🎯 目标

您的核心任务是开发一个“大数据采集、存储与分析系统”。根据《大数据部分项目指导书》和《实验步骤.pdf》，您可以按照以下7个主要任务逐步推进：

1. 搭建系统框架 (VUE)

- **任务：** 基于VUE设计并搭建大数据采集与分析系统框架。
- **步骤：**
  1. **创建项目：** 使用 `vue ui` 命令创建新的VUE项目。
  2. **文件结构：** 在 `components` 文件夹中，新建 `CoreSource.vue` (数据采集界面)、`ShowData.vue` (数据显示界面)、`DataProcess.vue` (数据分析界面) 等核心文件。同时，新建 `pyt` 文件夹及 `backend.py` 文件用于后端开发。
  3. **界面元素：** 在 `main.js` 中导入 Element UI 库。然后在 `.vue` 文件的 `<template>` 部分使用 `<el-button>`、`<el-card>` 等组件搭建用户界面。
  4. **页面路由：** 在 `main.js` 中引入并配置 `VueRouter`。在 `App.vue` 中使用 `<router-view>` 来显示不同页面，并通过 `<el-button>` 的点击事件 (`@click="onSubmit"`) 配合 `this.$router.push` 实现页面跳转。

2. 运行虚拟数据源 (MQTT)

- **任务：** 建立与边缘设备（虚拟）的通信。在开发过程中，您将使用虚拟数控系统作为数据源。
- **步骤：**
  1. **启动MQTT服务器：** 在 `emqx-5.0.11-windows-amd64\bin` 路径下，通过 `cmd` 运行 `emqx start` 命令。
  2. **启动虚拟数控服务：** 在 `StressTest-INC-Cloud` 路径下，通过 `cmd` 运行 `StressTest-INC-Cloud -n 1 -b 0` 命令。这将模拟一个序号为0的数控系统。
  3. **(可选) 测试连接：** 使用 MQTTX 软件连接到 `127.0.0.1:1883`，通过发布和订阅指令（如 `Probe/Query/Request/STRESS_TEST_00000`）来测试虚拟服务是否正常工作。

3. 实现前后端通信与数据获取

- **任务：** 利用Python后端连接MQTT服务器并获取数据，实现前后端通信。
- **步骤：**
  1. **后端 (Python + Flask)：**
     - 使用 `flask` 框架搭建后端服务。
     - 使用 `@app.route('/connect/', ...)` 定义接口，通过 `request.get_json()['data']` 接收前端发送的数据。
     - 使用 `flask_mqtt` 库根据前端传递的配置（如服务器地址、主题）来连接MQTT服务器、订阅(Subscribe)和发布(Publish)消息，以获取实时数据。
  2. **前端 (VUE + Axios)：**
     - 在 `vue.config.js` 中配置 `devServer proxy` 以解决跨域问题。
     - 导入 `axios` 库，使用 `axios.post('/connect/', {data:this.connection})` 将数据采集界面的配置信息发送给后端。

    - 通过 `.then(res => { ... })` 接收后端返回的响应。

4. 实现数据可视化 (Echarts)

- **任务：** 设计Web网页，实现数据的实时可视化。
- **步骤：**
    1. **引入库：** 在 `main.js` 中导入 `echarts`。
    2. **定义图表容器：** 在 `ShowData.vue` 的 `<template>` 中，放置一个 `div` 并设置 `id` (例如 `<div id='x.actualspeed' ...>`)。
    3. **初始化图表：** 在 `<script>` 中，通过 `echarts.init(document.getElementById('x.actualspeed'))` 初始化图表。
    4. **更新数据：** 后端程序解析MQTT获取的JSON数据并发送至前端。前端在 `axios` 的回调中获取数据，并使用 `this.chart.setOption({ series: [{ data: received_data }] })` 来动态更新图表。

5. 实现数据存储 (InfluxDB & MySQL)

- **任务：** 基于InfluxDB和MySQL数据库实现数据存储。
- **步骤：**
    1. **InfluxDB (时序数据)：**
        - **本地：** 运行 `influxd` 启动本地服务。访问 `http://localhost:8086`，注册并创建 `bucket` (数据桶)。
        - **云端：** 访问云端InfluxDB网址，注册登录，并同样创建 `bucket`。
        - **实现：** 在后端Python或前端Node.js中，使用InfluxDB的API token和配置信息，将采集到的数据写入本地和云端的 `bucket` 中。
    2. **MySQL (关系数据)：**
        - **配置：** 确保MySQL服务已安装并运行。使用 Navicat Premium 等工具，连接到 `localhost` (用户名 `root`，输入您的密码)。
        - **创建：** 新建数据库 (如 `mqtt_data`)，并创建表结构。
        - **实现：** 在后端Python程序中，使用 `pymysql` 库。连接数据库 (`pymysql.connect(...)`)，创建游标 (`conn.cursor()`)，执行 `insert` 语句 (`cur.execute(sql)`)，并提交事务 (`conn.commit()`)。

6. 实现大数据分布式存储

- **任务：** 基于工业大数据平台实现实时获取数据的分布式存储。
- **步骤：**
    - 该任务要求将您存入 MySQL 的数据，与工业大数据平台（基于Hadoop HDFS）建立连接。
    - 您需要开发数据通信接口，将 MySQL 数据库中的数据写入大数据平台，实现分布式存储。

7. 实现数据分析与结果集成

- **任务：** 利用算法分析数据，并将分析结果集成到系统中。
- **步骤：**
    1. **训练模型：** 根据提供的数据集，使用机器学习或深度学习算法训练一个设备状态预测模型。
    2. **集成模型：** 将训练好的模型集成到 `backend.py` 后端程序中。

3. **界面展示：** 在 `DataProcess.vue` 界面 中，设计界面元素，允许用户触发分析。后端调用模型进行预测，并将分析结果（如设备状态）返回给前端，在网页上显示。

相关启动操作

1. **cmd启动：** emqx start
   StressTest-INC-Cloud -n 1 -b 0
   influx
2. **python直接运行：** backend.py
3. **终端直接运行：** yarn serve
   vue ui：创建vue界面的基础代码

mysql密码：cps@CPS123
本地influxdbTOKEN：7ucc4S8rrzwu85NA5nUYb_CNG7C-03Rbuyf2A85A5leATuxcPH_UlFvrCNXSGQtxvZQTuY_C6O7BUWNg4oIH-g==
云端influxdbTOKEN：export zsOj7fEHWcI3DUOhFvhZ0iR39tRyWxko0oGAS5grC8PVyr-RhLe9X3WwHWPjdMJmiZRHwxqt2pE5XWi3TQ34Og==

vscode运行：
注意对应的环境在终端中打开

github配置：

> git config --global --unset https.proxy
> git config --global http.proxy http://127.0.0.1:7890
> git config --global https.proxy http://127.0.0.1:7890

任务一：搭建系统框架 (VUE)

将其从一个基础模板修改为符合我们实验要求的“大数据采集与分析系统”框架。

您当前的文件是使用`vue-cli`创建的一个标准Vue 2项目。

1. “解读”——分析现有代码

- `package.json`: 定义了项目依赖，核心是`vue: "^2.6.14"`。
- `src/main.js`: 项目的入口文件。它目前只加载了`App.vue`组件。
- `src/App.vue`: 项目的根组件。它目前显示一个Logo和`HelloWorld`组件。
- `src/components/HelloWorld.vue`: 默认的欢迎页面。
- `vue.config.js`: Vue CLI的配置文件，目前是空的。

2. “调试”——修改代码以搭建框架

“调试”在这里意味着我们需要修改和添加代码，以满足《实验步骤.pdf》和《大数据部分项目指导书.pdf》中的要求。

2.1: 安装核心依赖

根据实验指导书，我们需要**页面切换**（`vue-router`）、**UI界面**（`element-ui`）和**前后端通信**（`axios`）。

请在您的项目根目录（`juiceyang999/class2up/class2up-9730b2fcdf474be54786a1f787cc6b005a26d2fc/`）打开终端，运行以下命令：

```
yarn add vue-router@3 element-ui axios
```

(注意：因为您使用的是Vue 2，所以我们安装 `vue-router@3`)

2.2: 创建页面组件

在 `src/components/` 文件夹中，除了 `HelloWorld.vue`，请**新建**以下三个文件：

- `CoreSource.vue` (数据采集界面)
- `ShowData.vue` (数据显示界面)
- `DataProcess.vue` (数据分析界面)

为了让它们能显示内容，您可以暂时在每个文件中填入以下基础代码（以`CoreSource.vue`为例）：

```
<template>
  <div class="page">
    <h1>数据采集界面</h1>
    </div>
</template>

<script>
export default {
  name: 'CoreSource'
}
</script>
```

2.3: 配置路由 (src/router.js)

《实验步骤.pdf》中提到了 `vue router`。我们需要一个文件来管理页面之间的跳转。

在 `src/` 目录下**新建**一个文件 `router.js`，并填入以下内容：

```
import Vue from 'vue'
import VueRouter from 'vue-router'

// 引入我们的组件
import HelloWorld from './components/HelloWorld.vue'
import CoreSource from './components/CoreSource.vue'
import ShowData from './components/ShowData.vue'
import DataProcess from './components/DataProcess.vue'

Vue.use(VueRouter)

// 定义路由规则
const routes = [
  {
    path: '/',
```

```
    name: 'Home',
    component: HelloWorld
  },
  {
    path: '/core',
    name: 'CoreSource',
    component: CoreSource
  },
  {
    path: '/show',
    name: 'ShowData',
    component: ShowData
  },
  {
    path: '/process',
    name: 'DataProcess',
    component: DataProcess
  }
]

// 创建 router 实例
const router = new VueRouter({
  routes // (缩写) 相当于 routes: routes
})

export default router
```

2.4: 挂载路由和UI库 (修改 src/main.js)

[cite_start]现在我们需要告诉Vue"使用"我们刚刚安装的 `ElementUI` [cite: 1091] 和 `router.js`。

请**修改** `src/main.js` 文件，内容如下：

```
import Vue from 'vue'
import App from './App.vue'

// 1. 引入 Element UI
import ElementUI from 'element-ui';
import 'element-ui/lib/theme-chalk/index.css';

// 2. 引入路由
import router from './router'

Vue.config.productionTip = false

// 3. 使用插件
Vue.use(ElementUI);

new Vue({
  // 4. 挂载 router
  router,
  render: h => h(App),
}).$mount('#app')
```

