基于 Web 在线监测系统软件设计文档

1. 引言

1.1 目的

本文档旨在详细描述用于工业设备状态监测系统的设计和架构，该系统应用了深度学习技术来实现工业设备状态监测及实时故障分析。

1.2 范围

功能概述

本监测系统旨在实现实时数据获取、处理、分析及展示，以便对工业设备的状态进行实时监控。系统将收集传感器数据，使用深度学习模型进行分析以预测设备潜在故障，并提供直观的用户界面以供监控与管理。

主要组件

系统的主要组件如下：

- 前端用户界面：基于 VUE2 开发的动态网页应用程序，提供用户与监控系统交互的界面。
- web 后端服务：Python 编写的服务后台，负责处理业务逻辑、数据管理以及模型推理。
- 数据库系统：使用 Mysql8.0 关系数据库，用于存储系统配置、用户信息、设备状态、历史数据等。
- 深度学习模型：基于 Pytorch 框架训练的模型，用于从传感器数据中分析设备状态。

系统功能

系统的核心功能包括：

- 传感器数据采集：通过第三方接口获取传感器实时波形数据，包含温度和振动等类型
- 数据存储：实现传感器实时数据流的存储、检索和备份。
- 历史数据读取：根据时间、测点等多维度检索历史数据。
- 波形显示：展示每个传感器的温度和振动信号，支持时间范围选择和缩放。

- 故障预测与警报：使用深度学习模型进行故障预测，并将预测结果显示在界面上。
- 人员管理：提供用户认证、用户信息管理等。

2. 总体描述

2.1 系统环境

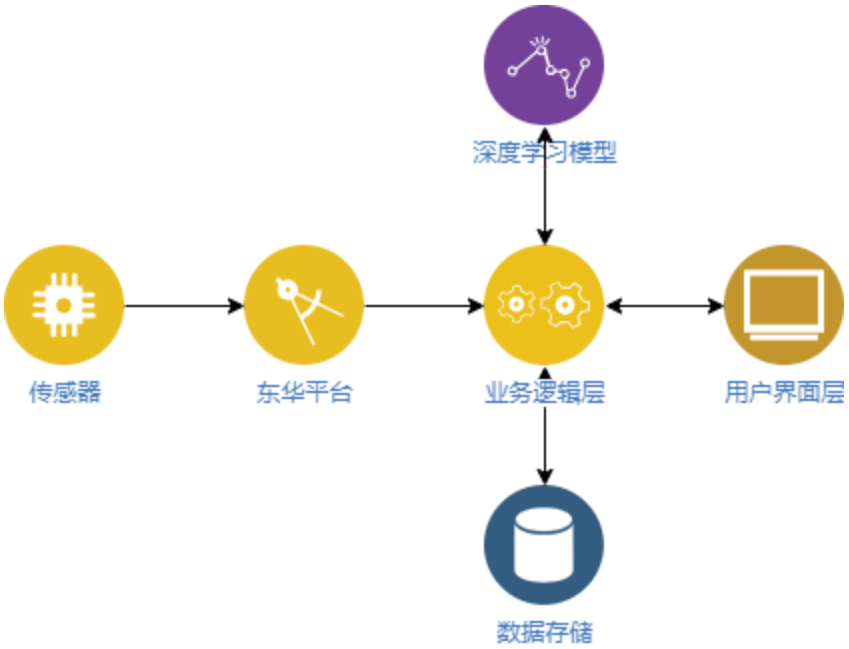
系统运行的主机配置为：i7-7700cpu@3.6g，16G 内存，操作系统为 Windows 10 LTSC 版本，网络环境为只能访问企业内网。主机登录名 work，密码 uestc

2.2 系统架构 / 设计

平台架构概述

该工业设备状态监测系统采用 B/S 架构，客户端使用 Web 浏览器与系统交互，后端服务器处理业务逻辑、数据存储及机器学习模型的推理。系统的部署考虑了可扩展性、安全性和高可用性。

主要架构组件



以下是系统的主要架构组件和它们的交互：

1. 用户界面层（Front-end）：
   - 使用 VUE2 构建的 SPA（单页面应用程序）允许用户与系统交互。
   - 前端通过 API 与后端服务器通信。
   - 主要功能模块包括用户登录界面、实时监测界面、历史数据展示、和系统配置。

