSQL)

接下来我们聊一聊向量数据库是如何存储数据以及如何检索与用户问题相关的数据片段，要搞清楚这些首先我们得搞清楚什么是向量。其实向量这个东西咱们高中数学都有学过，我在这里带着大家一块复习一下。

向量：是数学和物理学中表示大小和方向的量

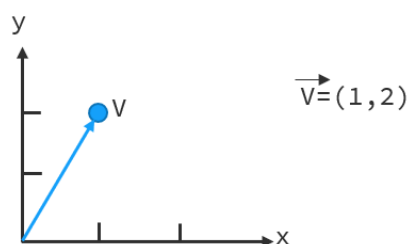
● 几何表示

有向线段，向量可以用一条带箭头的线段表示，线段的长度表示大小，箭头的方向表示方向。



● 代数表示

坐标表示，在直角坐标系中，向量可以用一组坐标来表示。

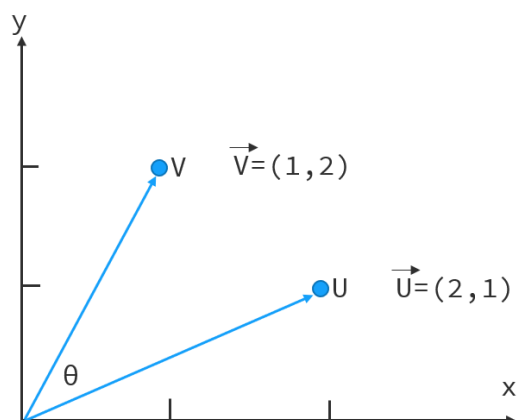


向量是数学和物理中表示大小和方向的量，常见的表示方式有两种：一种是几何表示，另外一种是用代数表示。在几何中，向量可以用一条带箭头的线段表示，线段的长度表示大小，箭头的方向表示方向。比如有两个点A点和B点，那么A点到B点之间的有向线段就可以记作向量AB。

在代数中向量可以表示为一组坐标，比如一个直角坐标系，横轴为X，纵轴为Y，在坐标系中有一个点V，我们记作向量V(1,2)，其中1是V点在X轴的取值，2是V点在Y轴的取值。其实在坐标系中表示的向量也可以转化为几何向量表示，V是终点，默认的起点是坐标原点，那么向量V表示的是原点到V点的有向线段。

了解完什么是向量我们聊一聊与向量有关的一个知识，叫做余弦相似度。

向量余弦相似度，用于表示坐标系中两个点之间的距离远近



$$\cos\theta(\text{向量余弦相似度}) = \frac{V \cdot U}{|V| |U|} = \frac{1 \cdot 2 + 2 \cdot 1}{\sqrt{1^2 + 2^2} * \sqrt{2^2 + 1^2}} = 0.8$$

假设：U点和V点重合，也就是 $\vec{V} = (1, 2)$ $\vec{U} = (1, 2)$

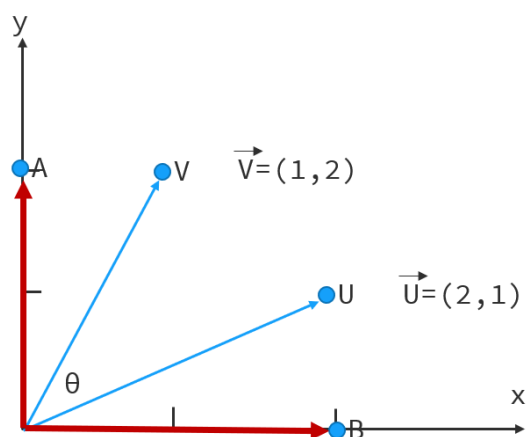
$$\cos\theta(\text{向量余弦相似度}) = \frac{V \cdot U}{|V| |U|} = \frac{1 \cdot 1 + 2 \cdot 2}{\sqrt{1^2 + 2^2} * \sqrt{2^2 + 1^2}} = 1$$

向量的余弦相似度用于标识坐标系中,两个点之间的距离远近。在直角坐标系中有两个点那个v和u，向量v和向量u之间有一个夹角 θ ，我们所说的向量的余弦相似度其实就是指这个夹角 θ 的cosin值，根据高中学过的公式，向量夹角的cosin值等于向量的内积除以向量模长的乘积。两个向量的内积为对应

坐标的乘积和，所以分母是 $1^2 + 2^2$ ，向量的模长为当前向量所有坐标的平方和再开方，所以分母为根号 $1^2 + 2^2$ 再乘以 根号 $2^2 + 1^2$ 。分母是4，分子是5，最终的结果是0.8。

接下来我们脑补一下，假设U点和V点重合了，也就是说两个向量重合了，两个点之间的距离为0，我们计算一下重合的两个向量的余弦相似度为多少？依然是内积除以模的乘积，分子为 $1*1 + 2*2$ 得到5，分母为根号 $1^2 + 2^2$ 乘以 $2^2 + 1^2$ ，得到也是5，所以余弦相似度为1，也就是说如果两个向量重合，对应的两个点之间的距离为0，此时余弦相似度为1，这是非常极端的一种情况。

接下来我们考虑另外一种非常极端的情况。

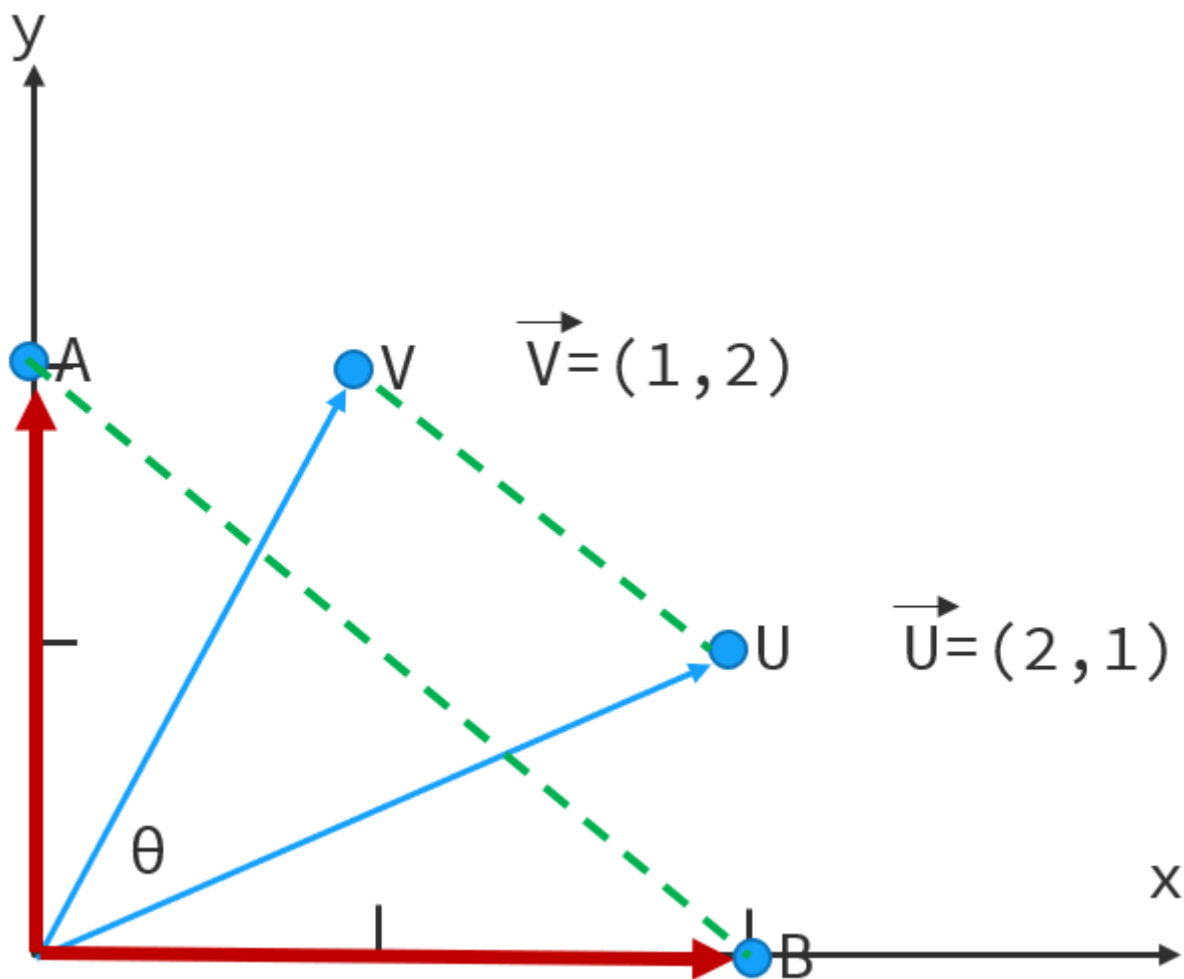


假设: $\vec{A} = (0, 2)$ $\vec{B} = (2, 0)$

$$\cos\theta (\text{向量余弦相似度}) = \frac{\vec{V} \cdot \vec{U}}{|\vec{V}| |\vec{U}|} = \frac{0*2 + 2*0}{\sqrt{0^2 + 2^2} * \sqrt{2^2 + 0^2}} = 0$$

假设有两个向量A和B，其中A向量的x坐标为0，y坐标为2，B向量的x坐标为2，y向量为0。此时两个向量处于正交状态，也就是夹角 θ 为90度。接下来我们算一下它们的余弦相似度。依然是内积除以模的乘积，分子为 $0*2 + 2*0$ ，得到0，分母是根号 $0^2 + 2^2$ 乘以 根号 $2^2 + 0^2$ ，得到4，最终余弦相似度为0。

