JavaScript 语言与 Web 程序设计

项目结题报告

金桥通 Web 移植

汪之立 2200012966

一、项目概述

本项目基于本人参与的另一个已完成项目——金桥通。主要改写了客户端的部分。将原本应用于 Android 平台的客户端迁移至 Web 端，扩展用户群体。

（一）项目目的

金桥通网页移植版本旨在将现有的金桥通移动应用移植到网页平台上，以扩大用户覆盖面，提供更便捷的访问方式，并提高用户的使用体验。通过将金桥通的部分功能迁移到网页端，我希望能够吸引更多的用户，特别是那些更偏好使用桌面设备进行操作的用户。

（二）项目概览

金桥通是一个结合了大模型的基于 P2P 借贷平台的项目，旨在为借款人和贷款人提供一个安全、便捷的在线交流和交易平台。该项目包括以下主要功能：

功能	描述
用户注册与登录	用户可以进行注册和登录，享受平台提供的所有服务。
实时聊天	用户可以与顾问和大模型进行实时聊天，获取即时帮助和信息。
信用评分	平台将通过数据分析为用户提供准确的信用评分，帮助贷款人评估借款人的信用风险。
贷款申请	借款人可以通过网页端提交贷款申请，简化了借贷流程。
投资机会	贷款人可以浏览并选择投资机会，通过平台进行资金投资。

（表一：金桥通项目主要功能）

这次的 Web 移植包括了上述全部的功能，并且考虑屏幕的变化重新设计了界面，争取通过优雅的用户界面，为用户提供更加简洁、高效、可靠的服务。

（三）技术概览

服务端使用的是原本项目基于 Rust 的服务端，采用 TCP 协议进行通信，具体内容参见原项目文档。

为适应服务端现有协议和 Web 端常用 http 协议不一致的问题，本项目除了移植的 Web 版本客户端，还有一个代理服务器用于转发接收的请求，同时实现 session 认证等问题。本项目基于 Javascript 语言实现，现阶段项目主要的 JavaScript 代码在 3500 行左右。

代理服务器部分

语言	文件数	代码行数	注释行数	空行数	总计
JSON	3	833	0	3	836
JavaScript	1	167	18	29	214

（表二：代理服务器部分代码统计）

代理服务器使用 express 框架实现，提供 HTTP 与 TCP 服务器之间的通信桥梁。它能够处理客户端请求，将其转换为 TCP 请求并发送给后端服务器，同时通过 express-session 管理会话和连接的生命周期，确保可靠性和安全性。通过不同的 API 端口，实现了对于不同请求的一致对外接口。

Web 端部分

语言	文件数	代码行数	注释行数	空行数	总计
JavaScript	32	2,980	44	289	3,313
CSS	19	1,300	0	174	1,474
Markdown	1	38	0	33	71
HTML	1	20	23	1	44

（表三：Web 服务端代码统计）

Web 客户端部分使用 React.js 框架。在迁移原有客户端功能的同时，通过重构 UI 界面，进一步增加动态效果等方式力图给用户提供更好的使用体验。

（四）项目代码

原项目金桥通代码仓库：

https://github.com/2668940140/GBridge

本项目 Web 移植代码仓库：

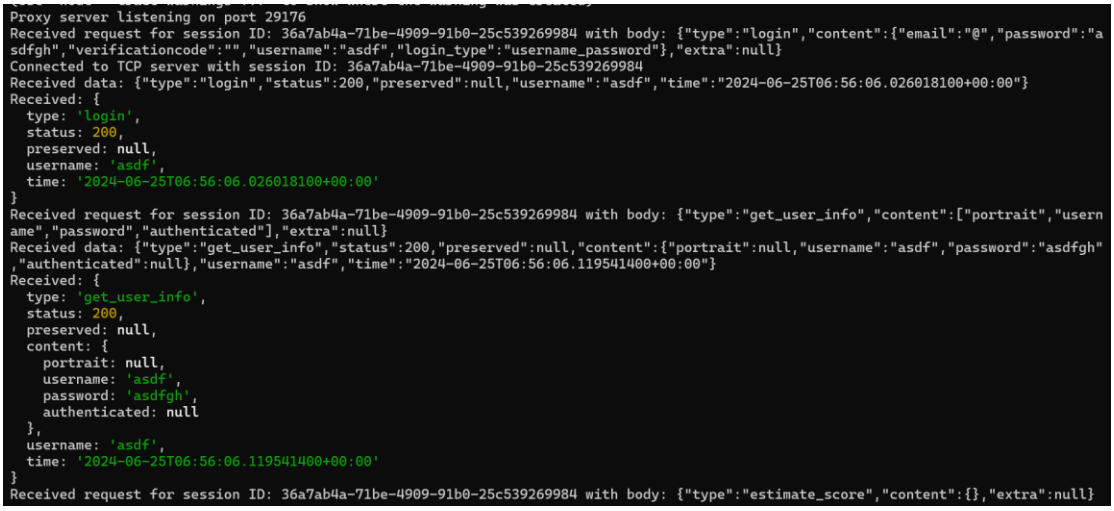
https://github.com/Litchi-Silhouette/GBridgeWeb

由于隐私相关问题，保存数据库 key 和大模型 API-key 的 config.json 并未包含在项目文件中。如有需要请联系本人（2200012966@stu.pku.edu.cn）。

二、代理服务端（gbridge-proxy）

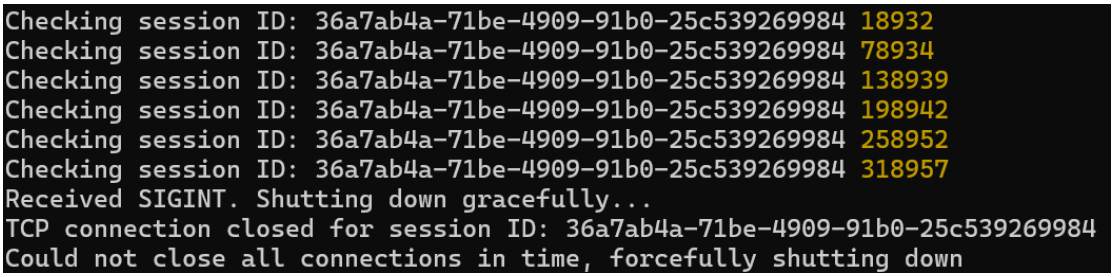
（一）使用实例

代理服务器没有相应的 UI 界面，在此用命令行界面展示。



（图一：代理服务器通讯内容）

开头一行表示代理开始。之后每收到一个客户端请求都先打印 ID 和相应请求，当收到服务端的数据时先打印原始数据，再打印解析的 json。



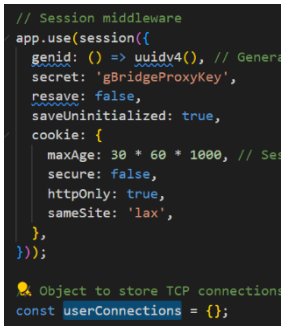
（图二：代理服务器清理内容和结束）

每分钟检查存储的所有 session 最后的使用时间，如果大于 30min 则进行清理。最后捕获 SIGINT 关闭代理服务器。

（二）代码详解

这部分所有的代码在 proxy-server.js 文件中，一些配置在 config/ config.json 中。Proxy 主要包含 set-up, TCP-connection, API-settings 和 session-cleaning 四部分。

