涉密论文 □　　公开论文 □

浙江大学

高级程序设计课设

题目 ZJU - TIMEBOX

目录

1 项目概述

ZJU_TIMEBOX 是一个创新的平台，旨在动态追踪和展示浙江大学（ZJU）相关的重要活动信息，包括出国交流项目、社会实践活动、讲座、选拔事项等。通过用户友好的日程表界面，ZJU_TIMEBOX 不仅提高了信息获取的效率，还通过智能问答系统为用户提供个性化的帮助和支持，确保每位用户都能迅速找到所需的信息。

ZJU_TIMEBOX 项目的核心目标是通过一个智能化的 Web 日程平台，帮助浙江大学的师生实现以下目标：

- **快速获取校园活动信息**：通过直观的日历界面展示各类活动信息，用户无需在多个平台之间切换，就能轻松浏览校园活动。

- **智能化信息检索**：通过集成智能问答系统，用户可以通过自然语言提问，系统会根据语义理解提供相关活动信息，突破传统关键词检索的局限。

- **实时信息更新**：通过自动化爬虫系统，实时抓取并更新浙江大学官方网站的活动信息，确保用户获取的是最新的活动内容。

可在 https://github.com/Bubblevan/ZJU_TimeBox 仓库查看源代码。

可在 https://pan.baidu.com/s/1uGOT9L2yOfQNBXvkxUA4DQ?pwd=xswl 查看展示视频。

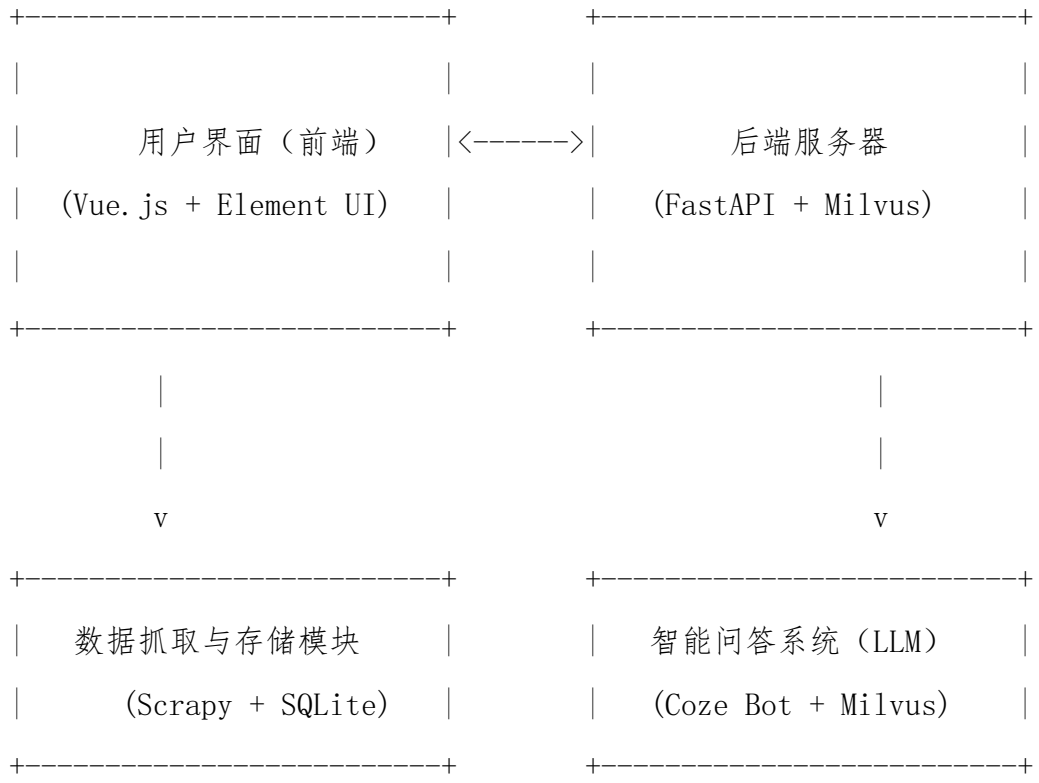
2 项目背景

在现代高校中，信息获取的效率对于学生和教职工的日常生活和工作至关重要。浙江大学作为一所国内顶尖高校，其各类活动信息繁多且更新频繁，包括学术讲座、出国交流项目、社会实践活动、各类选拔与竞赛等。然而，这些信息常常分散在多个不同的平台上，且更新不及时，给需要获取相关信息的师生带来了不便。

为了提高信息获取的效率，解决信息获取不便的问题，我们提出了 ZJU_TIMEBOX 项目。这一平台旨在为浙江大学师生提供一个统一、便捷的途径，通过动态追踪并展示各类重要活动信息，确保每位用户都能及时获得所需的活动信息。

3 项目设计与架构

3.1 系统架构



3.2 技术栈与选型理由

• Vue.js：

Vue.js 是一个极其轻量且高度灵活的 JavaScript 框架，专门设计用于构建现代化的前端界面。它的核心理念是通过简洁的语法和强大的功能，帮助开发者快速构建出交互性强、动态更新的用户界面。Vue.js 的双向数据绑定机制是其一大亮点，这意味着当用户在前端界面进行操作时，数据会自动同步更新，反之亦然。这种机制大大简化了前端开发的工作流程，使得开发者能够更加专注于业务逻辑的实现，而不必过多关注数据同步的问题。

此外，Vue.js 还引入了组件化开发的概念，使得前端代码可以被模块化地组织和管理。每个组件都是一个独立的单元，包含了自身的模板、逻辑和样式，开发者可以轻松地将这些组件组合起来，构建出复杂的用户界面。这种模块化的开发方式不仅提高了代码的可维护性，还使得团队协作变得更加高效。无论是初学者还是经验丰富的开发者，都能在 Vue.js 的帮助下，快速上手并构建出高质量的前端应用。

• FastAPI：

Python 作为一门广泛使用的编程语言，拥有多个主流的后端框架，比如全功能的 Django，轻量级的 Flask，不过这里我们选择 FastAPI 来进行后端开发。

FastAPI 是一种基于 Python 的现代化 Web 框架，专门用于高效地开发高性能的 API。它的设计理念是提供一种简单、直观的方式来构建 RESTful API，同时保证极高的性能和可靠性。FastAPI 的核心优势之一是其自动生成文档的功能，开发者无需手动编写文档，框架会根据代码自动生成详细的 API 文档，这大大减少了开发和维护文档的工作量。

FastAPI 的另一个重要特性是其强大的异步处理能力。在现代 Web 应用中，高并发请求是一个常见的挑战，而 FastAPI 通过支持异步编程，能够高效地处

理大量的并发请求，确保系统在高负载下依然能够快速响应。这种高性能的特性使得 FastAPI 非常适合用于构建需要处理大量请求的 Web 服务，例如实时数据处理、在线游戏服务器等。

此外，FastAPI 还提供了丰富的工具和插件，帮助开发者轻松地集成各种第三方服务和数据库，进一步提升了开发效率。无论是构建简单的 API 还是复杂的分布式系统，FastAPI 都能为开发者提供强大的支持。

• Scrapy：

Scrapy 是一个功能极其强大的 Web 爬虫框架，专门设计用于高效地抓取和处理动态更新的数据。它的核心理念是通过简洁的代码和强大的功能，帮助开发者快速构建出高性能的爬虫系统。Scrapy 支持分布式抓取和高并发请求，这意味着它能够同时处理多个网站的数据抓取任务，并且能够在短时间内完成大量的数据抓取工作。

