****

**应用基础实践二**

**实验报告**

|  |  |
| --- | --- |
| 学生姓名 | 张子洋 |
| 学 号 | 808221223 |
| 专业班级 | 计科2203班 |
| 指导教师 | 杨希 |
| 学 院 | 计算机学院 |
| 完成时间 | 2024.12.22 |

计算机学院

2024年12月

目录

[一、问题描述 3](#_Toc11321)

[二、需求分析 4](#_Toc4681)

[2.1 功能需求 4](#_Toc26066)

[2.2 非功能需求 4](#_Toc14906)

[三、设计思想 5](#_Toc28525)

[3.1 前后端分离 5](#_Toc6718)

[3.2 权限控制 5](#_Toc21887)

[3.3 数据流与状态管理 5](#_Toc24533)

[3.4 性能优化 5](#_Toc32500)

[四、概要设计 6](#_Toc20448)

[五、详细设计 7](#_Toc22579)

[5.1 环境配置 7](#_Toc31783)

[5.2 技术栈 7](#_Toc3825)

[5.2.1 前端： 7](#_Toc31317)

[5.2.2 后端： 7](#_Toc11728)

[5.2.3 开发工具： 7](#_Toc20380)

[5.3 Token 流程 8](#_Toc7659)

[5.4 功能模块 8](#_Toc7832)

[5.4.1 注册 8](#_Toc1460)

[5.4.2 登录 11](#_Toc20779)

[5.4.3 发微博 12](#_Toc30672)

[5.4.4 查看微博 13](#_Toc1647)

[5.4.5 删除 14](#_Toc14813)

[5.4.6 点赞 15](#_Toc3303)

[5.4.7 修改信息 17](#_Toc27194)

[六、 调试与结果 21](#_Toc13213)

[6.1 调试 21](#_Toc17744)

[6.2 结果 21](#_Toc20689)

[七、 心得 22](#_Toc3568)

[7.1 前后端分离架构的价值 22](#_Toc4953)

[7.2 用户权限管理的重要性 22](#_Toc31163)

[7.3 数据交互与实时性优化 22](#_Toc7153)

[7.4 业务逻辑的复杂性 22](#_Toc18864)

[7.5 界面设计与用户体验 22](#_Toc1536)

[7.6 总结 23](#_Toc14425)

[附录 24](#_Toc933)

[参考文献 24](#_Toc29228)

[源代码 24](#_Toc3847)

# 一、问题描述

微博类社交平台在现代互联网中占据重要地位，核心挑战在于如何通过前后端分离的方式实现常见的微博功能，如用户认证、微博管理、评论与点赞等，并确保系统安全性、性能优化和代码的可维护性。

1. **权限管理**：如何保护敏感操作，防止未经授权的用户访问关键资源或滥用功能。
2. **数据管理**：如何高效存储和处理用户数据、微博内容以及相关的点赞和评论记录。
3. **性能优化**：如何提升系统的响应速度和用户体验，特别是在加载微博列表或动态更新内容时。
4. **前后端分离**：如何清晰划分前端与后端的职责，确保代码的维护性与扩展性。
5. **功能实现**：如何在一个小型项目中实现微博的核心功能，如用户登录、注册、发帖、删帖、点赞和评论等。

# 二、需求分析

## 2.1 功能需求

**① 用户模块**

* 用户可以删除自己的微博。注册、登录、登出功能。
* 登录后提供 JWT Token，后续请求通过 Token 验证用户身份。
* 修改个人信息，如用户名、头像等。

**② 微博模块**

* 用户可以发布微博，内容包括文本、图片或视频。
* 微博列表按时间倒序显示，支持分页加载。

**③ 评论模块**

* 用户可以对微博发表评论，评论内容为文本格式。
* 用户可以删除自己的评论。

**④ 点赞模块**

* 用户可以点赞或取消点赞微博，系统统计每条微博的点赞数。

## 2.2 非功能需求

**① 安全性**

* 使用加密算法存储用户密码（如 bcrypt）。
* 前端通过 HTTPS 与后端通信，防止数据在传输过程中被篡改。

**② 性能**

* 微博图片或视频内容按需加载（懒加载）。
* 微博列表分页展示，避免一次性加载过多数据。

**③ 可维护性**

* 使用 TypeScript 提高代码类型安全性，降低潜在的运行时错误。
* 前端组件化设计，后端模块化开发，提升可读性和扩展性。

# 三、设计思想

## 3.1 前后端分离

* **前端职责**：通过 React 构建用户界面，处理与用户的交互，发起 HTTP 请求获取或提交数据。
* **后端职责**：提供 RESTful API 接口，处理业务逻辑，操作数据库并返回 JSON 格式的数据响应。

