系统概要设计方案

1. 引言

1.1. 编写目的

本文档旨在从技术实现层面，阐明系统的整体架构、关键技术选型、核心模块划分、数据流转机制以及部署策略，为项目的详细设计和开发实施提供统一的技术蓝图和指导原则。

1.2. 设计范围

本文档覆盖了系统的前端、后端服务网关、后端计算引擎三个主要部分的设计，并对它们之间的接口和交互方式进行了定义。

1.3. 关键设计目标

- **模块化**: 各层及内部模块应高度解耦，便于独立开发、测试和维护。
- **可伸缩性**: 架构应支持未来对无状态服务的水平扩展。
- **易用性**: 面向工程师用户，提供清晰、高效的工作流和交互体验。
- **环境一致性**: 通过容器化技术，确保开发、测试和生产环境的统一。

1.4. 定义

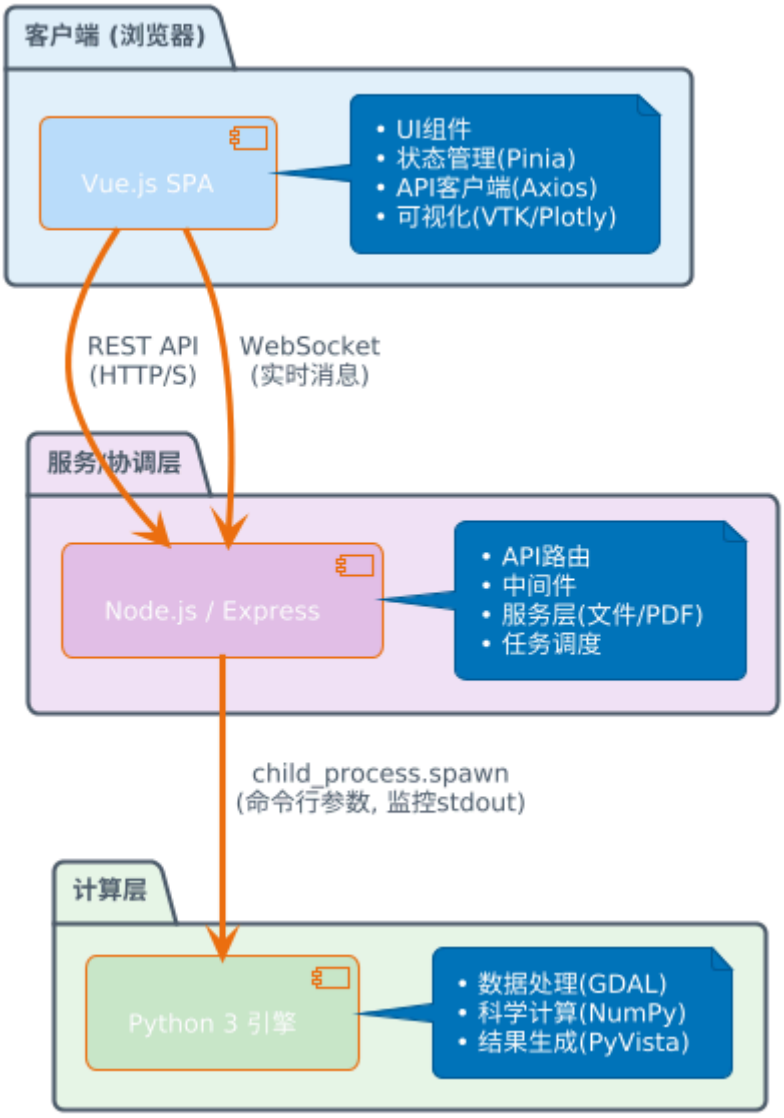
- **工况 (Case)**: 系统中的核心概念，指一个独立的、自包含的仿真项目。它以一个文件夹的形式存在，内部包含了该项目的所有输入数据、配置文件和输出结果。
- **DEM**: 数字高程模型 (Digital Elevation Model)，描述地表高程的栅格数据文件。
- **VTK**: The Visualization Toolkit，一个用于3D计算机图形、图像处理和可视化的开源软件系统。
- **info.json**: 关键的配置文件。由系统根据用户的所有输入（参数、风机、地形等）自动生成，是驱动后端Python计算引擎的唯一、最终的输入文件。
- **BFF**: Backend-for-Frontend，一种架构模式，即为前端定制的后端服务。
- **SPA**: 单页应用 (Single-Page Application)。