Scrapy 的另一个重要特性是其灵活的扩展性。开发者可以通过编写自定义的中间件和插件，轻松地扩展 Scrapy 的功能，以适应各种复杂的抓取需求。无论是抓取静态网页还是动态生成的内容,Scrapy 都能提供强大的支持。此外，Scrapy 还内置了强大的数据处理和存储功能,开发者可以轻松地将抓取到的数据存储到数据库或导出为各种格式的文件。

Scrapy 的高性能和灵活性使得它成为数据抓取领域的首选工具，无论是用于商业数据分析、市场调研还是学术研究，Scrapy 都能为开发者提供强大的支持。

• SQLite：

SQLite 是一款轻量级的关系型数据库，专门设计用于存储和管理相对较小的活动数据。它的核心理念是通过简洁的配置和高效的性能，帮助开发者快速构建出轻量级的数据存储解决方案。SQLite 的最大特点是其无需复杂的数据库

配置，开发者可以直接在应用中嵌入 SQLite 数据库，无需安装和配置独立的数据库服务器。

SQLite 的另一个重要特性是其高效的读写性能。尽管它是一个轻量级的数据库，但在处理小规模数据时，其性能表现非常出色。无论是存储用户配置、日志数据还是简单的应用数据，SQLite 都能提供稳定且高效的数据存储服务。此外,SQLite 还支持事务处理和数据完整性约束,确保数据的一致性和可靠性。

SQLite 的轻量级特性和高效性能使得它非常适合用于小规模项目，例如移动应用、桌面应用和嵌入式系统。无论是初学者还是经验丰富的开发者，都能在 SQLite 的帮助下，快速构建出高效的数据存储解决方案。

- **Milvus:**

Milvus 是一个高效的向量数据库，专门设计用于处理大规模的向量数据。它的核心理念是通过高性能的向量检索算法，帮助开发者快速构建出高效的相似性搜索系统。Milvus 的高性能 ANN（Approximate Nearest Neighbor）检索算法是其一大亮点，这意味着它能够在极短的时间内，从海量的向量数据中找到与目标向量最相似的结果。

Milvus 的另一个重要特性是其强大的扩展性。开发者可以通过配置不同的索引类型和参数，轻松地优化系统的检索性能，以适应各种复杂的搜索需求。无论是处理图像、文本还是音频数据，Milvus 都能提供强大的支持。此外，Milvus 还支持分布式部署和高并发请求,确保系统在处理大规模数据时依然能够保持高性能。

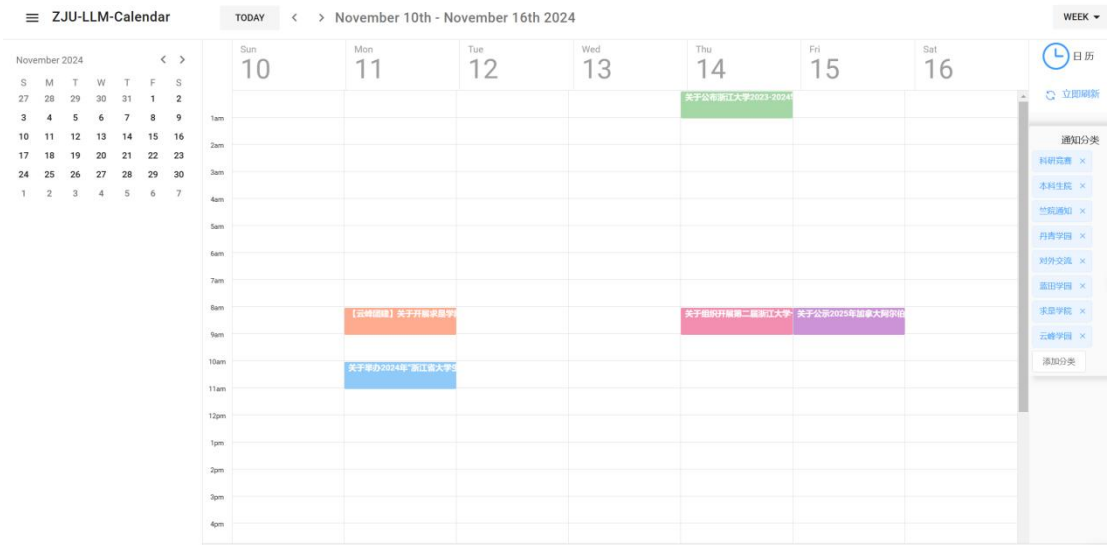
Milvus 的高性能和灵活性使得它非常适合用于智能问答系统、推荐系统、图像检索等需要高效相似性搜索的应用场景。无论是构建复杂的机器学习模型还是实现实时的数据检索，Milvus 都能为开发者提供强大的支持。

3.3 功能模块设计

ZJU_TIMEBOX 系统的设计包括多个核心功能模块,每个模块都实现了独立的功能，同时通过 RESTful API 进行集成，确保了系统的高效性和扩展性。

3.3.1 前端用户界面（UI）

前端采用 Vue.js 框架，结合 Element UI 设计了用户友好的日历界面，用户可以通过简单的点击、浏览操作查看活动信息。



- **日历界面**：展示活动信息，使用不同颜色的色块来区分不同类型的活动（如讲座、交流项目、社会实践等）。
- **活动详情查看**：用户点击色块后，可以查看活动的详细信息，包括标题、时间、地点、活动内容等。
- **智能问答界面**：提供一个文本输入框，用户可以向系统提出问题（如“明天有何讲座？”），系统通过智能问答模块返回相关的活动信息。

3.3.2 后端服务器（FastAPI）

端点采用 FastAPI 框架，提供的 RESTful API，高效处理前端请求和数据交互。

• **活动信息 API**：提供活动数据的查询、添加、删除、修改接口，支持接口与数据库的交互。

• **智能问答 API**：接收用户问题，调用智能问答系统进行处理，并返回生成的答案。

• **实时更新 API**：提供定时任务接口，定期触发数据抓取任务，确保系统活动信息的实时更新。

3.3.3 数据抓取与存储模块（Scrapy + SQLite）

Scrapy 是一个强大的 Python 爬虫框架，负责从浙江大学的官方网站（如学工部、外事处、招生处等）抓取活动信息。系统通过 Scrapy 定期抓取最新活动数据，确保信息更新及时。