## 3.2 权限控制

* 使用 React Router 实现路由守卫，拦截未登录用户的访问请求。
* 通过 JWT 实现用户登录状态的无状态管理，确保 API 的安全访问。

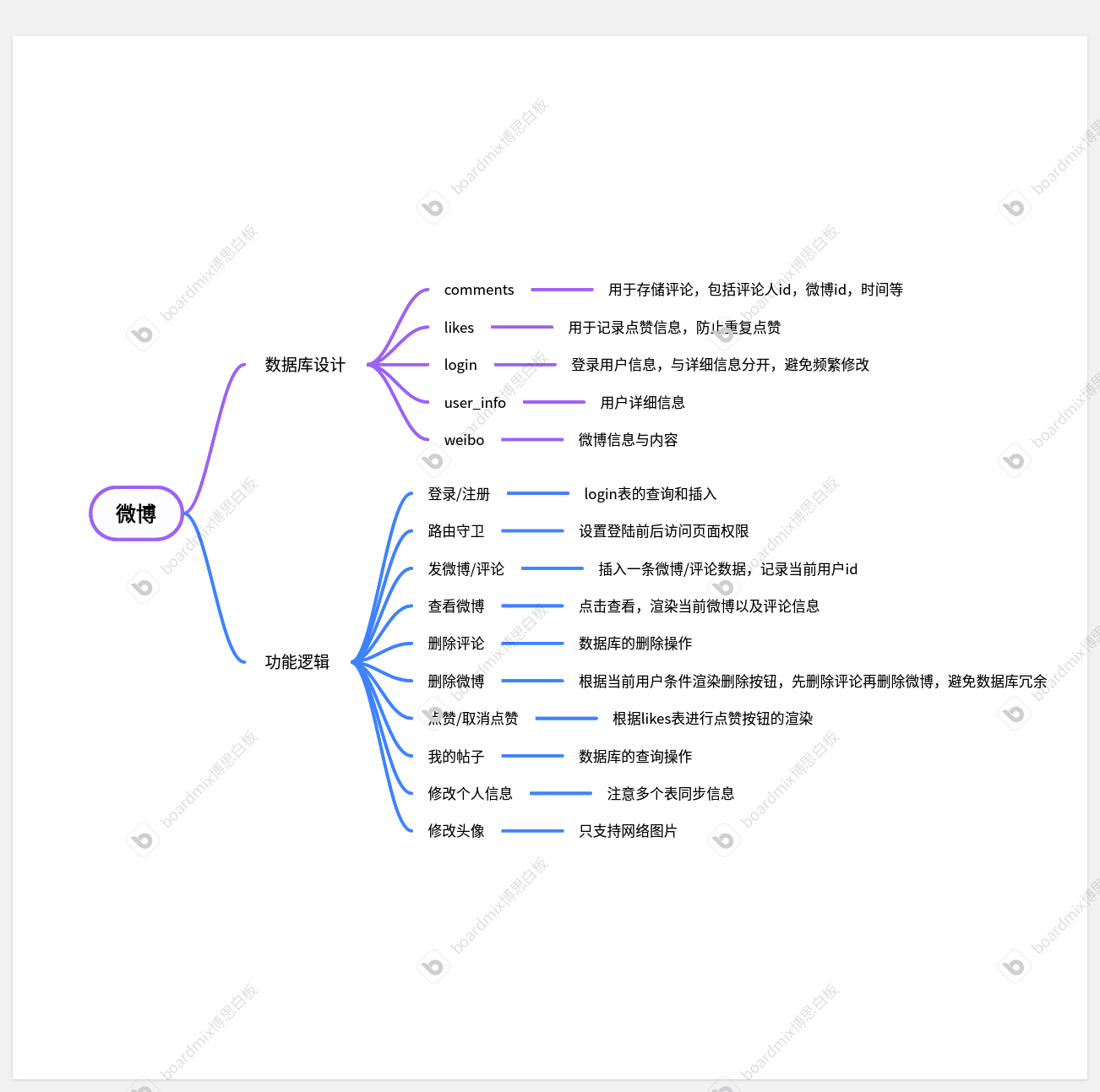
## 3.3 数据流与状态管理

* 使用 React 的 useState 和 useEffect 管理页面状态和数据流动。
* 通过组件间的数据传递实现动态更新，例如评论区的局部刷新。

## 3.4 性能优化

* 前端采用懒加载技术按需加载资源。
* 后端通过分页加载和数据库索引优化查询性能。

# 四、概要设计



# 五、详细设计

## 5.1 环境配置

创建react项目

npm install antd

npm install axios express jwt-simple

npm install axios react-navigation react-navigation-stack react-native-gesture-handler react-native-reanimated

