Python大作业

基于Python的B站舆情分析系统

实验报告

姓 名： 张楚轩

学 号： 2023211472

班 级： 2023211803

摘要

本报告详细阐述了一个基于Python语言的Bilibili（B站）舆情分析系统的设计与实现过程。该系统旨在通过自动化程序，实现对B站特定舆情信息进行采集、处理、分析和可视化。系统核心功能包括：根据指定“主题”或“用户UID”爬取B站动态及评论数据；设计并实现合理的数据库结构对数据进行持久化存储；构建多维度负面舆情关键词库，通过“关键词匹配+情感分析”的二级策略，筛选和识别潜在的负面内容；最后，对分析结果进行统计和可视化展示。

本项目综合运用了Selenium自动化测试框架、SQLite数据库、Flask Web框架以及Pyecharts可视化库等多种技术，在macOS环境下完成开发与测试。报告将重点围绕系统各模块的设计思路、技术选型理由、开发过程中遇到的关键问题及其解决方案进行深入探讨，并总结本次课程设计的收获与未来可行的优化方向。

关键词： Python；网络爬虫；Bilibili；舆情分析；数据可视化；Selenium；Flask

1. 实验目的

本次课程设计的主要目的在于综合运用本学期所学的Python编程知识，结合网络爬虫、数据库管理、数据分析等技术，构建一个功能完整的小型应用系统。通过该项目，旨在达成以下具体目标：

1. **掌握网络数据采集技术：** 熟练使用Selenium等工具，针对B站这类动态加载内容的网站，实现对特定主题动态、用户动态及其相关评论、“楼中楼”回复的精准爬取。

2. **学习数据库设计与应用：** 能够根据B站动态与评论的层级关系设计合理的数据表结构，并使用Python的`sqlite3`库进行高效的数据增删改查操作。
3. **探索文本数据分析方法：** 学习并实践基于关键词规则的文本分类方法，理解其在负面舆情发现等场景中的应用，并结合情感分析算法提升识别准确性。
4. **提升系统设计与工程能力：** 体验一个软件项目从需求分析、技术选型、模块化开发到最终集成的完整流程，培养模块化编程思想和解决复杂问题的能力。
5. **实践数据可视化：** 学习使用Pyecharts等可视化库，将抽象的数据分析结果以图表形式直观地呈现出来，提升报告的可读性和说服力。

2. 实验设置

2.1 实验环境

- **硬件环境：** Apple MacBook Air4(M4 Chip, 16GB RAM)
- **软件环境：**
  - 操作系统：macOS 15.4
  - 编程语言：Python 3.12
  - 主要开发库（详见`requirements.txt`）
    - `Flask (2.2.2)`: 用于构建Web应用后端，提供API接口并与前端交互。
    - `Selenium (4.8.3)`: 核心爬虫工具，用于驱动浏览器模拟用户行为，处理JavaScript动态加载的内容。
    - `Pyecharts (2.0.3)`: 用于生成交互式的数据可视化图表。
    - `Jieba (0.42.1)`: 中文分词库，用于在舆情分析时对文本进行分词处理。
    - `SnowNLP (0.12.3)`: 简化的中文自然语言处理库，用于情感分析。
    - `Pandas (1.5.3)`: 用于数据处理和分析。
- **开发工具：** Trae

2.2 实验数据

- **数据来源：** 哔哩哔哩（bilibili.com）。选择B站作为数据源是因为其拥有庞大且活跃的年轻用户社群，内容形式多样（视频、动态），评论区互动频繁，是当代网络舆情的重要发源地。
- **数据范围：**
  1. **按主题爬取：** 输入特定关键词，爬取B站综合搜索结果下“动态”板块的前100条动态。

2. **按用户爬取：** 输入特定用户的UID，爬取其个人空间“动态”板块下的所有动态。
3. **关联数据：** 爬取上述动态下的所有评论，以及评论下的所有“楼中楼”式回复，并完整保留其层级关系。

- **数据字段：** 爬取的数据主要包括动态ID、UP主昵称、UP主UID、动态内容、发布时间、评论ID、评论内容、评论时间、被回复的评论ID等。

3. 实验方法与过程

3.1 系统总体设计

【a】为什么要这么做（设计思路）：

为了构建一个结构清晰、易于维护和扩展的系统，我采用了经典且高效的**前后端分离架构**。这种模式将负责用户交互的**前端（Presentation Layer）与负责业务逻辑和数据处理的**后端（Business Logic & Data Layer）彻底分离开。它们之间通过定义好的API（应用程序编程接口）进行通信。

这样的设计带来了几个核心优势：

- **职责单一：** 前端专注于UI/UX（界面与体验），后端专注于核心功能实现。开发人员可以并行工作，互不干扰。
- **易于维护与调试：** 当出现问题时，可以快速定位是前端展示错误还是后端逻辑错误。各模块可以独立测试。
- **高扩展性与灵活性：** 未来可以轻易地更换或升级某一层。例如，可以为同一个后端开发一个全新的前端（如手机App），而无需改动后端代码。
  在本项目中，这一架构的具体体现为：

前端 (Frontend)

前端是用户与本系统交互的直接入口，位于 `frontend/` 目录下。我使用 **HTML** (`templates/index.html`) 搭建页面骨架，**CSS**（内联或外部）美化样式，并利用 **JavaScript** (`static/js/main.js`) 赋予页面动态交互的能力。

前端的核心职责是：

1. 提供清晰的用户操作界面（输入框、按钮）。
2. 捕获用户的输入（如要搜索的主题或用户UID）。
3. 当用户点击按钮时，通过 **AJAX (Asynchronous JavaScript and XML)** 技术向后端Flask服务器发送异步API请求。