1. set-up（1-40 行）



主要完成框架建立和存储变量定义。

引入 config 中端口号，等待时间，重复尝试次数等变量，定义浏览器中应该存储的 cookie 信息和 sessionId,使用 cors 进行跨域资源共享，应对客户端之间并行多请求的不同回调函数的 userConnections 对象。

（图三：代理服务器框架建立）

2. TCP-connection（42-145）

```
// create a tcp connection
const handleTcpConnection = async (sessionId) => {
  const tcpClient = new net.Socket();
  userConnections[sessionId].client = tcpClient;
  userConnections[sessionId].lastActive = Date.now();

  tcpClient.connect(TCP_SERVER_PORT, TCP_SERVER_HOST, () => {
    console.log(`Connected to TCP server with session ID: ${sessionId}`);
    userConnections[sessionId].retryCount = 0; // Reset retry count
  });

  tcpClient.on('data', (data) => {
    // Append the received data to the buffer
    let dataBuffer = userConnections[sessionId].dataBuffer;
    dataBuffer += data.toString();
    console.log('Received data:', dataBuffer);
    userConnections[sessionId].dataBuffer = dataBuffer;

    let jsonResponse;
    try {
      jsonResponse = JSON.parse(dataBuffer);
      console.log('Received:', jsonResponse);
      userConnections[sessionId].dataBuffer = '';
```

（图四：代理服务器创建连接，处理回复信息部分）

创建新的 TCP 连接的函数，在失败或者意外关闭时依据 config 中定义的 retry 信息试图重新创建 TCP 连接。图中展示的是接收服务器 data 部分的处理。使用 buffer 缓存收到的原始数据以防由于网络问题传回的数据被截断的情况。

```
// handle the tcp request
const handleTcpRequest = async (sessionId, requestBody, requestType, res) => {
  if (!userConnections[sessionId])
    userConnections[sessionId] = {
```

（图五：代理服务器判断一个请求如何处理）

处理接收的请求的函数。通过 ID 和相应状态判断是否需要使用新的 TCP 连接。设置相应的回调函数，发送请求并更新 session 访问时间。

这部分是代理的核心内容，整体使用异步的方式实现一定程度上的并行处理。同时在代码的错误处理和健壮性上也进行了一定的优化。

3. API-settings（147-169）

```
// handle the common request
app.post('/api/common', (req, res) => {
  const sessionId = req.sessionID;
  const requestBody = req.body;
  const requestType = requestBody.type; // Use request type as th
  console.log(`Received request for session ID: ${sessionId} with
  handleTcpRequest(sessionId, requestBody, requestType, res);
});

// handle the logout request, delete session and cookie
app.post('/api/logout', (req, res) => {
```

（图六：代理服务器 API 接口，实现不同的逻辑）

设置了两种处理方式的 API 端口，用于和客户端通信。

4. session-cleaning（171-214）

```
171  // Cleanup inactive sessions
172  setInterval(() => {
173    const now = Date.now();
174    const timeout = 30 * 60 * 1000; // 30 minutes
175
176    for (const sessionId in userConnections) {
177      console.log(`Checking session ID: ${sessionId}`, now - userC
178      if (now - userConnections[sessionId].lastActive > timeout) {
```

（图七：代理服务器设定定时器清理）

设置每分钟历遍 session 检查长时间不使用的 sessionID，将相应的 session 删除，取消用户的认证状态。

```
186  const server = app.listen(PORT, () => {
187    console.log(`Proxy server listening on port ${PORT}`);
188  });
189
190  // gracefully shut down the server
191  const gracefulShutdown = () => {
192    console.log('Received SIGINT. Shutting down gracefully...');
193
194    // Close all TCP connections
195    for (const sessionId in userConnections) {
```

（图八：代理服务器注册信号监听和相应处理）

检查 SIGINT 信号关闭代理。关闭前清空 session 并关闭所有创建的 TCP 连接。同时设置强制停止的计时用于处理未能正常退出的情况。

（三）反思

在现有服务器和 express 框架的基础下完成简单的转发任务较为基础。通过缓存的形式解决了遇到的回复接收不全的问题，以及 session 机制进行简单的认证，总体完成较为顺利。

现在的代理服务器具有完整的初始化、运行时 log、数据清理和关闭检查，基本流程完整。提供给客户端的 API 接口统一，与客户端通信具有一定的鲁棒性，完整实现了转发的功能。

缺陷主要在于没有完整的错误处理。现在仅仅处理的简单的连接中断问题，对于其他大多数错误未能有完整的反馈，同时没有完整的和客户端发送错误内容的能力。此外，会话清理部分现在是直接遍历判断，比较低效，仅适合于小型场景。后续可以拆分成两个队列并降低检查频率以提高效率。