npm install react-router-dom

npm install --save-dev @types/react-router-dom

npm install jsonwebtoken

## 5.2 技术栈

### 5.2.1 前端：

* **React**：用于构建用户界面，利用其组件化特性实现微博应用的动态页面。
* **TypeScript**：为 React 项目添加静态类型检查，提高代码的可维护性和可读性。
* **CSS**：用于页面的样式和布局。
* **Axios**：用于与后端 API 进行通信，处理 HTTP 请求。
* **Ant design：**一个ui组件库，用于界面设计。
* **Vite：**用于react项目的搭建。

### 5.2.2 后端：

* **Node.js**：作为服务器端环境，使用 Express 框架提供 API。
* **Express**：用于搭建 API，处理请求和响应。
* **MySQL**：作为关系型数据库管理系统，存储用户数据和微博内容。

### 5.2.3 开发工具：

* **Visual Studio Code**：作为开发 IDE，支持 React 和 Node.js 的开发。
* **Navicat：**数据库可视化工具。

## ****5.3 Token 流程****

* **生成 Token**：后端在登录成功时，用 JWT 为用户生成一个 Token 并返回给前端。
* **存储 Token**：前端将 Token 存储在 localStorage 中，便于后续使用。
* **携带 Token**：前端在每次请求受保护的资源时，从 localStorage 中读取 Token，并在请求头中携带。
* **验证 Token**：后端通过中间件解析请求中的 Token，验证其有效性，并将用户信息注入到请求对象中。
* **访问保护资源**：通过前端路由守卫和后端中间件的双重保护，确保只有登录状态下的用户可以访问相关页面和数据。

## 5.4 功能模块

### 5.4.1 注册

通过表单得到数据，对login表进行插入。

后端接口：

app.post('/register', (req, res) => {

    const { username, password, profile\_image\_url, nickname } = req.body;

    // 检查用户名是否已存在

    const checkUsernameSql = 'SELECT \* FROM login WHERE username = ?';

    db.query(checkUsernameSql, [username], (err, result) => {

      if (err) {

        console.error('查询用户名时发生错误:', err);

        return res.status(500).json({ message: '注册失败，请稍后再试' });

      }

      if (result.length > 0) {

        return res.status(400).json({ message: '用户名已存在' });

      }

      // 插入用户信息

      const insertUserInfoSql = 'INSERT INTO login (username, password) VALUES (?, ?)';

      db.query(insertUserInfoSql, [username, password], (err, result) => {

        if (err) {

          console.error('注册用户时发生错误:', err);

          return res.status(500).json({ message: '注册失败，请稍后再试' });

        }

        // 插入用户信息

        const insertWeiboSql = 'INSERT INTO user\_info (id, nickname, profile\_image\_url) VALUES (?, ?, ?)';

        db.query(insertWeiboSql, [username, nickname, profile\_image\_url], (err, result) => {

          if (err) {

            console.error('插入微博数据时发生错误:', err);

            return res.status(500).json({ message: '微博数据插入失败' });

          }

          res.json({ success: true, message: '注册成功' });

        });

      });

    });

  });

前端代码：

import React, {  useState } from 'react';

import { Form, Input, Button, message, FloatButton, } from 'antd';

import { UserOutlined, LockOutlined, LeftOutlined } from '@ant-design/icons';

import './index.css'; // 引入样式文件

import headPicture from '../../assets/head\_picture.png'; // 正确引入图片

import { useNavigate } from 'react-router-dom';

const LoginPage: React.FC = () => {

  const [loading, setLoading] = useState<boolean>(false); // 显示加载状态

  const navigate = useNavigate(); // 用于路由跳转

  // 异步登录处理

  const handleLogin = async (values: any) => {

    const { username, password } = values; // 解构表单数据

    setLoading(true); // 开启加载状态

    try {

      // 模拟向后端发送请求

      const response = await fetch('http://localhost:5000/login', {

        method: 'POST',

        headers: {

          'Content-Type': 'application/json',

        },

        body: JSON.stringify({ username, password }), // 发送数据

      });

      const data = await response.json(); // 解析响应

      if (data.success) {

        message.success('登录成功'); // 成功提示

        localStorage.setItem('token', data.token); // 存储 Token

        navigate('/explore'); // 跳转到首页

      } else {

        message.error(data.message || '用户名或密码错误'); // 失败提示

      }

    } catch (error) {

      message.error('网络请求失败，请稍后再试'); // 网络错误提示

    } finally {

      setLoading(false); // 请求结束，关闭加载状态

    }

  };

  return (

    <div className="login-container">

            {/\* 后退按钮 \*/}

      <FloatButton

        icon={<LeftOutlined />}

        onClick={() => navigate(-1)} // 返回上一页

        className="back-button"