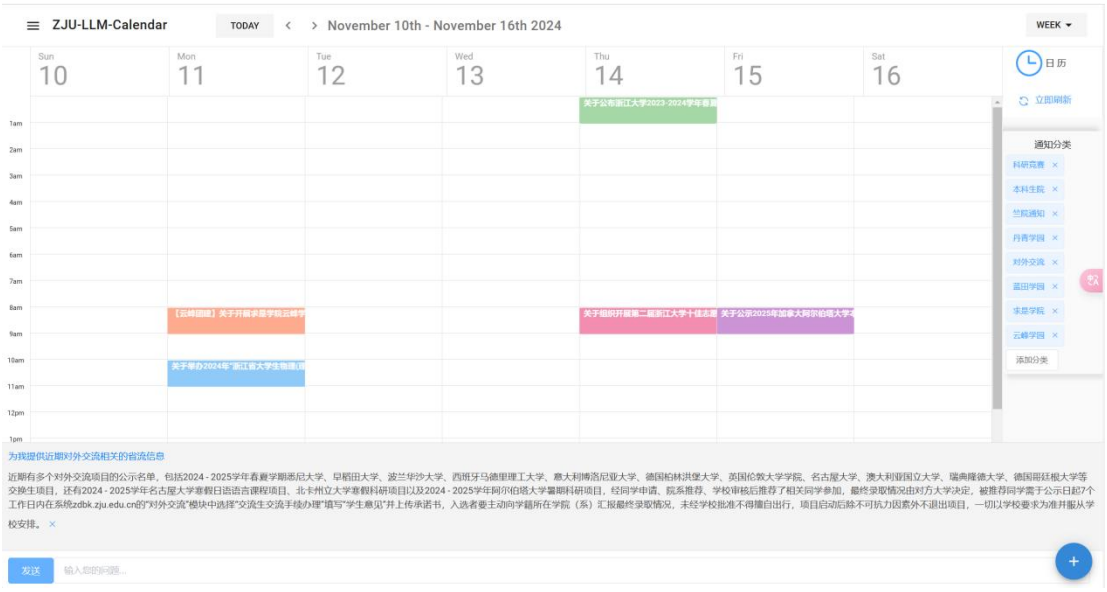
抓取到的活动信息将被存储到 SQLite 数据库中，数据包括活动的标题、日期、描述、链接等。

```
清华 19567096699 浙江大学 团委 相关通知： https://zjutw.zju.edu.cn/2024/1024/c32438a2979425/
page.psp （ 官网 通知 ） https://mp.weixin.qq.com/s/BFy78vkygpziAy49zFMD0g （ 公众号 推文 ）
 往 年优秀作息表 可 参考： https://mp.weixin.qq.com/s/4gZ2pNBq1rgIGRFlR2lTgg https://mp.weix
in.qq.com/s/nAbcx-kNk8O05_tu_GwoHA 共青团浙江大学求是学院紫云碧峰学园分委员会 202 4 年 10 月
 31 日
sqlite> .exit
PS D:\MyLab\ZJU_TimeBox\llm_backend> sqlite3 notices.db
SQLite version 3.45.3 2024-04-15 13:34:05 (UTF-16 console I/O)
Enter ".help" for usage hints.
sqlite> .table
bksy     danqing  kyjs     notices  yunfeng
ckc      jiaoliu  lantian  qsxy
sqlite> .schema kyjs
CREATE TABLE kyjs (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            title TEXT,
            date TEXT,
            link TEXT,
            content TEXT
        );
```

3.3.4 智能问答系统（Coze Bot + Milvus）

在 ZJU_TIMEBOX 项目的智能问答部分，我们使用了一个基于 **检索增强生成（RAG，Retrieval-Augmented Generation）** 的问答系统。尽管方法上与常见的 RAG 实现方式有所不同，但我们仍然借鉴了其核心思想，通过结合向量数据库的检索能力与大语言模型的生成能力，为用户提供精准且高效的答案。

字节跳动推出的 AI 平台扣子（与阿里百炼、百度千帆、质谱 AI 等定位相同），提供强大的大语言模型服务。系统通过自建的 Coze Bot 生成智能答案，根据用户提问的内容提供与活动信息相关的答案。



Milvus 作为支持数据库，用于存储活动信息的嵌入支持。活动内容通过 Jina-embeddings-v3 模型生成嵌入支持后，存储到 Milvus 数据库中。当用户提问时，系统将问题转换为可用，利用 Milvus 进行相似性搜索，找到最相关的活动信息片段。

```
Processing question: 为我提供近期对外交流相关的省流信息
Loaded collection into memory.
Generated embeddings for 1 texts.
Normalized embeddings.
Question embedding (first 5 dimensions): [0.05773918330669403, -0.18111781775951385, 0.05461
801588535309, -0.0194048210978508, 0.0019836463034152985, 0.006919271312654018]
Number of search results: 5
Hit ID: 454192918794884270, Distance: 0.6861059665679932
Hit ID: 454192918794884272, Distance: 0.6917312145233154
Hit ID: 454192918794884267, Distance: 0.6951735019683838
Hit ID: 454192918794884265, Distance: 0.705828070640564
Hit ID: 454192918794884271, Distance: 0.7277748584747314
Loaded collection into memory for querying.
Released collection from memory.
Generated prompt for LLM.
```

3.3.5 定时任务与数据更新

为了确保活动信息的实时更新，系统通过定时任务自动启动 Scrapy 爬虫，定期从浙江大学官网抓取最新的活动数据。

3.4 数据库设计

1、活动信息表（Activity_Info）

```
CREATE TABLE kyjs (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            title TEXT,
            date TEXT,
            link TEXT,
            content TEXT
        );
```

2、活动支持表（Activity_Vectors）

```
CREATE TABLE kyjs (
            title TEXT,
            date TEXT,
```

```
    link TEXT,
    content TEXT(embedding_vector)
);
```

3.5 系统安全与错误处理

·输入验证与防输入：所有用户输入均经过验证，以防止恶意代码输入。

·异常处理机制：系统各个模块都实现了完善的异常捕获与处理机制，确保在出现错误时能够友好地提示用户，并记录日志进行后续调试。

·数据备份与恢复：定期备份数据库中的重要数据，避免因系统故障导致数据丢失。

4 功能实现与技术细节

4.1 数据抓取模块

数据抓取模块主要负责从浙江大学相关通知网站抓取信息。为了高效地抓取数据，我们选用了 Scrapy 框架，可以通过定义爬虫类来提取结构化数据。它提供了强大的数据抓取、处理和存储功能，使得抓取大量网页变得高效且易于管理。Scrapy 有许多内置组件，比如 Spider（爬虫），Pipeline（管道），Item（数据模型），用于定制抓取流程。

比如通过 scrapy startproject zju_notices 就可以创建一个名为 zju_notices 的项目文件夹：

```
zju_notices/
    scrapy.cfg            # 部署配置文件
    zju_notices/          # 项目 Python 模块
        __init__.py
        items.py          # 定义数据模型
        middlewares.py    # 定义中间件
        pipelines.py      # 定义数据处理管道
```

```
        settings.py        # 项目设置
        spiders/           # 存放爬虫文件
            __init__.py
```

在 items.py 文件中，你可以定义数据模型，用于存储从网页中提取的数据。例如，假设我们要抓取浙江大学通知网站的通知信息，可以定义一个 NoticeItem：

```
import scrapy


class NoticeItem(scrapy.Item):
    title = scrapy.Field()    # 通知标题
    date = scrapy.Field()     # 发布日期
    link = scrapy.Field()     # 通知链接
    content = scrapy.Field()  # 通知内容
```

在 pipelines.py 文件中，可以定义数据处理管道，用于处理抓取到的数据。在 settings.py 文件中，则可以配置 Scrapy 项目的各种设置，例如启用管道、设置下载延迟等。下面以求是学院的爬虫为例来介绍核心组件逻辑：

爬虫的目标是从“求是学院”通知页面抓取最近三个月的通知信息，并将数据以 JSON 格式保存到本地。每个通知包括标题、发布时间、内容和链接。

```
class QsxySpider(scrapy.Spider):
    name = 'qsxy'  # 爬虫名称
    allowed_domains = ['qsxy.zju.edu.cn']  # 限制爬虫只爬取该域名下的网页
    start_urls = ['http://qsxy.zju.edu.cn/30845/list1.htm']  # 起始URL

    counter = 1  # 用于给保存的文件命名
```

```
    def __init__(self):
        self.stop_crawling = False  # 用于控制何时停止抓取
        self.base_url = 'http://qsxy.zju.edu.cn'  # 基本URL
        self.today = datetime.now()  # 当前日期
        self.three_months_ago = self.today - timedelta(days=90)  # 三个月前的日期
        self.result_dir = os.path.abspath(os.path.join(os.path.dirname(__file__), '..', '..', 'result', 'qsxy'))  # 结果保存目录
        os.makedirs(self.result_dir, exist_ok=True)  # 创建保存目录
```