2.5: 修改主布局 (修改 src/App.vue)

最后，我们修改根组件 `App.vue`，将它从显示 "HelloWorld" 改为显示**导航菜单**和**路由页面**。

请**替换** `src/App.vue` 的全部内容为：

```
<template>
  <div id="app">
    <el-menu :router="true" mode="horizontal" default-active="/">
      <el-menu-item index="/">欢迎界面</el-menu-item>
      <el-menu-item index="/core">数据采集</el-menu-item>
      <el-menu-item index="/show">数据显示</el-menu-item>
      <el-menu-item index="/process">数据分析</el-menu-item>
    </el-menu>

    <router-view></router-view>
  </div>
</template>

<script>
export default {
  name: 'App'
}
</script>

<style>
/* 保持原有样式，并移除顶部的60px边距 */
#app {
  font-family: Avenir, Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
  /* margin-top: 60px; */ /* 移除或注释掉此行 */
}
</style>
```

2.6: 配置跨域代理 (修改 vue.config.js)

[cite_start]为了准备后续的“前后端通信”任务，我们需要配置一个代理（proxy）来解决跨域问题，这在《实验步骤.pdf》中也有提及 [cite: 1149, 1151]。

请**修改** `vue.config.js` 文件，内容如下：

```
const { defineConfig } = require('@vue/cli-service')
module.exports = defineConfig({
  transpileDependencies: true,

  // 添加 devServer 配置
  devServer: {
    proxy: {
      // '/api' 是一个自定义的标识，所有以 /api 开头的请求都会被代理
      '/api': {
        target: 'http://127.0.0.1:5000', // 这是我们后端 Python Flask 服务的地址
        changOrigin: true, // 允许跨域
```

```
        pathRewrite: {
          '^/api': '' // 在请求时，将 /api 替换为空字符串
        }
      }
    }
  }
})
```

---

3. 验证框架

完成以上所有修改后，请在终端运行：

```
yarn serve
```

[cite_start]现在访问浏览器（通常是 `http://localhost:8080` 或 `http://localhost:8081` [cite: 901]），您应该能看到一个包含“欢迎界面”、“数据采集”、“数据显示”、“数据分析”的导航栏。点击它们，下方的页面内容会随之切换。

这标志着**任务1：搭建系统框架**已成功完成。

接下来，我们将开始**任务2：运行虚拟数据源 (MQTT)**。请确认您是否已准备好 `emqx` 和 `StressTest-INC-Cloud` 软件。

任务二：运行虚拟数据源 (MQTT)

这个任务不需要修改代码，是纯粹的环境运行步骤。根据《实验步骤.pdf》，您需要启动两个服务来模拟数控机床的数据源。

当您回到主机后，请按以下步骤操作：

1. 启动 EMQX (MQTT Broker/服务器)

- **目的：** 启动MQTT消息代理服务器。这是所有数据发布和订阅的中转站。
- **步骤：**
  1. 打开 `cmd` (命令提示符)。
  2. `cd` (切换目录) 到您的 EMQX `bin` 文件夹。根据文档，路径类似：`....../emqx-5.0.11-windows-amd64\bin`。
  3. 输入并执行以下命令：

     ```
     emqx start
     ```

  4. **验证：** 启动成功后，您会看到相关的启动信息。您可以关闭此 `cmd` 窗口，服务将在后台继续运行。

2. 启动虚拟数控系统 (MQTT Client/数据发布者)

- **目的：** 启动模拟的数控机床服务，它会作为客户端连接到 EMQX，并持续发布（Publish）模拟的机床数据。
- **步骤：**

1. 打开一个新的 `cmd` (命令提示符)。
2. `cd` (切换目录) 到您的 `StressTest-INC-Cloud` 文件夹。根据文档，路径类似：`....../StressTest-INC-Cloud`。
3. 输入并执行以下命令：

```
StressTest-INC-Cloud -n 1 -b 0
```

4. **参数说明：**
   - `-n 1`：表示模拟1个数控系统。
   - `-b 0`：表示序号从0开始。
5. **验证：** 启动成功后，您会看到类似 `STRESS_TEST_00000 is Connecting` 和 `STRESS_TEST_00000 Connected` 的输出。**请保持此 `cmd` 窗口打开**，关闭它会中止数据发送。

3. 使用 MQTTX 测试数据源

在进行下一步编码前，您可以（在主机上）使用 MQTTX 软件来验证数据是否在正常发布。

1. **安装：** 确保您已安装 MQTTX。
2. **连接：** 新建连接，IP地址填 `127.0.0.1`，端口填 `1883`。
3. **订阅：**
   - 订阅主题（Topic）：`Probe/Query/Response/STRESS_TEST_00000`。
   - 点击“订阅”。
4. **发布：**
   - 切换到发布（Publish）区域。
   - 发布主题（Topic）：`Probe/Query/Request/STRESS_TEST_00000`。
   - 指令字（Payload）留空（即发送 `""`）。
   - 点击“发布”。
5. **验证：** 回到订阅页面，您应该能收到一大串JSON格式的数据，这代表虚拟服务工作正常。

以上就是任务二的全部内容。

接下来，我们将进行**任务三：实现前后端通信与数据获取。**这包括修改VUE界面以发送连接请求，以及编写Python后端来接收请求并连接MQTT。

任务三：实现前后端通信与数据获取

此任务分为两个部分：

1. **前端 (VUE)**：修改 `CoreSource.vue` 文件，创建一个表单，让用户能输入MQTT服务器信息，并通过 `axios` 将这些信息发送到后端。
2. **后端 (Python)**：创建 `pyt/backend.py` 文件，使用 `Flask` 框架搭建一个Web服务，该服务会接收前端发来的数据，并使用 `flask-mqtt` 库尝试连接到MQTT服务器。

3.1 前端 (VUE) 代码编写

请在 `src/components/` 文件夹中，**修改 `CoreSource.vue` 文件。**
将 `CoreSource.vue` 的**全部内容**替换为以下代码。

文件路径： `juiceyang999/class2up/class2up-9730b2fcdf474be54786a1f787cc6b005a26d2fc/src/components/CoreSource.vue`

```
<template>
  <el-container>
    <el-header>
      <h2>数据采集与服务器连接</h2>
    </el-header>
    <el-main>
      <el-row :gutter="20">
        <el-col :span="12">
          <el-card class="box-card">
            <div slot="header" class="clearfix">
              <span>MQTT 服务器连接配置</span>
            </div>
            <el-form ref="form" :model="connection" label-width="120px">
              <el-form-item label="服务器地址">
                <el-input v-model="connection.host"></el-input>
              </el-form-item>
              <el-form-item label="端口号">
                <el-input v-model.number="connection.port" type="number"></el-
input>
              </el-form-item>
              <el-form-item label="Client ID">
                <el-input v-model="connection.clientid"></el-input>
              </el-form-item>
              <el-form-item>
                <el-button type="primary" @click="onConnect">连接</el-button>
                <el-button @click="onDisconnect">断开</el-button>
              </el-form-item>
              <el-form-item label="连接状态">
                <el-tag :type="connectStatus ? 'success' : 'danger'">
                  {{ connectStatus ? '已连接' : '未连接' }}
                </el-tag>
              </el-form-item>
            </el-form>
          </el-card>
        </el-col>

        <el-col :span="12">
          <el-card class="box-card">
            <div slot="header" class="clearfix">
              <span>订阅与发布</span>
            </div>
            <el-form label-width="120px">
              <el-form-item label="订阅主题">
                <el-input placeholder="例如:
Probe/Query/Response/STRESS_TEST_00000"></el-input>
              </el-form-item>
              <el-form-item>
```

```
                <el-button type="success">订阅</el-button>
              </el-form-item>
              <el-form-item label="收到的消息">
                <el-input
                  type="textarea"
                  :rows="5"
                  placeholder="等待消息..."
                  readonly>
                </el-input>
              </el-form-item>
            </el-form>
          </el-card>
        </el-col>
      </el-row>
    </el-main>
  </el-container>
</template>

<script>
// 1. 引入 axios 用于前后端通信
import axios from "axios";

// 2. 设置 axios 的基础URL，所有请求都会自动加上 /api 前缀
// 这对应《实验步骤.pdf》P9的 "axios.defaults.baseURL = "/api";"
// 我们在 vue.config.js 中配置了 /api 代理
axios.defaults.baseURL = "/api";

export default {
  name: "CoreSource",
  data() {
    return {
      // 对应《实验步骤.pdf》P10 "前端定义输入变量: connection"
      connection: {
        host: "127.0.0.1",
        port: 1883,
        clientid: "vue_client_" + Math.random().toString(16).substr(2, 8),
      },
      connectStatus: false, // 用于显示连接状态
    };
  },
  methods: {
    // 3. onConnect 函数，对应《实验步骤.pdf》P10
    onConnect() {
      console.log("正在连接到MQTT服务器...");
      // 使用 axios.post 发送数据到后端
      // '/connect/' 对应后端 Flask 的路由
      // {data: this.connection} 对应后端 request.get_json()['data']
      axios
        .post("/connect/", { data: this.connection })
        .then((res) => {
          // 这里的 res.data 对应后端 `return {'rc_status': ...}`
          console.log("后端返回的数据:", res.data);
          // 我们用 rc_status 的内容来判断是否成功
          if (res.data && res.data.rc_status === "success") {
            this.connectStatus = true;
            console.log("连接成功! ");
```

```
            this.$message({
              message: 'MQTT服务器连接成功！',
              type: 'success'
            });
          } else {
            this.connectStatus = false;
            this.$message.error('连接失败: ' + res.data.rc_status);
          }
        })
        .catch((err) => {
          // 如果后端服务没启动或出错，会在这里捕获
          console.error("连接请求失败:", err);
          this.connectStatus = false;
          this.$message.error('后端服务连接失败，请检查Python后端是否已运行。');
        });
    },
    onDisconnect() {
      // (功能待定，目前仅用于前端演示)
      console.log("断开连接");
      this.connectStatus = false;
      this.$message({
        message: '已断开连接（前端模拟）',
        type: 'info'
      });
      // 实际开发中，这里也应该向后端发送一个 /disconnect/ 请求
    }
  },
};
</script>

<style scoped>
.el-header {
  background-color: #b3c0d1;
  color: #333;
  line-height: 60px;
}
.box-card {
  text-align: left;
}
</style>
```