      >

      </FloatButton>

      <h1 className="welcome-title">🎉欢迎来到微博🌈</h1>

      <img src={headPicture} alt="logo" className="logo" />

      <div className="login-desc">

      <Form

        name="login\_form"

        onFinish={handleLogin} // 表单提交事件

        initialValues={{ remember: true }}

        autoComplete="off"

      >

        <Form.Item

          name="username"

          rules={[{ required: true, message: '请输入用户名' }]}

        >

          <Input

            prefix={<UserOutlined className="site-form-item-icon" />}

            placeholder="用户名"

          />

        </Form.Item>

        <Form.Item

          name="password"

          rules={[{ required: true, message: '请输入密码' }]}

        >

          <Input

            prefix={<LockOutlined className="site-form-item-icon" />}

            type="password"

            placeholder="密码"

          />

        </Form.Item>

        <Form.Item>

          <Button

            type="primary"

            htmlType="submit"

            color="default"

            variant="solid"

            block

            loading={loading} // 显示加载状态

          >

            登录

          </Button>

        </Form.Item>

      </Form>

      </div>

    </div>

  );

};

export default LoginPage;

### 5.4.2 登录

// 登录请求处理

app.post('/login', (req, res) => {

  const { username, password } = req.body;

  if (!username || !password) {

    return res.status(400).json({ success: false, message: '用户名和密码是必填的' });

  }

  const query = 'SELECT \* FROM login WHERE username = ?';

  db.query(query, [username], (err, results) => {

    if (err) {

      console.error('查询失败:', err);

      return res.status(500).json({ success: false, message: '服务器错误' });

    }

    if (results.length === 0) {

      return res.status(400).json({ success: false, message: '用户名或密码错误' });

    }

    const user = results[0];

    if (password === user.password) {

      // 生成 JWT

      const token = jwt.sign({ id: user.id, username: user.username }, SECRET\_KEY, {

        expiresIn: '5h', // token 有效期 5 小时

      });

      return res.json({ success: true, message: '登录成功', token });

    } else {

      return res.status(400).json({ success: false, message: '用户名或密码错误' });

    }

  });

});

### 5.4.3 发微博

// 发布微博接口

app.post('/posts', (req, res) => {

  const { publisher\_id, publish\_content } = req.body;

  // 查询用户信息

  const queryUserInfo = `

    SELECT nickname, profile\_image\_url

    FROM user\_info

    WHERE id = ?`;

  db.query(queryUserInfo, [publisher\_id], (err, result) => {

    if (err) return res.status(500).json({ message: 'Database error', error: err });

    if (result.length === 0) {

      return res.status(404).json({ message: 'User not found' });

    }

    const { nickname, profile\_image\_url } = result[0];

    // 插入微博信息

    const insertWeibo = `

      INSERT INTO weibo (publisher\_id, publisher\_nickname, publisher\_avatar\_url, publish\_content, publish\_time)

      VALUES (?, ?, ?, ?, NOW())`;

    db.query(insertWeibo, [publisher\_id, nickname, profile\_image\_url, publish\_content], (err) => {

      if (err) return res.status(500).json({ message: 'Database error', error: err });

      res.status(200).json({ message: 'Weibo posted successfully' });

    });

  });

});

### 5.4.4 查看微博

我们获取当前的信息，根据微博id获取评论，在评论区显示。

// 获取指定微博的详细信息

app.get('/weibo/:id', (req, res) => {

  const weiboId = req.params.id;

  if (!weiboId) {

    return res.status(400).json({ error: '微博 ID 不能为空' });

  }

  // 获取微博详细内容

  const query = `

    SELECT

      weibo\_id,

      publisher\_id,

      publisher\_nickname,

      publisher\_avatar\_url,

      like\_count,

      publish\_content,

      publish\_time

    FROM weibo

    WHERE weibo\_id = ?