4.1.1 解析页面数据

我们使用 CSS 选择器从页面中提取所有通知的列表项。提取通知的日期 date_text，并转换为 datetime 对象进行比较。如果通知日期早于三个月前，则停止抓取。对每个通知发起新的请求（scrapy.Request），获取详情页面数据，如果还没完，尝试提取并访问分页的“下一页”链接，继续抓取。

```
def  parse ( self, response ):
     if  self .stop_crawling:
         return   # 如果标志为True，则停止抓取
    # 查找并处理每个通知
    notices = response.css( '.news_list li' )   # 使用CSS选择器提取通知
    for notification in notification:
        date_text = notification.css( '.news_meta::text' ).get()   # 获取通知的日期
        if date_text:
            date = datetime.strptime(date_text.strip(), '%Y-%m-%d' )
             if date < self .third_months_ago:
```

```
                self.stop_crawling = True    # 如果日期小于三个月前，停止抓取
                return
            title = notification.css('.news_title a::attr(title)').get()    # 获取通知标题
            link = notice.css('.news_title a::attr(href)').get()    # 获取通知链接
            if link:
                full_link = self.base_url + link    # 拼接完整链接
                yield scrapy.Request(full_link, callback=self.parse_activity_page, meta={'title': title, 'date': date_text, 'link': full_link})

    # 查找下一页链接并继续爬取
    next_page = response.css('li.page_nav a.next::attr(href)').get()
     if next_page and  not  self.stop_crawling:
        next_page_url = self.base_url + next_page
        scrapy.Request(next_page_url, callback=self.parse)
```

4.1.2 提取和清理正文内容

使用 BeautifulSoup 解析 extract_content 中的 HTML 内容，找到包含正文的 div 元素。随后在 clean_text 函数中清理正文中的 <script> 和 <style> 标签，并返回纯文本内容。

```
def extract_content(self, html_content):
    soup = BeautifulSoup(html_content, 'html.parser')
    content_div = soup.find('div', class_='wp_articlecontent')  # 提取正文所在的 div
    if content_div:
        return self.clean_text(content_div)
```

```
    return ''
def clean_text(self, element):
    for script in element(["script", "style"]):
        script.decompose()  # 删除多余的标签
    return ' '.join(element.stripped_strings)  # 返回清理后的文本
```

4.2 日历界面实现

在本项目中，日历界面采用了 Vue.js 作为前端框架，并结合了第三方库 dayspan 来实现日历的功能。我们还使用了 Element UI 来构建用户界面，以提供丰富的交互组件。

main.js 是 Vue 项目的入口文件，负责初始化 Vue 实例并将它挂载到 DOM 上。DOM（Document Object Model，文档对象模型）是 HTML 和 XML 文档的编程接口。它将文档解析为一个由节点和对象（包含属性和方法）组成的树状结构，使得开发者可以通过编程语言（如 JavaScript）来操作文档的内容、结构和样式。如 、<p>、<a> 等就是元素节点，<a href="https://example.com"> 中的 href 属性就是属性节点。在此文件中，我们导入了 Vue、Element UI 以及其他需要的插件或库，然后创建并挂载 Vue 实例。

随后是另一个关键的主应用组件 App.vue，在 App.vue 中，我们定义了日历界面的核心部分，包括日历的展示、事件的展示以及侧边栏的标签管理功能。

```
<template>
  <v-app id="dayspan" v-cloak>
    <div class="calendar-container">
      <ds-calendar-app
        ref="app"
        :calendar="calendar"
        :read-only="readOnly"
```

```
        @change="saveState"
        :types="[{ type: 'week', size: 1, label: 'Week' }, { type:
'month', size: 1, label: 'Month' }]"
      >
        <template slot="title"> ZJU-LLM-Calendar </template>
        <!-- 事件弹窗模板 -->
        <template slot="eventPopover" slot-scope="slotData">
          <ds-calendar-event-popover
            v-bind="slotData"
            :read-only="readOnly"
            @finish="saveState"
          ></ds-calendar-event-popover>
        </template>
        <!-- 事件创建弹窗模板 -->
        <template slot="eventCreatePopover" slot-scope="{ placeholder,
calendar }">
          <ds-calendar-event-create-popover
            :calendar-event="placeholder"
            :calendar="calendar"
            :close="$refs.app.$refs.calendar.clearPlaceholder"
            @create-edit="$refs.app.editPlaceholder"
            @create-popover-closed="saveState"
          ></ds-calendar-event-create-popover>
        </template>
        <!-- 事件时间标题模板 -->
        <template slot="eventTimeTitle" slot-scope="{ details }">
          <div>
            <v-icon
              class="ds-ev-icon"
```

```
              v-if="details.icon"
              size="14"
              :style="{ color: details.forecolor }"
            >
              {{ details.icon }}
            </v-icon>
            <strong class="ds-ev-title" style="word-wrap:
break-word">{{ details.title }}</strong>
          </div>
        </template>
      </ds-calendar-app>

      <!-- 侧边栏和其他功能 -->
      <div class="empty-space">
        <el-menu default-active="2">
          <el-menu-item index="1" @click="fetchAllTablesData">
            <img src="../src/assets/logo.png" alt="ZJU" style="width:
40px; height: 40px"/>
            <span slot="title" style="margin-left: 5px">日历</span>
          </el-menu-item>
          <el-menu-item index="2">
            <i class="el-icon-refresh"></i>
            <span slot="title" @click="refreshData">立即刷新</span>
          </el-menu-item>
        </el-menu>
      </div>
    </div>
  </v-app></template>
```

ds-calendar-app 是日历界面的核心组件，负责日历的展示及事件管理。通过 ref="app" 引用日历实例，可以在后续操作中访问和修改日历的状态。

calendar 是日历的核心数据对象，readOnly 用于设置日历是否为只读模式。

@change="saveState" 事件处理器用于保存用户更改的状态（如活动的增删改）。

4.2.1 日历数据的处理

```
data: () => ({
  calendar: Calendar.weeks(),
  readOnly: true,
  dynamicTags: [
    { name: "科研竞赛", tableName: "kyjs" },
    { name: "本科生院", tableName: "bksy" },
    { name: "竺院通知", tableName: "ckc" },
    // 更多标签...
  ],
  inputVisible: false,
  inputValue: "",
}),
```

在 data 部分，定义了日历需要的数据，包括 calendar、dynamicTags（动态标签）、inputVisible 等。

首先，Calendar.weeks() 初始化日历为“周视图”。可以根据需求修改为“月视图”或其他视图类型。

dynamicTags 定义了不同类型的通知标签，如“科研竞赛”、“本科生院”等。每个标签对应一个 tableName，可以用来过滤和展示不同类型的活动信息。

4.2.2 数据的加载和展示

```
mounted() {
  this.fetchNotices(); // 默认加载所有通知
},
methods: {
  async fetchNotices() {
    try {
      const allNotices = [];
      for (const tag of this.dynamicTags) {
        const response = await
axios.get(`http://127.0.0.1:8000/notices/${tag.tableName}`);
        allNotices.push(...response.data);
      }

      this.defaultEvents = allNotices.map(notice => {
        const dateObj = new Date(notice.date);
        return {
          data: {
            title: notice.title,
            color: '#FF5722', // 自定义颜色
            description: notice.content,
            link: notice.link,
          },
          schedule: {
            month: [Month[dateObj.toLocaleString('default', { month:
'long' })]],
            dayOfMonth: [dateObj.getDate()],
            times: [dateObj.getHours()],
            duration: 60,
```

```
            durationUnit: "minutes",
          },
        };
      });

      this.loadState();
    } catch (error) {
      console.error("Error fetching notices:", error);
    }
  },
},
```