2. 业务逻辑层（Back-end）：

- 由 Python Flask 框架构建的 RESTful API 服务器提供安全的数据访问点。

- API 服务器处理来自前端的请求并返回数据或执行命令。

- 后台服务器将定期从传感器接口收集数据，处理后存入数据库，并提供对深度学习模型的访问。

3. 深度学习模型层（Deep Learning Models）：

- 使用 PyTorch 框架训练的深度学习模型负责数据的实时分析和故障预警。

- 模型根据传感器的数据输出状态评估结果。

- 后端服务会调用预训练的模型进行推理，并将结果传递给前端展示。

4. 数据存储层（Database）：

- MySQL 数据库用于存储用户信息、系统配置、设备、传感器数据以及模型预测结果。

- 提供数据持久化支持，保证数据安全和完整性。

组件交互

- 前端应用通过 Web API 与后端进行通信，发送请求和接收响应。

- 后端 API 服务器处理逻辑操作，并将数据请求转发到数据库服务。

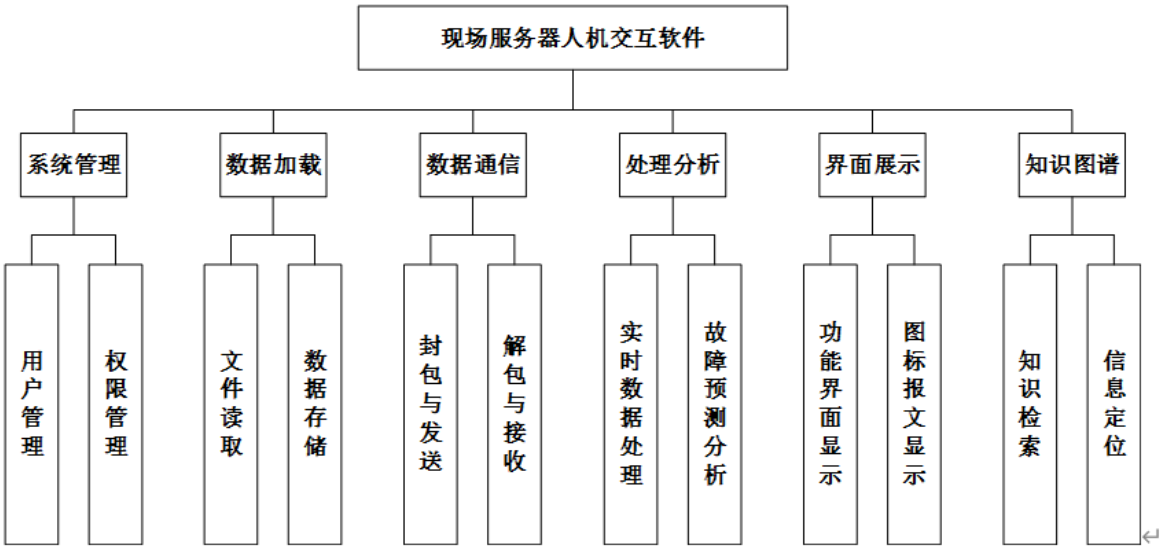
- API 服务器还负责从第三方设备数据接口获取数据，然后将数据整合并通过深度学习模型进行推理。

- 数据库不仅存储来自第三方接口的数据，也存储模型推理的输出。

安全性设计

- 使用 JWT（JSON Web Tokens）和 OAuth 用于管理用户会话和 API 访问控制。

3. 功能需求



3.1 用户角色

定义系统中的用户信息及认证。

系统管理员（Admin）

- 权限：
  - 管理用户账号，包括创建、编辑、禁用和删除用户。
  - 管理系统设置，包括配置参数。
- 目的：确保系统的正常运行，并处理用户相关的事务。

操作人员（Operator）

- 权限：
  - 查看实时数据和状态信息。
  - 访问历史数据。
  - 接收和响应系统预警。
- 目的：监控设备状态，确保生产安全和效率。

3.2 功能模块

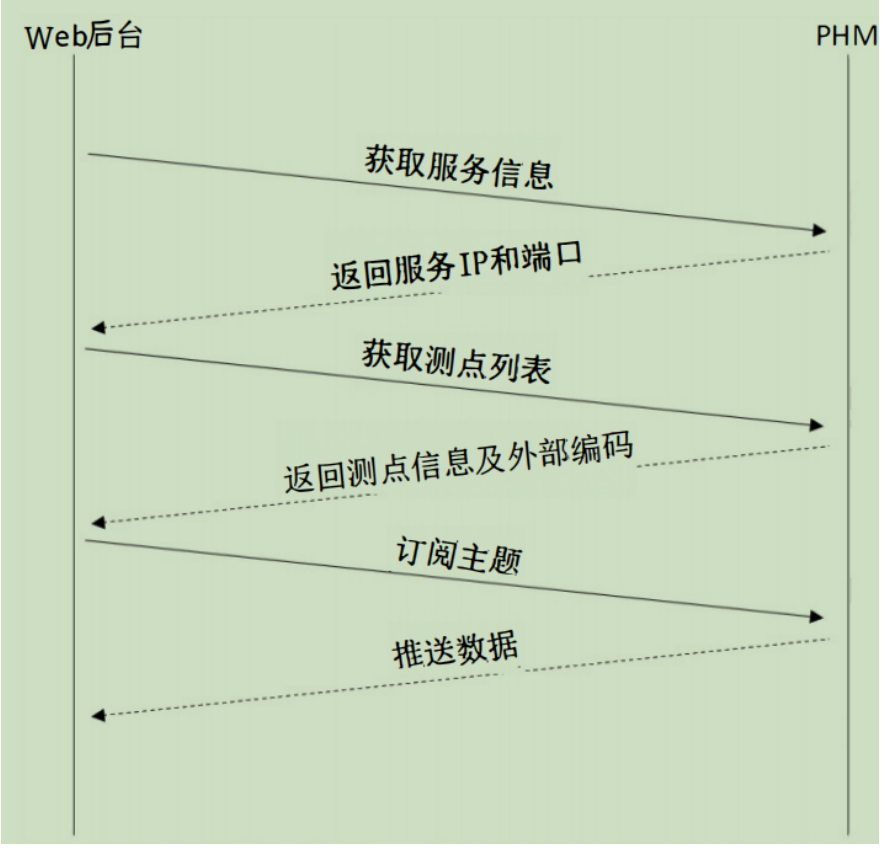
- 传感器数据采集：通过第三方接口获取传感器实时波形数据，包含温度和振动等类型
- 数据存储：实现传感器实时数据流的存储、检索和备份。
- 历史数据读取：根据时间、测点等多维度检索历史数据。
- 波形显示：展示每个传感器的温度和振动信号，支持时间范围选择和缩放。