3.2 后端 (Python) 代码编写

现在，我们在项目**根目录**（`juiceyang999/class2up/class2up-9730b2fcdf474be54786a1f787cc6b005a26d2fc/`）下，**新建**一个名为 `pyt` 的文件夹。

然后，在 `pyt` 文件夹中**新建**一个 `backend.py` 文件。

3.2.1 安装 Python 依赖

当您回到主机时，需要先为您的Python环境安装 `Flask`（Web框架）、`flask-cors`（解决跨域）、`flask-mqtt`（连接MQTT）和 `pymysql`（后续连接MySQL时使用）。

安装命令：

```
pip install Flask flask-cors flask-mqtt pymysql
```

3.2.2 编写 `backend.py` 代码

请将以下代码**复制**到 `backend.py` 文件中。

文件路径： `juiceyang999/class2up/class2up-9730b2fcdf474be54786a1f787cc6b005a26d2fc/pyt/backend.py`

```
from flask import Flask, request, jsonify
from flask_cors import CORS
from flask_mqtt import Mqtt
import time

# 1. 初始化 Flask 应用
app = Flask(__name__)
# 允许所有域名的跨域请求（在开发中很有用）
CORS(app)

# 2. 配置 Flask-MQTT
# 我们先设置默认值，稍后会用前端传来的数据覆盖它们
app.config['MQTT_BROKER_URL'] = '127.0.0.1'  # 默认服务器地址
app.config['MQTT_BROKER_PORT'] = 1883       # 默认端口
app.config['MQTT_CLIENT_ID'] = 'flask_mqtt_client'
app.config['MQTT_KEEPALIVE'] = 60
app.config['MQTT_TLS_ENABLED'] = False

# 3. 初始化 MQTT 客户端
mqtt = Mqtt(app)

# --- MQTT 事件回调 ---
# 当连接成功时
@mqtt.on_connect()
def handle_connect(client, userdata, flags, rc):
    if rc == 0:
        print("MQTT 连接成功 (rc=0)")
    else:
        print(f"MQTT 连接失败，返回码: {rc}")

# 当收到消息时 (我们将在下一步任务中用到)
@mqtt.on_message()
def handle_message(client, userdata, message):
    data = dict(
        topic=message.topic,
        payload=message.payload.decode()
    )
    print(f"收到消息: {data}")

# --- Flask API 路由 ---

# 4. 创建 /connect/ 接口，对应《实验步骤.pdf》P10
@app.route('/connect/', methods=['POST', 'GET'])
def make_connect():
    try:
        # 5. 接收前端发来的JSON数据
```

```
        # 对应 axios.post 中的 {data: this.connection}
        data_connect = request.get_json()['data']

        # 6. 在后端控制台打印收到的数据
        print(f"收到来自前端的连接请求: {data_connect}")

        # 7. 更新 MQTT 配置并尝试连接
        app.config['MQTT_BROKER_URL'] = data_connect['host']
        app.config['MQTT_BROKER_PORT'] = data_connect['port']
        app.config['MQTT_CLIENT_ID'] = data_connect['clientid']

        # 重新配置并连接
        mqtt.client.disconnect() # 先断开旧连接
        mqtt.client._client_id = data_connect['clientid'].encode()
        mqtt.client.reinitialise()
        mqtt.client.connect(data_connect['host'], data_connect['port'])

        print(f"正在尝试连接到 {data_connect['host']}:{data_connect['port']}...")

        # 8. 向前端返回成功响应
        # 对应 axios.then(res => ...)
        # 我们返回 'success' 字符串，前端会据此更新 "连接状态"
        return jsonify({'rc_status': 'success'})

    except Exception as e:
        print(f"连接失败: {e}")
        # 如果出错，返回错误信息
        return jsonify({'rc_status': str(e)})

# (我们将在任务4中添加 /subscribe/ 接口)

# 9. 启动 Flask 服务
if __name__ == '__main__':
    # 确保端口为 5000，以匹配 vue.config.js 中的代理设置
    app.run(debug=True, host='127.0.0.1', port=5000)
```

3.3 运行与调试（当您回到主机时）

1. **启动环境（共4个）：**
   - **终端 1：** 运行 `emqx start` (任务 2.1)。
   - **终端 2：** 运行 `StressTest-INC-Cloud -n 1 -b 0` (任务 2.2)，并保持窗口开启。
   - **终端 3：** `cd` 到 VUE 项目目录 (... `/class2up-9730b2fcdf474be54786a1f787cc6b005a26d2fc`)，运行 `yarn serve`。
   - **终端 4：** `cd` 到 VUE 项目目录 (... `/class2up-9730b2fcdf474be54786a1f787cc6b005a26d2fc`)，运行 `python pyt/backend.py`。
2. **测试：**
   - 打开浏览器访问 VUE 页面（如 `http://localhost:8080`）。
   - 点击导航栏的“数据采集”。
   - 表单中应已填好 `127.0.0.1` 和 `1883`。

- 点击 “连接” 按钮。
- **预期结果：**
    1. VUE 页面右上角弹出 “MQTT服务器连接成功！” 提示，“连接状态” 变为 “已连接”。
    2. 在 **终端 4** (Python 后端) 的输出中，您会看到 `收到来自前端的连接请求: ...` 和 `MQTT 连接成功 (rc=0)` 的打印信息。

任务三完成。我们成功打通了 VUE -> Flask -> MQTT Broker 的整条链路。

接下来，我们将开始**任务四：实现数据可视化 (Echarts)。**我们将修改后端以订阅数据，并修改前端 `ShowData.vue` 页面以接收并显示数据。

任务四：Web 网页的设计与数据可视化

此任务的目标是：

1. **订阅数据：** 让后端订阅 MQTT 主题以接收数据。
2. **请求数据：** 从前端按时发起数据请求（发布消息）。
3. **获取数据：** 后端将收到的数据传递给前端。
4. **可视化：** 前端 `ShowData.vue` 页面使用 `Echarts` 实时显示数据。

我们将采用“**前端定时轮询**”的方式：

- 前端 `ShowData.vue` 页面每隔3秒向后端 **发布** 一个数据请求。
- 同时，`ShowData.vue` 向后端 **获取** 上一次收到的数据。
- 后端 `backend.py` 负责订阅和收发消息，并暂存数据。

4.1 安装 Echarts 依赖

在您回到主机后，请在项目根目录（`.../class2up-9730b2fcdf474be54786a1f787cc6b005a26d2fc/`）打开终端，运行以下命令来安装 `echarts`：

```
yarn add echarts
```

(这会更新您的 `package.json` 和 `yarn.lock` 文件)

4.2 后端 (Python) 代码更新

我们需要大幅更新 `backend.py`，使其具备 **订阅**、**发布** 和 **数据暂存/提供** 的能力。

请**替换** `pyt/backend.py` 文件的**全部内容**为以下代码：

文件路径： `juiceyang999/class2up/class2up-9730b2fcdf474be54786a1f787cc6b005a26d2fc/pyt/backend.py`

```
from flask import Flask, request, jsonify
from flask_cors import CORS
from flask_mqtt import Mqtt
import json # 引入 json 库
```

```python
# 1. 初始化 Flask 应用
app = Flask(__name__)
CORS(app)

# 2. 配置 Flask-MQTT
app.config['MQTT_BROKER_URL'] = '127.0.0.1'
app.config['MQTT_BROKER_PORT'] = 1883
app.config['MQTT_CLIENT_ID'] = 'flask_mqtt_client'
app.config['MQTT_KEEPALIVE'] = 60
app.config['MQTT_TLS_ENABLED'] = False

# 3. 初始化 MQTT 客户端
mqtt = Mqtt(app)

# 4. (新增) 全局变量，用于暂存从MQTT收到的最新数据
# 我们使用字典，以数据ID为键，方便前端按ID查询
latest_data_store = {}

# --- MQTT 事件回调 ---

@mqtt.on_connect()
def handle_connect(client, userdata, flags, rc):
    if rc == 0:
        print("MQTT 连接成功 (rc=0)")
        # 5. (新增) 一旦连接成功，立刻订阅“响应”主题
        # 这是虚拟数控系统返回数据的主题
        response_topic = "Query/Response/STRESS_TEST_00000"
        mqtt.subscribe(response_topic)
        print(f"已自动订阅主题: {response_topic}")
    else:
        print(f"MQTT 连接失败，返回码: {rc}")

@mqtt.on_message()
def handle_message(client, userdata, message):
    # 6. (更新) 当收到消息时，解析并存储它
    try:
        payload_str = message.payload.decode()
        print(f"收到消息 (Topic: {message.topic}): {payload_str}")

        # 解析JSON数据
        data = json.loads(payload_str)

        # 根据《项目课程2025.pdf》P30 的数据格式 {"values":[{"id":"...","values":
[...]}]}
        if 'values' in data and isinstance(data['values'], list):
            for item in data['values']:
                if 'id' in item and 'values' in item:
                    # 我们只存储最新的值
                    latest_data_store[item['id']] = item['values'][0] # 假设我们只
关心第一个值

            print(f"已更新数据存储: {latest_data_store}")

    except Exception as e:
        print(f"处理消息失败: {e}")
```

```python
# --- Flask API 路由 ---

@app.route('/connect/', methods=['POST', 'GET'])
def make_connect():
    try:
        data_connect = request.get_json()['data']
        print(f"收到来自前端的连接请求: {data_connect}")

        app.config['MQTT_BROKER_URL'] = data_connect['host']
        app.config['MQTT_BROKER_PORT'] = data_connect['port']
        app.config['MQTT_CLIENT_ID'] = data_connect['clientid']

        # 重新配置并连接
        # 注意：在生产环境中，更健壮的连接管理是必要的
        if mqtt.client.is_connected():
            mqtt.client.disconnect()

        mqtt.client.username_pw_set(None, None) # 清除旧凭证
        mqtt.client._client_id = data_connect['clientid'].encode()
        mqtt.client.reinitialise()
        mqtt.client.connect(data_connect['host'], data_connect['port'])

        print(f"正在尝试连接到 {data_connect['host']}:{data_connect['port']}...")
        return jsonify({'rc_status': 'success'})

    except Exception as e:
        print(f"连接失败: {e}")
        return jsonify({'rc_status': str(e)})

# 7. (新增) /publish/ 接口
# 用于让前端“请求”虚拟数控系统发送数据
@app.route('/publish/', methods=['POST'])
def do_publish():
    try:
        data = request.get_json()
        topic = data.get('topic')
        payload = data.get('payload', "{}") # payload 应该是 JSON 字符串

        if not topic:
            return jsonify({'status': 'error', 'message': 'Topic is required'}),
400

        print(f"正在发布消息到 (Topic: {topic}): {payload}")
        mqtt.publish(topic, payload)

        return jsonify({'status': 'published', 'topic': topic})
    except Exception as e:
        print(f"发布失败: {e}")
        return jsonify({'status': 'error', 'message': str(e)}), 500

# 8. (新增) /get_data/ 接口
# 用于让前端获取已存储的最新数据
@app.route('/get_data/', methods=['POST'])
def get_data():
    try:
        data = request.get_json()
```

```
        data_id = data.get('id') # 前端将请求特定ID的数据

        if not data_id:
            return jsonify({'status': 'error', 'message': 'ID is required'}), 400

        # 从我们的存储中获取最新值
        value = latest_data_store.get(data_id)

        if value is not None:
            return jsonify({'status': 'ok', 'id': data_id, 'value': value})
        else:
            return jsonify({'status': 'not_found', 'id': data_id, 'value': None})

    except Exception as e:
        print(f"获取数据失败: {e}")
        return jsonify({'status': 'error', 'message': str(e)}), 500

# 9. 启动 Flask 服务
if __name__ == '__main__':
    app.run(debug=True, host='127.0.0.1', port=5000)
```