在 JavaScript 里面，生命周期钩子是组件在不同阶段自动调用的函数，用于执行特定的操作。而 mounted 钩子主要在在实例挂载到 DOM 后被调用。此时，DOM 元素已经可用，可以进行 DOM 操作。

在这里的 mounted 钩子中，我们通过 fetchNotices 方法从后端获取所有通知，并将其转化为适合 dayspan 的事件数据结构。从后端接口获取通知数据后，遍历每个通知，并将其转化为 dayspan 事件对象的格式。这包括设置标题、颜色、时间、持续时间等属性。然后可以使用 this.loadState() 恢复日历的状态，包括事件和视图设置等。该方法使用浏览器的 localStorage 来存储和恢复日历的用户状态。

4.2.3 交互与标签管理

标签功能允许用户动态管理和筛选活动信息。用户可以添加新的标签、关闭标签，并根据标签筛选不同的通知。

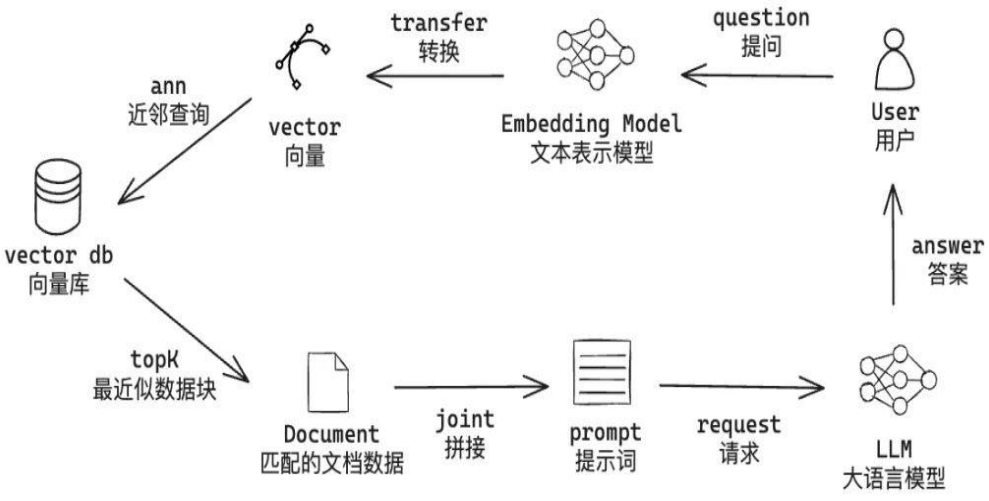
```
methods: {
  handleTagClick(tableName) {
    this.selectedTableName = tableName;
    this.fetchNoticesByCategory(tableName);
  },
```

```
  async fetchNoticesByCategory(tableName) {
    try {
      const response = await
axios.get(`http://127.0.0.1:8000/notices/${tableName}`);
      // 根据类别加载通知数据
    } catch (error) {
      console.error(`Error fetching notices for ${tableName}:`,
error);
    }
  },
},
```

handleTagClick 方法用于根据标签选择不同的通知类型。每次选择标签时，都会调用 fetchNoticesByCategory 来加载该标签下的活动。

4.3 智能问答系统实现

RAG 系统的核心任务是根据用户输入的问题，从存储的通知数据中检索相关的信息，并生成合适的回答。为此，系统利用了深度学习模型来生成文本的嵌入向量，并使用 Milvus 向量数据库进行高效的相似度检索，结合 Coze 大语言模型提供流式响应。



4.3.1 本地数据加载

在 ZJU_TIMEBOX 的实现中，数据来自于浙江大学各大官方网站。通过使用 Scrapy 框架，我们定期从网站抓取校园活动信息，并将其保存为结构化的 JSON 文件。爬取的数据包含了活动的标题、日期、链接和内容等信息。此数据文件将作为后续步骤中的输入，用于进一步的文本处理和向量化。

4.3.2 文档分割（split_documents）

为了高效处理大规模文本数据，并保证嵌入模型的处理效率，我们需要将长文本切分成更小的片段。**文本切分**的目的是将长文本分割成较小的文本块，以便更好地捕捉文本的局部语义特征。预处理步骤通常包括去除停用词、标点符号、特殊字符等，以及将文本转换为小写形式。

在 ZJU_TIMEBOX 中，我们将活动信息的内容分割成多个短文本块，每个块的最大长度设置为 2048 个 token。这一做法避免了大文本块对嵌入模型和检索系统的负担，同时也使得数据在 Milvus 数据库中的存储更加高效。

```
def split_text(text, max_tokens=2048, max_length=7000):
    sentences = nltk.sent_tokenize(text)
    chunks = []
    current_chunk = ""
    for sentence in sentences:
        if len(current_chunk) + len(sentence) <= max_length:
            current_chunk += sentence + ' '
        else:
            chunks.append(current_chunk.strip())
            current_chunk = sentence + ' '
    if current_chunk:
        chunks.append(current_chunk.strip())
    return chunks
```

我们使用了 NLTK 的 sent_tokenize 函数将文本切分成句子，然后根据句子数量限制将它们组合成多个块。这些文本块最终作为检索和向量化的输入。

4.3.3 向量化（embedding）

切分后的文本块需要通过嵌入模型转换成向量，以便在向量数据库中进行高效的相似度检索。在 ZJU_TIMEBOX 中，我们采用了 Jina-embeddings-v3 模型，该模型能够将文本转换为低维的嵌入向量，这些向量能够捕捉文本的语义特征，从而为后续的检索和生成提供支持。

```
def generate_embeddings(texts, task="text-matching"):
    inputs = tokenizer(texts, padding=True, truncation=True,
return_tensors="pt")
    with torch.no_grad():
        outputs = model(**inputs)
    embeddings = outputs.last_hidden_state
    embeddings = embeddings.mean(dim=1)  # 对序列维度进行平均池化
    embeddings = F.normalize(embeddings, p=2, dim=1).numpy()
    return embeddings
```

首先，将预处理后的文本块输入到 Jina-embeddings-v3 模型中，通过模型的前向传播过程，生成每个文本块的隐藏状态表示。然后，对隐藏状态进行池化操作，以获得每个文本块的固定长度的向量表示。常用的池化方法包括平均池化和最大池化。最后，我们对生成的向量进行归一化处理，以确保向量在单位超球面上均匀分布。归一化有助于提高向量之间的余弦相似度计算的准确性。

通过将文本输入到预训练的模型中，我们获得了每个文本块的嵌入向量。这些嵌入向量是向量数据库 Milvus 存储和检索的基础。

4.3.4 数据入库（store data in database）

生成的嵌入向量被存储到 **Milvus 向量数据库**中，这样可以实现高效的相似度搜索。这里可以通过切换路径到 llm_backend/milvus 路径下通过 docker-compose 安装 Milvus 服务。