- 故障预测与警报：使用深度学习模型进行故障预测，并将预测结果显示在界面上。
- 人员管理：提供用户认证、用户信息管理等。

3.2.1 传感器数据采集

本系统通过第三方接口获取传感器实时波形数据，包含温度和振动等类型。第三方平台为东华测试的智能设备维保管理平台，该平台提供了 4 种实时数据获取方式，包括 TCP 实时数据、SingalR 实时数据、MQTT 实时数据、Modbus 实时数据，鉴于本项目是基于 web 前端的监测系统，使用 MQTT 实时数据获取方式更适配本系统。

具体获取流程如下：



1. 登录 PHM 系统，确认 PHM 系统中已配置了 MQTT 推送服务。

2. 请求 MQTT 服务信息，获取服务 IP 与端口。

3. 请求测点列表，获取测点信息以及测点外部编码。

4. 使用上述步骤 3 中的测点信息以及外部编码订阅主题，支持订阅特征值数据、波形

数据与报警数据。

5. 侦听 Mqtt 服务地址与端口号，接收推送数据。

详细的接口定义在东华测试提供的接口文档中有说明。

东华设备平台用户名 :admin, 密码：phmpassword

3.2.2 数据存储

本系统使用开源版本的 MySQL8.0 数据库进行数据的存储和管理，数据库安装在本地主机上，与前端和后端共享主机资源。

3.2.3 历史数据读取

历史数据的存储将通过一个 拥有索引的优化查询系统实现快速检索功能。查询方法会包含时间范围选择、按设备类型过滤以及其他关键指标的筛选。历史数据的导出功能将允许用户将查询结果导出到 CSV 或 Excel 文件，以便进行报告或进一步分析。

3.2.4 波形显示

波形显示模块应具备实时和历史数据的图形化可视能力，包括温度和振动信号。用户应能够通过缩放和滚动工具深入分析具体时间段的数据，同时模块将支持数据点的选取，方便用户准确读取指定时刻的数值信息。

3.2.5 人员管理

人员管理模块将提账户注册。

3.3 界面要求

3.3.1 实时监控页面

3.3.1.1 单设备主页

展示单个设备，图片（可接视频通道）、基本参数（型号、功率、安装位置等）、工况参数（压力、排量、转速、电流、负载、负载率等）、状态参数（振动、温度等）、报警参数、故障参数等。可选取时间段访问历史数据。多以仪表盘、驾驶舱型式展示。

3.3.1.2 单场景主页（详细版）——用于较少设备在一个屏幕展示

展示多个设备，图片（可接视频通道）、基本参数（型号、功率、安装位置等）、工况参数（压力、排量、转速、负载、负载率等）、状态参数（振动、温度等）、报警参数、故障参数等。多以数据型式展示。