“异步通信”是提升用户体验的关键。它意味着前端在等待后端处理（例如，一个耗时较长的爬虫任务）时，页面不会被冻结或刷新，用户依然可以进行其他操作。这在 `main.js` 中通过 `fetch` API 实现：

```
// frontend/static/js/main.js 关键代码片段
document.getElementById('crawl-topic-
btn').addEventListener('click', () => {
    const topic = document.getElementById('topic-input').value;
    // ...省略了加载状态的UI更新代码...

    // 使用fetch向后端发起异步POST请求
    fetch('/start_crawl_by_topic', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
        },
        body: JSON.stringify({ topic: topic }), // 将主题作为JSON数
据发送
    })
    .then(response => response.json())
    .then(data => {
        // 收到后端响应后，更新页面信息
        console.log(data.message);
        document.getElementById('status').innerText =
data.message;
    })
    .catch(error => {
        console.error('Error:', error);
        document.getElementById('status').innerText = '爬取失败，请
查看控制台。';
    });
});
```

这段代码展示了当用户点击“按主题爬取”按钮时，前端如何将输入框中的主题打包成JSON，并通过`fetch`发送到后端的`/start_crawl_by_topic`这个API端点。

后端 (Backend)

后端是整个系统的大脑，使用 **Python** 和轻量级的 **Flask** Web框架搭建。它不处理任何具体的HTML页面渲染，而是作为一个纯粹的API服务器。其内部又被精心划分为多个各司其职的模块：

1. API控制器 (`app.py`)

`app.py` 文件是后端的入口和总调度中心。它使用Flask的路由装饰器（`@app.route()`）来定义前端可以访问的各个API端点。当一个请求（如前端发来的`/start_crawl_by_topic`）到达时，`app.py`中对应的函数会被触发。这个函数本身不执行复杂的逻辑，而是像一个“项目经理”，负责调用其他专业模块（如爬虫、分析模块）来完成具体工作。

PYTHON

```python
# app.py 关键代码片段
from flask import Flask, request, jsonify, render_template
from backend.crawler_improved import Crawler
# ...其他导入...

app = Flask(__name__, template_folder='../frontend/templates',
static_folder='../frontend/static')

# 定义API端点，处理按主题爬取的请求
@app.route('/start_crawl_by_topic', methods=['POST'])
def start_crawl_by_topic():
    data = request.get_json()
    topic = data.get('topic')
    if not topic:
        return jsonify({"status": "error", "message": "主题不能为
空"}), 400

    # 1. 实例化爬虫对象
    crawler = Crawler()
    # 2. 调用爬虫方法，将具体工作委托给爬虫模块
    crawler.start_by_topic(topic)

    return jsonify({"status": "success", "message": f"主题
'{topic}' 的爬取任务已启动。"})
```

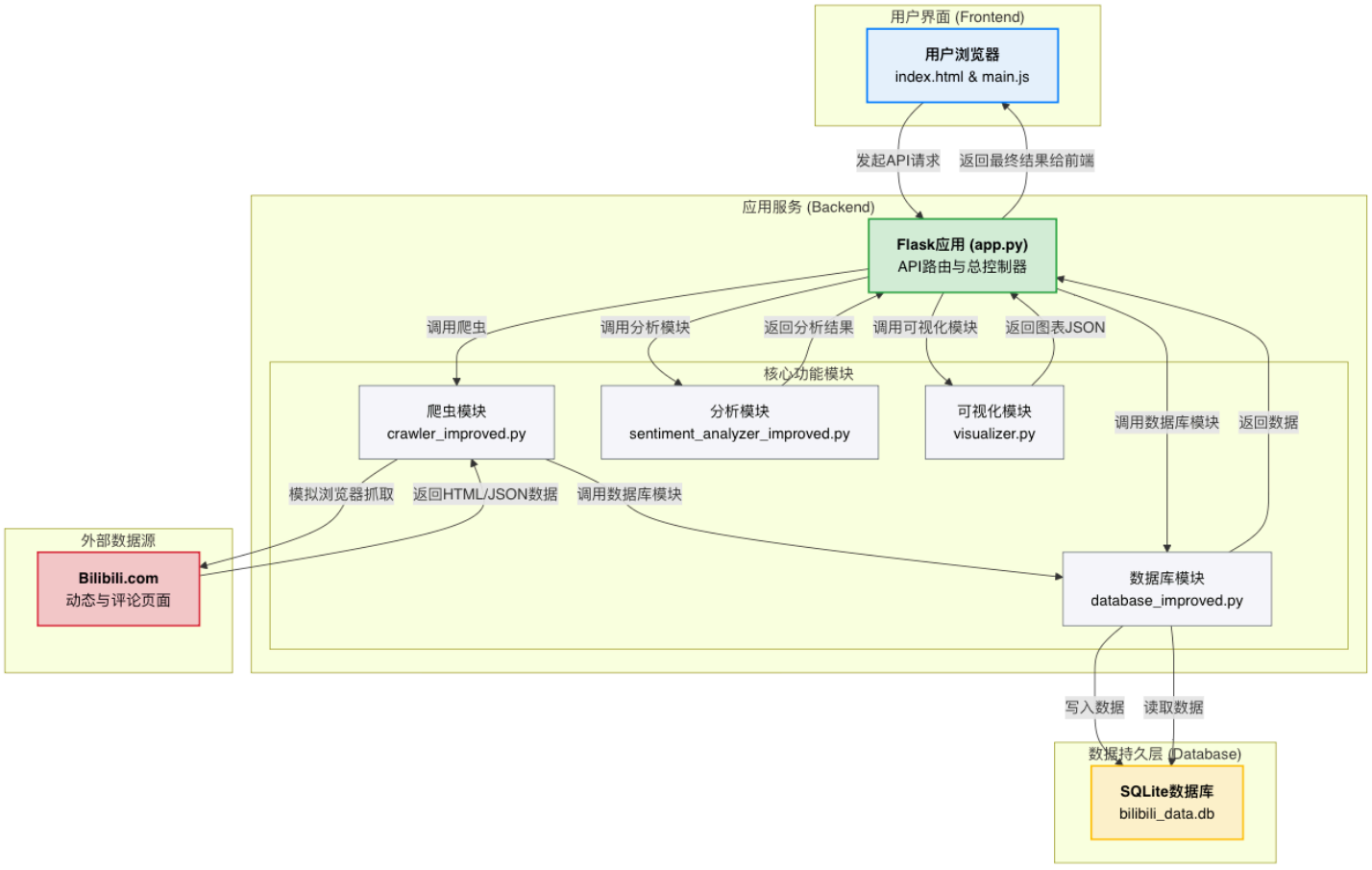
从代码可见，`app.py`的角色是解析请求、验证参数，然后将任务转交给`Crawler`类。