三、Web 客户端（gbridge-web-client）

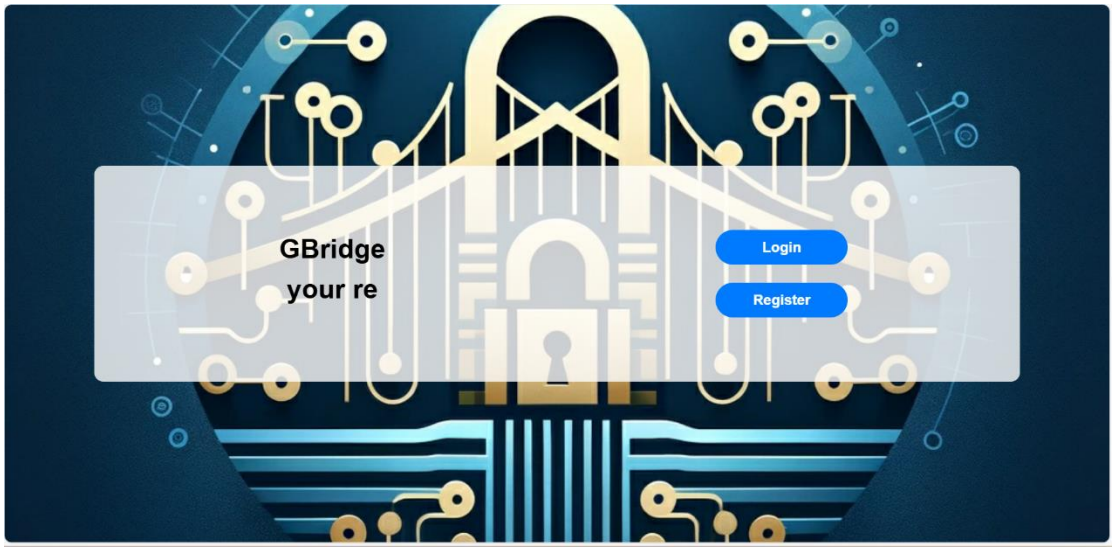
（一）使用实例

按功能分区客户端主要分为认证界面、主界面、市场、用户请求、评分和聊天与用户信息六个部分。除了认证部分其余五个部分都使用最上方的导航栏跳转。在对应界面导航栏会变蓝，最右侧是退出登录的按钮。

（图九：Web 导航栏）

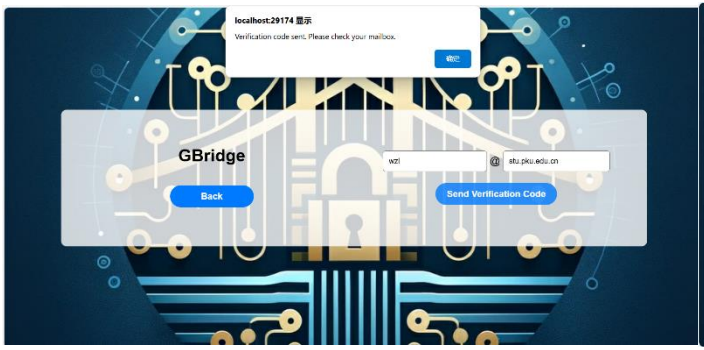
- **认证界面**

认证界面是用户登录或者注册的界面，以项目 Logo 为背景，左侧动态循环显示欢迎语。

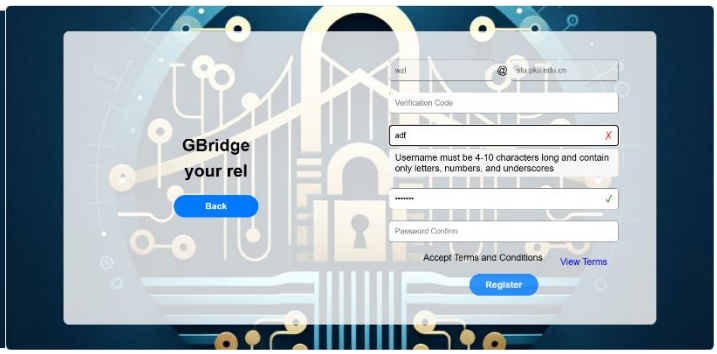


（图十：Web 认证界面展示）

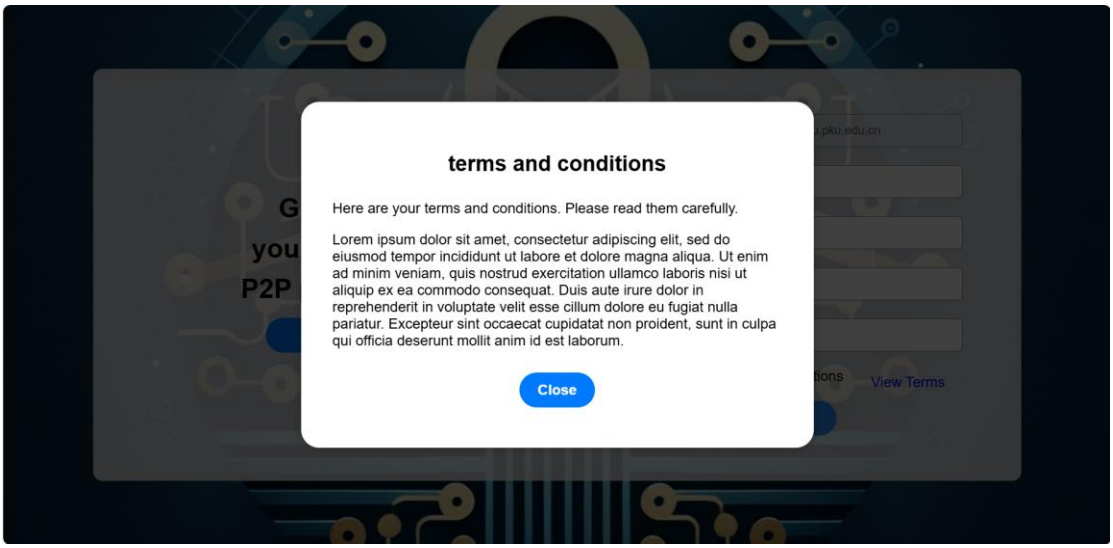
注册界面输入邮箱获取验证码，然后会显示注册表单。选中用户名和密码输入框会显示相应格式要求。符合要求的输入会以绿色对勾提示，否则是红叉。下方是用户条款同意栏，点击可显示条款框。信息填写不全或未同意条款将不能提交，密码不一致同样有提示。注册不成功也会将错误信息显示在下方。