此时大家有没有发现保持模长不变的情况下，当余弦相似度为0时，两点之间的距离最远，当余弦相似度为1时，两点之间的距离最近，而余弦相似度处于0~1之间时，两点之间的距离也是介于两种极端情况中间的。

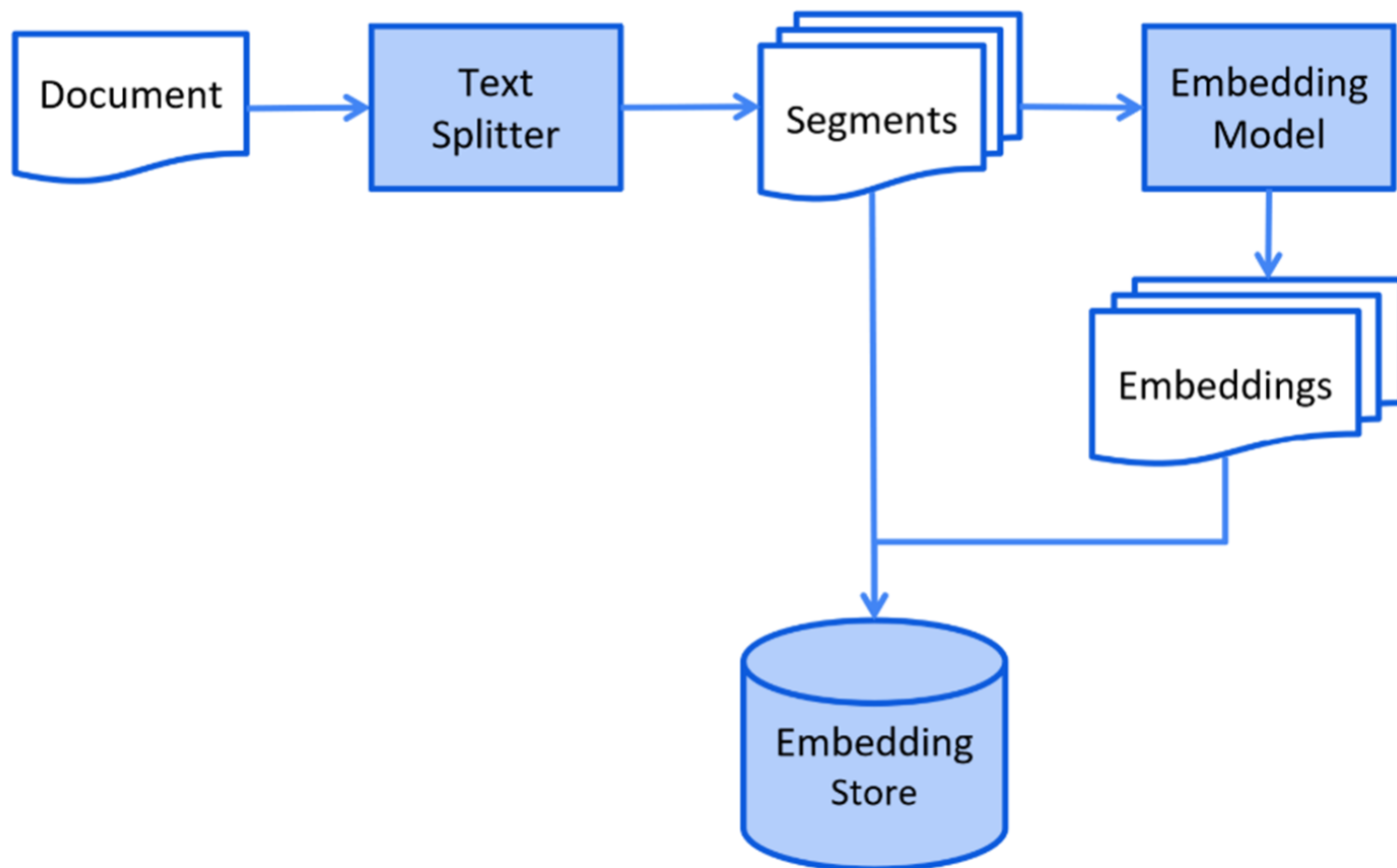


因此我们得出一个结论，在第一象限中，向量之间的余弦相似度的取值范围为0~1，而且余弦相似度越大，说明向量的方向越接近，对应的两点之间的距离越小。

刚才我们举得例子都是二维向量，其实当你把二维向量搞明白了，多维向量也是一模一样的。比如三维向量的记法就是记录三个轴的坐标，四维向量的记法就是记录四个轴的坐标，N维向量的记法就是记录N个轴的坐标。而我们RAG知识库中使用的向量，一般是几百个维度到几千个维度不等，不管他们有多少个维度，咱们之前得出的公式和结论都是通用的。余弦相似度的算法都是内积除以模的乘积，而且**两个向量的余弦相似度越大，向量方向越接近，两点之间的距离也就越小。**

- 二维向量： $V=(v1,v2)$
- 三维向量： $V=(v1,v2,v3)$
- 四维向量 $V=(v1,v2,v3,v4)$
- N维向量： $V=(v1,v2,v3...vn)$

聊完了向量相关的知识接下来我们聊一聊RAG中如何使用向量数据库存储数据。下面的流程图是LangChain4j官网给出的针对于RAG知识库存储流程的解释，我们简单的看一看。



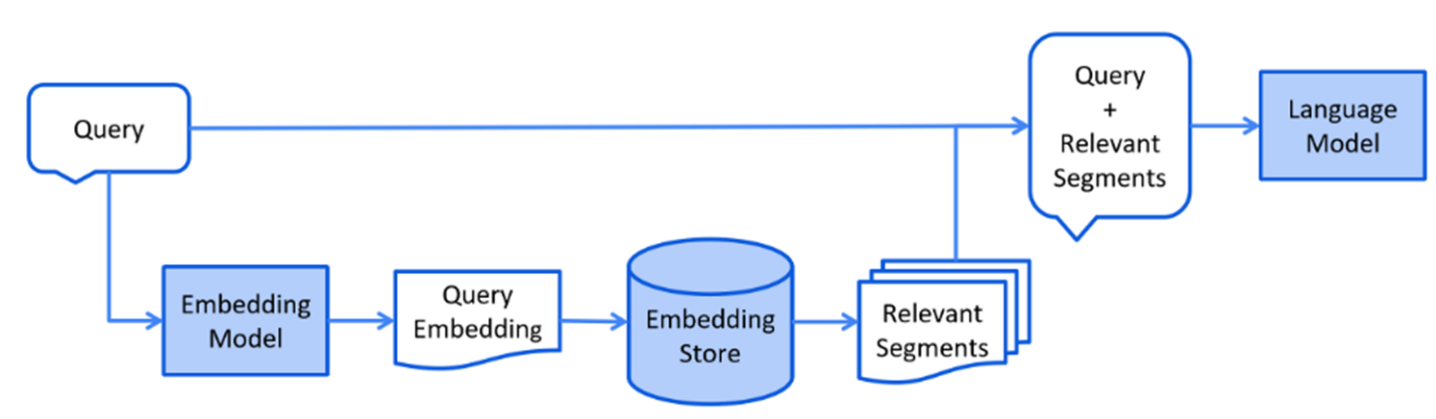
首先我们需要把最新的数据或者专业的数据存储在文档中，接下来借助于文本分割器把一个大的文档分割成一个一个小的文本片段，然后这些小的文本片段要使用一种专门的大模型：向量模型，之前我们介绍大模型的时候有讲过，不同的大模型擅长的领域不一样，有擅长文本处理的、有擅长图片处理的，其中就有一种大模型擅长文本向量化。借助于向量模型把一个一个的文本片段转换成向量，接下来把每一个向量和其对应的文本片段一块存储到向量数据库中。

为了大家更好的理解, 我给大家举个例子。

我爱上班
上班真好
我爱工作
拒绝加班
我要躺平

比如我有一个大的文档，里面存储了一些文本信息，接下来借助于文本分割器把大的文档切割成一个一个的文本片段，比如这里切割为我爱上班、上班真好、我爱工作、拒绝加班、我要躺平这五个小片段。紧接着使用向量模型把文本片段转化为向量，那我们之前聊过所谓的向量在坐标系中表示就是记录每一个轴的坐标，说白了就是一堆数字，最后再把每一个向量和其对应的文本片段组合成一条一条的数据存储到向量数据库中。

这样, 就给大家介绍完了在RAG中往向量数据库中存储数据的过程。接下来给大家介绍一下如何从向量数据库中检索出跟用户问题相关的文本片段，这里同样是一副LangChain4j提供的流程图，用于说明整个检索过程的，我们简单的看一看。



用户提交的消息需要使用向量模型转换为向量，接下来拿着该向量和向量数据库中已经存在的向量进行比对，计算他们之间的余弦相似度，把满足要求的向量筛选出来得到其对应的文本片段，最后结合用户提交的消息和从向量数据库中检索到的文本片段，组织数据发送给大模型。

同样的，为了大家更好的理解我在这里也给大家举个例子。

你爱上上班吗？

假如用户提交了一条消息，你爱上上班吗？接下来需要使用向量模型将这条消息转换成向量，其实得到的就是一组坐标数据。紧接着拿着该向量和向量数据库中的向量比对，计算余弦相似度，假设最终计算的结果分别为 0.8、0.6、0.7、0.3、0.2，之前我们讲过，两个向量的余弦相似度越大说明向量方向越接近，两点之间的距离越小。由于RAG中，向量都是由文本转换过来的，不同文本对应的向量余弦相似度越大说明对应文本之间的距离越近，那么对应文本的相似度就越高也就是说该向量对应的文本片段跟用户问的问题相关度越高。假设我设置一个标准：只有余弦相似度超过0.5的文本能被查出来。

此时,我爱上班、上班真好、我爱工作这三个片段就被检索出来了,最后再把用户的问题和检索出来的这三个文本片段一并发送给大模型,让大模型生成结果即可。

截此为止,有关RAG知识库的原理就给大家解释完了,有了这个理解基础,接下来我们的操作你就会理解的更加透彻!