    ORDER BY publish\_time DESC`;

  db.query(query, [weiboId], (err, weiboResults) => {

    if (err) {

      console.error('查询微博数据失败:', err);

      return res.status(500).json({ error: '无法获取微博数据' });

    }

    if (weiboResults.length === 0) {

      return res.status(404).json({ error: '微博不存在' });

    }

    const weibo = weiboResults[0];

    weibo.publish\_time = formatTime(weibo.publish\_time);

    // 获取该微博的所有评论

    const commentsQuery = `

      SELECT

        comment\_id,

        commenter\_id,

        commenter\_nickname,

        commenter\_avatar\_url,

        like\_count,

        comment\_content,

        comment\_time

      FROM comments

      WHERE weibo\_id = ?`;

    db.query(commentsQuery, [weiboId], (err, commentsResults) => {

      if (err) {

        console.error('查询评论数据失败:', err);

        return res.status(500).json({ error: '无法获取评论数据' });

      }

      const formattedComments = commentsResults.map(comment => ({

        ...comment,

        comment\_time: formatTime(comment.comment\_time)

      }));

      res.json({ weibo, comments: formattedComments });

    });

  });

});

### 5.4.5 删除

根据当前用户token，条件渲染删除按钮，进行权限控制。

app.delete('/delete/comments/:comment\_id', (req, res) => {

  const token = req.headers['authorization'];

  if (!token) {

    return res.status(400).send('缺少 token');

  }

  const tokenParts = token.split(' ');

  if (tokenParts[0] !== 'Bearer' || !tokenParts[1]) {

      return res.status(400).send('无效的 token 格式');

  }

  const username = tokenParts[1];

  const comment\_id = req.params.comment\_id;

  const query = 'DELETE FROM comments WHERE comment\_id = ? AND commenter\_id = ?';

  db.query(query, [comment\_id, username], (err) => {

    if (err) {

      console.error('删除评论失败:', err);

      return res.status(500).json({ error: '删除评论失败' });

    }

    res.status(200).json({ message: '删除评论成功' });

  });

});

app.delete('/delete/weibo/:weibo\_id', (req, res) => {

  const token = req.headers['authorization'];

  if (!token) {

    return res.status(400).send('缺少 token');

  }

  const tokenParts = token.split(' ');

  if (tokenParts[0] !== 'Bearer' || !tokenParts[1]) {

    return res.status(400).send('无效的 token 格式');

  }

  const username = tokenParts[1];

  const weibo\_id = req.params.weibo\_id;

  const deleteCommentsQuery = 'DELETE FROM comments WHERE weibo\_id = ?';

  db.query(deleteCommentsQuery, [weibo\_id], (err) => {

    if (err) {

      console.error('删除评论失败:', err);

      return res.status(500).json({ error: '删除评论失败' });

    }

    const deleteWeiboQuery = 'DELETE FROM weibo WHERE weibo\_id = ? AND publisher\_id = ?';

    db.query(deleteWeiboQuery, [weibo\_id, username], (err) => {

      if (err) {

        console.error('删除微博失败:', err);

        return res.status(500).json({ error: '删除微博失败' });

      }

      res.status(200).json({ message: '删除微博及其评论成功' });

    });

  });

});

### 5.4.6 点赞

这个需要创建一个likes表，确保不出现重复点赞。

// 判断用户是否点赞了微博

app.get('/weibo/:weibo\_id/judgeLiked', (req, res) => {

  const { weibo\_id } = req.params;

  const token = req.headers['authorization'];

  if (!token) {

    return res.status(400).send('缺少 token');

  }

  const userId = token.split(' ')[1]; // 从 Authorization header 中提取 token 部分

  const checkQuery = 'SELECT \* FROM likes WHERE user\_id = ? AND target\_id = ? AND target\_type = "post"';

  db.query(checkQuery, [userId, weibo\_id], (err, results) => {

    if (err) {

      console.error('查询点赞状态时发生错误:', err);

      return res.status(500).json({ success: false, message: '服务器错误' });

    }

    const isLiked = results.length > 0;

    res.json({ success: true, isLiked });

  });

});

// 判断用户是否点赞了评论

app.get('/weibo/:id/comments/:comment\_id/judgeLiked', (req, res) => {

  const { id,comment\_id } = req.params;

  const token = req.headers['authorization'];

  if (!token) {

    return res.status(400).send('缺少 token');

  }

  const userId = token.split(' ')[1]; // 从 Authorization header 中提取 token 部分

  const checkQuery = 'SELECT \* FROM likes WHERE user\_id = ? AND target\_id = ? AND target\_type = "comment"';

  db.query(checkQuery, [userId, comment\_id], (err, results) => {

    if (err) {

      console.error('查询点赞状态时发生错误:', err);

      return res.status(500).json({ success: false, message: '服务器错误' });

    }

    const isLiked = results.length > 0;

    res.json({ success: true, isLiked });