在 Milvus 中，**集合（collection）**类似于关系型数据库中的**表**。我们需要创建一个集合来存储向量和其他相关信息。在这里不像其他可以通过终端交互的关系型数据库，我们需要通过 pymilvus 库来在 Milvus 中执行操作。Milvus 支持对高维向量进行近似最近邻（ANN）检索，能够快速地找出与查询问题语义最相似的活动信息块。

```
def insert_into_milvus(collection, notifications):
    try:
        titles = []
        dates = []
        links = []
        chunks = []
        embeddings = []

        for notification in notifications:
            title = notification['title']
            date = notification['date']
            link = notification['link']
            content = notification['content']
            chunks_split = split_text(content)

            # 跳过 content=0 的情况
            if len(chunks_split) == 0:
                logging.warning(f"Skipping notification with empty
content: {title}")
                continue
```

```
        emb = generate_embeddings(chunks_split)
        for chunk, vector in zip(chunks_split, emb):
            titles.append(title)
            dates.append(date)
            links.append(link)
            chunks.append(chunk)
            embeddings.append(vector.tolist())

# 构建数据
entities = [
    titles,
    dates,
    links,
    chunks,
    embeddings,
]

# 批量插入数据
batch_size = 1000
for i in range(0, len(titles), batch_size):
    batch_entities = [
        titles[i:i+batch_size],
        dates[i:i+batch_size],
        links[i:i+batch_size],
        chunks[i:i+batch_size],
        embeddings[i:i+batch_size],
    ]
    collection.insert(batch_entities)
```

```
            logging.info(f"Inserted batch {i//batch_size + 1} of
{len(titles)//batch_size + 1}")

        # 刷新集合，确保数据可搜索
        collection.load()
        logging.info("Data inserted into Milvus successfully.")
    except Exception as e:
        logging.error(f"Error inserting data into Milvus: {e}")
```

将嵌入向量批量插入到 Milvus 中，并为后续的查询提供高效的向量存储和检索功能。每个活动的标题、日期、链接以及文本块和嵌入向量都会被一起存储，便于后续检索和回答生成。

4.3.5 检索与增强（Retrieve and Augment）

一旦数据被存储在 Milvus 中，用户的问题会被转化为嵌入向量，并在数据库中进行检索。检索的结果是与问题最相关的活动信息块，这些结果将作为上下文信息传递给大语言模型。这个过程即为**增强（Augment）**，它通过为模型提供相关上下文信息，增强模型的知识库，使得模型能够给出更加精准的回答。

```
def answer_question(question):
    """
    根据用户问题，生成回答。
    """
    print(f"Processing question: {question}")

    # Load the collection into memory
    collection.load()
    print("Loaded collection into memory.")

    # 获取当前时间
```

```
    current_time =
datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    print(f"Current time: {current_time}")

    # 生成问题的 Embedding 向量
    question_embedding = generate_embeddings([question],
task="text-matching")[0].tolist()
    print(f"Question embedding (first 5 dimensions):
{question_embedding[:6]}")

    # 在 Milvus 中搜索相似的向量
    search_results = collection.search(
        data=[question_embedding],
        anns_field="embedding",
        param={"metric_type": "L2", "params": {"nprobe": 10}},
        limit=5,
        expr=None,
        output_fields=["id"]  # 只返回 id 字段
    )[0]

    print(f"Number of search results: {len(search_results)}")
    for hit in search_results:
        print(f"Hit ID: {hit.id}, Distance: {hit.distance}")

    # 检查是否有检索结果
    if not search_results:
        print("No search results found.")
        return "抱歉，我无法找到相关的通知信息。"
```

```
    # 构建 Prompt
    prompt = generate_prompt(search_results, question)

    # 将当前时间添加到 Prompt 中
    prompt += f"\n 当前时间: {current_time}\n"

    # 获取回答
    answer = get_answer_from_coze(prompt)

    # Release the collection from memory if it's no longer needed
    collection.release()
    print("Released collection from memory.")

    return answer
```

这里，answer_question 函数会将用户的提问转化为嵌入向量，然后在 Milvus 数据库中检索最相似的活动信息块。检索到的文本块将被用作对大语言模型的增强上下文。

4.3.6 生成（Generate）

最后，经过增强的上下文与用户问题一起构成了 **生成（Generate）** 阶段的输入。通过与 LLM（如 **Coze Bot**）的交互，系统生成最终的答案。大语言模型会根据提供的上下文，结合自己的语言生成能力，给出与问题相关的答案。

```
def generate_prompt(results, question):
    """
    根据检索到的结果和用户问题，生成用于 LLM 的 Prompt。
    """
    prompt = "使用标记中的内容作为你的知识:\n\n"
    # 加载集合到内存中
```

```
    collection.load()
    print("Loaded collection into memory for querying.")
    for hit in results:
        # 获取实体数据
        entity = collection.query(
            expr=f"id == {hit.id}",
            output_fields=["title", "date", "link", "chunk"]
        )
        if entity:
            entity = entity[0]
            prompt += f"标题: {entity.get('title', 'N/A')}\n时间:
{entity.get('date', 'N/A')}\n链接: {entity.get('link', 'N/A')}\n内容:
{entity.get('chunk', 'N/A')}\n\n"
        else:
            print(f"No entity found for ID: {hit.id}")
    # 释放集合
    collection.release()
    print("Released collection from memory.")

    prompt += (
        "回答要求: \n\n"
        "如果你不清楚答案，绝对不能澄清。\n"
        "避免提及你是从获取的知识。\n"
        "保持答案与中描述的一致。\n"
        "使用不分点的一段话(可使用分号)优化回答格式。\n"
        "使用与问题相同的语言回答。\n\n"
        f"问题: \"{question}\"\n"
    )
    print("Generated prompt for LLM.")
```

```
    return prompt
```

根据检索到的活动信息，我们动态生成一个 **prompt**，并将其与用户的问题一起传递给 Coze Bot，获取模型生成的答案。这就是 RAG 的全部流程。

4.4 服务端操作

4.4.1 FastAPI 应用初始化

首先，我们创建了一个 FastAPI 实例 app，并通过 CORSMiddleware 配置允许来自不同前端应用的跨域请求。这样，即使前端和后端部署在不同的服务器上，仍然可以顺利进行数据交互。

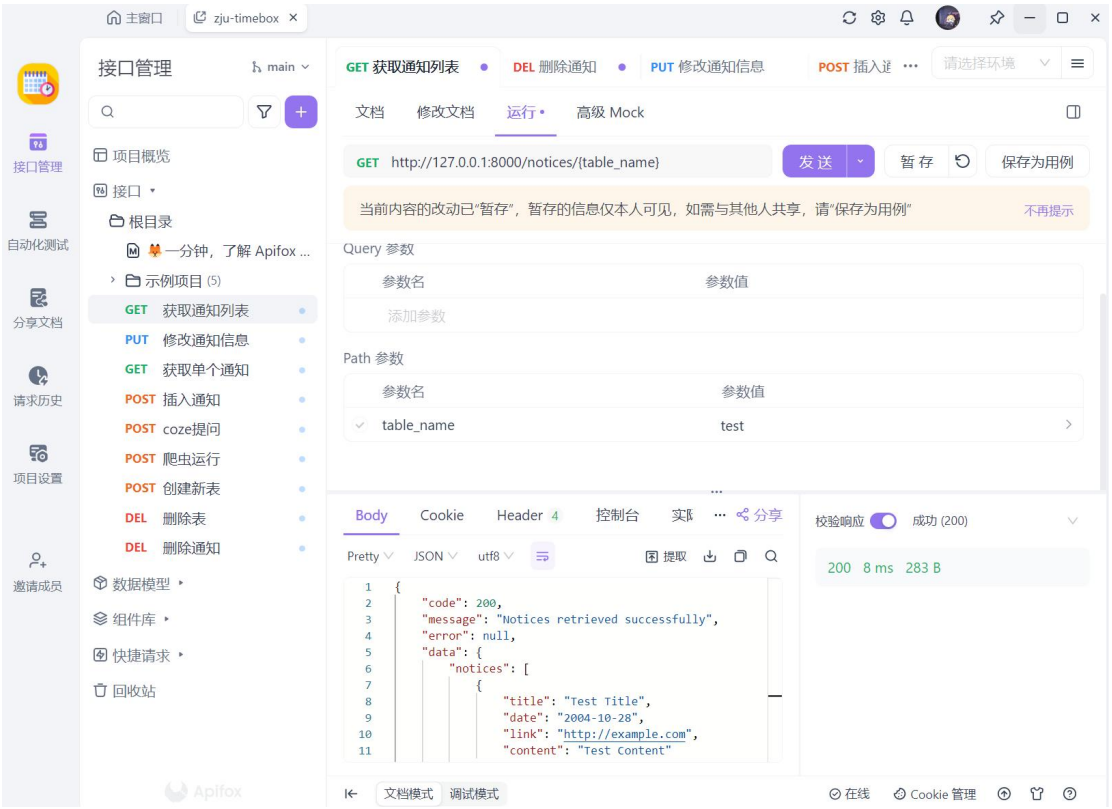
```
# FastAPI 应用
app = FastAPI()
# 添加 CORS 中间件，允许跨域请求
origins = ["http://localhost:8081", "http://10.162.254.38:8081",
"http://localhost:8082"]

app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

4.4.2 SQLite 数据库操作

我们定义了几个 API 端点，用于读取数据库中的通知数据。数据库连接通过 get_db_connection 函数实现，该函数创建了一个 SQLite 连接，并使用 SQL 查询获取通知数据：