4.3 前端 (VUE) 界面修改

4.3.1 修改 `CoreSource.vue`（数据采集界面）

我们需要在 `CoreSource.vue` 页面添加“发布数据请求”的功能，以模拟《实验步骤.pdf》P2 中 MQTTX 的“发布”操作。

请**替换** `src/components/CoreSource.vue` 文件的**全部内容**为以下代码：

文件路径： `juiceyang999/class2up/class2up-9730b2fcdf474be54786a1f787cc6b005a26d2fc/src/components/CoreSource.vue`

```
<template>
  <el-container>
    <el-header>
      <h2>数据采集与服务器连接</h2>
    </el-header>
    <el-main>
      <el-row :gutter="20">
        <el-col :span="12">
          <el-card class="box-card">
            <div slot="header" class="clearfix">
              <span>MQTT 服务器连接配置</span>
            </div>
            <el-form ref="form" :model="connection" label-width="120px">
              <el-form-item label="服务器地址">
                <el-input v-model="connection.host"></el-input>
              </el-form-item>
              <el-form-item label="端口号">
                <el-input v-model.number="connection.port" type="number"></el-
input>
              </el-form-item>
              <el-form-item label="Client ID">
```

```
              <el-input v-model="connection.clientid"></el-input>
            </el-form-item>
            <el-form-item>
              <el-button type="primary" @click="onConnect">连接</el-button>
              <el-button @click="onDisconnect" :disabled="!connectStatus">断开
</el-button>
            </el-form-item>
            <el-form-item label="连接状态">
              <el-tag :type="connectStatus ? 'success' : 'danger'">
                {{ connectStatus ? '已连接' : '未连接' }}
              </el-tag>
            </el-form-item>
          </el-form>
        </el-card>
      </el-col>

      <el-col :span="12">
        <el-card class="box-card">
          <div slot="header" class="clearfix">
            <span>(调试) 手动发布数据请求</span>
          </div>
          <el-form label-width="120px">
            <el-form-item label="请求主题">
              <el-input v-model="publish.topic"></el-input>
            </el-form-item>
            <el-form-item label="请求 Payload">
              <el-input type="textarea" :rows="3" v-model="publish.payload">
</el-input>
            </el-form-item>
            <el-form-item>
              <el-button type="warning" @click="onPublish"
:disabled="!connectStatus">
                手动发布
              </el-button>
            </el-form-item>
          </el-form>
          <div style="font-size: 12px; color: #909399;">
            <p>说明：后端连接成功后会自动订阅响应主题。此卡片仅用于手动调试数据请求。</p>
            <p>例如，根据《实验步骤.pdf》P2，Payload 可填：</p>
            <p>{"ids":[{"id":"0103502202"}]}</p>
          </div>
        </el-card>
      </el-col>
    </el-row>
  </el-main>
</el-container>
</template>

<script>
import axios from "axios";
axios.defaults.baseURL = "/api";

export default {
  name: "CoreSource",
  data() {
    return {
```

```
      connection: {
        host: "127.0.0.1",
        port: 1883,
        clientid: "vue_client_" + Math.random().toString(16).substr(2, 8),
      },
      connectStatus: false,
      // (新增) 存储发布信息
      publish: {
        topic: "Query/Request/STRESS_TEST_00000",
        // '0103502202' 是X轴实际速度
        payload: '{"ids":[{"id":"0103502202"}]}'
      }
    };
  },
  methods: {
    onConnect() {
      console.log("正在连接到MQTT服务器...");
      axios
        .post("/connect/", { data: this.connection })
        .then((res) => {
          if (res.data && res.data.rc_status === "success") {
            this.connectStatus = true;
            this.$message.success('MQTT服务器连接成功！');
          } else {
            this.connectStatus = false;
            this.$message.error('连接失败: ' + res.data.rc_status);
          }
        })
        .catch((err) => {
          console.error("连接请求失败:", err);
          this.connectStatus = false;
          this.$message.error('后端服务连接失败，请检查Python后端是否已运行。');
        });
    },
    onDisconnect() {
      // (功能待定，目前仅用于前端演示)
      console.log("断开连接");
      this.connectStatus = false;
      this.$message.info('已断开连接（前端模拟）');
      // 实际开发中，这里也应该向后端发送一个 /disconnect/ 请求
    },
    // (新增) 发布方法
    onPublish() {
      axios.post("/publish/", {
        topic: this.publish.topic,
        payload: this.publish.payload
      })
      .then(res => {
        if (res.data && res.data.status === 'published') {
          this.$message.success('请求已发布！请切换到“数据显示”页面查看。');
        } else {
          this.$message.error('发布失败: ' + res.data.message);
        }
      })
      .catch(err => {
        this.$message.error('发布请求失败: ' + err);
```

```
      });
    }
  },
};
</script>

<style scoped>
.el-header {
  background-color: #b3c0d1;
  color: #333;
  line-height: 60px;
}
.box-card {
  text-align: left;
}
</style>
```

4.3.2 编写 `ShowData.vue`（数据显示界面）

这是本任务的核心文件。我们将在这里初始化 Echarts 并定时请求和刷新数据。

请**替换** `src/components/ShowData.vue` 文件的**全部内容**为以下代码：

文件路径： `juiceyang999/class2up/class2up-9730b2fcdf474be54786a1f787cc6b005a26d2fc/src/components/ShowData.vue`

```
<template>
  <el-container>
    <el-main>
      <el-card>
        <div slot="header">
          <span>实时数据显示 (X轴实际速度)</span>
          <span style="font-size: 12px; color: #909399; margin-left: 20px;">
            (数据ID: {{ dataIdToFetch }})
          </span>
        </div>
        <div id="data-chart" style="width: 100%; height: 400px;"></div>
      </el-card>
    </el-main>
  </el-container>
</template>

<script>
import axios from "axios";
// 2. 引入 Echarts
import * as echarts from "echarts";

axios.defaults.baseURL = "/api";

export default {
  name: "ShowData",
  data() {
    return {
      chart: null, // Echarts 实例
      chartData: [], // 存储图表的数据
```

```
      dataTimer: null, // 存储 setInterval 的ID
      // 3. 我们要抓取的数据ID：X轴实际速度
      // (来自《项目课程2025.pdf》P29, "0103502202" "SPEED" "实际速度")
      dataIdToFetch: '0103502202',
      // 请求发布的主题和payload
      publishConfig: {
        topic: "Query/Request/STRESS_TEST_00000",
        payload: '{"ids":[{"id":"0103502202"}]}' // 动态生成
      },
    };
  },
  mounted() {
    // 4. 页面加载时：初始化图表并开始抓取数据
    // 动态设置我们要请求的ID
    this.publishConfig.payload = `{"ids":[{"id":"${this.dataIdToFetch}"}]}`;

    this.initChart();
    this.startFetching();
  },
  beforeDestroy() {
    // 5. 页面销毁时：停止定时器，防止内存泄漏
    if (this.dataTimer) {
      clearInterval(this.dataTimer);
    }
  },
  methods: {
    // 6. 初始化图表
    initChart() {
      // 对应《实验步骤.pdf》P12/P13的初始化代码
      const chartDom = document.getElementById("data-chart");
      this.chart = echarts.init(chartDom);

      const option = {
        xAxis: {
          type: "category",
          data: [], // X轴数据（时间戳或索引）
        },
        yAxis: {
          type: "value",
          scale: true // 自动缩放
        },
        series: [
          {
            data: [], // Y轴数据（速度值）
            type: "line",
            smooth: true,
          },
        ],
        tooltip: {
          trigger: 'axis'
        }
      };

      this.chart.setOption(option);
    },
```

```
    // 7. 开始轮询
    startFetching() {
      // 每3秒钟执行一次 requestAndFetch
      this.requestAndFetch(); // 立即执行一次
      this.dataTimer = setInterval(this.requestAndFetch, 3000); // 3000毫秒 = 3秒
    },

    // 8. (新增) 组合方法：先请求，再获取
    requestAndFetch() {
      // 8.1. 第一步：向后端发请求，让后端去 "问" 虚拟数控机
      axios.post("/publish/", this.publishConfig)
        .catch(err => {
          console.error("发布数据请求失败:", err);
        });

      // 8.2. 第二步：稍等片刻（例如500ms），再去后端 "取" 数据
      // (给MQTT一个来回的时间)
      setTimeout(() => {
        this.fetchData();
      }, 500);
    },

    // 9. (新增) 从后端获取数据并更新图表
    fetchData() {
      axios.post("/get_data/", { id: this.dataIdToFetch })
        .then(res => {
          if (res.data && res.data.status === 'ok' && res.data.value !== null) {
            const newValue = parseFloat(res.data.value); // 确保是数字

            if (this.chartData.length > 50) {
              this.chartData.shift();
            }
            this.chartData.push(newValue);

            const xAxisData = this.chartData.map((_, index) => index);

            // 9.3. (已修正) 更新 Echarts 图表
            this.chart.setOption({
              xAxis: {
                data: xAxisData,
              },
              series: [
                {
                  data: this.chartData,
                  type: 'line',    // <--- ！！！(修正) 加上这一行
                  smooth: true   // <--- ！！！(修正) 加上这一行
                },
              ],
            });

          } else if (res.data.status === 'not_found') {
            // 这就是你现在在F12里看到的日志
            console.log(`数据ID ${this.dataIdToFetch} 尚未收到.`);
          }
        })
        .catch(err => {
```

```
          console.error("获取数据失败:", err);
        });
    }
  },
};
</script>

<style scoped>
/* 可以在这里添加此页面专属的样式 */
</style>
```