2. 核心功能模块 (`backend/` 目录)

这是具体“干活”的模块，它们被设计为可重用的类或函数库。

- **爬虫模块 (`backend/crawler_improved.py`):** 负责所有与B站的交互。它被`app.py`调用后，会启动Selenium浏览器，执行搜索、滚动、点击、数据提取等一系列复杂操作。一个重要的设计是，爬虫模块在获取到数据后，会**直接调用数据库模块**来完成数据持久化，而不是将庞大的数据返回给`app.py`。这体现了模块间的直接协作，提高了效率。
- **数据库模块 (`backend/database_improved.py`):** 该模块封装了所有与SQLite数据库的交互。它提供了一套简洁明了的接口（如`save_post()`, `get_all_posts_and_comments()`），供其他模块（主要是爬虫和分析模块）使用。这种封装将SQL语句与业务逻辑分离，使得代码更易读，且未来如果想更换数据库（如从SQLite换成MySQL），只需修改这一个文件即可。
- **分析与可视化模块 (`backend/sentiment_analyzer_improved.py` & `backend/visualizer.py`):** 当需要进行数据分析时，`app.py`会先通过数据库模块获取所需数据，然后将数据传递给这两个模块。`sentiment_analyzer`负责执行关键词匹配和情感分析，返回被标记的负面文本。`visualizer`则接收分析后的统计数据，利用Pyecharts生成图表所需的JSON配置。

通过这样的分层和模块化设计，整个系统形成了一个清晰的数据流：**用户输入 → 前端JS → Flask控制器 → 爬虫/分析模块 → 数据库 → 分析/可视化结果 → Flask控制器 → 前端JS → 页面呈现**。这个流程中的每一步都由专门的模块负责，确保了系统的高内聚、低耦合，为项目的成功奠定了坚实的架构基础。



3.2 数据爬取模块 (`backend/crawler_improved.py`)

【a】为什么要这么做（技术选型）：

B站的页面，无论是动态列表还是评论区，都大量使用AJAX技术动态加载内容。用户必须向下滚动页面，才能触发新的动态或评论加载出来。特别是B站的评论区，点击“查看更多回复”才会展示“楼中楼”内容。传统的requests库无法执行JavaScript，只能获取到页面的静态骨架。因此，我选择使用Selenium。Selenium可以驱动一个真实的浏览器（如Chrome），模拟用户的滚动、点击等操作，从而触发JavaScript加载数据，确保能够获取到所有需要的信息。

【b】遇到了什么问题与【c】如何解决：

1. 问题：登录验证与反爬虫。

   B站在未登录状态下，可查看的内容非常有限，且频繁访问容易触发验证码。自动化程序需要一种稳定的免登录策略。

   解决方案： 我采用Selenium加载本地浏览器用户数据的方式实现自动登录。在config.py中配置好Chrome的用户数据路径，首次运行时，手动登录一次B站，浏览器会保存登录状态（Cookies）。后续Selenium启动时加载这个带有登录信息的用户配置，即可自动保持登录状态，绕过登录验证。

   ```
   # backend/crawler_improved.py 部分代码
   from selenium.webdriver.chrome.options import Options

   chrome_options = Options()
   # 通过加载用户数据目录实现免登录
   chrome_options.add_argument(f"user-data-dir={USER_DATA_PATH}")
   driver = webdriver.Chrome(options=chrome_options)
   ```

2. 问题：动态加载与“无限滚动”。

   B站的动态列表和评论列表都是“无限滚动”的。

   解决方案： 我编写了一个循环，在循环中通过执行JavaScript window.scrollTo(0, document.body.scrollHeight) 来将页面滚动到底部。每次滚动后，等待一小段时间（time.sleep）让新内容加载，然后通过比较滚动

前后的页面高度来判断是否已到达底部。如果高度不再增加，则说明所有内容已加载完毕。

3. 问题：评论与“楼中楼”回复的层级关系。

   B站的回复是嵌套在主评论内部的，形成了“楼中楼”结构。在爬取时必须准确记录这种父子关系。

   解决方案： 在爬取时，我首先定位到每一条主评论的HTML元素（div.reply-item）。然后，在该元素内部，先提取主评论自身的信息，再查找其下属的回复元素（div.sub-reply-item）。在存入数据库时，将主评论的ID作为其所有下属回复的parent_id字段值，从而在数据库中清晰地建立起层级关联。

【d】收获和心得体会：

通过实现B站的爬虫，我深刻理解了动态网页爬取的复杂性。它不仅需要分析HTTP请求，更需要分析前端的JavaScript交互逻辑。编写健壮的爬虫必须考虑到各种异常情况，如网络延迟、元素未找到（B站页面结构偶尔会微调）、反爬验证等，并加入适当的显式等待（WebDriverWait）和try-except异常处理机制，这比简单的time.sleep更加高效和可靠。

3.3 数据存储模块 (`backend/database_improved.py`)