---

2. 系统架构设计

2.1. 整体架构

系统采用经典的前后端分离架构，并引入了API网关和独立的计算引擎，形成了一个职责清晰的三层模型：表示层、服务/协调层、计算层。



2.2. 技术栈选型

层次	技术	目的
前端	Vue 3 (Composition API), Vite, Pinia, Vue Router, Axios, Element Plus	构建响应式、高性能的现代化用户界面
可视化	VTK.js, Three.js, ECharts, Plotly.js, Leaflet	实现专业的2D/3D科学计算可视化和地理信息展示
后端网关	Node.js, Express.js, Socket.io, Multer, Winston	高并发I/O处理、任务调度、实时通信
计算引擎	Python 3, NumPy, Pandas, GDAL, Rasterio, PyVista, Matplotlib	高性能科学计算、地理空间数据处理、数据分析
部署	Docker, Docker Compose	提供一致、可移植、可伸缩的运行环境

2.3. 部署设计

- **容器化编排**: 项目根目录下的 `docker-compose.yml` 文件定义了应用的所有服务（如 `frontend`, `backend`），以及它们之间的网络和数据卷依赖。
- **数据持久化**: 用户数据（如工况目录）通过Docker数据卷（Volumes）映射到宿主机的文件系统上，确保容器销毁后数据不会丢失。
- **一键启动**: `start.sh` 封装了启动命令，实现了整个应用环境的一键式构建和启动。

---

3. 数据设计

3.1. 核心数据模型：工况 (Case)

系统不采用传统数据库，其核心数据模型是“工况”。一个工况在物理上对应一个独立的文件夹，包含了该仿真项目从输入到输出的全部信息，具有高度的内聚性和可移植性。

3.2. 持久化方案：目录结构

每个工况目录（`/backend/uploads/{caseId}`）的内部结构是固定的，作为其数据模式（Schema）：

- **输入层**: `terrain.tif`, `wind_turbines.json`, `parameters.json`, `customCurves/`, `rou`
- **配置层**: `info.json` (由系统生成的最终计算配置文件)
- **输出层**: `run/` (包含 `VTK/` 和 `Output/` 子目录)
- **缓存层**: `visualization_cache/` (用于加速前端可视化的预处理数据)

3.3. 核心数据流

以“配置固化并启动计算”为例，数据流转如下：

1. **[前端] 汇集**: 用户在 `ParameterSettings.vue` 点击“提交参数”。
2. **[前端->后端] 请求**: 前端向 `POST /api/cases/:caseId/info` 发送请求，请求体中包含所有从 `caseStore` 中获取的参数和风机数据。
3. **[后端] 生成配置**: 后端API读取请求数据，并结合服务器上存储的地形、粗糙度等文件，生成最终的 `info.json` 并保存。
4. **[后端] 锁定**: `info.json` 生成成功，工况进入“已配置”状态。
5. **[前端] 启动**: 用户点击“开始计算”。
6. **[后端->计算引擎] 任务派发**: 后端API通过 `child_process.spawn` 调用Python计算脚本，并将 `info.json` 的路径作为命令行参数传入。
7. **[计算引擎->后端->前端] 结果回传**: 计算结果文件被写入 `run/` 目录。计算进度和日志通过 `stdout` -> `Node.js` -> `WebSocket` 的链路实时返回给前端。

---

4. 模块化设计

4.1. 前端模块 (`frontend/`)

- **视图层 (`src/views/`)**: 定义了"工况列表 -> 新建工况 -> 工况详情"的核心页面流程。
- **组件层 (`src/components/`)**: 提供了可复用的功能单元，如 `ParameterSettings.vue` (参数配置)、`WindTurbineManagement.vue` (风机管理与分析)、`VTKViewer.vue` (3D渲染)。
- **状态管理层 (`src/store/`)**: 使用Pinia进行状态管理。`caseStore.js`是架构核心，作为客户端当前工况所有数据的"单一事实来源"，解耦了各个复杂组件。
- **API服务层 (`src/api/`)**: 使用Axios封装对后端API的调用，实现了业务逻辑与网络请求的分离。