4.4 运行与调试（当您回到主机时）

1. **启动环境（共4个）：** (同任务三)
   - **终端 1：** `emqx start`
   - **终端 2：** `StressTest-INC-Cloud -n 1 -b 0` (保持运行)
   - **终端 3：** `yarn serve` (VUE 前端)
   - **终端 4：** `python pyt/backend.py` (Python 后端)
2. **测试：**
   - 打开浏览器访问 VUE 页面（如 `http://localhost:8080`）。
   - **步骤 1:** 点击导航栏 “**数据采集**”。
   - **步骤 2:** 点击 “**连接**” 按钮。确保提示 “MQTT服务器连接成功！”。
   - **步骤 3:** 点击导航栏 “**数据显示**”。
   - **预期结果：**
     1. 您会看到一个空的 Echarts 图表。
     2. 等待几秒钟...
     3. 图表上开始**实时**出现一条线，并不断向右滚动，显示来自虚拟数控机床的 "X轴实际速度" 数据。
     4. 在 **终端 4** (Python 后端) 中，您会看到 `收到消息...` 和 `已更新数据存储...` 的打印信息。

任务四完成。我们已经成功地从前端请求数据，通过后端与MQTT通信，并将获取到的实时数据展示在了Echarts图表上。

接下来，我们将进行**任务五：基于数据库的数据存储 (MySQL & InfluxDB)。**

任务五：基于数据库的数据存储与管理

此任务的目标是将我们在任务四中获取到的实时数据，同步写入到 **MySQL**（关系型数据库）和 **InfluxDB**（时序数据库）中。所有操作都在后端 `backend.py` 文件中完成，前端 VUE 页面无需修改。

这是一个纯后端的任务，我们将分三步进行：

1. **数据库环境准备。**
2. **安装 Python 依赖库。**
3. **修改 `backend.py` 代码** 以实现双数据库写入。

5.1 数据库环境准备（主机操作步骤）

当您回到主机后，在启动 `backend.py` 之前，必须先配置好两个数据库。

5.1.1 MySQL 数据库配置

根据《实验步骤.pdf》P19-P20，您需要使用 Navicat 等工具完成以下操作：

1. **确保服务运行：** 确保您的 MySQL 服务正在运行。
2. **创建连接：** 在 Navicat 中，创建一个到 `localhost` (主机)、端口 `3306`、用户名为 `root`、并输入您密码的连接。
3. **新建数据库：** 右键点击连接，选择"新建数据库"。
   - 数据库名：`mqtt_data`
   - 字符集：`utf8`
   - 排序规则：`utf8_general_ci`
4. **新建数据表：**
   - 双击进入 `mqtt_data` 数据库，右键点击"表"，选择"新建表"。
   - 根据《实验步骤.pdf》P21的案例，我们创建一个表（我建议添加一个 `id` 主键和 `data_id` 字段以便区分数据）：
     - `id`：类型 `INT`，勾选 `自动递增` (Auto Increment)，设为 `主键`。
     - `data_id`：类型 `VARCHAR(50)` (例如 '0103502202')。
     - `payload`：类型 `VARCHAR(255)` (用于存储数据值)。
     - `time`：类型 `DATETIME` (用于存储时间戳)。
   - 点击"保存"，表名填写：`mac_data`。

5.1.2 InfluxDB (边缘端) 配置

根据《实验步骤.pdf》P14-P15，您需要完成以下操作：
安装Influxdb：
https://docs.influxdata.com/influxdb/v2/install/?t=Windows#download-and-install-influxdb-v2

1. **启动服务：**
   - 打开 `cmd`， `cd` 到您的 InfluxDB 配置文件夹（例如 `E:\MQTT-test\Influxdb配置文件`）。
   - 运行命令：`influxd`
   - **保持此 `cmd` 窗口开启。**
2. **Web端配置：**
   - 打开浏览器，访问 `http://localhost:8086`。
   - 您将看到 InfluxDB 的**首次设置界面。**
   - 设置您的**用户名**、**密码**、**组织名称** (Organization, 例如 `my-org`) 和**存储桶名称** (Bucket, 例如 `mqtt_bucket`)。
3. **获取 Token：**
   - 设置完成后，您会进入 InfluxDB 的主界面。
   - 在左侧菜单栏找到 `Data` -> `API Tokens`。
   - 您会看到一个为您自动生成的 `<username>'s Token`。

- 点击它，然后点击 “Copy to Clipboard” **复制这个 Token。**
- **将这个 Token 字符串**、您的**组织名称(Org)**和存储桶名称(Bucket)**保存好，我们马上要在 `backend.py` 中使用它们。

5.2 安装 Python 依赖库

您需要安装用于连接 InfluxDB v2 的 Python 客户端（`pymysql` 我们在任务三已安装）。

当您回到主机后，请在**终端 4** (运行后端的终端) 中，`cd` 到 VUE 项目根目录，并执行：

```
pip install influxdb-client
```

5.3 后端 (Python) 代码更新

现在，我们将修改 `backend.py`，在 `handle_message` 函数中（即每次收到MQTT消息时）添加将数据写入这两个数据库的逻辑。

请**替换** `pyt/backend.py` 文件的**全部内容**为以下代码：

文件路径： `juiceyang999/class2up/class2up-9730b2fcdf474be54786a1f787cc6b005a26d2fc/pyt/backend.py`

```
from flask import Flask, request, jsonify
from flask_cors import CORS
from flask_mqtt import Mqtt
import json
import pymysql       # (新增) 引入 MySQL 库
import datetime      # (新增) 引入时间库
from influxdb_client import InfluxDBClient, Point, WritePrecision # (新增) 引入
InfluxDB 库
from influxdb_client.client.write_api import SYNCHRONOUS

# --- (新增) 数据库配置 ---
# 1. MySQL 配置 (请根据您的设置修改)
MYSQL_HOST = 'localhost'
MYSQL_USER = 'root'
MYSQL_PASSWORD = 'xjtu2025'  # ！！！请修改为您的MySQL密码！！！
MYSQL_DB = 'mqtt_data'
MYSQL_PORT = 3306

# 2. InfluxDB v2 配置 (请根据您在 5.1.2 中设置和复制的值修改)
INFLUX_URL = "http://localhost:8086"
INFLUX_TOKEN = "YOUR_API_TOKEN_HERE"  # ！！！请修改为您复制的 Token！！！
INFLUX_ORG = "my-org"                 # ！！！请修改为您的组织名称！！！
INFLUX_BUCKET = "mqtt_bucket"         # ！！！请修改为您的存储桶名称！！！

# 3. (新增) 初始化 InfluxDB 客户端
try:
    influx_client = InfluxDBClient(url=INFLUX_URL, token=INFLUX_TOKEN,
org=INFLUX_ORG)
    # 使用同步写入模式
    write_api = influx_client.write_api(write_options=SYNCHRONOUS)
```

```
    print("InfluxDB 客户端初始化成功。")
except Exception as e:
    print(f"InfluxDB 客户端初始化失败: {e}")
# --------------------------

# 初始化 Flask 应用
app = Flask(__name__)
CORS(app)

# 配置 Flask-MQTT
app.config['MQTT_BROKER_URL'] = '127.0.0.1'
app.config['MQTT_BROKER_PORT'] = 1883
app.config['MQTT_CLIENT_ID'] = 'flask_mqtt_client'
app.config['MQTT_KEEPALIVE'] = 60
app.config['MQTT_TLS_ENABLED'] = False

mqtt = Mqtt(app)

# 暂存最新数据
latest_data_store = {}

# --- (新增) 数据库写入函数 ---

def save_to_mysql(data_id, value):
    """
    将数据保存到 MySQL 数据库
    """
    conn = None
    cur = None
    try:
        # 获取当前时间
        time_str = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")

        # 建立连接
        conn = pymysql.connect(
            host=MYSQL_HOST,
            user=MYSQL_USER,
            password=MYSQL_PASSWORD,
            database=MYSQL_DB,
            port=MYSQL_PORT,
            charset='utf8'
        )
        # 创建游标
        cur = conn.cursor()

        # SQL 插入语句 (使用参数化查询防止SQL注入)
        sql = "INSERT INTO mac_data (data_id, payload, time) VALUES (%s, %s, %s)"

        # 执行 SQL
        cur.execute(sql, (data_id, str(value), time_str))

        # 提交事务
        conn.commit()
        print(f"MySQL 写入成功: ID={data_id}, Value={value}")

    except Exception as e:
```

```
        print(f"MySQL 写入失败: {e}")
        if conn:
            conn.rollback() # 出错时回滚
    finally:
        # 关闭游标和连接
        if cur:
            cur.close()
        if conn:
            conn.close()

def save_to_influxdb(data_id, value):
    """
    将数据保存到 InfluxDB
    """
    try:
        # 转换为 float，InfluxDB 推荐使用数值类型
        value_float = float(value)

        # 创建一个数据点 (Point)
        p = Point("machine_data") \
            .tag("device_id", "STRESS_TEST_00000") \
            .tag("data_id", data_id) \
            .field("value", value_float) \
            .time(datetime.datetime.utcnow(), WritePrecision.NS)

        # 写入数据
        write_api.write(bucket=INFLUX_BUCKET, org=INFLUX_ORG, record=p)
        print(f"InfluxDB 写入成功: ID={data_id}, Value={value_float}")

    except Exception as e:
        print(f"InfluxDB 写入失败: {e}")

# --- MQTT 事件回调 ---

@mqtt.on_connect()
def handle_connect(client, userdata, flags, rc):
    if rc == 0:
        print("MQTT 连接成功 (rc=0)")
        response_topic = "Query/Response/STRESS_TEST_00000"
        mqtt.subscribe(response_topic)
        print(f"已自动订阅主题: {response_topic}")
    else:
        print(f"MQTT 连接失败，返回码: {rc}")

@mqtt.on_message()
def handle_message(client, userdata, message):
    # 4. (更新) 当收到消息时，解析、存储、并写入数据库
    try:
        payload_str = message.payload.decode()
        print(f"收到消息 (Topic: {message.topic}): {payload_str}")

        data = json.loads(payload_str)

        if 'values' in data and isinstance(data['values'], list):
            for item in data['values']:
                if 'id' in item and 'values' in item:
```

```
                    data_id = item['id']
                    value = item['values'][0] # 假设我们只关心第一个值

                    # 4.1. 暂存数据 (供前端Echarts使用)
                    latest_data_store[data_id] = value
                    print(f"已更新数据存储: {latest_data_store}")

                    # 4.2. (新增) 写入 MySQL
                    save_to_mysql(data_id, value)

                    # 4.3. (新增) 写入 InfluxDB
                    save_to_influxdb(data_id, value)

    except Exception as e:
        print(f"处理消息失败: {e}")

# --- Flask API 路由 (保持不变) ---

@app.route('/connect/', methods=['POST', 'GET'])
def make_connect():
    try:
        data_connect = request.get_json()['data']
        print(f"收到来自前端的连接请求: {data_connect}")
        app.config['MQTT_BROKER_URL'] = data_connect['host']
        app.config['MQTT_BROKER_PORT'] = data_connect['port']
        app.config['MQTT_CLIENT_ID'] = data_connect['clientid']
        if mqtt.client.is_connected():
            mqtt.client.disconnect()
        mqtt.client.username_pw_set(None, None)
        mqtt.client._client_id = data_connect['clientid'].encode()
        mqtt.client.reinitialise()
        mqtt.client.connect(data_connect['host'], data_connect['port'])
        print(f"正在尝试连接到 {data_connect['host']}:{data_connect['port']}...")
        return jsonify({'rc_status': 'success'})
    except Exception as e:
        print(f"连接失败: {e}")
        return jsonify({'rc_status': str(e)})

@app.route('/publish/', methods=['POST'])
def do_publish():
    try:
        data = request.get_json()
        topic = data.get('topic')
        payload = data.get('payload', "{}")
        if not topic:
            return jsonify({'status': 'error', 'message': 'Topic is required'}),
400
        print(f"正在发布消息到 (Topic: {topic}): {payload}")
        mqtt.publish(topic, payload)
        return jsonify({'status': 'published', 'topic': topic})
    except Exception as e:
        print(f"发布失败: {e}")
        return jsonify({'status': 'error', 'message': str(e)}), 500

@app.route('/get_data/', methods=['POST'])
def get_data():
```

```
    try:
        data = request.get_json()
        data_id = data.get('id')
        if not data_id:
            return jsonify({'status': 'error', 'message': 'ID is required'}), 400
        value = latest_data_store.get(data_id)
        if value is not None:
            return jsonify({'status': 'ok', 'id': data_id, 'value': value})
        else:
            return jsonify({'status': 'not_found', 'id': data_id, 'value': None})
    except Exception as e:
        print(f"获取数据失败: {e}")
        return jsonify({'status': 'error', 'message': str(e)}), 500

# 启动 Flask 服务
if __name__ == '__main__':
    app.run(debug=True, host='127.0.0.1', port=5000)
```