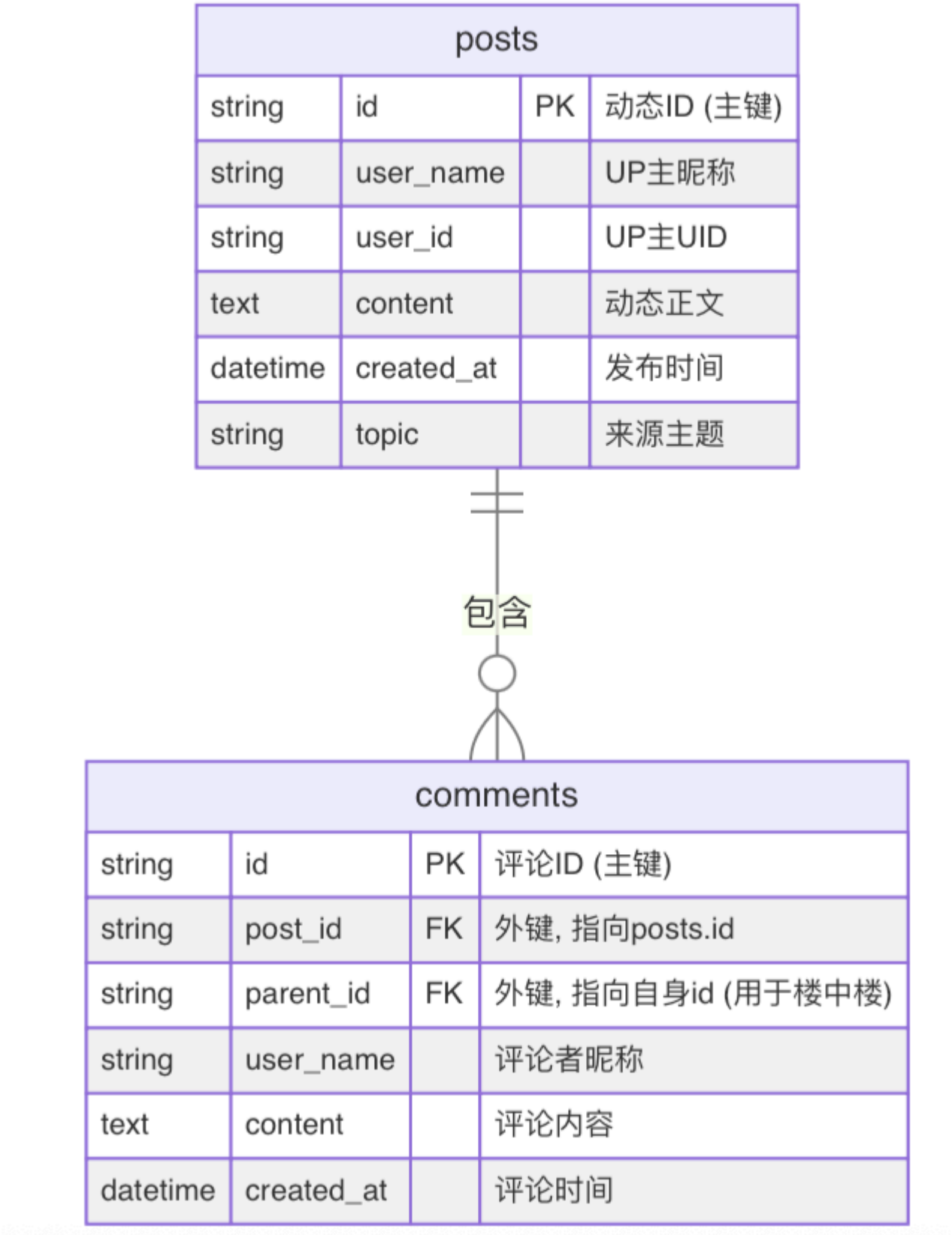
【a】为什么要这么做（设计思路）：

为了以“合理的结构”存储数据，特别是保留B站评论的“楼中楼”关系，我设计了两张核心数据表：posts（动态表）和comments（评论表）。

- **技术选型：** 我选择了**SQLite**作为数据库。因为它是一个轻量级的、无服务器的数据库，直接以单个文件（`bilibili_data.db`）的形式存在。对于本项目这种中小型数据量、单机运行的应用来说，SQLite无需复杂的配置，使用Python内置的`sqlite3`库即可操作，非常适合快速原型开发。
- **表结构设计：**
  1. `posts` **表：**（在代码中命名为posts，实际存储B站动态）
     - `id`: 主键，动态ID。
     - `user_name`: UP主昵称。
     - `user_id`: UP主UID。
     - `content`: 动态正文。

- `created_at`: 发布时间。
- `topic`: 动态来源的主题（如果是按主题爬取）。

2. `comments` **表:**
    - `id`: 主键，评论ID。
    - `post_id`: 外键，关联到`posts`表的`id`，表明该评论属于哪条动态。
    - `user_name`: 评论者昵称。
    - `content`: 评论内容。
    - `created_at`: 评论时间。
    - `parent_id`: **核心字段**，用于记录回复关系。如果是一条主评论，该值为NULL；如果是一条“楼中楼”回复，该值为其父评论的ID。



【b】遇到了什么问题与【c】如何解决：

- 问题：数据重复插入。 如果对同一个主题或用户执行多次爬取，可能会导致数据库中出现大量重复数据。
- **解决方案：** 在插入数据前进行检查。我为`posts`表和`comments`表的`id`字段都设置了`PRIMARY KEY`约束。在Python代码中，使用`INSERT OR IGNORE INTO ...`的SQL语句。这样，当尝试插入一个已存在的ID时，数据库会自动忽略该操作，而不会报错中断，从而高效地保证了数据的唯一性。

```
# backend/database_improved.py 部分代码
def save_post(self, post_data):
    # 使用 INSERT OR IGNORE 避免重复插入
    sql = ''' INSERT OR IGNORE INTO posts (id, user_name, user_id,
content, created_at, topic)
              VALUES (?,?,?,?,?,?) '''
    self.cursor.execute(sql, (post_data['id'], ...,
post_data['topic']))
    self.conn.commit()
```