```
# SQLite 数据库连接函数 def get_db_connection():
    conn = sqlite3.connect(SQLITE_DB_PATH)
    conn.row_factory = sqlite3.Row  # 使查询结果支持字典样式访问
    return conn
```

4.4.3 增删查改



这里使用 APIFox 来调试路由，当然也可以用 Postman。

```
@app.get("/notices/{table_name}", response_model=CustomResponse)
def read_notices(table_name: str, limit: int = 10):
    try:
        conn = get_db_connection()
        cursor = conn.cursor()
        cursor.execute(f"SELECT title, date, link, content FROM
{table_name} ORDER BY date DESC LIMIT ?", (limit,))
        notices = cursor.fetchall()
```

```
        conn.close()

        if not notices:
            return CustomResponse(code=404, message="No notices found",
error="Not Found")

        validated_notices = []
        for notice in notices:
            try:
                validated_notices.append(Notice(**dict(notice)))
            except ValidationError as e:
                logging.error(f"Validation error for notice:
{notice}")
                logging.error(e)

        return CustomResponse(code=200, message="Notices retrieved
successfully", data={"notices": validated_notices})
    except Exception as e:
        logging.error(f"Error reading notices: {e}")
        raise HTTPException(status_code=500, detail=str(e))
@app.get("/notices/{table_name}/{notice_id}",
response_model=CustomResponse)
def read_notice(table_name: str, notice_id: int):
    try:
        conn = get_db_connection()
        cursor = conn.cursor()
        cursor.execute(f"SELECT title, date, link, content FROM
{table_name} WHERE id = ?", (notice_id,))
        notice = cursor.fetchone()
```

```
        conn.close()

        if notice is None:
            return CustomResponse(code=404, message="Notice not found",
error="Not Found")

        try:
            notice_obj = Notice(**dict(notice))
            return CustomResponse(code=200, message="Notice retrieved
successfully", data={"notice": notice_obj})
        except ValidationError as e:
            logging.error(f"Validation error for notice: {notice}")
            logging.error(e)
            raise HTTPException(status_code=400, detail="Invalid
notice data")
    except Exception as e:
        logging.error(f"Error reading notice: {e}")
        raise HTTPException(status_code=500, detail=str(e))
@app.post("/notices/{table_name}", response_model=CustomResponse,
status_code=status.HTTP_201_CREATED)
def create_notice(table_name: str, notice: Notice):
    try:
        conn = get_db_connection()
        cursor = conn.cursor()
        cursor.execute(
            f"INSERT INTO {table_name} (title, date, link, content)
VALUES (?, ?, ?, ?)",
            (notice.title, notice.date, notice.link, notice.content)
        )
```

```
        conn.commit()
        notice_id = cursor.lastrowid
        conn.close()
        return CustomResponse(code=201, message="Notice created
successfully", data={"notice_id": notice_id})
    except Exception as e:
        conn.close()
        logging.error(f"Error creating notice: {e}")
        raise HTTPException(status_code=400, detail=f"Error creating
notice: {e}")
@app.post("/create_table/{table_name}",
response_model=CustomResponse, status_code=status.HTTP_201_CREATED)
def create_new_table(table_name: str):
    try:
        create_table_if_not_exists(table_name)
        return CustomResponse(code=201, message=f"Table
'{table_name}' created successfully")
    except Exception as e:
        logging.error(f"Error creating table {table_name}: {e}")
        raise HTTPException(status_code=400, detail=f"Error creating
table {table_name}: {e}")
@app.put("/notices/{table_name}/{notice_id}",
response_model=CustomResponse)
def update_notice(table_name: str, notice_id: int, updated_notice:
Notice):
    try:
        conn = get_db_connection()
        cursor = conn.cursor()
```

```
        cursor.execute(f"SELECT id FROM {table_name} WHERE id = ?",
(notice_id,))
        existing_notice = cursor.fetchone()

        if existing_notice is None:
            conn.close()
            return CustomResponse(code=404, message="Notice not found",
error="Not Found")

        cursor.execute(
            f"UPDATE {table_name} SET title = ?, date = ?, link = ?,
content = ? WHERE id = ?",
            (updated_notice.title, updated_notice.date,
updated_notice.link, updated_notice.content, notice_id)
        )
        conn.commit()
        conn.close()
        return CustomResponse(code=200, message="Notice updated
successfully")
    except Exception as e:
        conn.close()
        logging.error(f"Error updating notice: {e}")
        raise HTTPException(status_code=400, detail=f"Error updating
notice: {e}")
@app.delete("/notices/{table_name}/{notice_id}",
status_code=status.HTTP_204_NO_CONTENT)
def delete_notice(table_name: str, notice_id: int):
    try:
        conn = get_db_connection()
```

```
        cursor = conn.cursor()
        cursor.execute(f"SELECT id FROM {table_name} WHERE id = ?",
(notice_id,))
        existing_notice = cursor.fetchone()

        if existing_notice is None:
            conn.close()
            raise HTTPException(status_code=404, detail="Notice not
found")

        cursor.execute(f"DELETE FROM {table_name} WHERE id = ?",
(notice_id,))
        conn.commit()
        conn.close()
        return None  # 返回 None 以确保没有响应体
    except Exception as e:
        conn.close()
        logging.error(f"Error deleting notice: {e}")
        raise HTTPException(status_code=400, detail=f"Error deleting
notice: {e}")
    @app.delete("/delete_table/{table_name}",
status_code=status.HTTP_204_NO_CONTENT)
def delete_table_route(table_name: str):
    try:
        delete_table(table_name)
        return CustomResponse(code=204, message=f"Table
'{table_name}' deleted successfully")
    except Exception as e:
        logging.error(f"Error deleting table {table_name}: {e}")
```

```
        raise HTTPException(status_code=400, detail=f"Error deleting table {table_name}: {e}")
```

4.4.4 API Scrapy 异步接口

爬虫任务通过 run_crawlers 端点触发。在此端点中，我们启动了一个后台线程来执行爬虫操作，避免阻塞 FastAPI 主线程。爬虫完成后，爬取的数据会被插入到数据库中：