5.4 运行与调试（当您回到主机时）

1. **启动环境（共 5 个）：**
   - **终端 1：** 运行 `emqx start` (MQTT Broker)。
   - **终端 2：** 运行 `StressTest-INC-Cloud -n 1 -b 0` (数据源)。
   - **终端 3：** 运行 `influxd` (InfluxDB 数据库)。
   - **终端 4：** 运行 `yarn serve` (VUE 前端)。
   - **终端 5：** **(重要!)** 确保您已将 `backend.py` 中的 `MYSQL_PASSWORD` 和 `INFLUX_TOKEN` 修改为您自己的配置。然后运行 `python pyt/backend.py` (Python 后端)。
2. **测试：**
   - 打开浏览器访问 VUE 页面（如 `http://localhost:8080`）。
   - 点击 “**数据采集**” 页面，点击 “**连接**”，确保连接成功。
   - 点击 “**数据显示**” 页面，您会看到 Echarts 图表开始实时显示数据。
3. **验证存储：**
   - **MySQL 验证：**
     - 打开 **Navicat**。
     - 刷新 `mqtt_data` 数据库下的 `mac_data` 表。
     - **预期结果：** 您会看到表中**每隔几秒就新增一行数据**，包含 `data_id` ('0103502202')、`payload` (速度值) 和 `time` (时间戳)。
   - **InfluxDB 验证：**
     - 打开浏览器访问 `http://localhost:8086`。
     - 点击左侧菜单的 `Explore` (Data Explorer)。
     - 在查询构建器 (Query Builder) 中：
       - **FROM:** 选择您的存储桶 `mqtt_bucket`。
       - **Filter (Measurement):** 选择 `machine_data`。
       - **Filter (Field):** 选择 `value`。

        - **Filter (Tag):** 选择 `data_id` = `0103502202`。
    - 点击 “**Submit**”。
    - **预期结果：** 您会看到一个与 VUE 页面中 Echarts 几乎一样的实时图表，证明时序数据已成功写入。

补充：云端 Influxdb 时序数据库存储

总述：
打开云端 Influxdb 时序数据库服务网页 https://us-east-1-1.aws.cloud2.influxdata.com，注册账号登录。点击 buckets 可以设置数据存储桶，与边缘端相同。点击 source-node 进入配置界面。根据指引可以进行编程，也可参考本地数据库配置代码，功能相同。在 Dataexplorer中可以查看上传的数据。

见“任务五（补充）：搭建云端inflexdb.md”

任务五（补充）：搭建云端inflexdb

我们将分四步走：

1. **注册 InfluxDB Cloud 并创建 Bucket (在浏览器)**
2. **获取云端 Token 和 Org (在浏览器)**
3. **修改 `pyt/backend.py` 代码 (在 VSCode)**
4. **运行和验证**

步骤 1: 注册和创建 Bucket (在 InfluxDB Cloud 网站)

1. **打开网页**:
   打开《实验步骤.pdf》中提供的云端服务网页：`https://us-east-1-1.aws.cloud2.influxdata.com`。
2. **注册账号**:
    - 点击 "Sign Up" 或 "Get Started"。
    - 填写你的邮箱、密码、公司名（随便填）等信息完成注册。
    - 登录后，它可能会让你选择一个云服务商（AWS, GCP, Azure）和区域（Region），**你可以随便选一个免费的**（比如 AWS 的 us-east-1）。
3. **创建 Bucket**:
    - 登录成功后，你会进入 InfluxDB Cloud 的主界面。
    - 在左侧菜单栏，找到一个像 “桶” 一样的图标，点击 “**Load Data**” -> “**Buckets**”。
    - 点击右上角的 “**+ Create Bucket**”。
    - 给你的云端存储桶起一个新名字，例如：`my_cloud_bucket`。（不要和本地的 `mqtt_bucket` 搞混了）
    - 点击 "Create"。

步骤 2: 获取云端 Token 和 Org (在 InfluxDB Cloud 网站)

现在，你需要 3 个关键信息才能让 Python 连接到云端。**请把这 3 个信息复制到一个记事本里**：

1. **你的 Token (API 令牌)**：
   - 在左侧菜单栏，点击 “**Load Data**” -> “**API Tokens**”。
   - 你会看到一个已经存在的、以你名字命名的 Token（例如 `Your_Name's Token`）。
   - **点击这个 Token 的名字。**
   - 点击 “**Copy to Clipboard**”（复制到剪贴板）按钮。
   - **这就是你的** `INFLUX_CLOUD_TOKEN`。
2. **你的 Org (组织名称)**：
   - 你的组织名称通常显示在**左上角**，在你的邮箱地址下面。
   - **这就是你的** `INFLUX_CLOUD_ORG`。
3. **你的 URL (云端地址)**：
   - 在左侧菜单栏，点击最下方的“个人”图标，然后选择 “**About**”。
   - 你会看到 “Cloud URL”，它就是 `https://us-east-1-1.aws.cloud2.influxdata.com`（或者根据你选择的区域而定）。
   - **这就是你的** `INFLUX_CLOUD_URL`。

步骤 3: 修改 `pyt/backend.py` 代码 (在 VSCode)