3.3.1.3 单场景主页（概述版）——用于较多设备在一个屏幕展示

展示多个设备，图片主要参数、报警信息、故障信息等。

3.3.2 数据管理页面

3.3.2.1 数据管理

查看、查询、监控各数据的运行情况

3.3.2.2 历史数据

单设备主页，选取时间回放数据，数据列表展示数据。

3.3.2.3 数据回传

配置需要上传云端数据。

3.3.3 故障诊断页面

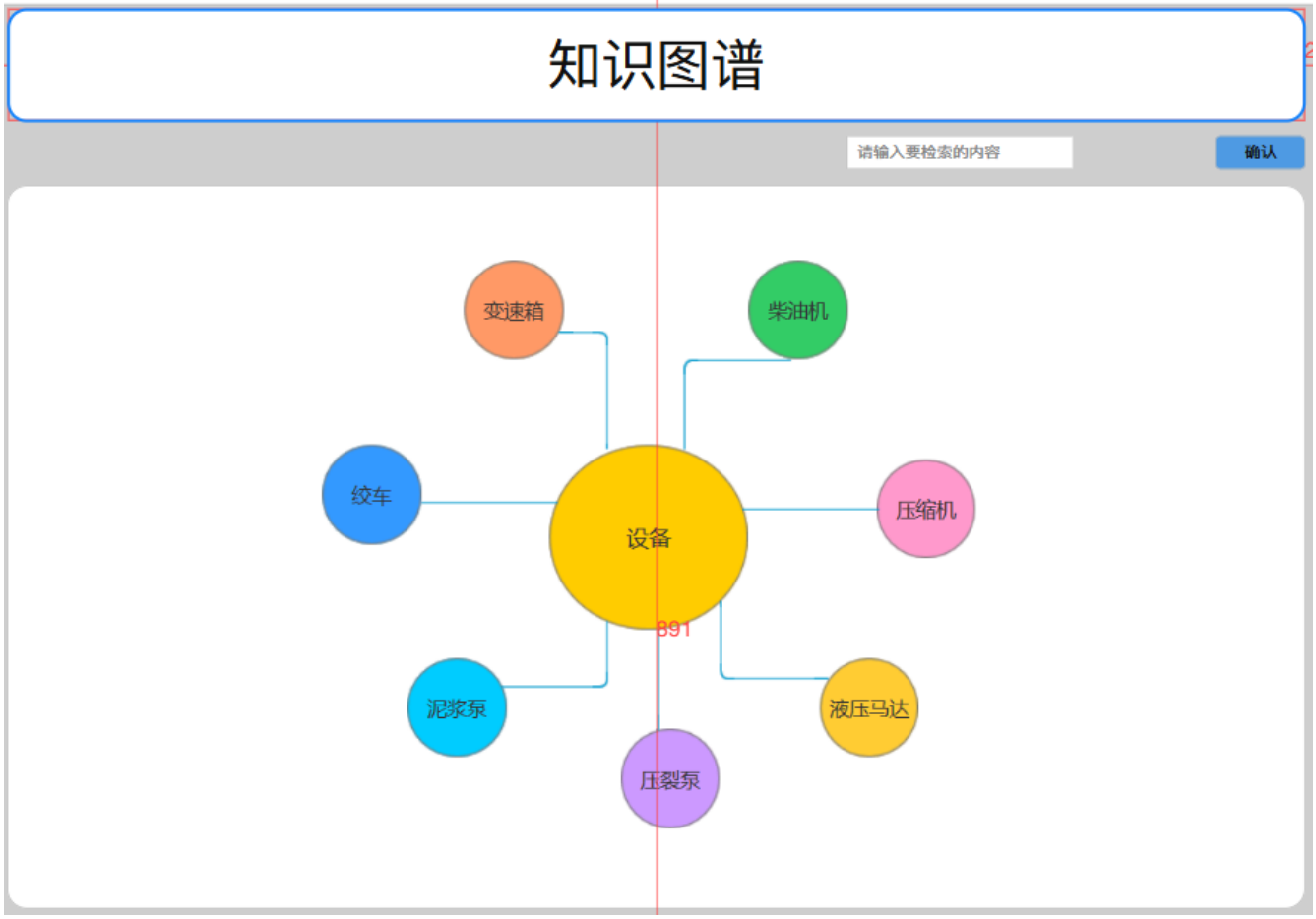
3.3.3.1 预警设置

对工况参数（压力、排量、转速、电流、负载、负载率等）进行分区间，设置振动、温度的报警参数值（分高报和高高报二级预警）。

3.3.3.2 故障诊断

选择数据，选择模型，开展故障分析工作。

3.3.4 知识图谱页面



3.3.5 系统配置页面

3.3.5.1 创建任务

公司（属性值）、部门（属性值）、场景（属性值）、设备（属性值）；属性值可缺省、可添加、可删除。

推荐属性值：公司：名称、地址、联系人、联系电话；

部门：名称、地址、联系人、联系电话；

场景：名称、地址、联系人、联系电话；

设备：名称、型号、... ...