```
@app.post("/run_crawlers")def run_crawlers():
    try:
        # 使用线程运行爬虫，避免阻塞 FastAPI 的主线程
        def run_and_populate():
            run_all_crawlers()  # 运行爬虫
            populate_database_from_json()  # 在爬虫完成后填充数据库

        thread = threading.Thread(target=run_and_populate)
        thread.start()
        return {"message": "Crawlers started successfully and data will be inserted into the database upon completion."}

    except Exception as e:
        logging.error(f"Error running crawlers: {e}")
        raise HTTPException(status_code=500, detail=str(e))
```

通过 threading.Thread 我们可以将爬虫任务放到独立的线程中执行，保证主线程不受影响。

4.4.5 问答系统接口

在 /ask 端点，我们接收用户提出的问题，并调用 answer_question 函数返回问题的答案。

```
@app.post("/ask")async def ask_question(request: QuestionRequest):
    question = request.question
    if not question:
        raise HTTPException(status_code=400, detail="请输入问题哦！")


    try:
        answer = answer_question(question)  # 调用 answer_question 生成回答
        return {"question": question, "answer": answer}
    except Exception as e:
        logging.error(f"Error processing question: {e}")
        raise HTTPException(status_code=500, detail="处理问题时出现异常")
```

4.4.6 监听端口

uvicorn 是一个 ASGI 服务器，用于运行 FastAPI 应用。我们将其设置为监听 127.0.0.1:8000，通过以下代码启动 FastAPI 服务：

```
if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="127.0.0.1", port=8000)
```

5 总结与展望

5.1 创新性

5.1.1 智能化信息检索系统

传统的活动信息查询通常依赖关键词匹配，这种方式在面对多样化的查询需求时难以提供精准的答案。ZJU_TIMEBOX 通过结合 Milvus 向量数据库 和大语言模型（Coze Bot），实现了基于语义的智能问答系统。该系统不仅能够对活动信息进行向量化处理，还能根据用户问题进行语义匹配，从而提供更加精准和个性化的回答。这种基于向量搜索和大语言模型的创新结合，突破了传

统关键词检索的局限，提升了信息检索的准确性和用户体验。

5.1.2 实时数据抓取与更新机制

由于活动信息往往需要实时更新，传统的人工或手动更新方式无法满足高效、准确的需求。ZJU_TIMEBOX 利用 Scrapy 爬虫框架 实现了对浙江大学各类活动信息的自动化抓取，并结合 Milvus 向量数据库 的高效存储和检索，确保了活动信息能够实时更新，并且快速呈现给用户。这一机制极大地提高了信息管理的效率，确保了用户能够随时获得最新的校园活动数据。

5.1.3 多功能集成平台

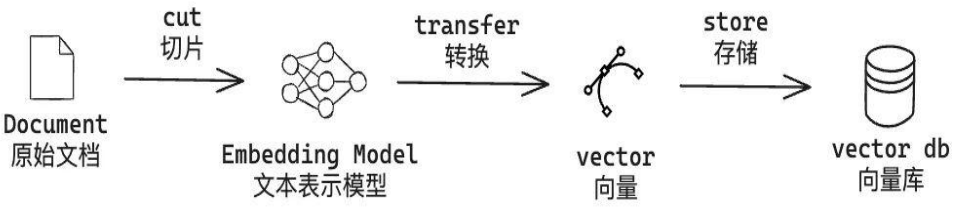
ZJU_TIMEBOX 不仅集成了日历界面展示、智能问答功能，还通过整合多种技术栈（如 Vue.js、FastAPI、Scrapy、Milvus）构建了一个功能齐全的动态信息追踪平台。通过使用前端框架 Vue.js，我们为用户提供了直观、易用的日历界面，帮助他们轻松浏览和获取活动信息。同时，后端通过 FastAPI 提供高效的 API 服务，保证了数据的高可用性和可扩展性。

5.1.4 智能问答系统的可扩展性

通过集成 Coze Bot 和 Milvus 的组合，系统不仅实现了高效的活动信息检索和语义理解，还具备了强大的扩展能力。Coze Bot 的自然语言处理能力可以轻松适应不同领域的问答需求，而 Milvus 数据库的分布式向量存储和近似最近邻（ANN）算法则使得该系统能够处理大规模数据。未来可以通过更强大的大语言模型或自定义知识库，进一步提升问答系统的智能化水平。

5.1.5 面向多语言用户的设计

项目使用 Jina-embeddings-v3 模型对文本进行多语言向量化处理，确保系统不仅支持中文用户，也能够满足英语或其他语言用户的需求。随着国际化需求的增长，这种多语言支持为 ZJU_TIMEBOX 提供了更广泛的应用场景和市场潜力。



5.2 未来展望

5.2.1 移动端应用开发

目前，ZJU_TIMEBOX 主要通过 Web 端展示活动信息和问答服务。未来，可以考虑将其转化为一个移动端应用（如 iOS/Android APP），以便为用户提供更便捷的使用方式（就像 Celechron 那样）。通过推送通知、活动提醒、即时问答等功能，提升用户体验，使得师生在忙碌的校园生活中能够更及时地获取信息。

5.2.2 个性化适用性与推广

虽然当前系统聚焦于浙江大学的活动信息，但其架构和设计具有较强的可扩展性，未来可以推广至其他活动。通过开放 API 或二次开发，其他信息源也可以进一步多样化，帮助（包括浙江大学之外的）学生和教职工便捷地获取校园活动信息，以及更多的社交平台（如微信群、微信公众号等）或校内管理系统的数据，进一步扩展活动信息的覆盖面，并提供更加全面的服务。

5.2.3 RAG 系统的深度优化

当前的 RAG 系统已经能够通过语义匹配提供高效的答案，但在面对一些模糊、复杂或上下文依赖较强的问题时，仍有提升空间。未来可以进一步优化模型的推理能力，结合上下文理解、对话管理等技术，使得系统能够提供更加准确和自然的对话体验。此外，还可以加入用户行为分析，通过历史问题和交互数据，优化 LLM 系统的推荐算法，提前推送用户可能感兴趣的活动信息。

5.2.4 增强的数据可视化功能

虽然当前系统通过日历界面展示活动信息，但对于数据的进一步分析和挖掘还有很大的提升空间。未来可以通过集成数据可视化模块，展示活动的统计数据（如活动参与人数、活动频率等），还有 icalender 日历订阅链接的生成，以及活动类型分布等内容。

我们的项目也受到了上面 ZJU-ical 的启发。这不仅可以帮助用户更好地了解校园活动的全貌，还能为学校的管理者提供有价值的决策支持。

5.2.5 增强的用户自定义功能

目前，系统主要以预设的活动类型展示信息，未来可以允许用户自定义信息展示的方式。例如，用户可以选择关注某些特定类型的活动（如讲座、实践活动等），系统根据用户的偏好推荐相关的活动。此外，用户还可以设置定期提醒、活动收藏等个性化功能，进一步提升平台的用户粘性和满意度。

5.2.6 技术架构的升级与优化

随着用户规模的扩大和数据量的增长，当前的技术架构可能需要进一步优化。例如，可以考虑将数据库从 SQLite 迁移到更为高效和可扩展的数据库（如 MySQL、PostgreSQL 等），并优化 Milvus 的集群部署以应对更大规模的数据存储和查询需求。此外，还可以通过引入更高效的模型（如更强大的 LLM）和优化向量存储方式，进一步提升问答系统的响应速度和准确度。

ZJU_TIMEBOX 项目从技术架构到功能设计，通过整合多种技术手段，包括爬虫抓取、检索增强生成、自然语言处理等，为浙江大学的师生提供了一个高效、智能、实时的信息获取平台。