现在我们来修改 Python 后端，让它**同时**向**本地 InfluxDB** 和**云端 InfluxDB** 写入数据。

请**替换** `pyt/backend.py` 文件的**全部内容**为以下代码：

文件路径： `.../class2up/pyt/backend.py`

```
from flask import Flask, request, jsonify
from flask_cors import CORS
from flask_mqtt import Mqtt
import json
import pymysql
import datetime
from influxdb_client import InfluxDBClient, Point, WritePrecision
from influxdb_client.client.write_api import SYNCHRONOUS
import requests
import atexit
import joblib
import numpy as np

# --- 数据库配置 ---
# 1. MySQL 配置 (保持不变)
MYSQL_HOST = 'localhost'
MYSQL_USER = 'root'
MYSQL_PASSWORD = 'xjtu2025'  # ！！！请修改为您的MySQL密码！！！
MYSQL_DB = 'mqtt_data'
MYSQL_PORT = 3306

# 2. InfluxDB (本地) 配置 (保持不变)
```

```
INFLUX_LOCAL_URL = "http://localhost:8086"
INFLUX_LOCAL_TOKEN = "YOUR_API_TOKEN_HERE"  # ！！！请修改为您的本地 Token！！！
INFLUX_LOCAL_ORG = "my-org"                 # ！！！请修改为您的本地组织！！！
INFLUX_LOCAL_BUCKET = "mqtt_bucket"         # ！！！请修改为您的本地存储桶！！！

# 3. (新增) InfluxDB (云端) 配置
# ！！！ 请用你在 步骤 2 中复制的信息替换这些占位符 ！！！
INFLUX_CLOUD_URL = "https://us-east-1-1.aws.cloud2.influxdata.com"
INFLUX_CLOUD_TOKEN = "YOUR_CLOUD_TOKEN_HERE" # ！！！请修改为您的云端 Token！！！
INFLUX_CLOUD_ORG = "YOUR_CLOUD_ORG_HERE"     # ！！！请修改为您的云端 Org！！！
INFLUX_CLOUD_BUCKET = "my_cloud_bucket"      # (你刚刚创建的云端 Bucket)

# --- (新增) HDFS 平台配置 (占位符) ---
HDFS_WEB_API_URL = "http://YOUR_HADOOP_HOST:9870/webhdfs/v1"
HDFS_TARGET_PATH = "/user/your_name/machine_data.csv"
HDFS_USER = "your_username"

# --- (新增) ML 模型配置 ---
MODEL_PATH = 'pyt/status_predictor.pkl'
DATA_WINDOW_SIZE = 50
model = None

# --- 初始化所有客户端 ---
try:
    # (修改) 初始化本地 InfluxDB 客户端
    influx_client_local = InfluxDBClient(url=INFLUX_LOCAL_URL,
token=INFLUX_LOCAL_TOKEN, org=INFLUX_LOCAL_ORG)
    write_api_local = influx_client_local.write_api(write_options=SYNCHRONOUS)
    print("InfluxDB 本地客户端初始化成功。")
except Exception as e:
    print(f"InfluxDB 本地客户端初始化失败: {e}")

try:
    # (新增) 初始化云端 InfluxDB 客户端
    influx_client_cloud = InfluxDBClient(url=INFLUX_CLOUD_URL,
token=INFLUX_CLOUD_TOKEN, org=INFLUX_CLOUD_ORG)
    write_api_cloud = influx_client_cloud.write_api(write_options=SYNCHRONOUS)
    print("InfluxDB 云端客户端初始化成功。")
except Exception as e:
    print(f"InfluxDB 云端客户端初始化失败: {e}")

# (新增) 加载 ML 模型
try:
    model = joblib.load(MODEL_PATH)
    print(f"机器学习模型 {MODEL_PATH} 加载成功。")
except Exception as e:
    print(f"模型加载失败: {e} (请先运行 _create_dummy_model.py)")

# HDFS 本地临时文件
local_temp_file_path = "hdfs_temp_data.csv"
local_file = None
try:
    local_file = open(local_temp_file_path, "a", encoding="utf-8")
    if local_file.tell() == 0:
        local_file.write("timestamp,data_id,value\n")
    print(f"打开本地临时文件 {local_temp_file_path} 成功。")
```

```
except Exception as e:
    print(f"打开本地临时文件失败: {e}")

@atexit.register
def upload_to_hdfs_on_exit():
    # ... (此函数保持不变) ...
    global local_file
    if local_file:
        local_file.close()
        print(f"程序退出，正在将 {local_temp_file_path} 上传到 HDFS...")
        try:
            create_url = f"{HDFS_WEB_API_URL}{HDFS_TARGET_PATH}?
op=CREATE&overwrite=true&user.name={HDFS_USER}"
            r_create = requests.put(create_url, allow_redirects=False)
            if r_create.status_code == 307:
                data_node_url = r_create.headers['Location']
                with open(local_temp_file_path, "rb") as f:
                    r_upload = requests.put(data_node_url, data=f)
                    if r_upload.status_code == 201:
                        print(f"HDFS 文件上传成功: {HDFS_TARGET_PATH}")
                    else:
                        print(f"HDFS 文件上传失败: {r_upload.status_code} -
{r_upload.text}")
            else:
                 print(f"HDFS 'CREATE' 请求失败: {r_create.status_code} -
{r_create.text}")
        except Exception as e:
            print(f"HDFS 上传时发生严重错误: {e}")

# --- Flask 和 MQTT 初始化 (保持不变) ---
app = Flask(__name__)
CORS(app)
app.config['MQTT_BROKER_URL'] = '127.0.0.1'
app.config['MQTT_BROKER_PORT'] = 1883
app.config['MQTT_CLIENT_ID'] = 'flask_mqtt_client'
mqtt = Mqtt(app)

# --- 数据存储 (保持不变) ---
latest_data_store = {}
data_window_store = {}

# --- 数据库写入函数 ---
def save_to_mysql(data_id, value, time_str):
    # ... (此函数保持不变) ...
    conn = None
    cur = None
    try:
        conn = pymysql.connect(host=MYSQL_HOST, user=MYSQL_USER,
password=MYSQL_PASSWORD, database=MYSQL_DB, port=MYSQL_PORT, charset='utf8')
        cur = conn.cursor()
        sql = "INSERT INTO mac_data (data_id, payload, time) VALUES (%s, %s, %s)"
        cur.execute(sql, (data_id, str(value), time_str))
        conn.commit()
        print(f"MySQL 写入成功: ID={data_id}, Value={value}")
    except Exception as e:
        print(f"MySQL 写入失败: {e}")
```

```
    finally:
        if cur: cur.close()
        if conn: conn.close()

def save_to_influxdb_local(data_id, value, time_obj): # (函数名修改)
    try:
        value_float = float(value)
        p = Point("machine_data").tag("device_id",
"STRESS_TEST_00000").tag("data_id", data_id).field("value",
value_float).time(time_obj, WritePrecision.NS)
        write_api_local.write(bucket=INFLUX_LOCAL_BUCKET, org=INFLUX_LOCAL_ORG,
record=p) # (变量名修改)
        print(f"InfluxDB 本地 写入成功: ID={data_id}, Value={value_float}")
    except Exception as e:
        print(f"InfluxDB 本地 写入失败: {e}")

# (新增) 写入 InfluxDB 云端
def save_to_influxdb_cloud(data_id, value, time_obj):
    try:
        value_float = float(value)
        p = Point("machine_data_cloud") \
            .tag("device_id", "STRESS_TEST_00000") \
            .tag("data_id", data_id) \
            .field("value", value_float) \
            .time(time_obj, WritePrecision.NS)

        # 使用云端配置的 API 写入
        write_api_cloud.write(bucket=INFLUX_CLOUD_BUCKET, org=INFLUX_CLOUD_ORG,
record=p)
        print(f"InfluxDB 云端 写入成功: ID={data_id}, Value={value_float}")
    except Exception as e:
        print(f"InfluxDB 云端 写入失败: {e}")

def save_to_distributed_platform(data_id, value, time_obj):
    # ... (此函数保持不变) ...
    global local_file
    if local_file:
        try:
            time_iso = time_obj.isoformat()
            line = f"{time_iso},{data_id},{value}\n"
            local_file.write(line)
            local_file.flush()
        except Exception as e:
            print(f"本地临时文件写入失败: {e}")

# --- MQTT 回调 (更新) ---
@mqtt.on_connect()
def handle_connect(client, userdata, flags, rc):
    # ... (此函数保持不变) ...
    if rc == 0:
        print("MQTT 连接成功 (rc=0)")
        response_topic = "Query/Response/STRESS_TEST_00000"
        mqtt.subscribe(response_topic)
        print(f"已自动订阅主题: {response_topic}")
    else:
        print(f"MQTT 连接失败，返回码: {rc}")
```

```python
@mqtt.on_message()
def handle_message(client, userdata, message):
    # (更新)
    try:
        payload_str = message.payload.decode()
        data = json.loads(payload_str)

        time_now_utc = datetime.datetime.utcnow()
        time_now_local_str = datetime.datetime.now().strftime("%Y-%m-%d
%H:%M:%S")

        if 'values' in data and isinstance(data['values'], list):
            for item in data['values']:
                if 'id' in item and 'values' in item:
                    data_id = item['id']
                    value = item['values'][0]

                    # 1. 暂存 (供 Echarts)
                    latest_data_store[data_id] = value

                    # 2. 存储数据窗口 (供 ML)
                    if data_id not in data_window_store:
                        data_window_store[data_id] = []
                    data_window_store[data_id].append(float(value))
                    if len(data_window_store[data_id]) > DATA_WINDOW_SIZE:
                        data_window_store[data_id].pop(0)

                    # 3. 写入 MySQL
                    save_to_mysql(data_id, value, time_now_local_str)

                    # 4. 写入 InfluxDB (本地)
                    save_to_influxdb_local(data_id, value, time_now_utc)

                    # 5. (新增) 写入 InfluxDB (云端)
                    save_to_influxdb_cloud(data_id, value, time_now_utc)

                    # 6. 写入 HDFS 缓存
                    save_to_distributed_platform(data_id, value, time_now_utc)

    except Exception as e:
        print(f"处理消息失败: {e}")

# --- Flask API 路由 (保持不变) ---
@app.route('/connect/', methods=['POST', 'GET'])
def make_connect():
    # ... (此函数保持不变) ...
    try:
        data_connect = request.get_json()['data']
        app.config['MQTT_BROKER_URL'] = data_connect['host']
        app.config['MQTT_BROKER_PORT'] = data_connect['port']
        app.config['MQTT_CLIENT_ID'] = data_connect['clientid']
        if mqtt.client.is_connected():
            mqtt.client.disconnect()
        mqtt.client.username_pw_set(None, None)
        mqtt.client._client_id = data_connect['clientid'].encode()
```

```
        mqtt.client.reinitialise()
        mqtt.client.connect(data_connect['host'], data_connect['port'])
        return jsonify({'rc_status': 'success'})
    except Exception as e:
        return jsonify({'rc_status': str(e)})

@app.route('/publish/', methods=['POST'])
def do_publish():
    # ... (此函数保持不变) ...
    try:
        data = request.get_json()
        topic = data.get('topic')
        payload = data.get('payload', "{}")
        mqtt.publish(topic, payload)
        return jsonify({'status': 'published', 'topic': topic})
    except Exception as e:
        return jsonify({'status': 'error', 'message': str(e)}), 500

@app.route('/get_data/', methods=['POST'])
def get_data():
    # ... (此函数保持不变) ...
    global latest_data_store
    try:
        data = request.get_json()
        data_id = data.get('id')
        value = latest_data_store.get(data_id)
        if value is not None:
            return jsonify({'status': 'ok', 'id': data_id, 'value': value})
        else:
            return jsonify({'status': 'not_found', 'id': data_id, 'value': None})
    except Exception as e:
        return jsonify({'status': 'error', 'message': str(e)}), 500

@app.route('/predict/', methods=['POST'])
def predict_status():
    # ... (此函数保持不变) ...
    global model
    if model is None:
        return jsonify({'status': 'error', 'message': '模型未加载'}), 500
    try:
        data = request.get_json()
        data_id = data.get('id')
        window_data = data_window_store.get(data_id)
        if not window_data or len(window_data) < DATA_WINDOW_SIZE:
            return jsonify({'status': 'wait', 'message': f'数据采集中，当前
{len(window_data) if window_data else 0}/{DATA_WINDOW_SIZE} 点，请稍候...'})
        features_vector = np.array(window_data).reshape(1, -1)
        prediction_code = model.predict(features_vector)
        status_map = {0: '设备状态正常', 1: '设备故障预警'}
        prediction_text = status_map.get(prediction_code[0], '未知状态')
        return jsonify({'status': 'ok', 'prediction': prediction_text})
    except Exception as e:
        return jsonify({'status': 'error', 'message': str(e)}), 500

# 启动 Flask 服务
if __name__ == '__main__':
```

```
    app.run(debug=True, host='127.0.0.1', port=5000)
```