“创建任务”纳入“系统配置”，不单独设置。

3.3.5.2 配置传感器

配置设备状态数据（压力、排量、转速等）、状态数据（振动、温度等）通道。

3.3.5.3 模型管理

创建上传模型，分设备、传感器配置故障诊断模型。

3.3.5.4 用户管理

创建、删除、管理用户

4. 系统详细设计

4.1 前端设计

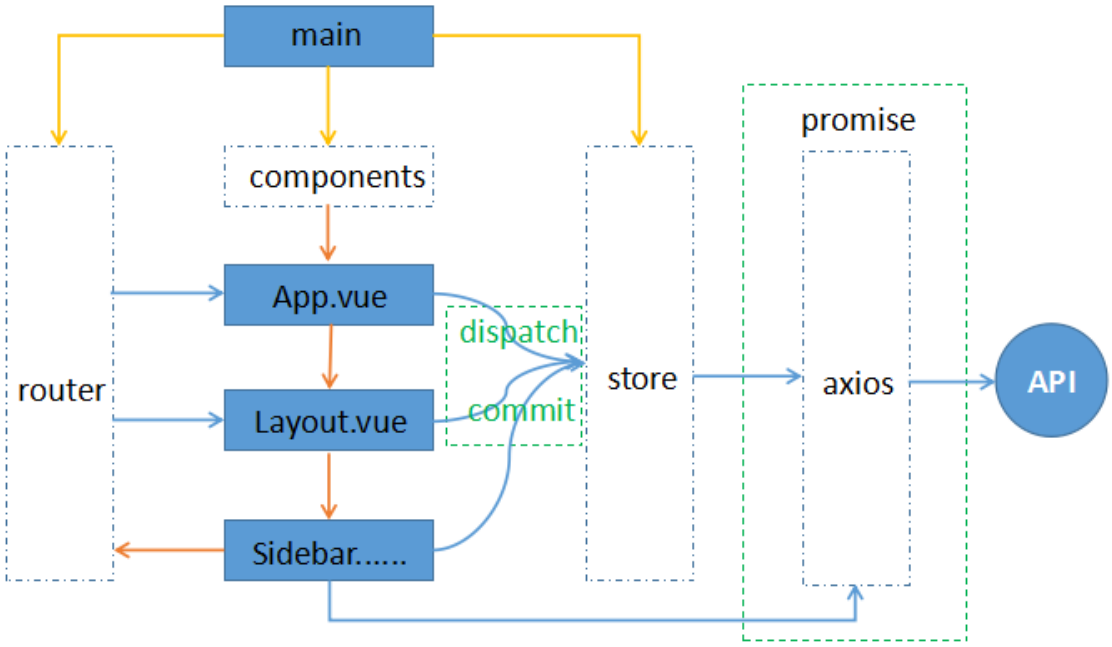
使用 VUE2 的详细设计说明。

- **项目结构**
  - **public 文件夹**：包含公共资源，如 `index.html` 文件和其他静态资源。
  - **src 文件夹：**
    - `assets` 文件夹：存放项目中使用的静态资源，例如图片、样式文件等。
    - `components` 文件夹：存放可复用的 Vue 组件，例如 Header 和 Footer。
    - `views` 文件夹：存放页面级别的 Vue 组件，例如 Home 和 About 页面。
    - `router` 文件夹：存放路由配置文件。
    - `store` 文件夹：存放 Vuex 状态管理相关文件。
    - `App.vue`：主应用组件。
    - `main.js`：项目的入口文件。
  - package.json 文件：定义项目的依赖和脚本。
- **主要依赖**
  - **vue**：Vue.js 的核心库。
  - **vue-router**：用于处理单页面应用的路由。
  - **vuex**：用于状态管理。
  - **axios**：用于进行 HTTP 请求。
  - **webpack/vite**：用于模块打包。
  - **babel-loader**：用于将 ES6+ 代码转化为向后兼容的 JavaScript 代码。
  - **vue-loader**：用于加载和编译 Vue 组件。
  - **vue-template-compiler**：用于编译 Vue 模板。
  - **css-loader** 和 **vue-style-loader**：用于处理 CSS 样式。
  - **echarts/heighcharts**：用于图表绘图。
  - **sass-loader** 和 **node-sass**：用于处理 Sass 样式。
  - 以及其他需要用到的工程库等等；

- **路由配置**
  - src 文件夹下的 router 文件
- **主应用文件** 文件名 main.js
  - **功能**：引入并初始化 Vue 实例，将 `App.vue` 挂载到 DOM 中的 `#app` 元素上。
  - **实现**：使用 `render` 函数渲染 `App` 组件，并将 `router` 和 `store` 注入到 Vue 实例中。
- **主要项目结构如下（示例）**

```
project-vue-app/
├── public/
│   ├── index.html
│   └── ...
├── src/
│   ├── assets/
│   ├── components/
│   │   ├── Header.vue
│   │   └── ...
│   ├── views/
│   │   ├── Home.vue
│   │   └── ...
│   ├── router/
│   │   └── index.js
│   ├── store/
│   │   └── index.js
│   ├── App.vue
│   ├── main.js
│   └── ...
├── package.json
└── ...
```

- **vue 原理图如下**



- **部署**

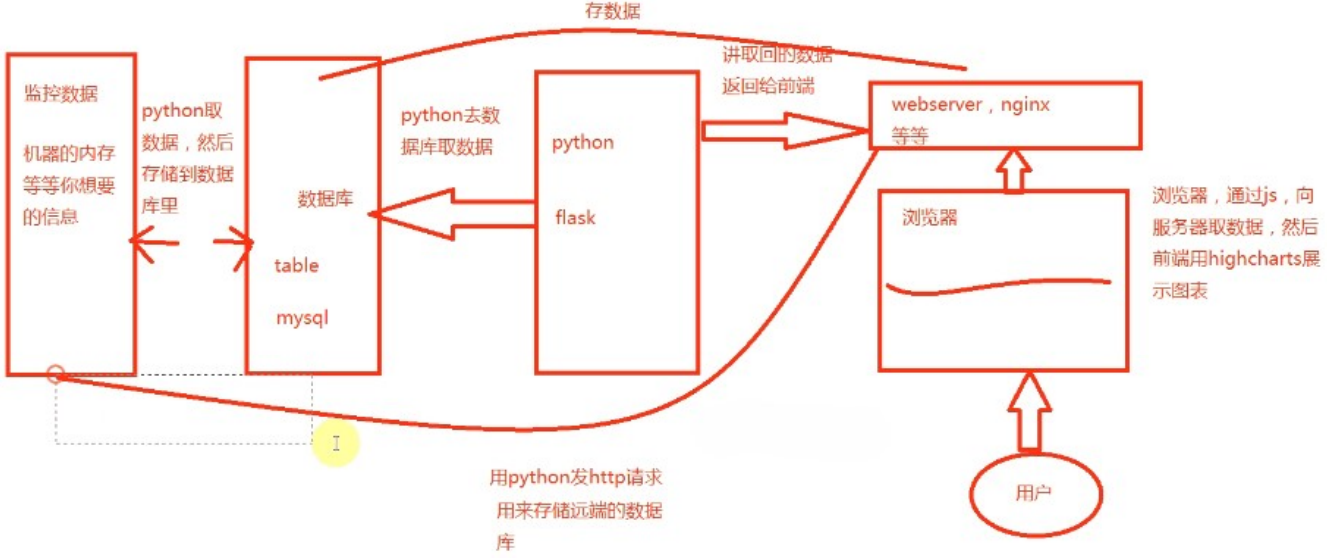
  采用 nginx 高性能服务器部署前端项目

  - 首先打包好 vue 前端项目，build 后的 dist 文件夹
    - npm run build:pro --> 生成 dist 文件
    - 将 dist 文件夹整体放入服务器中指定的地方
  - 安装好 nginx
    - 步骤省略
  - 配置 nginx 服务
    - 编辑 nginx.conf 文件，保存
    - 重启 nginx 服务