3.7.2 RAG快速入门

要在我们的案例中通过rag的方式增强大模型的生成能力,从而让我们能够查询出最新的2024年的录取分数线,我们有两个工作要完成,分别是存储和检索。

3.7.2.1 存储

3.7.2.1.1 引入依赖

这一块我们引入的依赖是langchain4j-easy-rag,看名字我们就知道这是一个简易版本的rag实现方案,这个依赖中提供了内存版的向量数据库和向量模型供我们使用。

代码块

```
1 <dependency>
2   <groupId>dev.langchain4j</groupId>
3   <artifactId>langchain4j-easy-rag</artifactId>
4   <version>1.0.1-beta6</version>
5 </dependency>
```

3.7.2.1.2 加载知识数据文档

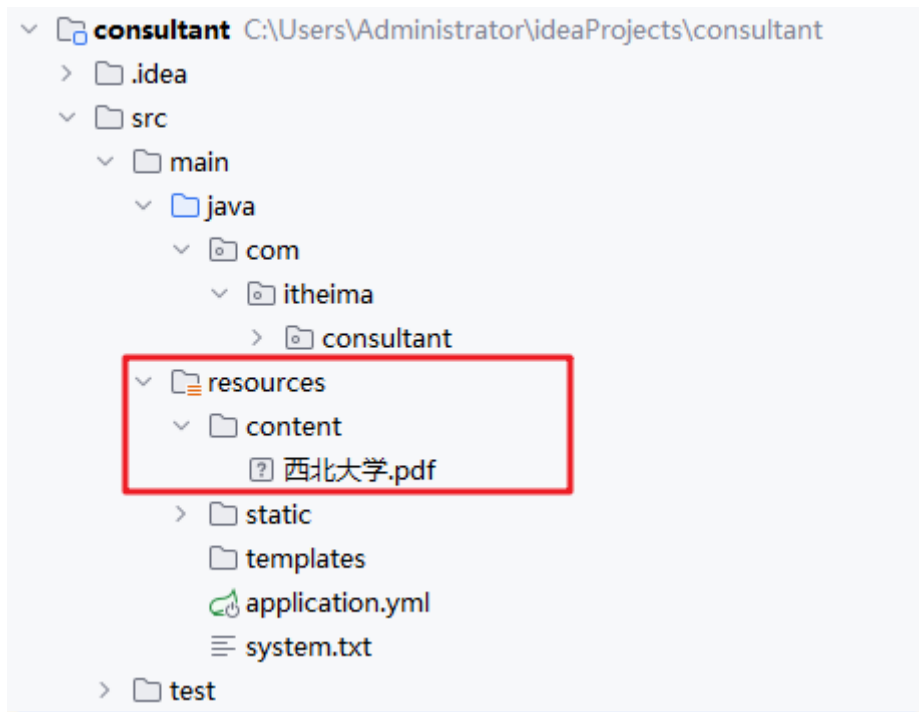
A. 将资料中准备的《西北大学.pdf》拷贝到当前工程的 resources/content目录下

LangChain4J > 03_资料 > 02_素材 > 03_各大高校本科招生信息 > 普通文本

搜索"普通文本"

名称	修改日期	类型	大小
热门专业top20.md	2025/5/20 20:33	Markdown File	4 KB
陕西师范大学.md	2025/5/26 14:43	Markdown File	13 KB
天坑专业top10.md	2025/5/20 20:33	Markdown File	3 KB
西安电子科技大学.md	2025/5/26 14:44	Markdown File	7 KB
西安交通大学.md	2025/5/26 14:46	Markdown File	11 KB
西北大学.md	2025/6/10 11:43	Markdown File	9 KB
西北工业大学.md	2025/5/26 14:47	Markdown File	6 KB
西北农林科技大学.md	2025/5/26 14:48	Markdown File	12 KB
长安大学.md	2025/5/26 14:49	Markdown File	6 KB

结



B. LangChain4j提供的ClassPathDocumentLoader可以让我们快速的将指定目录下的文档加载进内存中，并且每一个文档，都会对应的生成一个Document对象来记录文档的内容。这一部分工作需要在CommonConfig.java中完成。

CommonConfig.java

```
1  @Bean
2  public EmbeddingStore store(){
3      //1.加载文档进内存
4      List<Document> documents =
        ClassPathDocumentLoader.loadDocuments("content");
5      return null;
6  }
```

3.7.2.1.3 构建向量数据库操作对象EmbeddingStore

其实在我们引入的依赖中已经提供了一个用于操作内存版本的向量数据库的类 InmemoryEmbeddingStore，Inmemory是内存的意思，Embedding翻译过来是嵌入/向量的意思，Store是存储的意思，顾名思义，操作内存向量数据库。我们只要new出来一个对象即可。

CommonConfig.java

```
1  @Bean
2  public EmbeddingStore store(){
3      //1.加载文档进内存
4      List<Document> documents =
        ClassPathDocumentLoader.loadDocuments("content");
5      //2.构建向量数据库操作对象 操作的是内存版本的向量数据库
6      InMemoryEmbeddingStore store = new InMemoryEmbeddingStore();
7      return store;
```

3.7.2.1.4 切割文档、向量化并存储到向量数据库

LangChain4j中给我们提供了一个类EmbeddingStoreIngestor，它把很多细节都封装起来了，可以帮助我们快速的完成这一步的操作。首先我们构建EmbeddingStoreIngestor对象，构建的时候告诉它我要把向量化的数据存储到哪里？也就是把第三步构建的EmbeddingStore设置给它，接下来调用它的ingest方法，把需要存储数据的文档对象documents给它传递进去。在这个方法的内部会使用它内置的文本分割器先分割，然后使用内置的向量模型完成向量化，最后再把向量存储到向量数据库中。

CommonConfig.java

```

1  @Bean
2  public EmbeddingStore store(){
3      //1.加载文档进内存
4      List<Document> documents =
        ClassPathDocumentLoader.loadDocuments("content");
5      //2.构建向量数据库操作对象 操作的是内存版本的向量数据库
6      InMemoryEmbeddingStore store = new InMemoryEmbeddingStore();
7      //3.构建一个EmbeddingStoreIngestor对象,完成文本数据切割,向量化,存储
8      EmbeddingStoreIngestor ingestor = EmbeddingStoreIngestor.builder()
9          .embeddingStore(store)
10         .build();
11     ingestor.ingest(documents);
12     return store;
13 }
```

3.7.2.2 检索

3.7.2.2.1 构建ContentRetriever对象

LangChain4j提供的向量数据库检索对象叫做EmbeddingStoreContentRetriever，构建的时候我们可以设置三个内容。第一个得调用embeddingStore方法告诉它从哪里检索，其实就是我们刚才构建的这个InMemoryEmbeddingStore给他即可；第二个我们可以设置一下最小余弦相似度的值，之前我们讲过检索的时候会把用户的问题向量化，然后与向量数据库中已经存在的向量计算余弦相似度，值越大，相似度越高，这里通过minScore方法设置一个最低的相似度分数，可以确保检索出来的内容跟用户问题的相关度比较高；第三个可以设置一个最大检索出来的片段数量值，因为将来如果检索出来的片段太多，一并发送给大模型，token的消耗是比较大的，而且分数低的片段你发送给大模型还会影响生成的结果，这里通过maxResults方法设置最大的片段数量后，它会保留分数最高的前几个片段使用。这些操作也是在CommonConfig.java中完成

CommonCnfig.java

```

1  @Bean
```

```

2  public ContentRetriever contentRetriever(EmbeddingStore store){
3      return EmbeddingStoreContentRetriever.builder()
4          .embeddingStore(store)//设置向量数据库操作对象
5          .minScore(0.5)//设置最小分数
6          .maxResults(3)//设置最大片段数量
7          .build();
8  }

```

3.7.2.2.2 配置ContentRetriever对象

跟我们前面是类似的，在AiService注解中借助于contentRetriver这个属性完成配置即可。

代码块

```

1  @AiService(
2      wiringMode = AiServiceWiringMode.EXPLICIT,//手动装配
3      chatModel = "openAiChatModel",//指定模型
4      streamingChatModel = "openAiStreamingChatModel",
5      //chatMemory = "chatMemory",//配置会话记忆对象
6      chatMemoryProvider = "chatMemoryProvider",//配置会话记忆提供者对象
7      contentRetriever = "contentRetriever"//配置向量数据库检索对象
8  )
9  //@AiService
10 public interface ConsultantService {
11     //用于聊天的方法
12     //public String chat(String message);
13     //@SystemMessage("你是东哥的助手小月月,人美心善又多金!")
14     @SystemMessage(fromResource = "system.txt")
15     //@UserMessage("你是东哥的助手小月月,人美心善又多金!{{it}}")
16     //@UserMessage("你是东哥的助手小月月,人美心善又多金!{{msg}}")
17     public Flux<String> chat(*@V("msg")*/*@MemoryId String memoryId,
18     @UserMessage String message);
19 }