步骤 4: 运行和验证

1. **保存代码**:
   - 确保你把从 InfluxDB Cloud 网站复制的 `URL`, `TOKEN`, 和 `ORG` **正确地**粘贴到了 `backend.py` 的 `INFLUX_CLOUD_...` 变量里。
2. **重启后端**:
   - **停止**你的 Python 后端（终端 5）(按 `Ctrl+C`)。
   - **重新运行** `python pyt/backend.py`。
   - **（重要！）** 检查终端输出，你现在应该看到**两个**成功信息:

     ```
     InfluxDB 本地客户端初始化成功。
     InfluxDB 云端客户端初始化成功。
     ```

   - 如果云端初始化失败，请回去**仔细检查**你的 URL, Token 和 Org 是不是复制错了。
3. **验证写入**:
   - 让所有服务（EMQX, StressTest, Influxd, VUE, Python）运行一两分钟。
   - 你的 Python 终端（终端 5）现在应该**同时**打印:

     ```
     InfluxDB 本地 写入成功: ...
     InfluxDB 云端 写入成功: ...
     ```

4. **在云端查看数据**:
   - 回到你的 **InfluxDB Cloud 浏览器窗口**。
   - 点击左侧菜单的 “**Data Explorer**” (或 "Explore")。
   - 在查询构建器 (Query Builder) 中:
     - **FROM:** 选择你的云端存储桶 `my_cloud_bucket`。
     - **Filter (Measurement):** 选择 `machine_data_cloud`。
     - **Filter (Field):** 选择 `value`。
   - 点击 “**Submit**”。

预期结果:
你会在 **InfluxDB Cloud 网站**上看到一个**一模一样**的实时数据图表，证明你本地的数据已经成功同步到云端了!

任务五完成。我们现在已经具备了数据采集、前后端通信、实时可视化和双数据库存储的能力。

接下来，我们将进行**任务六：大数据分布式存储**，这需要我们将 MySQL 中的数据对接到工业大数据平台（Hadoop）。

在此之前，对任务一到任务五遇到的问题进行一个小的回顾与总结:

任务一到六遇到主要问题:

阶段一：VUE 前端启动与配置（“我的网页跑不起来”）

这是我们最早遇到的问题，我们当时的目标是先把 VUE 框架跑起来。

问题 1： `'vue-cli-service' 不是内部或外部命令...`

- **你遇到的现象：** 在终端里运行 `npm run serve`，直接报错。
- **背后的原因：**
  1. 你用了 `npm`，但项目文件里有 `yarn.lock`，说明它是个 `yarn` 项目。
  2. 你没有运行 `yarn install`，所以 `node_modules` 文件夹是空的，系统根本找不到 `vue-cli-service` 这个程序。
- **解决方案：** 严格按照 `README.md` 文件，先运行 `yarn install` 安装依赖，再运行 `yarn serve` 启动。

问题 2： `Module not found: Error: Can't resolve 'element-ui', 'vue-router'` **等...**

- **你遇到的现象：** `yarn serve` 运行时，编译失败，报了一大堆“模块未找到”的错误。
- **背后的原因：** 我们在“任务一”里给 `main.js` 和 `router.js` 添加了 `import ElementUI` 和 `import VueRouter`，但是我们**只 `import`（引用）了**，却**没有真正把这些库“安装”到项目里。**
- **解决方案：** 运行 `yarn add vue-router@3 element-ui axios echarts`，把我们代码中用到的所有新依赖一次性全装上。

问题 3：浏览器显示 `Proxy error: ... ECONNREFUSED`

- **你遇到的现象：** VUE 页面能打开，但F12控制台（或终端）报“代理错误，连接被拒绝”。
- **背后的原因：** `ECONNREFUSED` = 连接被拒绝。你的 VUE 前端（`localhost:8080`）正在尝试访问 `vue.config.js` 里配置的后端地址（`http://127.0.0.1:5000`），但那个地址上**什么服务都没有。**
- **解决方案：** 启动我们的 **Python 后端服务。**这证明了前后端分离项目，必须**至少**运行两个终端：一个 `yarn serve` (前端)，一个 `python pyt/backend.py` (后端)。

阶段二：Python 后端与环境配置（“我的后端也跑不起来”）

问题 4： `ModuleNotFoundError: No module named 'influxdb_client'`

- **你遇到的现象：** `python pyt/backend.py` 启动失败，报“模块未找到”。
- **背后的原因：** 和 VUE 的问题一样，我们 `import influxdb_client` 了，但你的 Python 环境里没有安装它。
- **解决方案：** 运行 `pip install influxdb-client`（以及 Flask, pymysql 等所有我们需要的库）。

问题 5：MySQL 报错 `1045 - Access denied for user 'root'`

- **你遇到的现象：** 你用 Navicat 测试连接 MySQL 失败。
- **背后的原因：** 你在 Navicat 里填的密码，和你电脑上安装 MySQL 时设置的 `root` 密码**不一致。**
- **解决方案：** 在 `pyt/backend.py` 和 Navicat 中，都统一改成你**正确的** MySQL `root` 密码。

问题 6： `'influxd' 不是内部或外部命令...`

- **你遇到的现象：** 启动 InfluxDB 数据库服务时，系统找不到 `influxd`。

- **背后的原因：**
  1. 《实验步骤.pdf》里的路径 `E:\MQTT-test` 是老师电脑上的，你电脑上没有。
  2. `influxd` 是一个**独立的服务程序**，和我们用 `pip` 装的 `influxdb-client`（库）是两回事。
  3. 你在 `cmd` 里从 C 盘切换到 D 盘时，命令没用对（你用了 `cd D:\...`，但正确姿势是先输 `D:` 再 `cd`）。
- **解决方案：**
  1. 从 InfluxDB 官网下载 `.zip` 包并解压（比如到 `D:\code9\class2down\influxd`）。
  2. 在 `cmd` 中先用 `D:` 切换盘符，再用 `cd D:\code9\class2down\influxd` 进入目录。
  3. 运行 `influxd` 启动服务。

阶段三：联调与“灵异事件”（“为什么它就是不工作？！”）

这是最“阴”的部分，也是我们刚刚才解决的！你所有的服务（5个终端）都跑起来了，但 VUE 页面就是没数据``。

问题 7：“无限重启” Bug (来自 `image_2b1be4.png`)

- **你遇到的现象：** Python 后端（终端5）不停地重复打印 `MQTT 连接成功``。
- **背后的原因：** 我们在 `pyt/backend.py` 的最后一行用了 `app.run(debug=True)`。
  - `debug=True` 会“监视”你项目里的所有文件，文件一改动，它就**自动重启**服务。
  - 我们的代码**恰好**会往项目里的 `pyt/hdfs_temp_data.csv` 文件**写数据**。
  - 这就导致：后端收到数据 -> 写入文件 -> 监视器发现文件被改 -> **立刻重启** -> 重启后打印 `MQTT 连接成功`` -> 收到下一条数据... 如此循环。
- **解决方案：** 修改 `pyt/backend.py` 的最后一行，改成 `app.run(debug=True, use_reloader=False)`，**关掉**这个自动重启功能。

问题 8：“图表没线” Bug (来自 `image_2b90d9.png`)

- **你遇到的现象：** VUE 页面图表是空的，F12 控制台显示 `数据ID ... 尚未收到`。
- **背后的原因（这个 Bug 有两个“元凶”）：**
  1. **Python 端的元凶**：`debug=True` 导致后端无限重启（问题7），`latest_data_store` 变量刚存上数据（比如 `10.9526`）就被**清空**了。VUE 来取数据时，库里永远是空的，所以后端返回 `not_found`（“尚未收到”）。
  2. **VUE 端的元凶**：（即使后端好了）`ShowData.vue` 里的 Echarts `setOption` 函数**忘记**加 `type: 'line'` 了。Echarts 拿到数据但不知道该画什么图，所以不画。
- **解决方案：**
  1. 解决“无限重启” Bug（问题7）。
  2. 在 `ShowData.vue` 里的 `setOption` 中**必须**加上 `type: 'line'` 和 `smooth: true`。

问题 9：InfluxDB 数据库没有数据 (来自 `image_2b8cfc.png` 和 `image_2b909d.png`)

- **你遇到的现象：** 本地和云端的 InfluxDB 都是 `No Results`，但后端日志却说 InfluxDB 写入成功`。
- **背后的原因（这个 Bug 也有两个“元凶”）：**

1. **无限重启的元凶**（问题7）：后端日志打印 `写入成功` 只是代表它**调用了**写入函数，但还没等数据包真正通过网络发到数据库，服务就被**重启**了，所以数据库啥也没收到。
2. **Token 格式的元凶**：你 `backend.py` 里的**云端 Token** 多了 `INFLUXDB_TOKEN=` 字样，导致云端连接认证失败。

- **解决方案：**
  1. 解决“无限重启” Bug（问题7）。
  2. **清理 Token**，把 `INFLUX_CLOUD_TOKEN = "INFLUXDB_TOKEN=zsOj..."` 改成 `INFLUX_CLOUD_TOKEN = "zsOj..."`。