（图十一：Web 注册验证码）

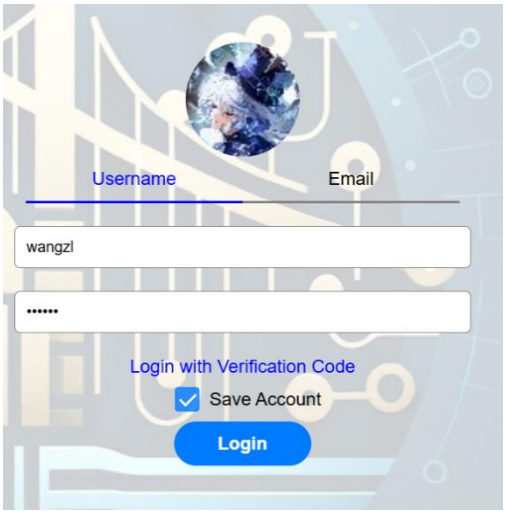


（图十二：Web 注册表单）

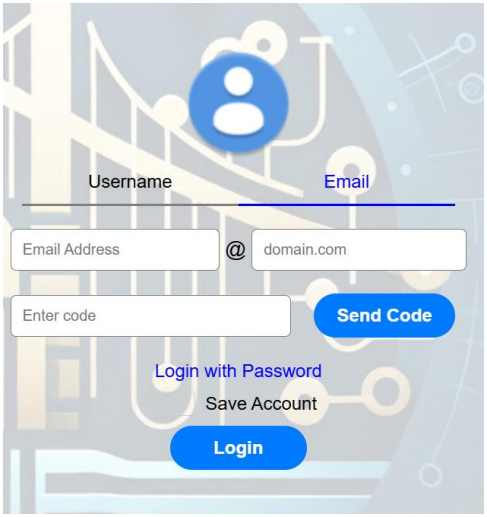


（图十三：Web 服务条款部分）

登录界面可以通过 Tab 和 link 选择 4 种登录方式，如果保存有以前的账号则会显示对应的头像提示。



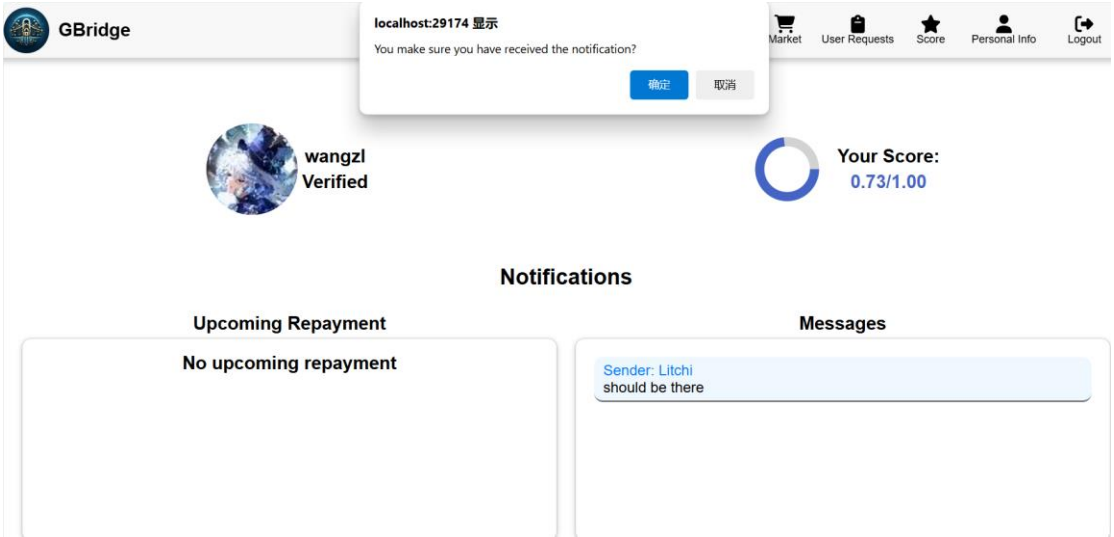
（图十四：Web 登录保存后）



（图十五：Web 登录普通）

- **主界面**

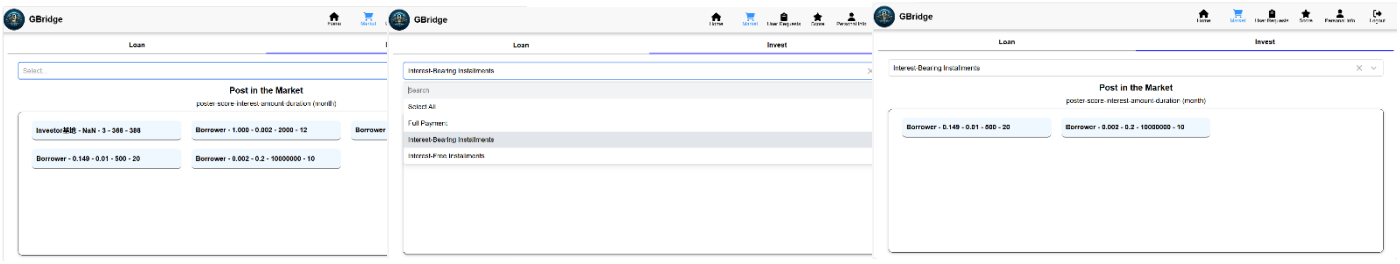
主界面包括最上方的问候语，用户信息栏（头像、用户名和认证状态），信用评分（加载界面时动态增加，同时按值更换颜色）以及下方的提醒板。提醒板包含近期还款项目以及其他用户发送的通知。点击通知接收。