```

3.7.2.3 测试

查询AI志愿填报顾问：西北大学2024年录取分数？已经可以正确的根据知识库的内容回答了。



可以在IDEA的控制台查看日志，会发现发送给大模型的用户消息中，格式是这样的：

用户问题\n\nAnswer using the following information:\n检索出来的知识片段



3.7.3 核心API

为了梳理RAG的核心API，我们再来回顾一下知识库的存储流程。

首先我们需要在项目中准备存储数据的文档，这些文档需要使用文档加载器 Document Loader 加载进内存，由于加载的过程中需要解析文档的内容，所以还要使用到文档解析器来解析文档的内容，最后在内存中生成一个一个的Document对象用于记录文档的内容。由于每个Document对象中记录的是对应文档中的全部内容，如果我们直接把整个文档的内容一次性向量化存储到向量数据库中，不利于检索，所以这些文档对象，需要使用文档分割器 Document Splitter分割成一个一个的文本片段，而每一个文本片段只是记录整个文档中的一小部分内容，这样将来根据用户问题检索相关片段的时候就会更精准。这些文本片段需要使用向量模型转化为一个一个向量，之前讲过其实就是一串一串的数字记录的是不同维度的坐标，LangChain4j中提供了Embedding对象用于记录这些坐标，因此这里得到的是一个一个的Embedding对象。最后再使用EmbeddingStore这种向量数据库操作对象将向量和对应的文本片段存储到向量数据库中。

在整个流程中，主要用到了文档加载器、文档解析器、文档分割器、向量模型以及向量数据库操作对象这五类API，等会儿咱们挨个讲解。其中有关文档分割器、向量模、还有向量数据库操作对象的具体方法的调用都被封装到了EmbeddingStoreIngestor中了，对于咱们来说无需过多关注，我们主要关注的是使用哪种文档分割器、哪种向量模型、哪种向量数据库操作对象即可，将来用哪种把哪种交给EmbeddingStoreIngestor就可以了。

3.7.3.1 文档加载器

文档加载器的作用是把磁盘或者网络中的数据加载进程序。LangChain4j给我们提供了多个文档加载器，其中常见的有以下三种：

- FileSystemDocumentLoader, 根据本地磁盘绝对路径加载
- ClassPathDocumentLoader, 相对于类路径加载
- UrlDocumentLoader, 根据url路径加载
-

大家可以把之前代码中的ClassPathDocumentLoader替换为FileSysteDocumentLoader做一个尝试。

代码块

```
1  @Bean
2  public EmbeddingStore store(){
3      //1.加载文档进内存
4      //List<Document> documents =
        ClassPathDocumentLoader.loadDocuments("content");
5      List<Document> documents =
        FileSystemDocumentLoader.loadDocuments("C:\\Users\\Administrator\\ideaProjects\\
        consultant\\src\\main\\resources\\content");
6      //2.构建向量数据库操作对象 操作的是内存版本的向量数据库
7      InMemoryEmbeddingStore store = new InMemoryEmbeddingStore();
8      //3.构建一个EmbeddingStoreIngestor对象,完成文本数据切割,向量化,存储
9      EmbeddingStoreIngestor ingestor = EmbeddingStoreIngestor.builder()
10          .embeddingStore(store)
11          .build();
12      ingestor.ingest(documents);
13      return store;
14  }
```

3.7.3.2 文档解析器










文档解析器就是用于解析文档中的内容，把原本非纯文本数据转化成纯文本。比如初始的文档是pdf格式的，它的内容就不是纯文本的，此时需要借助于文档解析器将非纯文本数据转化成纯文本。在LangChain4j中提供了几个常用的文档解析器：

- TextDocumentParser，解析纯文本格式的文件
- ApachePdfBoxDocumentParser，解析pdf格式文件
- ApachePoiDocumentParser，解析微软的office文件，例如DOC、PPT、XLS
- ApacheTikaDocumentParser（默认），几乎可以解析所有格式的文件

由于默认的ApacheTikaDocumentParser虽然可以解析所有格式的文件，但是它可能在纯PDF文件方面的表现没有那么优秀，或者使用起来没有那么方便，此时我们可以将默认的解析器切换成ApachePdfBoxDocumentParser，具体的操作如下：

A. 准备pdf格式的数据

将资料中准备的《西北大学.pdf》拷贝到resourcces/content目录下，删除原来的《西北大学.md》。

LangChain4J > 03_资料 > 02_素材 > 03_各大高校本科招生信息 > pdf					▼	🔄	搜索"pdf"
	名称	修改日期	类型	大小			
	 热门专业top20.pdf	2025/5/26 15:18	Microsoft Edge ...	325 KB			
	 陕西师范大学.pdf	2025/5/26 15:19	Microsoft Edge ...	267 KB			
	 天坑专业top10.pdf	2025/5/26 15:20	Microsoft Edge ...	369 KB			
	 西安电子科技大学.pdf	2025/5/26 15:20	Microsoft Edge ...	249 KB			
	 西安交通大学.pdf	2025/5/26 15:20	Microsoft Edge ...	278 KB			
	 西北大学.pdf	2025/6/10 11:45	Microsoft Edge ...	277 KB			
	 西北工业大学.pdf	2025/5/26 15:21	Microsoft Edge ...	264 KB			
	 西北农林科技大学.pdf	2025/5/26 15:21	Microsoft Edge ...	284 KB			
	 长安大学.pdf	2025/5/26 15:21	Microsoft Edge ...	257 KB			

B. 引入依赖

pom.xml

```
1 <dependency>
2   <groupId>dev.langchain4j</groupId>
3   <artifactId>langchain4j-document-parser-apache-pdfbox</artifactId>
4   <version>1.0.1-beta6</version>
5 </dependency>
```

C. 指定解析器

代码块

```
1 @Bean
2 public EmbeddingStore store(){
3     //1.加载文档进内存
4     //List<Document> documents =
5     ClassPathDocumentLoader.loadDocuments("content");
6     //加载文档的时候指定解析器
7     List<Document> documents =
8     ClassPathDocumentLoader.loadDocuments("content",new
9     ApachePdfBoxDocumentParser());
10    //2.构建向量数据库操作对象 操作的是内存版本的向量数据库
11    InMemoryEmbeddingStore store = new InMemoryEmbeddingStore();
12    //3.构建一个EmbeddingStoreIngestor对象,完成文本数据切割,向量化,存储
13    EmbeddingStoreIngestor ingestor = EmbeddingStoreIngestor.builder()
14        .embeddingStore(store)
15        .build();
16    ingestor.ingest(documents);
17    return store;
18 }
```


3.7.3.3 文档分割器

文档分割器主要用于把一个大的文档切割成一个一个小片段。在langchain4j中提供了多种文档分割器，大概有以下7种：

- DocuemntByParagraphSplitter，按照段落分割文本
- DocumentByLineSplitter，按照行分割文本
- DocumentBySentenceSplitter，按照句子分割文本
- DocumentByWordSplitter，按照词分割文本
- DocumentByCharacterSplitter，按照固定数量的字符分割文本
- DocumentByRegexSplitter，按照正则表达式分割文本
- DocumentSplitters.recursive(...)(默认)，递归分割器,优先段落分割，再按照行分割，再按照句子分割，再按照词分割

先说第一种按照段落分割文本，举个例子，假设我们文本中的内容是一片散文，总共由6个段落组成。

那么DocumentByParagraphSplitter就会把文档分割成6个部分，但是这里大家要注意的是这每一部分并不是将来进行向量化的文本片段，文本片段是根据这6部分的内容组合而成的。通常情况下LangChain4j是允许我们指定文本片段的字符容量的，假设我指定单个文本片段的字符容量为300，那么在组合文本片段的时候，第一部分的自然段和第二部分的自然段的字符总和不到300，可以放到同一个文本片段中，但是加上第三部分的自然段，字符总和超过了300，那么第三部分的自然段就不能再放到这个文本片段中了，而是放到下一个新的文本片段中。

当然除了按照段落分割文本，LangChain4j还提供了按行分割、按句子分割、按单词分割、按固定数量的字符分割等不同方式的文档分割器，都可以使用。这里我们关注一下最后一种文本分割器，它是通过一个静态方法recursive创建出来的，叫做递归分割器，它组合了段落分割器、行分割器、句子分割器以及词分割器，它会按照优先级进行分割文档，先按照段落分割，再按照行，再按照句子，最后按照词，有什么用呢？

咱们刚才按段落分割，第三个自然段是不是放不下了？此时如果是递归分割器的话它会继续使用行分割器，把第三个自然段进一步分割，尝试把得到的内容放到当前文本片段中，如果还是不行，再按照句子分割，这就是它的作用。

咱们默认使用的也是这种递归分割器，默认使用的单个文本最大字符个数就是300，当然了，我不想使用这个默认的切割器，我觉得300个字符太少了，我想多设置一点儿，行不行呢？也可以，接下来我们看应该如何操作。

3.7.3.3.1 构建文本分割器对象

代码块

```
1 DocumentSplitter documentSplitter = DocumentSplitters.recursive(  
2     每个片段最大容纳的字符，  
3     两个片段之间重叠字符的个数  
4 );
```