3.4 负面舆情分析模块 (`backend/sentiment_analyzer_improved.py`)

【a】为什么要这么做（定义“负面舆情”与实现方法）：

根据实验要求，负面舆情需要围绕“涉政有害”、“侮辱谩骂”等六个主题展开。由于缺乏专业的训练模型，我将“负面舆情”定义为：文本内容中包含了与这六个主题相关的特定关键词，并且整体情感倾向偏向消极的动态或评论。

基于这个定义，我设计了一个**关键词匹配 + 情感分析辅助**的二级分析策略：

1. **一级筛选（关键词匹配）：**
    - 首先，我借助网络资源和GPT等大模型，为六个主题分别构建了关键词表（每个不少于50个词），存储在`config.py`的`KEYWORD_LISTS`字典中。

```
class SentimentAnalyzer:
    def __init__(self):
        # 6个负面舆情类别的关键词表
        self.negative_keywords = {
            '涉政有害': [
                '政府', '官员', '腐败', '贪污', '受贿', '权力', '独裁', '专制', '民主', '自由',
                '人权', '抗议', '示威', '游行', '革命', '政变', '推翻', '颠覆', '反政府', '反党',
                '分裂', '独立', '台独', '港独', '藏独', '疆独', '法轮功', '六四', '天安门', '敏感',
                '封锁', '屏蔽', '删帖', '河蟹', '和谐', '404', '翻墙', 'VPN', '代理', '境外势力',
                '西方', '美帝', '反华', '汉奸', '卖国贼', '走狗', '间谍', '特务', 'CIA', 'FBI',
                '制裁', '封杀', '禁令', '黑名单', '监控', '监视', '跟踪', '窃听', '审查', '言论自由'
            ],
            '侮辱谩骂': [
                '傻逼', '白痴', '智障', '脑残', '弱智', '蠢货', '垃圾', '废物', '人渣', '败类',
                '畜生', '禽兽', '狗东西', '贱人', '婊子', '妓女', '鸡', '操', '草', '艹',
                '滚', '死', '去死', '该死', '找死', '活该', '报应', '天谴', '下地狱', '不得好死',
                '全家死光', '断子绝孙', '生孩子没屁眼', '你妈', '你爸', '你爷爷', '你奶奶', '祖宗',
                '骂街', '泼妇', '恶心', '恶心人', '讨厌', '烦人', '讨打', '欠揍', '找抽', '犯贱',
                '不要脸', '无耻', '卑鄙', '下流', '龌龊', '肮脏', '恶心', '变态', '神经病', '有病'
            ],
            '色情暴力': [
                '色情', '黄色', '淫秽', '猥亵', '性', '做爱', '性交', '强奸', '轮奸', '性侵',
                '性骚扰', '露点', '裸体', '脱衣', '三点', '胸部', '乳房', '下体', '生殖器', '阴茎',
                '阴道', '肛门', '自慰', '手淫', '口交', '肛交', '群交', '乱伦', '偷情', '出轨',
                '暴力', '打架', '斗殴', '械斗', '砍杀', '刺杀', '枪杀', '爆炸', '炸弹', '恐怖袭击',
                '血腥', '残忍', '虐待', '折磨', '酷刑', '处决', '枪决', '绞刑', '斩首', '肢解',
                '杀人', '谋杀', '凶杀', '血案', '惨案', '屠杀', '大屠杀', '种族灭绝', '战争', '武器'
            ],
            '事故灾难': [
                '事故', '车祸', '交通事故', '撞车', '追尾', '翻车', '坠车', '爆胎', '刹车失灵', '酒驾',
                '火灾', '爆炸', '煤气爆炸', '瓦斯爆炸', '化工爆炸', '起火', '燃烧', '烧死', '烧伤', '窒息',
                '地震', '海啸', '台风', '洪水', '泥石流', '山体滑坡', '雪崩', '冰雹', '龙卷风', '暴雨',
                '干旱', '饥荒', '瘟疫', '传染病', '病毒', '细菌', '感染', '死亡', '伤亡', '遇难',
                '坠落', '坠楼', '跳楼', '自杀', '溺水', '中毒', '食物中毒', '煤气中毒', '触电', '雷击',
                '矿难', '塌方', '坍塌', '建筑倒塌', '桥梁垮塌', '安全事故', '工伤', '职业病', '医疗事故', '误诊'
            ],
```

- 分析时，首先使用`jieba`库对文本内容进行分词。
- 然后，遍历分词结果，检查是否有词语出现在任何一个负面主题的关键词库中。如果匹配成功，则将该文本初步归类为对应的负面类别。

2. **二级修正（情感分析）：**

- **【b】遇到的问题：** 单纯的关键词匹配存在明显的**误判**问题。例如，在“侮辱谩骂”类别中，关键词“垃圾”可能出现在“这个游戏太垃圾了”中，也可能出现在“求助，该怎么垃圾分类”中。前者是负面，后者是中性。
- **【c】如何解决：** 为了提高准确率，我引入了`SnowNLP`库进行情感倾向分析。对于一级筛选出的文本，再进行一次情感打分。`SnowNLP`会返回一个0到1之间的值，越接近1表示情感越积极，越接近0表示情感越消极。我设定一个情感阈值（在代码中为0.3），只有当文本的情感分数低于该阈值时，才最终判定其为负面舆情。
- **【d】收获：** 这个过程让我认识到，文本分析任务远比想象的复杂。简单的规则虽然高效，但精度有限。结合多种方法（规则+算法）是提高系统智