4.2 后端设计

详细描述 Python 服务的设置，包括路由器、控制器、服务层和域模型。

后端项目使用 flaks 作为基础框架，结合 websocket 实现数据实时推送，使用 mysql 进行数据存储。



- **项目结构**

首先，我们需要设计项目的文件结构，使得代码清晰且可维护。典型的项目结构如下：

```
project_root/
├── app.py
├── controllers/
│   ├── __init__.py
│   ├── mqtt_controller.py
│   └── websocket_controller.py
├── services/
│   ├── __init__.py
│   ├── mqtt_service.py
│   └── websocket_service.py
├── models/
│   └── Basemanager.py
└── requirements.txt
```

- **路由**

路由器负责定义应用程序的 URL 路径，并将这些路径映射到相应的控制器方法上。路由器设置在主应用文件 `app.py` 中。 示例如下：

Python

```python
    from flask import Flask
    from controllers import mqtt_controller, websocket_controller
    app = Flask(__name__)
    # 定义 MQTT 相关路由
    app.add_url_rule('/mqtt/subscribe', 'mqtt_subscribe', mqtt_controller.subscribe, methods=['POST'])
    app.add_url_rule('/mqtt/publish', 'mqtt_publish', mqtt_controller.publish, methods=['POST'])

    # 定义 WebSocket 相关路由
    app.add_url_rule('/ws/connect', 'ws_connect', websocket_controller.connect)
    app.add_url_rule('/ws/send_message', 'ws_send_message', websocket_controller.send_message, methods=['POST'])
```

- **控制器（Controller）**

控制器处理来自路由器的请求，并调用相应的服务层方法来实现具体的业务逻辑。控制器通常位于 `controllers` 文件夹中，每个模块有单独的控制器文件。示例如下

Python

```python
# controllers/mqtt_controller.py
from flask import request
from services import mqtt_service
def subscribe():
    topic = request.json['topic']
    mqtt_service.subscribe(topic)
    return f'Subscribed to topic: {topic}'
```

- **服务层（Service Layer）**

服务层包含了具体的业务逻辑，并与控制器进行解耦。它们位于 `services` 文件夹中，这里包含对 MQTT 和 WebSocket 的操作。 示例如下

Python

```python
# services/mqtt_service.py
import paho.mqtt.client as mqtt

def on_connect(client, userdata, flags, rc):
    print(f'Connected to MQTT broker with result code {rc}')

def on_message(client, userdata, message):
    print(f'Received message {message.payload.decode("utf-8")} on topic {message.topic}')

mqtt_client = mqtt.Client()
mqtt_client.on_connect = on_connect
mqtt_client.on_message = on_message
mqtt_client.connect('localhost', 1883, 60)
```

- **数据库操作 /BaseMysqlManager**

数据库操作主要与 mysIq 数据库交互，封装定义了数据库的操作，与处理层解耦。

Python

```python
class BaseMysqlManager:
    def __init__(self, db_name=DB["DATABASE"], user=DB["USER"], pwd=DB["PASSWORD"], host=DB["HOST"],
                 port=3306, charset=DB["CHARSET"]):
        """初始化数据库配置"""
        self.__db_name = db_name
        self.__user = user
        self.__pwd = pwd
        self.__host = host
        self.__port = port
        self.__charset = charset
        self.__connect = None
        self.__cursor = None
```

- **集成 MQTT 和 WebSocket**

在上述设计中，已经通过 `paho-mqtt` 库实现了 MQTT 的订阅和发布功能，通过 `flask-sockets` 实现了 WebSocket 的基本连接和消息发送功能。这些服务可以根据具体需求进一步扩展和调整。

- **总结**

通过这种结构化的方式，可以有效地组织和管理一个复杂的 Web 应用程序。使用 Flask 作为后端框架，结合 Vue.js 前端框架，以及 MQTT 和 WebSocket 实现实时通信，使得应用具有更高的扩展性和维护性。每一层次（路由器、控制器、服务层和域模型 < 数据库操作 >）都各自负责特定的任务，共同协作实现整个平台的功能。

4.3 模型部署

4.3.1 服务部署

- 格式化模型：首先，将训练好的模型转换为适用于生产的格式，如将 PyTorch 模型保存为`torchscript`或`ONNX`格式以便跨平台部署。
- 服务部署：模型作为单独 1 个服务进行运行，由于运行环境为 windows 系统，不建议将模型服务容器化。如果需要将模型封装，则将模型部署在一个独立的服务层中，通常是作为一个 REST API，使用 Flask 或 FastAPI 之类的框架。

4.3.2 数据接口与预处理

- 数据采集：定期从 web 后台接口调取新的数据。
- 数据预处理：使用与训练时相同的预处理步骤清洗并格式化数据，以满足模型输入的需求。
- 临时存储：web 后台将预处理后的数据暂存于数据库或内存中，以供模型调用。

4.4 数据库设计

数据库设计原则

数据库设计遵循以下原则：

- 规范化：以减少冗余和提高数据一致性为目的，通过合理的设计减少数据保存的重复。
- 可扩展性：设计时考虑到未来可能增加的数据类型和量，并为可能的系统扩展留出空间。
- 性能优化：根据预期的查询模式设计索引策略，确保数据存取的效率。
- 安全性：敏感信息如用户数据需加密存储，确保只有授权用户才能访问相关数据。