构建的时候需要指定每个片段最大容纳的字符数量和两个片段之间重叠字符的个数，第一个好理解，给大家解释一下第二个是什么意思。

文本片段：最大300个字符

高考，这个被赋予太多意义的词汇，如同一座横亘在青春之路上的大山。我们习惯将高考比作独木桥，仿佛千万人只能挤过那唯一的通道。然而，教育的真谛不在于将所有人塑造成相同的模样，而在于让每个人都能在知识的密林中找到属于自己的小径。高考不是终点，而是起点；不是标准答案的复制，而是独特生命的绽放。

文本片段：最大300个字符

高考不是终点，而是起点；不是标准答案的复制，而是独特生命的绽放。纵观人类文明史，那些真正推动历史前进的人物，往往不是标准答案的复述者，而是敢于质疑、勇于创新的思想者。爱因斯坦在专利局做着小职员的工作时，没有被既定的物理定律束缚，而是让思想乘着光速的翅膀，最终颠覆了牛顿的经典物理学体系。

假如我有一篇以高考为题目的散文需要存储到向量数据库中,将来分割后得到的两个文本片段，第一个片段里写到高考.....而第二个片段中完全没有出现高考相关的字眼，那到时候我去检索高考相关的内容时第二个片段将不会被检索出来，但实质上按照语义它是应该被检索出来的。我们解决的办法就是让两个片段存储的内容有重叠的部分，上一个片段的末尾与下一个片段的开头重复，这样就可以保持语义的连贯性了。比如我把**高考不是重点，而是起点...**这句话存储到第二个片段的开头就能解决这个问题，咱们第二个参数就是用于指定重叠部分字符的数量。

3.7.3.3.2 配置文本分割器对象

真正分割文本的操作被封装到EmbeddingStoreIngstor中了，所以我们需要在构建该对象的时候，通过documentSplitter方法告诉它将来使用哪个文本分割器。

CommonConfig.java

```
1 @Bean  
2 public EmbeddingStore store(){  
3     //1.加载文档进内存
```

```

4      //List<Document> documents =
      ClassPathDocumentLoader.loadDocuments("content");
5      //加载文档的时候指定解析器
6      List<Document> documents =
      ClassPathDocumentLoader.loadDocuments("content",new
      ApachePdfBoxDocumentParser());
7      //2.构建向量数据库操作对象 操作的是内存版本的向量数据库
8      InMemoryEmbeddingStore store = new InMemoryEmbeddingStore();
9      //构建文档分割器对象
10     DocumentSplitter ds = DocumentSplitters.recursive(500,100);
11     //3.构建一个EmbeddingStoreIngestor对象,完成文本数据切割,向量化,存储
12     EmbeddingStoreIngestor ingestor = EmbeddingStoreIngestor.builder()
13         .embeddingStore(store)
14         .documentSplitter(ds)
15         .build();
16     ingestor.ingest(documents);
17     return store;
18 }

```

3.7.3.4 向量模型

向量模型的作用是把分割后的文本片段向量化或者把用户消息向量化。

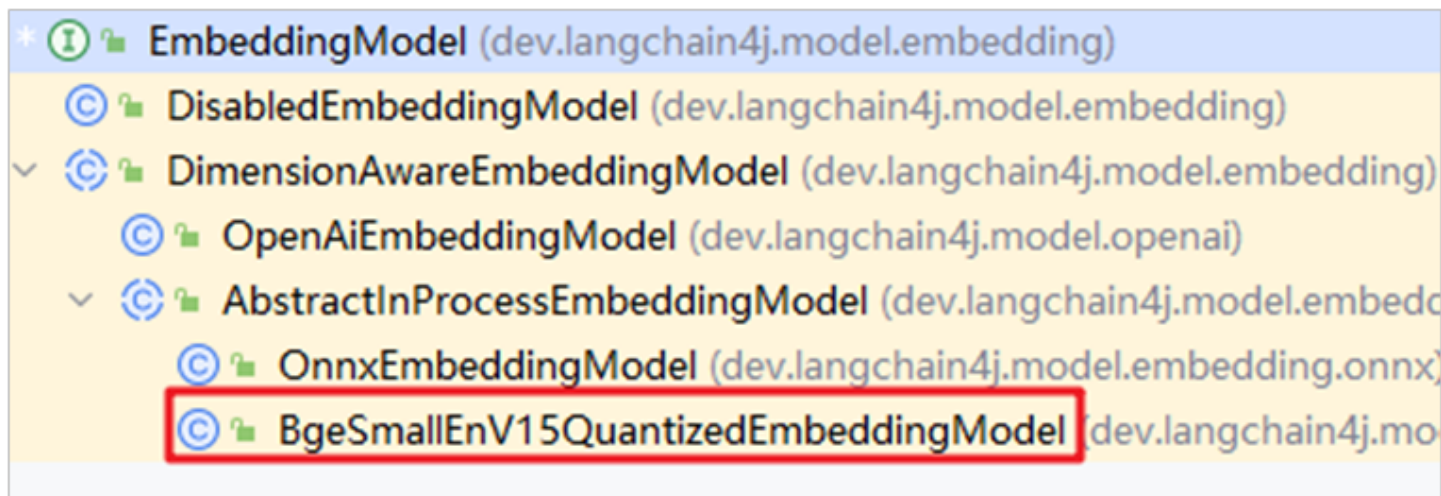
EmbeddingModel.java

```

1  public interface EmbeddingModel {
2      default Response<Embedding> embed(String text) {
3          return this.embed(TextSegment.from(text));
4      }
5
6      default Response<Embedding> embed(TextSegment textSegment) {
7      }
8
9      Response<List<Embedding>> embedAll(List<TextSegment> texts);
10
11     default int dimension() {
12         return ((Embedding)this.embed("test").content()).dimension();
13     }
14 }

```

LangChain4j中提供了EmbeddingModel接口用于定义有关向量模型的方法，例如有embed、embedall等等方法用于把文本片段向量化。LangChain4j提供了一个内存版本的向量模型实现方案，而咱们快速入门中使用的就是这个向量模型，只是咱们当时并没有指定这个向量模型，因为它被封装到EmbeddingStoreIngestor中了，所以我们并没有看到。



但是这种内置的向量模型内有时候功能没有那么强大，说白了就是支持的向量维度太少，检索的时候没有那么精准，所以有些情况下我们需要替换它，使用一些功能更强大的向量模型。阿里云百炼平台也提供了专门用于向量化的向量模型text-embedding-v3，接下来我们看应该如何把我们程序中内存版本的向量模型替换成阿里云百炼提供的向量模型。

3.7.3.4.1 配置向量模型

和咱们之前配置文本模型类似，只不过这里不再是chat-model或者streaming-chat-model，而是embedding-model，其它的配置一样，也需要配置url、apikey、modelname以及日志相关的配置。

代码块

```
1 langchain4j:
2   open-ai:
3     embedding-model:
4       base-url: https://dashscope.aliyuncs.com/compatible-mode/v1
5       api-key: ${API-KEY}
6       model-name: text-embedding-v3
7       log-requests: true
8       log-responses: true
```

3.7.3.4.2 设置向量模型

当我们配置完毕后，LangChain4j会自动的根据我们的配置信息往IOC容器中注入一个EmbeddingModel对象供我们使用，所以接下来我们只需要把这个EmbeddingModel对象交给EmbeddingStoreIngestor和EmbeddingStoreContentRetriever即可，一个是存储的时候使用，一个是检索的时候使用。

CommonConfig.java

```
1 @Autowired
2 private EmbeddingModel embeddingModel;
3
4 @Bean
```

```

5  public EmbeddingStore store(){
6      //1.加载文档进内存
7      //List<Document> documents =
      ClassPathDocumentLoader.loadDocuments("content");
8      //加载文档的时候指定解析器
9      List<Document> documents =
      ClassPathDocumentLoader.loadDocuments("content",new
      ApachePdfBoxDocumentParser());
10     //2.构建向量数据库操作对象 操作的是内存版本的向量数据库
11     InMemoryEmbeddingStore store = new InMemoryEmbeddingStore();
12     //构建文档分割器对象
13     DocumentSplitter ds = DocumentSplitters.recursive(500,100);
14     //3.构建一个EmbeddingStoreIngestor对象,完成文本数据切割,向量化, 存储
15     EmbeddingStoreIngestor ingestor = EmbeddingStoreIngestor.builder()
16         .embeddingStore(store)
17         .documentSplitter(ds)
18         .embeddingModel(embeddingModel)
19         .build();
20     ingestor.ingest(documents);
21     return store;
22 }
23
24 @Bean
25 public ContentRetriever contentRetriever(EmbeddingStore store){
26     return EmbeddingStoreContentRetriever.builder()
27         .embeddingStore(store)
28         .minScore(0.5)
29         .maxResults(3)
30         .embeddingModel(embeddingModel)
31         .build();
32 }

```

测试的时候大家可以查看IDEA控制台的日志，确保替换完成。

3.7.3.5 向量数据库操作对象EmbeddingStore

EmbeddingStore是用来操作向量数据库的API，将来不管是存储还是检索都需要借助于它来完成。LangChain4j提供的EmbeddingStore接口中提供了两组方法，分别是add用于存储数据，search用于检索数据。