能性的有效途径。

```
# backend/sentiment_analyzer_improved.py 核心逻辑代码
from snownlp import SnowNLP
import jieba

class SentimentAnalyzer:
    def analyze(self, text):
        # ... 加载关键词库 ...

        # 1. 关键词匹配
        matched_category = None
        words = jieba.lcut(text)
        for word in words:
            for category, keywords in self.keyword_lists.items():
                if word in keywords:
                    matched_category = category
                    break
            if matched_category:
                break

        if not matched_category:
            return "正常", 1.0

        # 2. 情感分析辅助判断
        sentiment_score = SnowNLP(text).sentiments
        if sentiment_score < 0.3: # 情感消极才判定为负面
            return matched_category, sentiment_score
        else:
            return "正常", sentiment_score # 情感偏积极，修正为正常
```

3.5 可视化与最终呈现 (`app.py`, `backend/visualizer.py`)

【a】为什么要这么做：

为了直观地展示分析结果，我使用Flask搭建了一个简单的Web界面。用户可以在网页上输入B站的搜索主题或用户UID，点击按钮后，后端执行爬取和分析，并将结

果动态呈现在页面上。

【b】遇到的问题与【c】如何解决：

- **问题：如何将后端Python生成的Pyecharts图表在前端HTML中展示?**
- **解决方案：** `Pyecharts`库可以非常方便地将生成的图表配置保存为一个JSON对象（通过`chart.dump_options()`方法）。在Flask的路由函数中，我调用`visualizer.py`模块生成图表所需的JSON数据，然后通过API接口将其返回给前端。前端的JavaScript代码（`main.js`）接收到这些JSON数据后，使用Echarts.js库将其渲染成动态图表。

最终呈现效果：

系统主界面为一个集成前端包含一个输入区和一个结果展示区。

1. **输入区：** 提供两个输入框，分别用于输入“主题”和“用户UID”，以及两个对应的执行按钮。

2. **结果展示区：**
   - 搜索主题为“特朗普”，爬取条数为10条，下图为实时更新状况

爬取状态

正在爬取视频评论: 特朗普视察新移民监狱 画面曝光：监狱位于沼泽地 室内布铁丝网...

24%

■ 停止爬取

实时爬取结果

1 特朗普“亲自剪彩”的“鳄鱼恶魔岛”究竟什么来头?

7 条负面评论

UP主: 新华社　　播放: 5.9万

发布: 2025/07/03 03:23　　点赞: 100

处理时间: 2025/07/03 02:24

负面评论:

飞羽如烟 2025/07/03 03:26　94

buff拉满了，沼泽地，关押严格，逃跑可能低（还有个非法移民大多都是雨林沼泽进去的）

拜了个~白 2025/07/03 06:41　0

佛罗里达不养闲人

DNA靛王酶 2025/07/03 06:52　0

之前这里好像也是个监狱，囚犯和狱警监还发生了大暴乱，后来还传出了诡异传闻

浮生若梦_欢几何 2025/07/03 05:21　2

《萝莉岛》：先来后到哈，不要乱。

长生不老娘 2025/07/03 05:35　0

要不说资本主义会赚钱呢，里里外外都赚一遍

一笺浅墨深 2025/07/03 03:36　55

关进去再拍一档“”逃出鳄鱼岛”真人秀，川普又大赚特赚了[笑哭][笑哭][笑哭]