（图十六：Web 主界面）

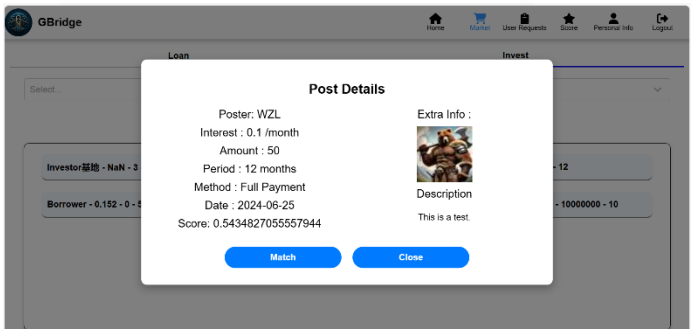
- **市场界面**

市场包含了当前贴出的所有帖子，分为贷款和投资两个板块。帖子按评分从高到低排序，通过选择筛选条件显示符合的帖子。

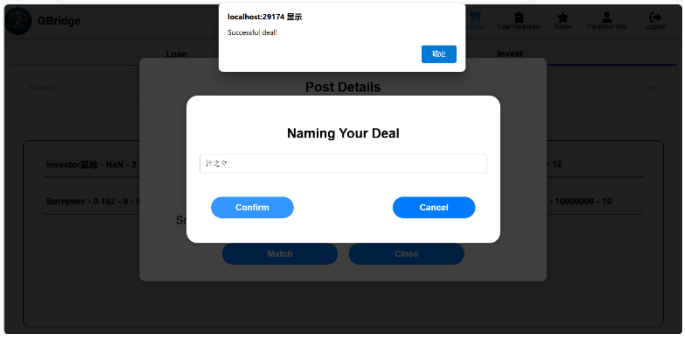


（图十七：Web 市场总体）　（图十八：Web 市场筛选）　（图十九：Web 市场筛选后）

点击帖子就可查看详情，进一步可以进行匹配。



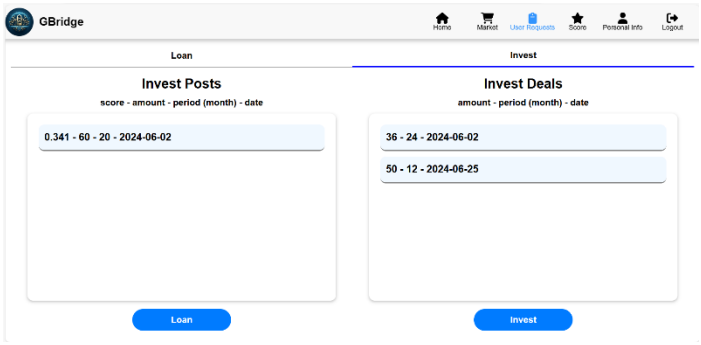
（图二十：帖子细节）



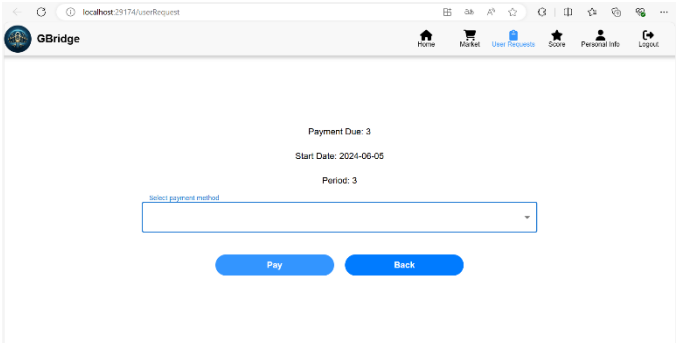
（图二十一：匹配帖子）

- **用户请求**

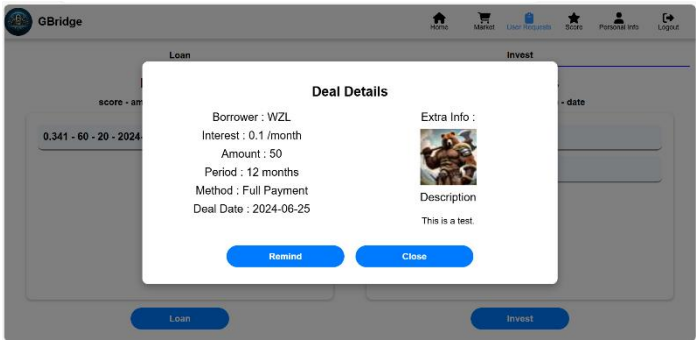
这一界面包含了用户所有的发帖和正在进行的交易，下方是发帖的转跳按钮。点击相应的项目可以查看具体信息。可以删除帖子、还款以及向借款人发送提醒。