  });

});

### 5.4.7 修改信息

这个也很简单，只需要注意修改相关表就可以了。

// 修改 `login` 表中根据 username 修改 password 的接口

app.put('/update-password', (req, res) => {

  const { username, newPassword } = req.body;

  if (!username || !newPassword) {

    return res.status(400).json({ message: '用户名和新密码不能为空' });

  }

  const sql = 'UPDATE login SET password = ? WHERE username = ?';

  db.query(sql, [newPassword, username], (err, result) => {

    if (err) {

      console.error('更新密码时发生错误:', err);

      return res.status(500).json({ message: '更新密码失败' });

    }

    if (result.affectedRows === 0) {

      return res.status(404).json({ message: '用户不存在' });

    }

    res.json({ message: '密码更新成功' });

  });

});

// 修改 `user\_info` 表中根据 id 修改 profile\_image\_url 的接口

app.put('/update-profile-image', (req, res) => {

  const { id, profileImageUrl } = req.body;

  console.log(id, profileImageUrl);

  if (!id || !profileImageUrl) {

    return res.status(400).json({ message: 'ID 和头像 URL 不能为空' });

  }

  // Begin transaction to ensure all updates happen atomically

  db.beginTransaction((err) => {

    if (err) {

      console.error('启动事务失败:', err);

      return res.status(500).json({ message: '更新头像事务启动失败' });

    }

    // Update user\_info table

    const updateUserInfoSql = 'UPDATE user\_info SET profile\_image\_url = ? WHERE id = ?';

    db.query(updateUserInfoSql, [profileImageUrl, id], (err, result) => {

      if (err) {

        return db.rollback(() => {

          console.error('更新 user\_info 失败:', err);

          return res.status(500).json({ message: '更新用户头像失败' });

        });

      }

      if (result.affectedRows === 0) {

        return db.rollback(() => {

          return res.status(404).json({ message: '用户不存在' });

        });

      }

      // Update profile\_image\_url in weibo table

      const updateWeiboSql = 'UPDATE weibo SET publisher\_avatar\_url = ? WHERE publisher\_id = ?';

      db.query(updateWeiboSql, [profileImageUrl, id], (err, result) => {

        if (err) {

          return db.rollback(() => {

            console.error('更新 weibo 表失败:', err);

            return res.status(500).json({ message: '更新微博失败' });

          });

        }

        // Update profile\_image\_url in comments table

        const updateCommentsSql = 'UPDATE comments SET commenter\_avatar\_url = ? WHERE commenter\_id = ?';

        db.query(updateCommentsSql, [profileImageUrl, id], (err, result) => {

          if (err) {

            return db.rollback(() => {

              console.error('更新 comments 表失败:', err);

              return res.status(500).json({ message: '更新评论失败' });

            });

          }

          // Commit the transaction if all updates are successful

          db.commit((err) => {

            if (err) {

              return db.rollback(() => {

                console.error('提交事务失败:', err);

                return res.status(500).json({ message: '提交事务失败' });

              });

            }

            res.json({ message: '用户头像更新成功' });

          });

        });

      });

    });

  });

});

// 修改 `user\_info` 表中根据 id 修改 nickname 的接口

app.put('/update-nickname', (req, res) => {

  const { id, nickname } = req.body;

  if (!id || !nickname) {

    return res.status(400).json({ message: 'ID 和昵称不能为空' });

  }

  // Begin transaction to ensure all updates happen atomically

  db.beginTransaction((err) => {

    if (err) {

      console.error('启动事务失败:', err);

      return res.status(500).json({ message: '更新昵称事务启动失败' });

    }

    // Update nickname in user\_info table

    const updateUserInfoSql = 'UPDATE user\_info SET nickname = ? WHERE id = ?';

    db.query(updateUserInfoSql, [nickname, id], (err, result) => {

      if (err) {

        return db.rollback(() => {

          console.error('更新 user\_info 失败:', err);

          return res.status(500).json({ message: '更新用户昵称失败' });

        });

      }

      if (result.affectedRows === 0) {

        return db.rollback(() => {

          return res.status(404).json({ message: '用户不存在' });

        });

      }

      // Update nickname in weibo table

      const updateWeiboSql = 'UPDATE weibo SET publisher\_nickname = ? WHERE publisher\_id = ?';

      db.query(updateWeiboSql, [nickname, id], (err, result) => {

        if (err) {

          return db.rollback(() => {

            console.error('更新 weibo 表失败:', err);

            return res.status(500).json({ message: '更新微博昵称失败' });

          });