EmbeddingStore.java

```

1  public interface EmbeddingStore<Embedded> {
2      String add(Embedding embedding);
3

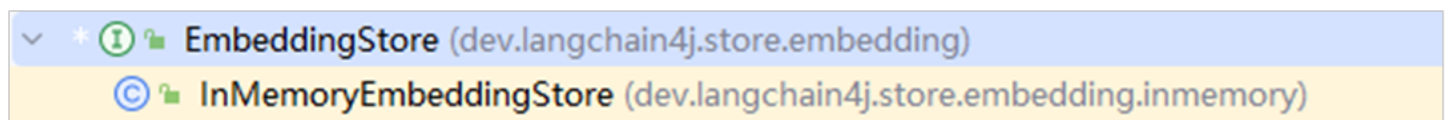
```

```

4      void add(String text, Embedding embedding);
5
6      String add(Embedding embedding, Embedded embedded);
7
8      List<String> addAll(List<Embedding> embeddings);
9
10     EmbeddingSearchResult<Embedded> search(EmbeddingSearchRequest request);
11 }

```

同时LangChain4j还提供了一个实现方案InMemoryEmbeddingStore，也就是咱们之前一直使用的方案，但是它操作的是内存向量数据库，有些情况下不能满足实际开发中的需求。大家可以想一下，如果我们使用内存向量数据库，一旦服务器重启数据就丢失了，又得重新加载文档、重新向量化，这样每次启动都会比较耗时，还有就是每次启动都会使用百炼平台提供的向量模型完成向量化，它是收费的，每次都这么干那是跟钱过不去，没必要对吧。



所以咱们得考虑把向量化后的数据存储到外部的向量数据库中。之前给大家介绍过常见的向量数据库有Milvus、Chroma、Pinecone、Redisearch以及pgvector,用哪一种都行，LangChain4j对这些向量数据库都做了支持。咱们本次课程中采用redisearch存储向量数据。接下来我们看看具体的操作。

3.7.3.5.1 准备向量数据库Redisearch

这一块我们依然使用docker来部署redisearch，由于redisearch是redis扩展的一个功能，所以我们得把之前部署的redis先卸载掉，然后部署一个扩展了redisearch的redis即可。这里我们需要执行三条命令：

安装redis

```

1  docker stop redis # 停止原有的redis镜像
2  docker rm redis #删除原有的redis镜像
3  docker run --name redis-vector -d -p 6379:6379 redislabs/redisearch #安装扩展
    redisearch功能的redis

```

3.7.3.5.2 引入依赖

pom.xml

```

1  <dependency>
2      <groupId>dev.langchain4j</groupId>
3      <artifactId>langchain4j-community-redis-spring-boot-starter</artifactId>
4      <version>1.0.1-beta6</version>
5  </dependency>

```

3.7.3.5.3 配置向量数据库连接信息

大家要注意的是这里的配置和我们之前配置的redis不相干，这里配置的是langchain4j.community下的，而之前配置的是spring.data下的。

```
application.yml

1  langchain4j:
2    community:
3      redis:
4        host: localhost
5        port: 6379
```

当引入的起步依赖检测我们这一段配置信息后，会自动的往IOC容器中注入一个RedisEmbeddingStore对象，这个对象实现了EmbeddingStore接口，封装了操作redisearch的API，我们可以直接使用。

3.7.3.5.4 注入RedisEmbeddingStore对象使用

和之前一样，将IOC容器中的RedisEmbeddingStore对象分别设置给EmbeddingStoreIngestor和EmbeddingStoreContentRetriever，用于存储和检索。

```
CommonConfig.java

1  @Autowired
2  private RedisEmbeddingStore redisEmbeddingStore;
3
4  @Bean
5  public EmbeddingStore store(){//embeddingStore的对象，这个对象的名字不能重复,所以这
    里使用store
6      //1.加载文档进内存
7      //List<Document> documents =
        ClassPathDocumentLoader.loadDocuments("content");
8      List<Document> documents =
        ClassPathDocumentLoader.loadDocuments("content",new
        ApachePdfBoxDocumentParser());
9      //List<Document> documents =
        FileSystemDocumentLoader.loadDocuments("C:\\Users\\Administrator\\ideaProjects\\
        consultant\\src\\main\\resources\\content");
10     //2.构建向量数据库操作对象 操作的是内存版本的向量数据库
11     //InMemoryEmbeddingStore store = new InMemoryEmbeddingStore();
12
13     //构建文档分割器对象
14     DocumentSplitter ds = DocumentSplitters.recursive(500,100);
15     //3.构建一个EmbeddingStoreIngestor对象,完成文本数据切割,向量化,存储
16     EmbeddingStoreIngestor ingestor = EmbeddingStoreIngestor.builder()
```

```










17         //.embeddingStore(store)
18         .embeddingStore(redisEmbeddingStore)
19         .documentSplitter(ds)
20         .embeddingModel(embeddingModel)
21         .build();
22     ingestor.ingest(documents);
23     return redisEmbeddingStore;
24 }
25
26 @Bean
27 public ContentRetriever contentRetriever(/*EmbeddingStore store*/){
28     return EmbeddingStoreContentRetriever.builder()
29         .embeddingStore(redisEmbeddingStore)
30         .minScore(0.5)
31         .maxResults(3)
32         .embeddingModel(embeddingModel)
33         .build();
34 }

```

3.7.4 收尾工作

3.7.4.1 完整知识库

将资料中提供的所有pdf文档，全部拷贝到reouserces/content目录下，重新启动测试，让向量数据库保存所有的数据。

LangChain4J > 03_资料 > 02_素材 > 03_各大高校本科招生信息 > pdf					搜索"pdf"
名称	修改日期	类型	大小		
 热门专业top20.pdf	2025/5/26 15:18	Microsoft Edge ...	325 KB		
 陕西师范大学.pdf	2025/5/26 15:19	Microsoft Edge ...	267 KB		
 天坑专业top10.pdf	2025/5/26 15:20	Microsoft Edge ...	369 KB		
 西安电子科技大学.pdf	2025/5/26 15:20	Microsoft Edge ...	249 KB		
 西安交通大学.pdf	2025/5/26 15:20	Microsoft Edge ...	278 KB		
 西北大学.pdf	2025/6/10 11:45	Microsoft Edge ...	277 KB		
 西北工业大学.pdf	2025/5/26 15:21	Microsoft Edge ...	264 KB		
 西北农林科技大学.pdf	2025/5/26 15:21	Microsoft Edge ...	284 KB		
 长安大学.pdf	2025/5/26 15:21	Microsoft Edge ...	257 KB		

3.7.4.2 避免每次启动程序都做向量化的操作

由于咱们准备向量数据库的操作是在CommonConfig配置类中完成的，在该类中我们提供了一个store方法，方法上添加了一个@Bean注解，所以每次启动程序，该方法都会执行一遍，文档就会重新加

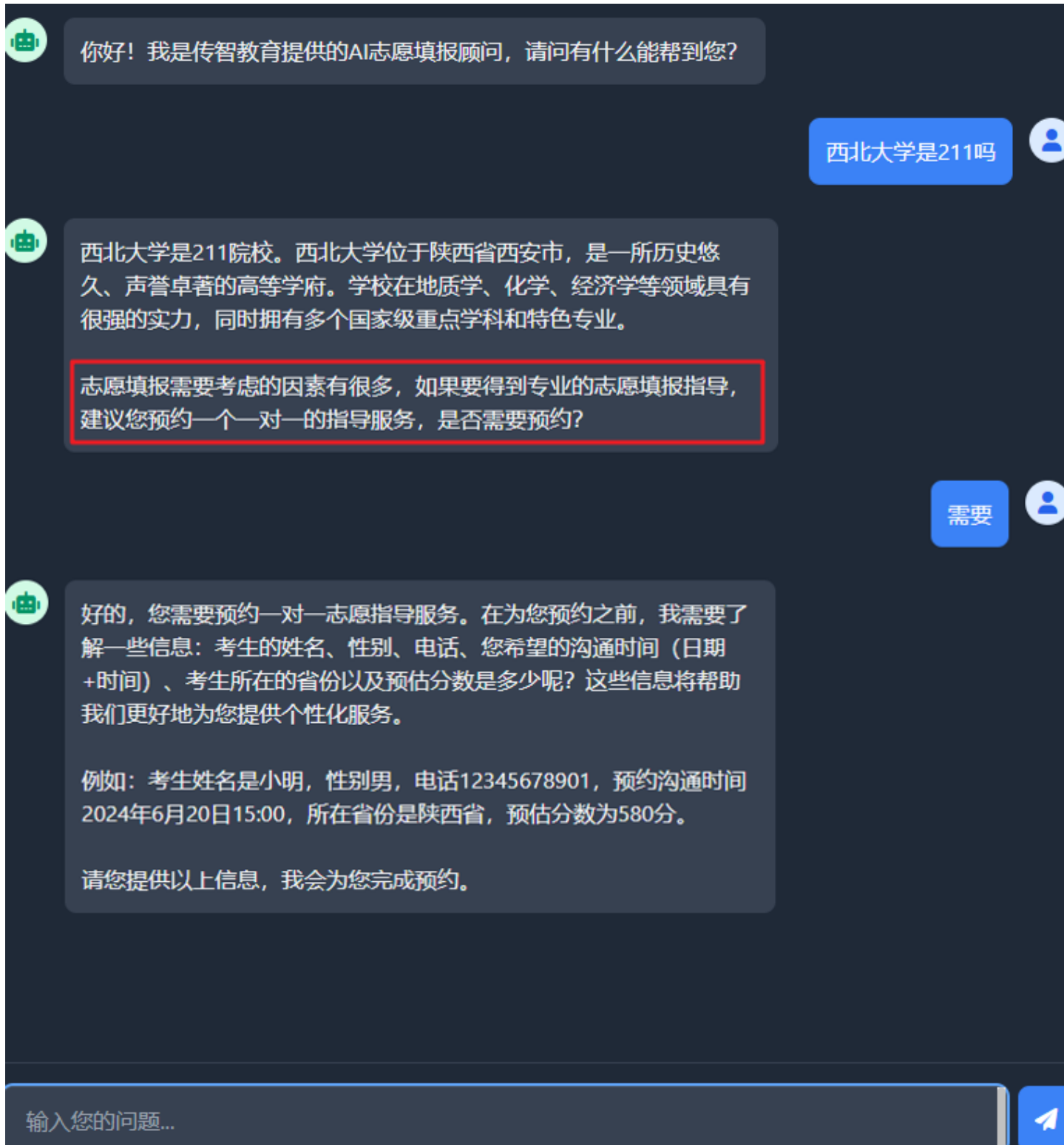
载，重新向量化，不合适。所以当我们把所有文档拷贝到content目录中，启动测试一遍后，redis中就已经存好了所有的数据，接下来把store方法上的@Bean注解注释掉，可以避免每次启动都做向量化的操作。