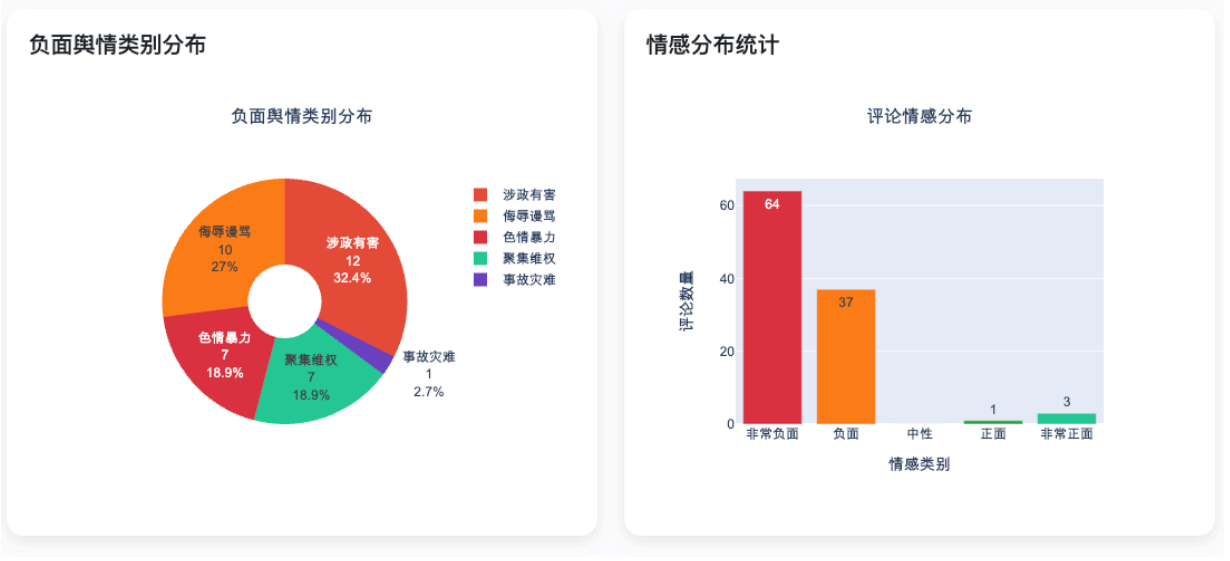
啵士顿圆脸 2025/07/03 03:40　2

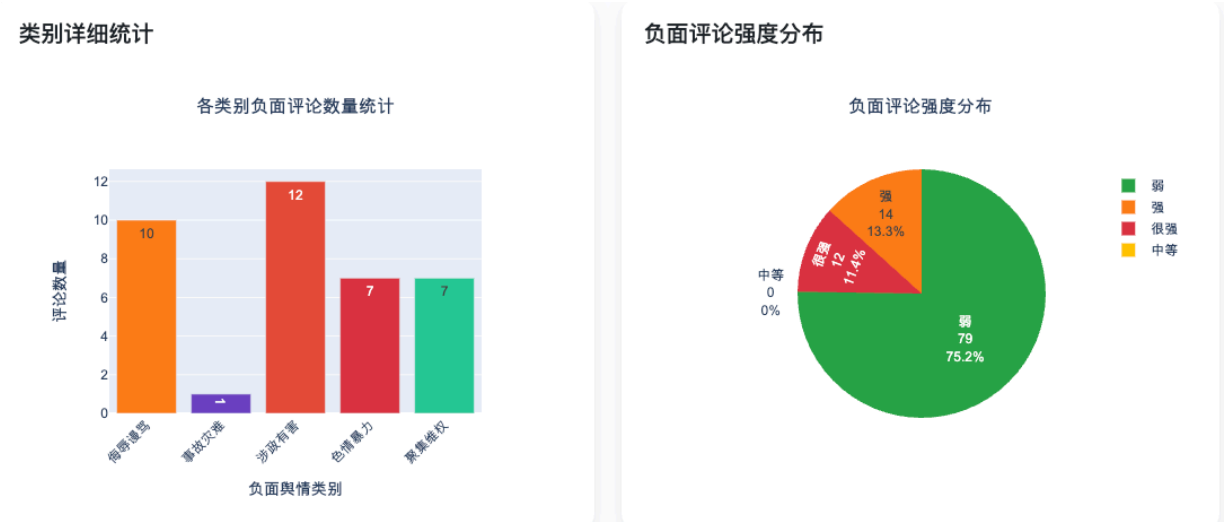
[星星眼]死亡飞车

侮辱谩骂　事故灾难

- 以列表形式展示筛选出的负面舆情文本及其分类。

- **可视化图表：**
  - 一个**饼图**，展示不同负面类别的分布情况。一个**柱状图**，展示不同负面类别的数量统计。





  - 词云图与关键词频率统计

- 最后有全部的负面文本合集

4. 实验结论与分析

本次实验成功构建了一个具备数据爬取、存储、分析和可视化能力的B站舆情分析系统。以B站用户进行测试，系统成功爬取了40条相关动态及600余条评论。

经过负面舆情分析模块处理，共识别出202条负面内容。分析结果如下：

按主题爬取

主题关键词

特朗普

爬取数量

10

▶ 开始爬取

按用户爬取

用户ID

73283747

用户ID可以在B站用户主页URL中找到

▶ 开始爬取

爬取状态

🎉 爬取完成！共处理 40 个视频 100%

查看分析结果

实时爬取结果

1 【MSI赛间单曲】《莫抽乡》⚡"陀螺，还得你当"~⚡他讲 "你也一样"~⚡

8 条负面评论

UP主: 往事在未来 ▶ 播放: 1.7万

发布: 2025/07/03 00:11 点赞: 126

处理时间: 2025/07/03 02:52

负面评论:

Doidoi丶 2025/07/03 02:08 5

寄、穷开摆这种名曲那可早了去了

往事在未来 2025/07/03 07:29 1

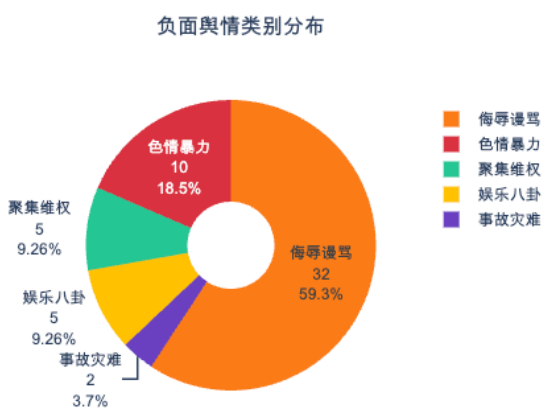
回复 @华语乐坛金曲榜 :6月都过完了，你的6月新歌速递（中）（下）呢？ 6月歌曲top10呢?