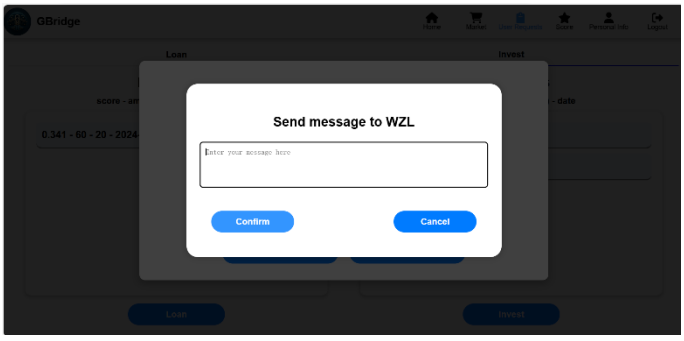
（图二十二：用户所有发帖交易）



（图二十三：还款界面）

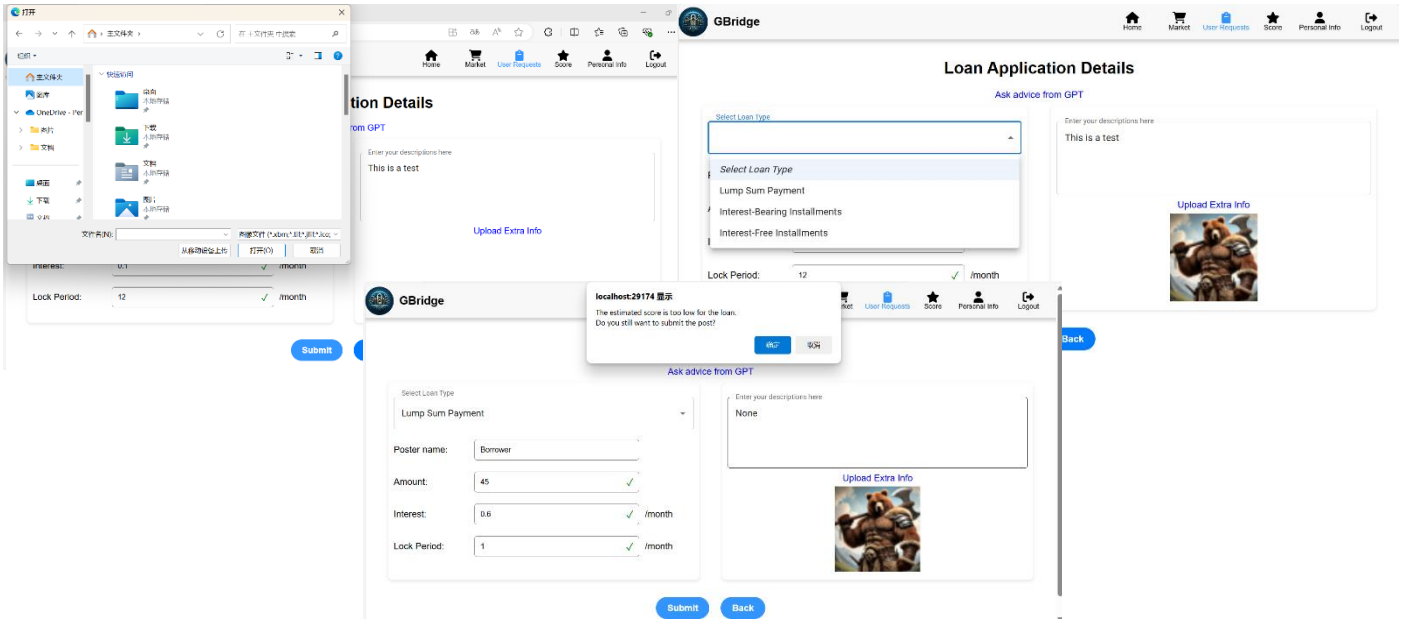


（图二十四：帖子详细信息）



（图二十五：签字匹配）

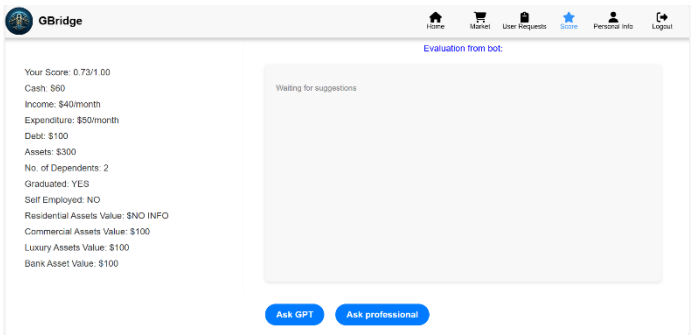
发布帖子，点击 link 可以向大模型询问推荐，会自动填充在表单中。如果发布的帖子的评分较低，会进一步发出一个提示是否继续发布帖子。



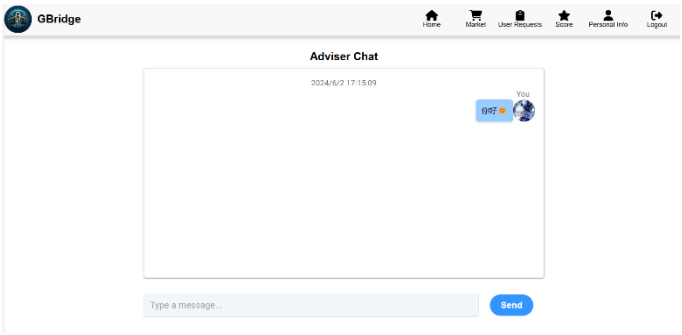
（图二十六到二十八：发布帖子过程展示）

- **评分与聊天**

在这里查看评分的具体细节，并可以向大模型询问对于此时经济状况的评价。下方是向大模型和专家聊天的转跳按钮。聊天界面类似普通聊天软件。



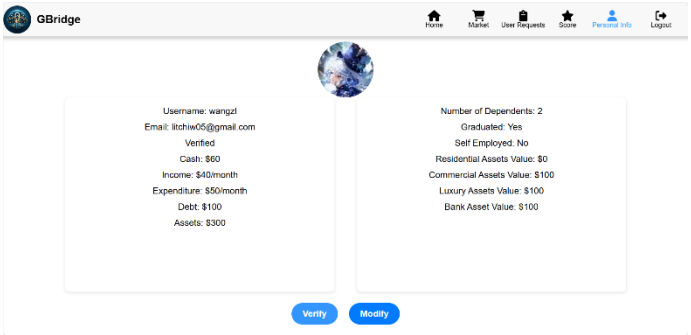
（图二十九：评分详情和评价）



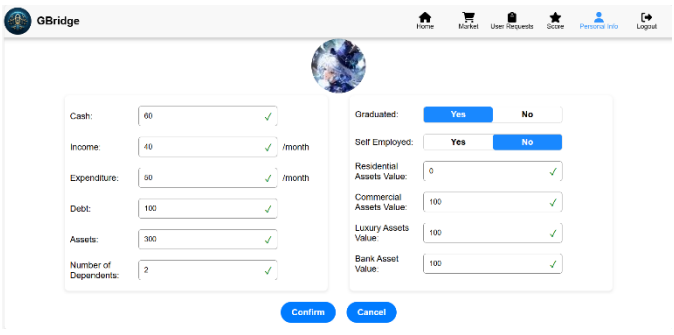
（图三十：聊天展示）

- **用户信息**

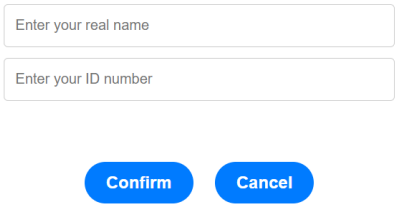
查看用户登记的各种信息。可以进行修改、认证等操作。



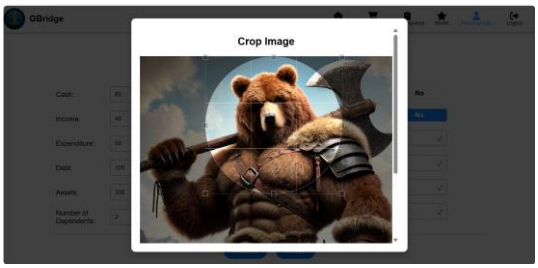
（图三十一：用户信息展示）



（图三十二：用户修改信息）

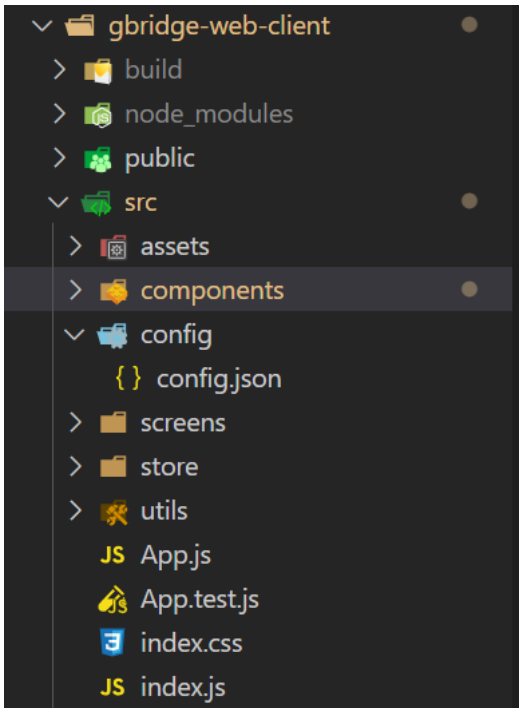


（图三十三：用户认证）



（图三十四：裁剪选取头像）

（二）代码简介



（图三十五：项目结构）

项目代码在 gbridge-web-client 中，主要包含一下几个部分：

1、 Assets：应用的图片等资源
2、 Config：外部的接口信息
3、 Store：定义全局状态和认证
4、 app & index：
设置界面包含内容和转跳表
5、 Utils/：包含三个处理类
AxiosContext：整体通信协议
ParseItem：重构 Post 的信息
localStorageUtils：本地存储
6、 components/：
由 React 延伸而出的构件。包含单纯的自定义组件或者页面的部分板块
7、 screens/：
会放在导航栏中的页面