        }

        // Update nickname in comments table

        const updateCommentsSql = 'UPDATE comments SET commenter\_nickname = ? WHERE commenter\_id = ?';

        db.query(updateCommentsSql, [nickname, id], (err, result) => {

          if (err) {

            return db.rollback(() => {

              console.error('更新 comments 表失败:', err);

              return res.status(500).json({ message: '更新评论昵称失败' });

            });

          }

          // Commit the transaction if all updates are successful

          db.commit((err) => {

            if (err) {

              return db.rollback(() => {

                console.error('提交事务失败:', err);

                return res.status(500).json({ message: '提交事务失败' });

              });

            }

            res.json({ message: '用户昵称更新成功' });

          });

        });

      });

    });

  });

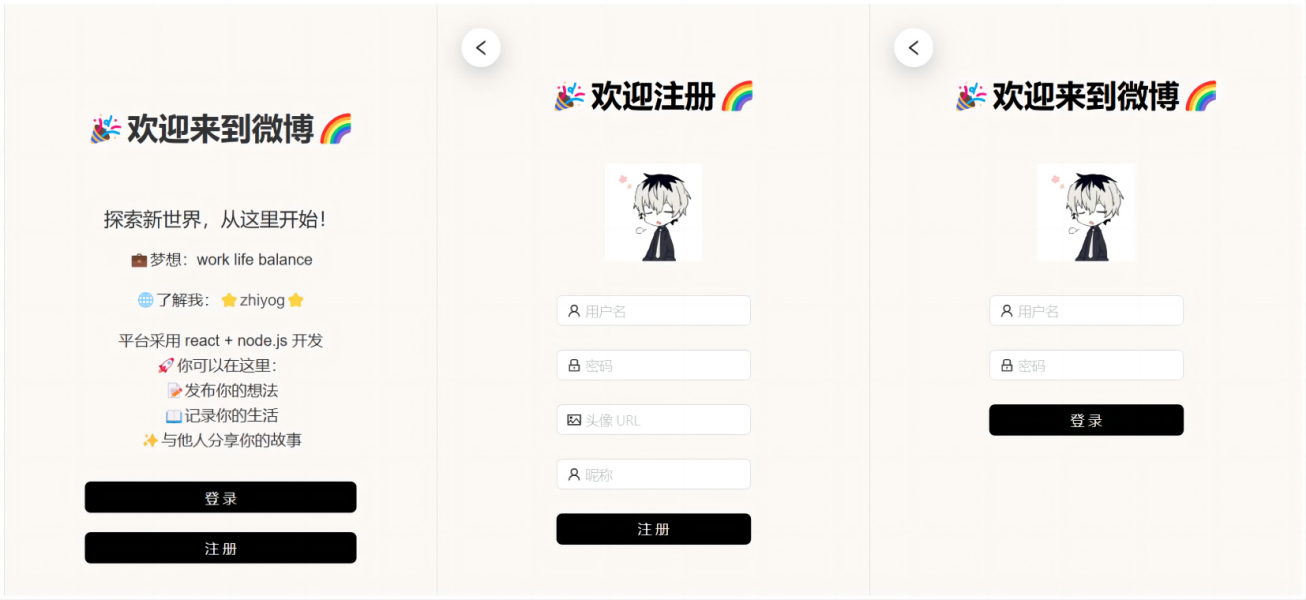
});