数据表结构

以下是核心数据表的结构设计，描述了每张表的主要字段和用途。

1. Users（用户信息表）

- UserId (INT, PK, Auto-Increment)
- Username (VARCHAR)
- PasswordHash (VARCHAR)
- Role (VARCHAR)
- CreatedAt (DATETIME)
- LastLogin (DATETIME)

- Description：存储用户登录相关信息，包括加密的密码、角色和登录记录。

2. Devices（设备信息表）

- DeviceId (INT, PK, Auto-Increment)
- Name (VARCHAR)
- Type (VARCHAR)
- Status (VARCHAR)
- InstallDate (DATETIME)

- Description：登记设备的基本信息及其运行状态。

3. Sensors（传感器信息表）

- SensorId (INT, PK, Auto-Increment)
- DeviceId (INT, FK)
- Type (VARCHAR)
- Location (VARCHAR)

- Description：记录与设备关联的传感器明细。

4. SensorData（传感器数据表）

- DataId (INT, PK, Auto-Increment)
- SensorId (INT, FK)
- Temperature (FLOAT)
- Vibration (FLOAT)
- Timestamp (DATETIME)

- Description：存储传感器的实时数据。

5. Alarms (报警信息表)

- AlarmId (INT, PK, Auto-Increment)
- DeviceId (INT, FK)
- Type (VARCHAR)
- Severity (VARCHAR)
- Timestamp (DATETIME)
- Cleared (BOOL)

- Description：保存报警情况，包括类型、严重性和状态。

6. SystemLogs（系统日志表）

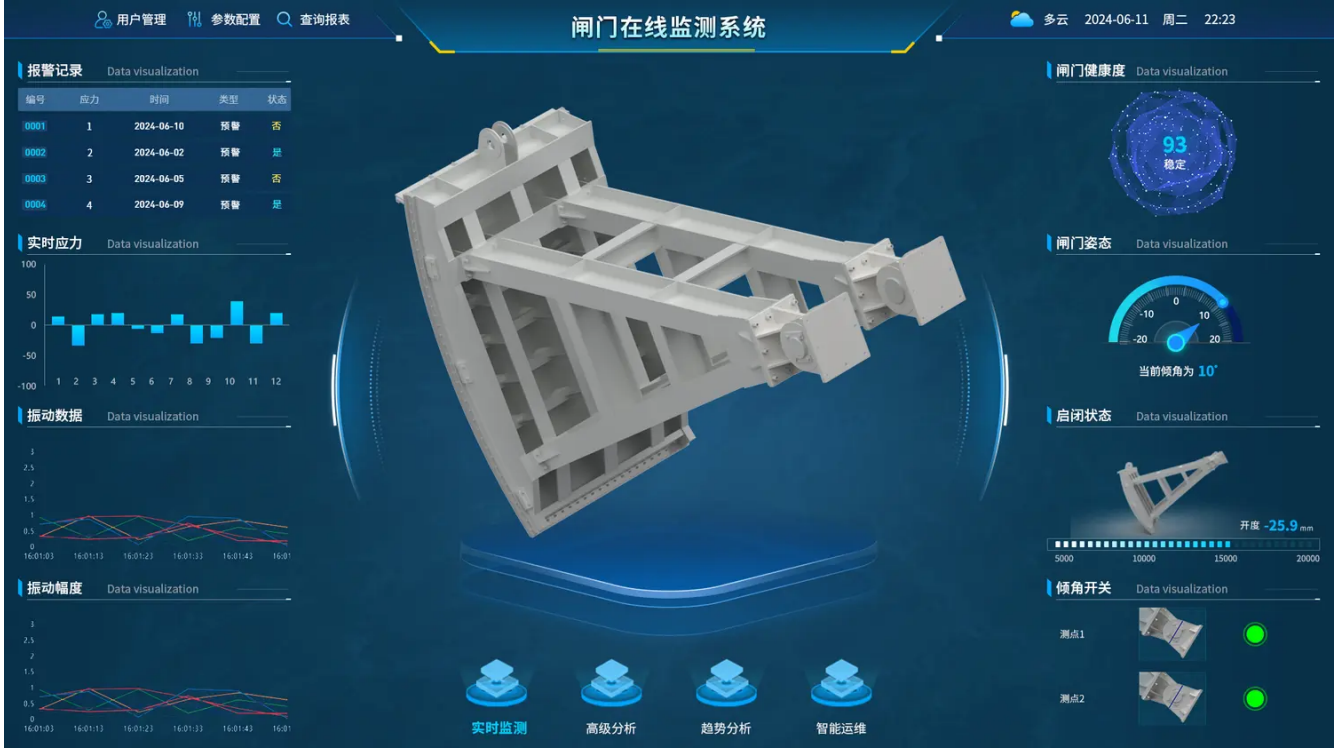
- LogId (INT, PK, Auto-Increment)
- UserId (INT, FK)
- Action (VARCHAR)
- Description (TEXT)
- Timestamp (DATETIME)