以下是部分重要代码：

- Store 部分

```
import { configureStore } from '@reduxjs/toolkit';
import authReducer from './authSlice';
import globalReducer from './globalSlice';

const store = configureStore({
  reducer: {
    global: globalReducer,
    auth: authReducer,
  },
});

export default store;
```

利用 redux 框架存储两个状态。认证状态存储注册或者登录是否成功。登出时重置状态。全局状态存储用户的用户名、头像、是否认证和可能保存的密码。用于减少图片带来的巨大数据传输量。每个状态包括相应是设置方法。

（图三十六：存储设置版块）

- App 部分、utils/AxiosContext.js

通过 AxiosContext 使整个应用使用统一的网络环境，即通过这样的方式保存同样的 cookie，从而使所有这一应用发出的请求都有同样的 sessionId，保证使用相同的 TCP 连接保持用户的认证状态。

应用的导航使用 BrowserRouter 设置所有页面的本地 Url 实现总体的页面跳转。所持有的 8 个页面中前六个是划分应用不同功能的板块，最后两个是两个 Chat 页面，虽然不独立于功能板块，但本身与依赖的 score 分离较强，所以单独拿出来。

```
const axiosInstance = axios.create({
    baseURL: config.proxy.url,
    withCredentials: true,
});

const AxiosContext = createContext();

export const AxiosProvider = ({ children }) => (
    <AxiosContext.Provider value={axiosInstance}>
        {children}
    </AxiosContext.Provider>
);
```

（图三十七：Axios 环境）

```
<AxiosProvider>
  <Router>
    <Routes>
      <Route path="/" element={<WelcomeInterface />} />
      <Route path="/home" element={<Home />} />
      <Route path="/userRequest" element={<UserRequests />} /
      <Route path="/market" element={<MarketComponent />} />
      <Route path="/score" element={<ScoreInterface />} />
      <Route path="/personalInfo" element={<PersonalInfo />}
      <Route path="/botChat" element={<BotChatInterface />} /
      <Route path="/adviserChat" element={<AdviserChatInterfa
    </Routes>
  </Router>
</AxiosProvider>
```

（图三十八：App 设置路径和页面）

- TextAnimation.js

通过套环的 setTimeout 逐字添加到显示的文本中，从而表现动态的效果。同时保证等待队列中永远只有一个字。useEffect 实现渲染时自动触发，return 退出清除还未触发的显示文本。

```
7     useEffect(() => {
8         let timeoutId;
9
10        const currentLine = lines[lineIndex];
11        const currentText = displayedTexts[lineIndex];
12
13        if (currentText.length < currentLine.length) {
14            timeoutId = setTimeout(() => {
15                const newTexts = displayedTexts.map((text, index) =>
16                    index === lineIndex ? text + currentLine[text.length] : text);
17                setDisplayedTexts(newTexts);
18            }, 100);
19        } else if (lineIndex < lines.length - 1) {
20            // Move to the next line after the current line is fully displayed
21            setTimeout(() => setLineIndex(lineIndex + 1), 300);
22        } else {
23            // After all lines are displayed, wait and then reset
24            timeoutId = setTimeout(() => {
25                setDisplayedTexts(new Array(lines.length).fill(''));
26                setLineIndex(0);
27            }, pauseDuration);
28        }
29
30        return () => clearTimeout(timeoutId);
31    }, [displayedTexts, lineIndex, lines, pauseDuration]);
```

（图三十九：动态文本显示部分代码）

- NumberInput/RuleInput

通过 forwardRef 将原本函数式组件可以仿照 React.component 加入 ref 从而让外部组件可以访问内部组件的部分函数。从而无论是外部还是内部状态都是可以同步的。从而将正则表达式可以固化在相应的组件中，而非放在外部组件传递。输入的规则通过正则表达式规定。

```
50  const NumberInput = forwardRef(({ iniValue, prompt, updateValue, tail, ...re
51      const [input, setInput] = useState(iniValue);
52      const [isValid, setIsValid] = useState(true);
53
54      useImperativeHandle(ref, () => ({
55          changeInitialValue(value) {
56              setInput(value);
57              setIsValid(true);
58              updateValue(value);
59          },
60      }));
61
62      const handleInputChange = (text) => {
63          setInput(text);
64          if (text.trim() !== '' && /^-?\d+(\.\d+)?$/.test(text.trim()) && !is
65              setIsValid(true);
66              updateValue(parseFloat(text));
67          } else {
68              setIsValid(false);
69              updateValue(null);
70          }
71      };
```

（图四十：包含规则的输入框部分代码）

- ProgessRing & Spinner

圆形进度条使用两个 svg 圆重叠，一个逐步打开实现渲染中的动画效果。同时传入差值得到的颜色代表相应的进度。Spinner 作为等待的提示符，通过 css 的 animation 属性定义不断运动的一小段。

```jsx
const Spinner = ({ size = 'large', color = '#0000ff' }) => {
    const spinnerSize = size === 'large' ? 80 : size === 'medium' ? 40 : s
    const borderSize = size === 'large' ? 16 : size === 'medium' ? 8 : siz

    const spinnerStyle = {
        width: `${spinnerSize}px`,
        height: `${spinnerSize}px`,
        borderWidth: `${borderSize}px`,
        borderTopColor: color,
    };

    return <div className="spinner" style={spinnerStyle}></div>;
};

export default Spinner;
```

（图四十一：等待提示符）

```jsx
// score ring component
const ProgressRing = ({ radius, stroke, progress, color }) => {
  const normalizedRadius = radius - stroke * 2;
  const circumference = normalizedRadius * 2 * Math.PI;

  const [offset, setOffset] = useState(circumference);

  useEffect(() => {
    const progressOffset = (1 - progress) * circumference;
    setOffset(progressOffset);
  }, [progress, circumference]);

  return (
    <svg
```

（图四十二：圆形进度条）

- ImageCropModal

主要分为两个部分：从本地文件中选择图片上传和剪切头像框位置。前者通过 HTML 自带的 input 直接实现，后者则需要自己弹出选择框。笔者采用的方式是通过弹出 Modal，使用固定比例的 ReactCrop 模块让用户自主选择想要从图片中截取的部分。然后通过 getCroppedImg 函数获取真正图片内容。创建画布画出现有图像，然后通过变换得到规定的图片大小内容，再次画出新图像获取最终的 base64 编码。

```jsx
const pickCropImage = () => {
    const input = document.createElement('input');
    input.type = 'file';
    input.accept = 'image/*';
    input.onchange = (event) => {
        const file = event.target.files[0];
        if (file) {
            const reader: FileReader
            reader.onload = () => {
                setSrc(reader.result);
                setModalVisible(true);
            }
            reader.onerror = error => console.error('Error: ', error);
        }
    };
    input.click();
};
```