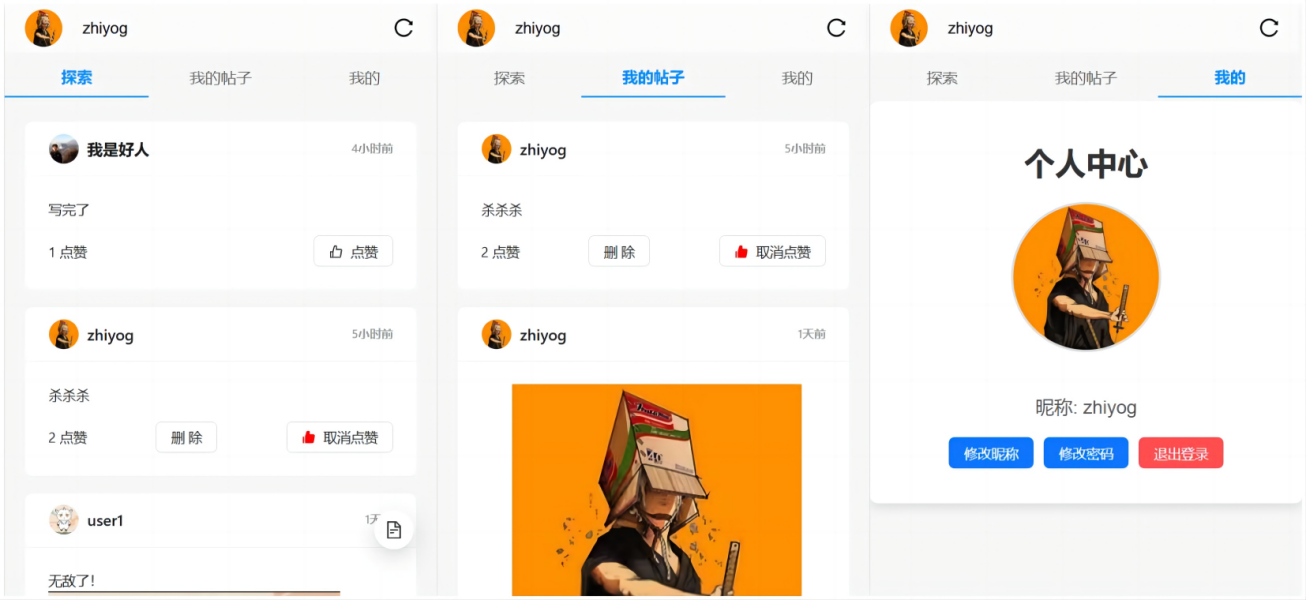
# 调试与结果

## 6.1 调试

1. 先启动server.js文件，连接数据库；
2. 再使用命令 npm run dev 运行前端项目。

## 6.2 结果





# 心得

## ****7.1 前后端分离架构的价值****

微博系统采用前后端分离的方式，前端负责用户交互和界面展示，后端负责业务逻辑和数据处理。

* **提升开发效率**：前后端独立开发、调试，互不干扰，效率显著提升。
* **模块化设计**：前端使用 React 组件化开发，后端使用 Express 路由设计，系统结构清晰，代码复用率高。

## ****7.2 用户权限管理的重要性****

在微博系统中，用户权限是核心功能之一，例如：

* **路由守卫**：通过 React Router 配合 JWT 实现权限控制，保证未登录用户无法访问受限页面。
* **动态操作校验**：如删除、点赞等操作，实时校验用户权限，防止滥用或非法操作。

这种细粒度的权限管理不仅提升了系统安全性，也让用户体验更加完善。

## 7.3 数据交互与实时性优化

微博系统的数据交互频繁，尤其是评论区和点赞功能。

* **局部刷新**：通过 React 的 useEffect 和条件渲染，实现页面局部刷新，避免整个页面重新加载，提升用户体验。
* **性能优化**：对图片内容使用懒加载技术，按需加载数据，有效减少了首屏加载时间，提高页面响应速度。

## ****7.4 业务逻辑的复杂性****

微博的核心功能（如发帖、评论、点赞）涉及多个模块的协作：

* **数据库设计**：使用 MySQL 存储微博、用户信息和交互数据（如点赞记录、评论内容等），合理的表结构设计是数据高效管理的关键。
* **事务管理**：在涉及多表操作时，通过事务确保数据一致性，避免部分成功、部分失败的情况。

## ****7.5 界面设计与用户体验****

一个优秀的微博系统，不仅需要实现功能，还需要用户操作顺畅、界面美观。

* **UI 设计**：使用 Ant Design 提供的组件快速构建响应式界面，既提高了开发效率，也提升了用户的视觉体验。
* **细节优化**：比如输入框的实时校验、操作提示信息、异步加载时的加载动画等，细节决定了用户的满意度。

## ****7.6 总结****

开发微博系统让我深入理解了如何从用户需求出发，结合技术实现高质量的功能。同时，这个项目让我认识到协作的重要性：前后端的良好配合、与数据库设计的紧密结合、对用户体验的关注，这些都让整个系统更加完善。

每一个小细节的改进，最终都会积累成系统的整体提升。微博系统不仅是一个技术实践，更是一场设计与实现的全方位考验！

# 附录

## 参考文献

一个 Vue 微博客户端 <https://github.com/werbhelius/Pixel-Web>

React官方中文文档 <https://react.docschina.org/>

Ant design 组件库 <https://ant-design.antgroup.com/index-cn>

切图仔做全栈：React&Nest.js社区平台(二)——👋手把手实现优雅的鉴

权机制 <https://juejin.cn/post/7345379924167262249>

## 源代码

由于代码过多，直接放在代码文件中了。

**注：node\_modules文件中的依赖项太多了，压缩不方便，我就删去了，只保留了源代码和版本依赖管理。**