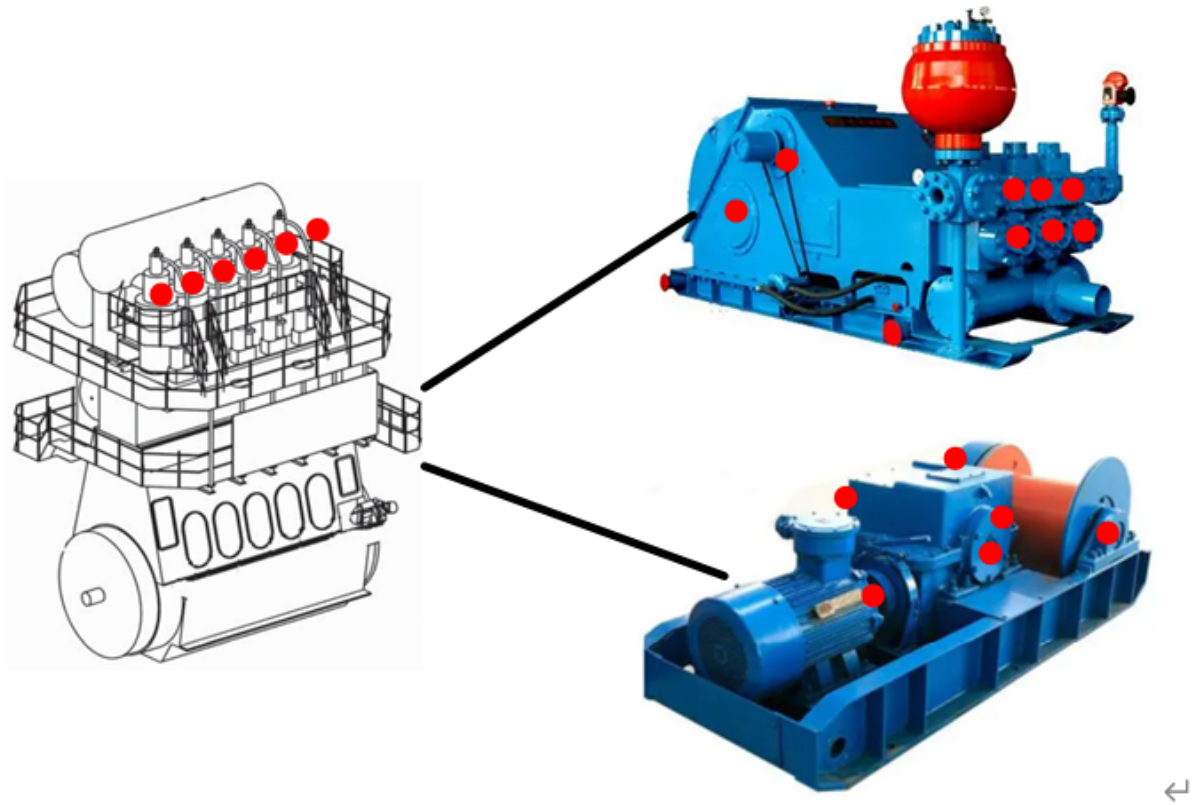
- Description：记录系统的操作日志，用于审计。

关系及索引

- 设备信息表(`Devices`)与传感器表(`Sensors`)以及报警信息表(`Alarms`)有关联（外键约束），表示它们之间的所属关系。
- 传感器表(`Sensors`)与传感器数据表(`SensorData`)相关联，实现数据与所属传感器之间的联系。
- 对于需要频繁查询的字段（如`Username`，`DeviceId`，`SensorId`，`Timestamp`等），将创建必要的索引以加速检索速度。

5. 界面设计





- 整体布局参考上面的设计图。
- 主界面：以模型静态图为中心，周围配以图表，少数据和文字，确保一目了然。
- 辅助界面：包括设置、历史记录和帮助文档等，通过顶部菜单栏和底部按钮访问。顶部可跟上述界面一致，底部采用我们自己的原有的方案，用按钮切换。底部按钮为：实时监测、模型分析、历史查询 3 个按钮。

6. 性能需求

本系统无多用户并发访问需要，主要是满足以上的功能要求，保持长时间的稳定运行。

7. 测试计划

测试计划的目的是确保软件的质量和稳定性。本项目主要包括完整功能流程的集成测试。

集成测试

- 目标：确保模块间接口连通，并在集成后表现出预期的协调行为。
- 实现：使用头部模式和无头模式进行测试， 这类工具通常能够在“头部”模式下运行（也就是打开实际的浏览器窗口操作），或者在“无头”模式下运行（无 UI 界面，适合 CI/CD pipelines）。
- 范围：
  - 用户操作流程（如登录、数据检索、数据提交）
  - 后端服务与数据库之间的交互
  - 深度学习模型的集成和数据响应

测试流程

- 模拟请求 / 响应：使用测试客户端发送 HTTP 请求到你的后端服务，并验证响应是否符合预期。这通常会涉及到调用 RESTful API 的端点。
- 数据库状态验证：在运行测试前后验证数据库的状态变化，确保业务逻辑被正确处理并反映在数据库中。
- 代码审查：在合并代码前进行人工代码审查。
- 功能测试：定期进行，特别是在系统升级后，以确保满足设计功能要求。

测试标准与文档

- 测试用例：编写明确的测试用例并记录测试结果。
- 文档化：所有的测试计划和结果应该被完整地记录并可用于审计。

8. 部署计划

本项目的部署计划主要包含后期生产环境的部署及维护文档的编写和用户培训。

生产环境部署

- 详细步骤：包括环境检查、代码部署、数据库迁移和服务启动。

训练和文档

- 用户培训：为系统管理员和运维人员提供培训。
- 文档编写：准备详细的部署手册，以便于故障排除和操作参考。部署计划的详细化有助于无缝和正确地将软件从开发阶段迁移到生产环境，并保证服务的高可用性和可靠性。