（图四十三：从本地选择文件）

```jsx
// get cropped image, resize to my size
const getCroppedImg = (image, crop) => {
    const canvas = document.createElement('canvas');
    const scaleX = image.naturalWidth / image.width;
    const scaleY = image.naturalHeight / image.height;
    canvas.width = crop.width;
    canvas.height = crop.height;
    const ctx = canvas.getContext('2d');

    ctx.drawImage(
        image,
        crop.x * scaleX,
        crop.y * scaleY,
```

（图四十四：裁剪获取的图片）

- 组件代码

大部分组件内部的函数由三部分构成。内部状态定义和状态转移函数部分、通信交互部分函数和组件定义（return 和 render）。组件的样式使用 module.css 的形式进行导入，从而防止 css 本身的样式重叠作用，将组件的样式完全独立开来。

通信交互使用 async 异步方式实现，主要用 axios 进行 http 通信，使用 try/catch 捕获过程中产生的异常，并且通过 alert 方式提出。

```jsx
    try {
      const response = await axios.post(config.proxy.common, {
        type: 'get_verificationcode',
        content,
        extra: null,
      });

      if (response.data.success) {
        dispatch(sendCodeSuccess());
        alert('Verification code sent. Please check your mailbox.');
      } else {
        dispatch(sendCodeFailure('Failed to send verification code. P
      }
    } catch (error) {
      dispatch(sendCodeFailure('Failed to send verification code. Plea
    }
  } else {
    alert(`Please enter your ${activeTab}`);
  }
```

（图四十五：通信交互异步处理样例）

以下是文件中所有模块相应的功能简介。

模块名称	主要功能
Cart	显示用户发布的帖子和完成的交易，可以进一步进行操作
Header	导航栏，各个页面的跳转和登出键
ImageCropModal	选择、裁剪头像和上传图片
InfoInputs	各类信息输入框
MyButton	各类自定义按钮
RuleTextInput	包含一定规则显示的输入框
RequestDetail	显示 post 相关的信息的弹出框
InputModal	输入文本的弹出框
Spinner	等待提示符
TextAnimation	逐字显示的文本
LoginInterface	处理登录的版块
RegisterInterface	处理注册版块
PostInterface	发布帖子的表格页面
RepayInterface	还款页面
ChatInterface	聊天界面基类，派生两种聊天
Home	主界面
MarketComponent	显示市场中可供挑选的 post
PersonalInfo	用户信息显示和设置
ScoreInterface	评分详情和聊天入口
UserRequests	显示个人帖子和发布帖子入口
WelcomeInterface	认证界面

（表四：项目涉及的所有主要模块功能展示）

（三）反思

利用 React 框架对于整体界面管理非常方便。同时，在@mui/material 中已经有较多已定义的优秀组件，将整体的代码量大幅度简化。在处理剪切头像时发现没有现成符合的组件，通过查询已有资料，和一些基本等价的改变，实现了自定义的组件。

移植版本充分利用 PC 端更大的屏幕，将原本移动端层叠的界面简化为单独界面，合并上下两部分的导航栏，简化了用户交互逻辑。大部分表单也通过分栏显示在统一页面中，整体表现上更加清晰。通过 redux 建立全局的状态记录，创建客户端层面的认证状态，缓存大数据量的信息并集中管理。

但是，Web 移植仍存在不少缺陷。作为网页端，移植版不存在未登录可以访问的一个宣传界面，包括移动版的下载和应用更新推广信息。同时下方也未标注相应的公司和条款信息，在正式程度上还有所欠缺。此外，为了认证的方便，应用没有允许通过回撤访问历史页面，一定程度上影响了用户体验。

四、总结与收获

项目总体耗时在三周时间。主要耗时在于对于原本客户端的重构和做对应 Web 的适应。项目难度中等，主要是有一定的代码量需求，同时需要对于 Express 框架和 React 框架有一定的了解。

（一）项目特色

本项目旨在通过移植到更容易接触到的平台从而扩大用户群体和增强用户体验。体现在本项目中主要有如下的几点：

1、完善的功能：移植版本实现了原版中作为用户方的所有功能，同时优化了部分流程，给用户完善的使用体验。
2、简洁统一的 UI 设计：网页中所有按钮等都统一样式处理，布局大气，功能指引简单明了，给用户便捷的使用体验。
3、丰富的提示符：当通讯延迟时，正在等待的组件会显示相应的等待提示。发生错误时也会通过弹窗的形式提示用户。
4、本地缓存提高速度：通过缓存信息，减少与服务器的通信，大大提高了页面跳转的流畅程度。同时将询问大模型等长耗时请求单独拆分，减少用户附加等待时间。

（二）项目可改进之处

在最终完成项目时， 我对于原项目中的非必要部分进行了部分删减以适应工期， 整个项目还有不少值得改进的部分。 现总结罗列如下：

1、adviser 的网页端登录（放弃的移动端功能）
2、包含宣传、更新、移动端下载的主页
3、UI 页面更多与用户动态的交互，如组件浮现效果
4、错误处理时自定义更符合项目风格的提示框
5、代理服务端与服务端通信的鲁棒性

（三）整体收获

通过推进这个项目，我对于 Javascript 基本语法和应用的熟练度有了不小的提升，较为基础的应用都较为熟悉。对于回调函数、异步编程和闭包这三部分的内容有了结合实际的更深一步的了解。同时我也积累了一定的对于这门语言的 debug 能力，初步掌握了这门语言的基础内容。

通过代理服务器的编写我接触了常用的 Express 框架，了解了如何使用 Javascript 利用 http 协议构建简单的服务器，理解了 API 端口拆分的应用、会话管理实现认证状态保持的方式。同时也认识到了网络编程中错误处理和容忍的重要性。

通过 Web 客户端的编写我进一步了解了 React 框架，熟悉了如何使用提供的模块自定义组件，如何在 css 样式中避免重叠。我也发现了以前并不知道的许多优秀组件和样例，为我快速构建 UI 帮助巨大。通过 css 样式试错般的编写我也真切地了解了不同值所代表的表现形式，对于前端编程有了一个初步的体验与理解。

一个项目接触了两个主流的框架，虽然涉及不深，但仍是收获巨大。

最后，特别感谢原本项目服务端的主要负责人张佳豪同学。非常感谢北京大学 Javascript 语言与 Web 程序设计课程的老师和助教提供一次自主开展项目的机会，也感谢老师和助教一学期的辛苦付出！