4.2. 后端网关模块 (`backend/`)

- **路由层 (`routes/`)**: 定义了RESTful API端点，以`cases.js`为核心，按资源对工况的整个生命周期进行管理。
- **中间件层 (`middleware/`)**: 提供了请求处理的横切关注点功能，如`fileUpload.js`处理文件上传，`statusCheck.js`防止任务重复执行，`errorHandler.js`进行统一异常捕获。

4.3. 计算引擎模块 (`backend/utils/` & `backend/base/solver`)

- **接口约定**: 所有Python脚本都遵循统一的调用约定：通过`argparse`解析命令行参数获取输入，通过向`stdout`打印JSON字符串来返回结构化的进度和结果。
- **职责划分**: 每个脚本都是一个独立的、功能单一的工具。例如，`precompute_visualization.py`专用于预处理可视化数据，`terrain_clipper.py`专用于裁剪地形。`base/solver`中的脚本则负责更底层的模拟计算任务。

---

5. 接口设计

5.1. 前后端通信接口

- **REST API**: 用于客户端发起的、有明确响应的请求-应答式交互。接口遵循HTTP动词规范（GET, POST, PUT, DELETE）。
- **WebSocket API**: 用于服务器主动向客户端推送消息。后端通过Socket.io房间机制（以`caseId`作为房间名）向特定客户端广播事件，如`calculation_progress`，`calculation_completed`。

5.2. 后端-计算引擎接口

- **通信方式**: 单向、异步的进程间通信。
- **协议**: Node.js通过`child_process.spawn()`调用Python进程，并通过**命令行参数**传递所有输入。Python进程通过向**标准输出(stdout)**打印JSON格式的字符串来回传信息。

---

6. 可视化设计

- **3D地形与风机渲染**: 使用 **Three.js** 作为核心3D渲染引擎，负责场景、光照、相机和对象的管理。通过加载 `terrain.tif` 生成地形网格，并加载 `.gltf` 格式的风机模型，实现风场的真实感渲染。
- **流场与数据可视化**:
  - **VTK.js**: 用于处理和渲染复杂的科学计算数据，如速度场、流线等 `.vtp` 文件。
  - **ECharts & Plotly.js**: 用于生成二维数据分析图表，如风速廓线、功率对比、性能变化率等，提供丰富的交互功能。
- **实时交互**: 通过 **OrbitControls** 实现场景的缩放、平移和旋转。通过 **Raycaster** 实现风机模型的点选和信息提示。

7. 安全性、错误处理与性能

- **安全性**:
  - 文件上传通过 **Multer** 中间件进行处理，限制文件大小和类型，防止恶意文件上传。
  - 对所有用户输入进行基础的验证和清理，防止注入攻击。
- **错误处理**:
  - **前端**: 使用组件的 `onErrorCaptured` 和全局错误处理器来捕获和显示错误信息。
  - **后端**: 使用统一的 `errorHandler` 中间件捕获所有路由和异步操作中的异常，返回标准格式的错误响应。
  - **计算引擎**: Python脚本通过向 `stderr` 输出错误信息，并以非零状态码退出，来向上层报告错误。
- **日志**:
  - **后端**: 使用 **Winston** 库记录不同级别的日志（info, warn, error）到文件和控制台，便于问题排查。
- **性能**:
  - **前端**: 使用 **Vite** 进行快速的冷启动和热更新。对大型数据集和复杂的可视化组件进行按需加载和懒加载。
  - **后端**: Node.js 的异步非阻塞I/O模型适合处理高并发的API请求和WebSocket连接。
  - **计算**: 计算密集型任务被委托给独立的Python进程，避免阻塞Node.js事件循环。

8. 扩展性

- **模块化设计**: 前后端和计算引擎的解耦设计，允许对任何一层进行独立的升级和替换。
- **容器化**: Docker化的部署方式使得应用可以轻松地在不同的环境中迁移和部署。
- **无状态服务**: 后端API被设计为无状态的，便于未来通过负载均衡进行水平扩展。