往事在未来 2025/07/03 07:30 0

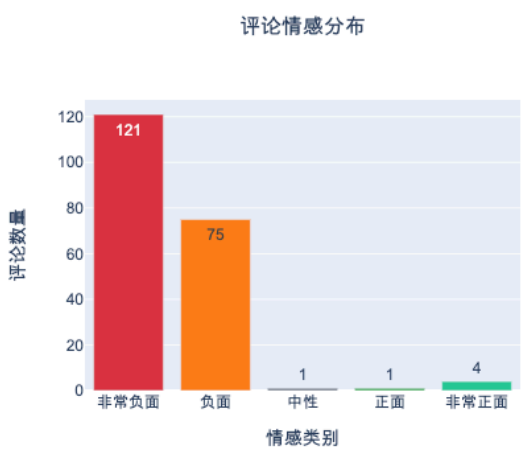
负面舆情分析结果

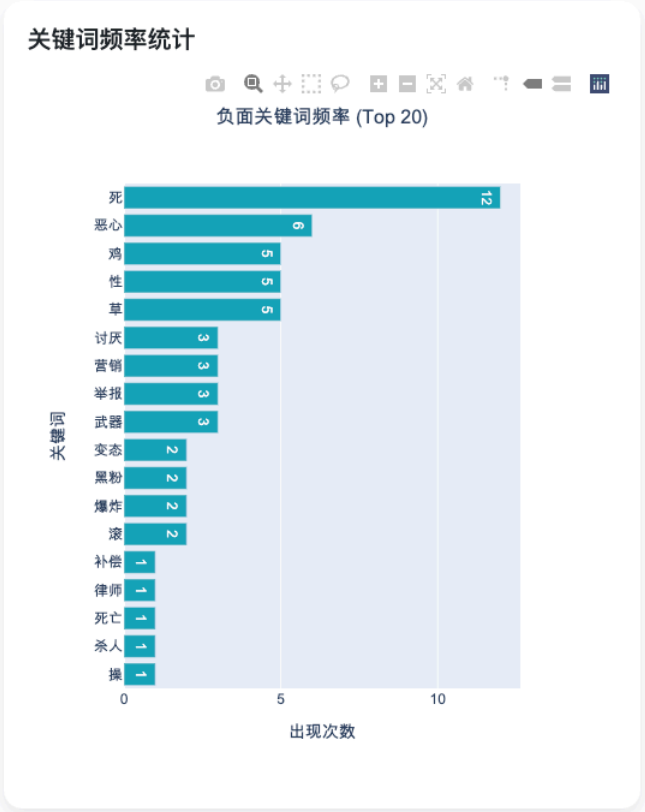
660	202	30.61%	5	18
总评论数	负面评论数	负面评论率	检测类别数	负面关键词数

负面舆情类别分布



情感分布统计





结果分析：

本次舆情分析仪表盘揭示了一个情绪激烈的网络舆论环境。在660条评论的样本中，高达30.61%（202条）的内容被判定为负面，这直接表明当前议题具有极大的争议性，并已引发了大规模的负面情绪。深入剖析负面内容的构成可以发现，舆论的核心并非理性的批评，而是以“侮辱谩骂”为主导的非理性情绪宣泄，该类别占比高达64.3%。情感分布统计进一步印证了这一点，其中“非常负面”的评论数量（120条）远超“负面”（75条），显示出大部分负面评论者的情绪已达到极点，讨论呈现出高度的对抗性。从关键词分析来看，“恶心”、“垃圾”、“举报”等攻击性词汇的高频出现，再次确认了舆论的焦点已从事件本身偏移至人身攻击和“站队”互撕；而“营销”、“黑粉”等词汇的出现，则暗示了背后可能存在有组织的舆论引导行为。值得注意的是，在202条负面评论中，仅有55条被成功归类，这说明大量负面言论以更隐晦的方式表达，虽未命中关键词，但其负面情绪不容忽视。

讨论：

本系统的分析结果为我们提供了一个观察特定网络事件舆论倾向的窗口。但需要注意的是，该结果具有局限性。首先，数据样本（100条动态）较小，不一定能完全代表整体舆论。其次，基于关键词和简单情感分析的判断仍可能存在误差。尽管如此，作为一次课程设计，本系统完整地实现了预设的全部功能，达到了实验目的。

5. 遇到的主要问题与解决方法总结

1. **B站反爬虫与动态加载：**
   - **问题：** 无法获取完整动态和评论，需要登录。
   - **解决：** 采用Selenium模拟浏览器行为，并通过加载本地Cookie缓存实现自动登录，有效解决了动态内容加载和身份验证问题。
2. **“楼中楼”数据层级关系存储：**
   - **问题：** 如何在数据库中表示B站评论与回复的父子关系。
   - **解决：** 在`comments`表中设计`parent_id`字段，巧妙地将层级结构扁平化存储到关系型数据库中，便于后续查询和重建。
3. **负面舆情误判：**
   - **问题：** 单纯的关键词匹配准确率低，容易误伤中性或正面言论。
   - **解决：** 引入`SnowNLP`进行情感分析作为二级校验，通过设定情感阈值过滤掉情感积极的“伪负面”文本，提升了系统判断的准确性。
4. **前后端数据交互：**
   - **问题：** 如何将后端Python生成的Pyecharts图表在前端HTML中展示。
   - **解决：** 利用Pyecharts将图表配置导出为JSON格式，后端通过Flask API将JSON数据传递给前端，前端再使用JavaScript的Echarts库进行渲染，实现了前后端的解耦和动态展示。

6. 心得体会与展望

【d】有什么收获和心得体会：

- **技术层面：** 我不仅巩固了Python基础语法，还结合大模型从零到一地实践了Web爬虫、数据库设计、Web开发和数据可视化等多个领域的知识，并将它们有机地整合到一个项目中。我学会了如何分析和解决开发过程中遇到的实际问题，例如如何调试B站的爬虫、优化数据库查询、设计API接口等。
- **工程层面：** 我对软件开发的整体流程有了更深刻的理解。从最初的需求分析，到模块化的系统设计，再到编码实现和最终测试，每一步都至关重要。良好的代码规范和项目结构（如本项目中的`backend`和`frontend`目录划分）对于项目的可维护性至关重要。
- **思维层面：** 我认识到技术是为解决问题服务的。在面对“负面舆情识别”这个复杂问题时，我没有盲目追求最尖端的人工智能模型，而是从实际需求和自身能力出发，选择了“规则+简单算法”的务实方案，并清晰地认识到其优点和局限性。这种在理想与现实之间寻找平衡的工程思维，我认为是本次实验最大的收获之一。

未来展望：

本系统虽然功能完整，但仍有许多可以改进和扩展的空间：

1. **提升爬虫性能：** 目前使用Selenium单线程爬取，速度较慢。未来可以研究结合B站的API接口进行爬取，或者使用`asyncio`配合`aiohttp`进行异步爬取，大幅提升数据采集效率。
2. **优化分析模型：** 引入更先进的NLP模型（如基于Transformer的BERT等）进行文本分类和情感分析，可以显著提高舆情判断的准确率，尤其能更好地理解B站用户独特的语言风格（如“梗”和“黑话”），但可能时间成本较高，分析将会消耗大量时间。
3. **增强可视化交互：** 开发功能更丰富的交互式仪表盘，用户可以自定义筛选时间范围、查看舆情趋势变化、下钻到具体动态等。
4. **部署与应用：** 将应用使用Docker打包，并部署到云服务器上，实现7x24小时不间断的舆情监控服务。