CommonConfig.java

```
1  // @Bean
2  public EmbeddingStore store(){
3      //.....
4      return redisEmbeddingStore;
5  }
```

3.8 Tools工具

在咱们的AI志愿填报顾问中，将来要做这么一个功能，每次回答完用户的问题后，都会在答案的最后附上这么一句话: 志愿填报需要考虑的因素有很多，如果要得到专业的志愿填报指导，建议您预约一个一对一的指导服务，是否需要预约？



当用户表达出需要预约的意愿并提交了姓名, 性别, 电话等信息后, 我们的程序就需要数据库中添加一条信息, 记录预约详情。

所以开发这个功能的前提是我们得先准备好mysql数据库环境, 把crud的代码开发好, 将来当用户提交了考生信息后才能调用这些代码往数据库中添加数据。

3.8.1 准备工作

3.8.1.1 准备数据库环境

这里依然采用docker部署，执行下面这条命令安装并运行mysql（注意，这里windows系统下映射了3307端口）

代码块

```
1 docker run --name mysql -d -p 3307:3306 -e MYSQL_ROOT_PASSWORD=1234 mysql
```

同时需要把资料中提供的sql脚本执行以下，导入到安装的mysql中，sql如下：

volunteer.sql

```
1 create database if not exists volunteer;
2 use volunteer;
3 create table if not exists reservation
4 (
5     id                bigint primary key auto_increment not null comment
6     '主键ID',
7     name              varchar(50) not null comment '考生姓名',
8     gender            varchar(2)  not null comment '考生性别',
9     phone             varchar(20) not null comment '考生手机号',
10    communication_time datetime  not null comment '沟通时间',
11    province           varchar(32) not null comment '考生所处的省份',
12    estimated_score    int         not null comment '考生预估分数'
13 )
```

3.8.1.2 引入依赖

pom.xml

```
1 <dependency>
2     <groupId>org.projectlombok</groupId>
3     <artifactId>lombok</artifactId>
4 </dependency>
5
6 <dependency>
7     <groupId>org.mybatis.spring.boot</groupId>
8     <artifactId>mybatis-spring-boot-starter</artifactId>
9     <version>3.0.3</version>
10 </dependency>
11
12 <dependency>
13     <groupId>com.mysql</groupId>
14     <artifactId>mysql-connector-j</artifactId>
15 </dependency>
```

3.8.1.3 配置数据库连接信息

application.yml

```
1  spring:
2    datasource:
3      username: root
4      password: 1234
5      url: jdbc:mysql://localhost:3307/volunteer?
        useUnicode=true&characterEncoding=utf-
        8&useSSL=false&serverTimezone=Asia/Shanghai&allowPublicKeyRetrieval=true
6      driver-class-name: com.mysql.cj.jdbc.Driver
7
8  mybatis:
9    configuration:
10     map-underscore-to-camel-case: true
```

3.8.1.4 准备实体类

Reservation.java

```
1  import lombok.AllArgsConstructor;
2  import lombok.Data;
3  import lombok.NoArgsConstructor;
4
5  import java.time.LocalDateTime;
6
7  @Data
8  @NoArgsConstructor
9  @AllArgsConstructor
10 public class Reservation {
11     private Long id;
12     private String name;
13     private String gender;
14     private String phone;
15     private LocalDateTime communicationTime;
16     private String province;
17     private Integer estimatedScore;
18 }
```

3.8.1.5 开发Mapper

ReservationMapper.java

```
1  import com.itheima.consultant.pojo.Reservation;
```

```

2  import org.apache.ibatis.annotations.Insert;
3  import org.apache.ibatis.annotations.Mapper;
4  import org.apache.ibatis.annotations.Select;
5
6  @Mapper
7  public interface ReservationMapper {
8
9      //1.添加预约信息
10     @Insert("insert into
reservation(name,gender,phone,communication_time,province,estimated_score)
values(#{name},#{gender},#{phone},#{communicationTime},#{province},#
{estimatedScore})")
11     void insert(Reservation reservation);
12     //2.根据手机号查询预约信息
13     @Select("select * from reservation where phone=#{phone}")
14     Reservation findByPhone(String phone);
15
16 }

```

3.8.1.6 开发Service

ReservationService.java

```

1  import com.itheima.consultant.mapper.ReservationMapper;
2  import com.itheima.consultant.pojo.Reservation;
3  import org.springframework.beans.factory.annotation.Autowired;
4  import org.springframework.stereotype.Service;
5
6  @Service
7  public class ReservationService {
8      @Autowired
9      private ReservationMapper reservationMapper;
10
11     //1.添加预约信息的方法
12     public void insert(Reservation reservation) {
13         reservationMapper.insert(reservation);
14     }
15
16     //2.查询预约信息的方法(根据手机号查询)
17     public Reservation findByPhone(String phone) {
18         return reservationMapper.findByPhone(phone);
19     }
20 }

```

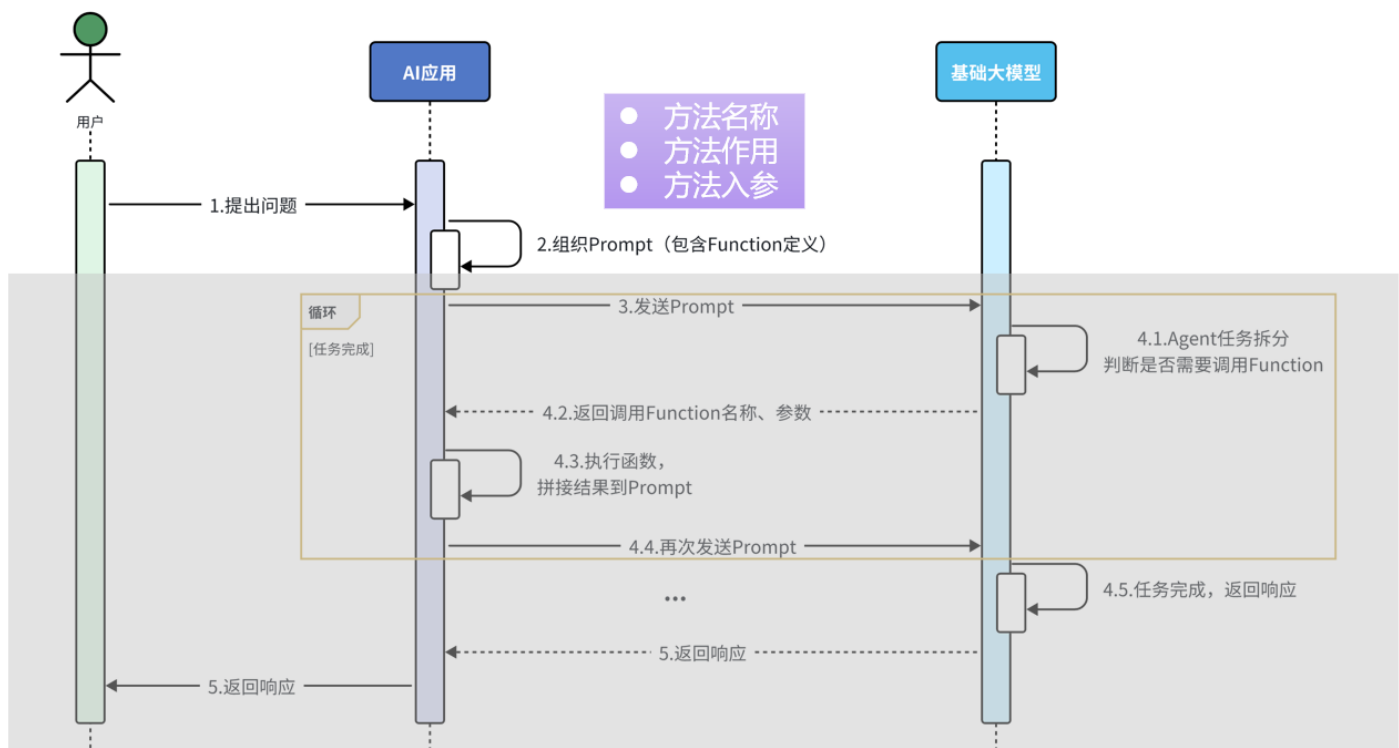
3.8.1.7 测试

ReservationServiceTest.java

```
1  import com.itheima.consultant.pojo.Reservation;
2  import com.itheima.consultant.service.ReservationService;
3  import org.junit.jupiter.api.Test;
4  import org.springframework.beans.factory.annotation.Autowired;
5  import org.springframework.boot.test.context.SpringBootTest;
6
7  import java.time.LocalDateTime;
8
9  @SpringBootTest
10 public class ReservationServiceTest {
11     @Autowired
12     private ReservationService reservationService;
13     //测试添加
14     @Test
15     void testInsert(){
16         Reservation reservation = new Reservation(null, "小王", "男",
17 "138000000001", LocalDateTime.now(), "上海", 580);
18         reservationService.insert(reservation);
19     }
20     //测试查询
21     @Test
22     void testFindByPhone(){
23         String phone = "138000000001";
24         Reservation reservation = reservationService.findByPhone(phone);
25         System.out.println(reservation);
26     }
27 }
```

3.8.2 Tools工具原理

Tools工具，以前也叫做function calling，翻译过来叫做函数调用，如果在我们的程序中添加了function calling功能，那整个工作流程会发生一些改变，我们简单的看一看。



当用户把问题发送给AI应用，在AI应用的内部需要组织提交给大模型的数据，而这些数据中需要描述清楚我们的AI应用中有哪些函数能够被大模型调用。每一个函数的描述都包含三个部分，方法名称、方法作用、方法入参。当AI应用把这些数据发送给大模型后，大模型会先根据用户的问题以及上下文拆解任务，从而判断是否需要调用函数，如果有函数需要调用，则把需要调用的函数的名称，以及调用时需要使用的参数准备好一并响应给AI应用。AI应用接收到响应后需要执行对应的函数，得到对应的结果，接下来把得到的结果和之前信息一块组织好再发送给大模型。

这里需要注意的是由于在一次任务的处理过程中可能需要根据顺序调用多个函数，所以当大模型接收到AI应用发送的数据继续拆解任务，如果发现还需要调用其他的函数，则会重复4.1~4.4这几个步骤，直到无需调用函数，最终把生成的结果响应该AI应用，并由AI应用发送给用户。

这就是增加了function calling 或者 Tools工具后整个AI应用的工作流程，比之前要复杂不少，不过好消息是下面的这些工作LangChain4j都能帮我们自动的完成，对于咱们来说只需要按照LangChain4j的规则描述清楚有哪些方法可以被大模型调用，方法名的名字是什么、有什么作用、以及都需要哪些参数？

3.8.2.1 准备工具方法

LangChain4j提供了Tool注解用于对方法的作用进行描述，还有P注解用于对方法的参数进行描述，将来LangChain4j就能通过反射的方式获取到Tool注解中的作用描述、P注解中的参数描述、以及方法的名称，组织数据,一并发送给大模型。这里需要注意，ReservationTool需要注入到IOC容器对象中。

ReservationTool.java

```
1 import com.itheima.consultant.pojo.Reservation;  
2 import com.itheima.consultant.service.ReservationService;  
3 import dev.langchain4j.agent.tool.P;  
4 import dev.langchain4j.agent.tool.Tool;
```

```

5  import org.springframework.beans.factory.annotation.Autowired;
6  import org.springframework.stereotype.Component;
7
8  import java.time.LocalDateTime;
9
10 @Component
11 public class ReservationTool {
12     @Autowired
13     private ReservationService reservationService;
14
15     //1.工具方法：添加预约信息
16     @Tool("预约志愿填报服务")
17     public void addReservation(
18         @P("考生姓名") String name,
19         @P("考生性别") String gender,
20         @P("考生手机号") String phone,
21         @P("预约沟通时间,格式为: yyyy-MM-dd'T'HH:mm") String
communicationTime,
22         @P("考生所在省份") String province,
23         @P("考生预估分数") Integer estimatedScore
24     ){
25         Reservation reservation = new Reservation(null,name,gender,phone,
LocalDateTime.parse(communicationTime),province,estimatedScore);
26         reservationService.insert(reservation);
27     }
28     //2.工具方法：查询预约信息
29     @Tool("根据考生手机号查询预约单")
30     public Reservation findReservation(@P("考生手机号") String phone){
31         return reservationService.findByPhone(phone);
32     }
33 }

```

3.8.2.2 配置工具方法

配置的方法和之前的类似，在AiService注解中过一个叫做tools的属性完成配置，值写上包含了工具方法的Bean对象的名字即可。

ConsultantService.java

```

1  @AiService(
2      wiringMode = AiServiceWiringMode.EXPLICIT, //手动装配
3      chatModel = "openAiChatModel", //指定模型
4      streamingChatModel = "openAiStreamingChatModel",
5      //chatMemory = "chatMemory", //配置会话记忆对象
6      chatMemoryProvider = "chatMemoryProvider", //配置会话记忆提供者对象
7      contentRetriever = "contentRetriever", //配置向量数据库检索对象
8      tools = "reservationTool"

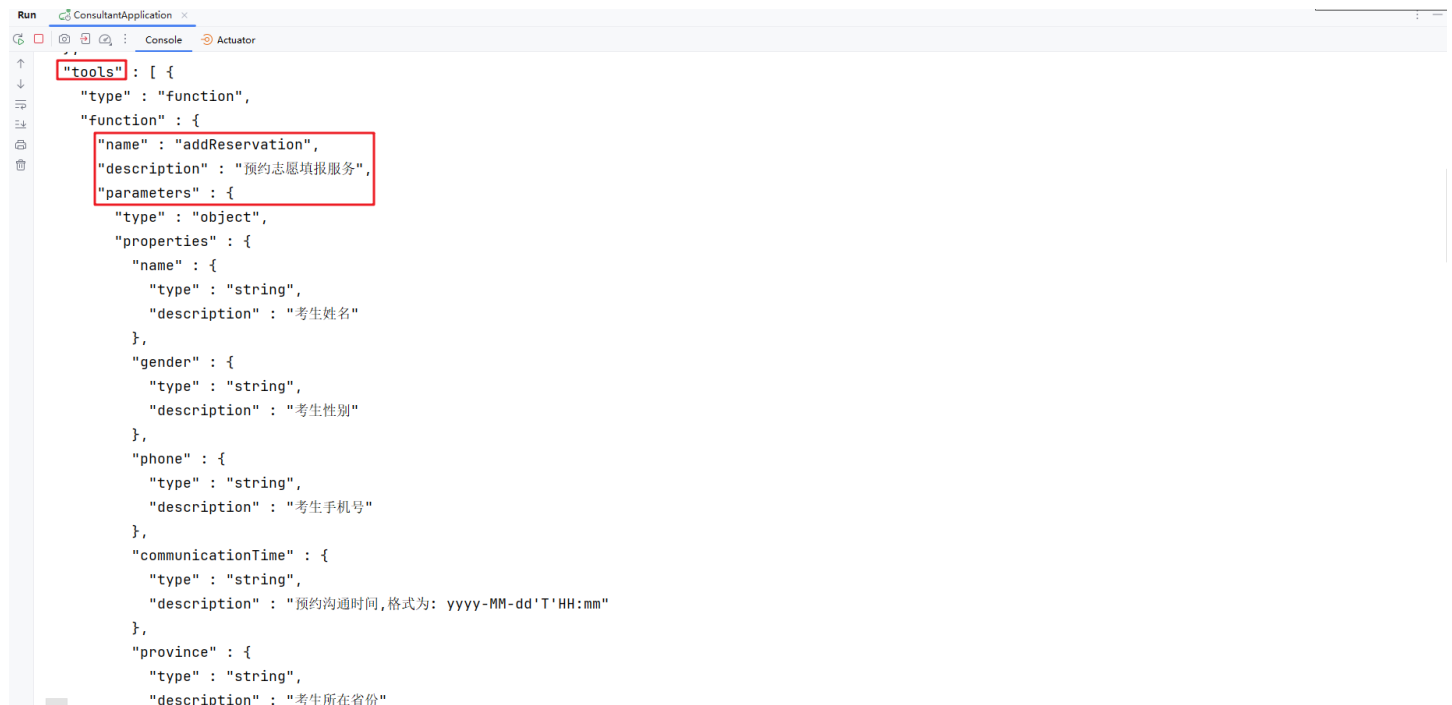
```

```

9 )
10 // @AiService
11 public interface ConsultantService {
12     // 用于聊天的方法
13     // public String chat(String message);
14     // @SystemMessage("你是东哥的助手小月月,人美心善又多金!")
15     @SystemMessage(fromResource = "system.txt")
16     // @UserMessage("你是东哥的助手小月月,人美心善又多金!{{it}}")
17     // @UserMessage("你是东哥的助手小月月,人美心善又多金!{{msg}}")
18     public Flux<String> chat(*@V("msg")*/@MemoryId String memoryId,
19     @UserMessage String message);
19 }

```

功能已经实现完毕了，测试的时候注意观察IDEA控制台的信息，langchain4j给大模型发送消息的时候会使用tools参数告诉大模型，有哪些函数可以调用。



```

Run ConsultantApplication x
Console
{"tools": [ {
  "type": "function",
  "function": {
    "name": "addReservation",
    "description": "预约志愿填报服务",
    "parameters": {
      "type": "object",
      "properties": {
        "name": {
          "type": "string",
          "description": "考生姓名"
        },
        "gender": {
          "type": "string",
          "description": "考生性别"
        },
        "phone": {
          "type": "string",
          "description": "考生手机号"
        },
        "communicationTime": {
          "type": "string",
          "description": "预约沟通时间,格式为: yyyy-MM-dd'T'HH:mm"
        },
        "province": {
          "type": "string",
          "description": "考生所在省份"
        }
      }
    }
  }
}

```